PROGRAMOVACÍ TECHNIKY

STUDIJNÍ MATERIÁLY URČENÉ PRO STUDENTY FEI VŠB-TU OSTRAVA

VYPRACOVAL:
MAREK BĚHÁLEK

OSTRAVA 2006

OBSAH

1 Průvodce studiem

V této kapitole budou stručně představeny požadavky kladené na studenta v předmětu Programovací techniky.

1.1 Charakteristika předmětu Programovací techniky

Předmět pokrývá oblast metod návrhu a realizace programových aplikací, jejich testování a dokumentace. Získáte praktické dovednosti při používání samostatných i integrovaných vývojových nástrojů. Také se seznámíte s principy komponentních technologií a hlavními představiteli těchto komponentních technologií. Poslední část kurzu tvoří některé pokročilé algoritmy používané při programovaní.

Obsah předmětu lze rozdělit do těchto čtyř částí.

1. **Nástroje pro vývoj softwaru** – v této části budou probrány různé nástroje, které se používají při vývoji softwarových produktů. V kurzu budou představeny nástroje pro sestavení, správu či uchovávání verzí aplikace. Jsou probírány například základní principy při kompilaci nebo třeba možnosti jak hledat, uchovávat a odstraňovat chyby.
2. **Komponentní technologie** – v této části budou představeny základní principy komponentně orientovaného programování. Dále se pak detailněji rozebírají tyto tři komponentní technologie.
   - **Component Object Model** – technologie vyvinuta firmou Microsoft. Je určená zejména pro platformu Windows. Tato technologie je postavená na binárně kompatibilních komponentách. Pro praktické testování jsou zvoleny ukázky implementované v jazyce C.
   - **Java Beans** – komponentní technologie firmy Sun. Tato komponentní technologie má přímou návaznost na programovací jazyk Java. Jsou představeny základní principy, na kterých je tato technologie postavena a schémata jejího použití.
   - **Komponenty v .NET** – nejnovější technologie firmy Microsoft. V rámci platformy .NET je implementovány nativní podpora komponentně orientovaného programování. V rámci této kapitoly bude realizován stručný tutoriál jazyka C#, na kterém jsou pak možnosti komponent u platformy .NET prakticky demonstrovány.
3. **Pokročilé algoritmy** – v rámci této kapitoly budou představeny algoritmy pro správu paměti, základní algoritmy pro kompresi dat a základní principy programování více vláknových aplikací.

1.2 Další informace k předmětu Programovací techniky

Hlavní zdroj informací pro předmět Programovací techniky je internetová stránka http://www.cs.vsb.cz/behalek/vyuka/pte/. Na těchto stránkách je

k dispozici nejen tento text, ale taká prezentace použité při přednáškách v tomto kurzu. Také zde najdete celou řadu praktických příkladů určených jak pro výuku v rámci cvičení tak pro další samostudium.

2 Nástroje pro tvorbu programu

V této kapitole se dozvíte:

- Jak vypadá životní cyklus aplikace.
- Jaké typy nástrojů se používají v jednotlivých etapách.
- Seznámíte se s některými zástupci těchto nástrojů. Zejména jde o nástroje pro kompilaci, správu verzí, sestavení aplikace, ladění, testování a dokumentaci aplikace.
- Jsou rozebrány možnosti a omezení jednotlivých nástrojů.

Po jejím prostudování byste měli být schopni:

- Používat různé nástroje určené pro zjednodušení vývoje aplikací.
- Najít a použít hlavní představitele různých tříd nástrojů.
- Být schopni zvolit nástroj vhodný pro řešení specifických situací při vývoji aplikace.
- Vyvíjet software v rámci týmu lidí.

Klíčová slova této kapitoly:

životní cyklus aplikace, správa projektu, stráva verzí, testování, CVS, Ant, IDE, debugger, refactoring, JUnit,…

Doba potřebná ke studiu: 10 hodin

Průvodce studiem

Studium této kapitoly je jednoduché a popisným způsobem zde nastudujete různé typy nástrojů. Prakticky si pak získané informace můžete ověřit na připravených úkolech. Ty nejsou součástí tohoto textu a jsou dostupné prostřednictvím Internetu.
Na studium této části si vyhraďte minimálně 10 hodin. Po celkovém prostudování doporučuji vyřešit praktické úkoly. Na tuto část si vyhraďte dalších 12 hodin. (kapitolu nemusíte studovat najednou).

Při vytváření softwarového produktu je prováděná celá řada aktivit. Následující výčet uvádí aktivity, které se pravděpodobně objeví při vývoji nějaké aplikace. Vlastní výčet je ovšem jen velmi nepřesný popis toho, co se skutečně děje při vytváření aplikace. Jednotlivé aktivity se nemusí realizovat v uváděném pořadí, často se provádějí paralelně či opakují vícekrát.

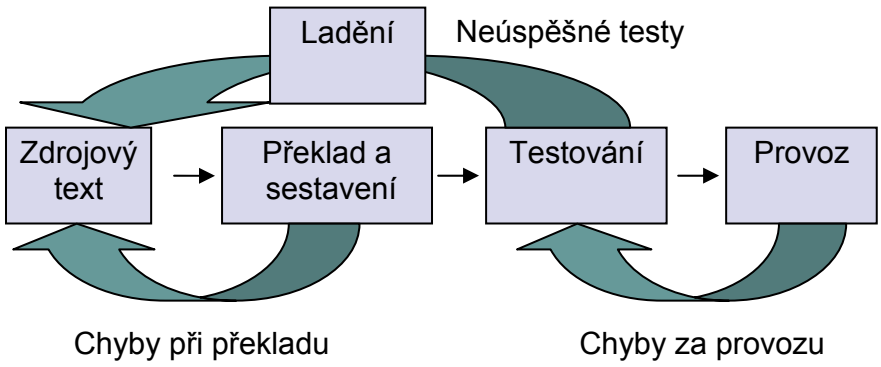
- Rozhodnutí o vytvoření produktu
- Specifikace požadavků
- Analýza a návrh aplikace
- **Implementace**
- **Testování a ladění**
- **Dokumentace**
- **Nasazení**
- Marketing, prodej a podpora
- **Údržba**
- Ukončení prodeje a podpory

Vlastní výčet aktivit je ovšem jen velmi nepřesný popis toho, co se skutečně děje při vytváření aplikace. Jednotlivé aktivity se nemusí realizovat v uváděném pořadí, často se provádějí paralelně či opakují vícekrát. To jakým způsobem vytvářet aplikaci, jak tento vývoj řídit, nebo jak se starat o marketing vytvářené aplikace je mimo rámec tohoto kurzu. Těmito oblastmi by se zabývaly vědní disciplíny jako Softwarové inženýrství či specifické oblasti Ekonomie. V této kapitole se zaměříme na nástroje, které nám můžou usnadnit zejména praktickou realizaci dané aplikace. Půjde například o nástroje spojené s tvorbou zdrojových kódů, jejich sestavením testováním laděním. Z uvedených aktivit se zaměříme zejména na nástroje, které nám můžou usnadnit implementaci, testování, dokumentování, nasazení a údržbu vytvářené aplikace (aktivity byly ve výčtu označeny tučně).

Existují také různé nástroje pro podporu ostatních aktivit. Například pro analýzu a návrh aplikace můžeme použít jazyk UML. Existuje celá řada různých nástrojů, které nám umožňují vytvořit diagramy v tomto jazyce. Těmito nástroji se v této kapitole zabývat nebudeme.

2.1 Tvorba aplikace

Pokud se při vývoji aplikace zaměříme na tvorbu zdrojových kódu a vytváření funkční verze, která pak bude nasazena do provozu, můžeme tento proces zjednodušit následujícím schématem.



V první fázi jsou vytvářeny zdrojové kódy. Při jejich vytváření použijeme nějaký vhodný editor. Pro uložení zdrojových kódu pak nějaký nástroj pro správu verzí. Ten nám umožní nejen sledovat a procházet různé verze zdrojových kódů, ale také usnadní týmovou práci s nimi.

V další fázi budeme vytvořené zdrojové kódy překládat. K tomu jistě využijeme služeb nějakého překladače. Vlastní překlad či lépe sestavení aplikace může být poměrně obtížné a výhodné může být použití nějakého nástroje pro správu aplikace. Primárním úkolem takovéhoto nástroje bude sestavení aplikace, ale kromě něj se většinou dokáže postarat i o další úkoly (například smazání dočasných souborů, spuštění testů či instalaci).

Před vlastním nasazením je potřeba aplikaci otestovat. Snažíme se najít všechny chyby, tedy nedostatky, které by omezovaly a nebo znemožnily provoz budované aplikace. Pokud objevíme nějakou chybu, snažíme se ladit vytvářenou aplikaci a nalézt a odstranit tuto chybu.

Poslední fázi bude nasazení aplikace do provozu. Tento krok nemusí být triviální. Zde nám může práci usnadnit například nástroj pro správu aplikace, nebo můžeme použít různé generátory instalátorů.

V další části budou představeny různé nástroje, které se používají při vývoji aplikace. V této kapitole se nebudeme zabývat celou řadou dalších nástrojů. Jednou z významných skupin nástrojů (kterou v tomto kurzu přesto přeskočíme) jsou integrovaná vývojová prostředí (IDE). Tyto nástroje integrují celou řadu funkcí současně. Může se jednat o funkce, které realizují nástroje představeny dále v tomto textu. V principu můžeme IDE rozdělit na dvě skupiny. Orientované na určitý jazyk (například Borland Pascal, C++, JBuilder, C# Builder), SharpDeveloper, JCreator, NetBeans) a nebo na univerzální prostředí podporující více programovacích jazyků (Elipse, MS Visual Studio). IDE často také podporují možnost přidávat nové funkce. Nejlepším příkladem je Eclipse. Eclipse je volně dostupný a „open source“ produkt. Díky tomu existuje celá řada rozšíření, které jsou dostupné prostřednictvím Internetu.

Další z tříd nástrojů, které nebudou součástí tohoto kurzu jsou Preconstructed Development Environments (PDE). Jiné používané jméno je Collaborative Development Environments. Jde o předpřipravenou ucelenou sada nástrojů pro vývoj aplikace. Takové prostředí obsahuje nástroje pro správu projektu, sestavení aplikace či generování dokumentace. Tvůrce aplikace nevytváří a neprovozuje tyto nástroje, ale používá prostředí předpřipravené někým jiným. Obvykle je pro komunikaci použito webové rozhraní. Nejznámější PDE je SourceForge (http://www.sourceforge.net). Je primárně určeno pro „open source“ projekty. V roce 2005 obsahovalo okolo 100 000 projektů a 1 milión uživatelů. Další PDE: GFroge, CollabNet, Savane, BerliOS.

Poslední skupinou nástrojů, která nebude součástí tohoto kurzu, jsou nástroje, které se snaží automatizovat celý proces vývoje aplikace. V dnešní době je poměrně značné úsilí věnované tomu, automatizovat proces vývoje aplikace a použití různých nástrojů při jeho vývoji. Jako příklad si můžete představit testování. Výsledkem testování je nějaká sada chyb. Ty pak musí tester vložit do systému pro správu verzí. Tento krok jsem ovšem v principu schopni provést automaticky. Tyto nástroje se snaží automatizovat použití ostatních nástrojů například pro správu verzí, sestavení aplikace nebo generování dokumentace. Představiteli takových nástrojů by byly produkty jako Mawen či Anthill.

2.2 Editor

Zdrojové kódy můžeme v principu vytvářet v libovolném textovém editoru (splňuje-li nároky na použité kódování a podobně). Ovšem použitím dobrého editoru, si můžeme usnadnit a zjednodušit psaní zdrojových kódů. Aktuálně existuje celá řada různých editorů. Takový editor může být samostatná aplikace. Také může být součástí komplexnějšího produktu. Tak je tomu například u integrovaných vývojových prostředí jako je třeba Eclipse. Editory bývají specializované na určitý programovací jazyk a nebo univerzální. Univerzální editory podporují více programovacích jazyků a často dávají uživateli možnost množinu podporovaných jazyků měnit či upravovat.

Typicky editor určený pro psaní zdrojových kódů podporuje následující vlastnosti.

- Zvýraznění syntaxe (syntax highlighting) – při vytváření zdrojových kódů je použit nějaký programovací jazyk. Editor je schopen graficky zvýraznit některé konstrukce tohoto jazyka. Tím zlepšuje čitelnost zdrojových kódů.
- Kontrola závorek
- Vyhledávání a nahrazování
- Šablony a makra
- Sbalení textu (folding)
- Spolupráce s různými externími nástroji například s nástroji pro správu verzí.

Konkrétní editor pak může obsahovat celou řadu dalších funkcí a vlastností. Jako příklady editorů můžeme použít: PsPad (ke stažení na http://pspad.cincura.net), gvim, emacs, a nebo JEdit.

2.3 Překladač

Důležitým nástrojem při implementaci aplikace je překladač. Díky němu jsem schopni použít při realizaci aplikace vyšší programovací jazyky. Úlohou překladače je nejčastěji právě překlad nějakého programovacího jazyka vyšší úrovně (jako je například C) do formy, kterou jsme schopni spouštět na platformě, pro kterou je vytvářena aplikace určena.

> ➢ **Obecně bychom překlad mohli definovat jako zobrazení ze zdrojového jazyka do jazyka cílového.**

Jde o proces, při kterém je program ve zdrojovém jazyce přeložen na ekvivalentní program v jazyce cílovém. V principu můžeme chtít překládat libovolný (například přirozený) jazyk. V oblasti informatiky se ovšem dnes nejčastěji setkáte s překladači programovacích jazyků, případně jiných specializovaných jazyků jako je LaTeX, WHLD, a nebo HTML. Tyto jazyky se vyznačují tím, že oproti například přirozenému jazyku je jsme schopni exaktně a formálně popsat. K takovému popisu nejčastěji slouží gramatiky (obecně uznávaný standard je BNF – Bacus-Noir Form). Fakt, že jsme schopni překládané jazyky formalizovat, usnadňuje vytváření příslušného překladače.

Pro vytváření aplikací je nejzajímavější překlad vyšších programovacích jazyků. Použití vyšších programovacích jazyků jako jsou C, C++, C#, Java, Haskel, Smal Talk, Self, Prolog, PHP a podobně značně usnadňuje vývoj aplikace. Můžeme se pohybovat na vyšší úrovni abstrakce. Překladač se pak postará o převod takovýchto programů do formy, která je spustitelná na dané platformě. Cílový jazyk tedy nejčastěji je nějaká forma strojově orientovaného jazyka nebo jazyk nějakého virtuálního procesoru.

Koncepce virtuálního procesoru vychází z toho, že zdrojový jazyk není překládán přímo do spustitelného binárního kódu dané platformy, ale do instrukcí virtuálního procesoru. Pro spuštění pak musíme mít k dispozici virtuální stroj, tedy jakýsi emulátor, který skutečně vykoná požadované operace. Díky této koncepci se programovací jazyk stává platformě nezávislý. Zdrojový kód je zkompilován do instrukcí virtuálního procesoru. Tento virtuální kód pak může být proveden na libovolné platformě, pro kterou máme k dispozici implementaci virtuálního procesoru. Představené schéma je implementováno například u programovacího jazyka Java či v platformě .NET.

Kromě vlastního překladu zdrojového jazyka do jazyka cílového se překladač také zabývá analýzou zdrojového kódu. Důležitou funkcí je, že programátora formou diagnostických zpráv informuje o chybách ve zdrojovém kódu.

2.3.1 Překlad zdrojových souborů

Překlad zdrojových kódů bychom mohli rozdělit na:

- **logické fáze** – fáze při transformaci zdrojového kódů na kód cílový. Obecně je problém překladu zdrojových kódů na cílové poměrně obtížný. Zjednodušit jej můžeme tím, že neprovádíme transformaci najednou, ale postupně zdrojové kódy transformujeme do cílové formy. Můžeme identifikovat tyto logické fáze.
    - Lexikální analýza – vstupem je zdrojový kód, tedy sekvence znaků. Při této analýze je vstup transformován na sekvenci lexikálních symbolů, jako jsou konstanty, operátory či identifikátory.
    - Syntaktická analýza – ze sekvence lexikálních symbolů jsou v této fázi vytvářeny hierarchické struktury. Ty mají v daném jazyce nějaký význam. Takovou strukturou může být výraz, příkaz a podobně.
    - Sémantická analýza – v této fázi se provádějí další kontroly a jsou zohledněny skutečnosti, které jsme nemohli zohlednit v předchozích fázích. Provádí se například typová kontrola.

  Výsledkem těchto analýz je nějaká vnitřní reprezentace. Vnitřní reprezentace (intermediární kód) je jakýmsi mezikrokem mezi zdrojovým a cílovým jazykem. Předcházející fáze jsou souhrnně označovány jako přední část překladače. Další logické fáze pak jsou označeny jako část zadní.
    - Optimalizace vnitřní reprezentace
    - Generování cílového kódu
    - Optimalizace cílového kódu

  Jednotlivé logické fáze se prakticky mohou realizovat najednou či nezávisle na sobě.
- **průchody** – vycházejí z praktické implementace překladače. V podstatě jde o rozdělení na průchody zdrojovým kódem či aktuální formou reprezentace. Lze implementovat jednoprůchodový překladač, který čte zdrojový kód a ihned generuje cílový (provádí všechny uvedené logické fáze najednou). Také můžeme implementovat víceprůchodový překladač. Některé jazyky (jako Java) nelze překládat jednoprůchodovým překladačem.

2.3.2 Typy překladače

Překladače bychom mohli rozdělit do dvou kategorií.

- **Kompilátory – ze zdrojových kódů je vygenerován cílový kód a tento kód je pak spouštěn.**
- **Interprety – interpret přečte zdrojové soubory, zkontroluje je a ihned vykoná implementované operace.**

Existuje řada rozdílů mezi kompilátory a interprety. Asi nejdůležitější rozdíl je, že interprety jsou v principu pomalejší. Na druhou stranu umožňují například měnit strukturu tříd přímo za běhu. U běžných programovacích jazyků jsou obvyklejší kompilační překladače. Jsou ovšem domény, kde se používají interprety (například skriptovací jazyky). Hlavní platformy jako Java či .NET kombinují oba přístupy. Kompilátor do instrukcí virtuálního procesoru a následná interpretace těchto kódů (s využitím JIT překladače, více níže…).

Kromě tohoto základního rozdělení existuje celá řada dalších typů překladačů. Například:

- **Just-In-Time překladač** – používají se například u virtuálního procesoru, virtuální procesor se v principu chová jako interpret. To by ovšem zpomalovalo běh aplikace. Proto je v době spuštění virtuálním procesorem zkompilován virtuální kód do binární formy, která je spustitelná na cílové platformě.
- **Zpětný překladač** – zajišťuje reversní překlad ze spustitelné binární formy zpět na (nejlépe) původní zdrojové kódy (v principu to vždy není možné).

2.4 Spojovací program (linker)

Spojovací program je nástroj, který z více modulů vygenerovaných překladačem sestaví jeden spustitelný program. Moduly obsahují zdrojový kód. Hlavní činnosti spojovacího programu je, že projde zdrojové kódy a vyhledá nedefinované symboly (například objekty obsažené v jiných modulech). Spojovací program potom vyhodnotí tyto závislosti, rekurzívně projde i tyto další moduly a definuje množinu modulů, které budou nutné pro sestavení aplikace. Spojovací program nemusí připojit všechny moduly. Například objekty základních knihoven spojovací programy obvykle nepřipojuje, jen vytvoří jakýsi zástupný objekt. Základní sada knihoven je obvykle připojena automaticky a nebo je součástí běhového prostředí.

Další důležitou činností spojovacího programu je seřazení těchto modulů v adresovém prostoru výsledné aplikace. Jednotlivé moduly obvykle používají jakousi relativní adresu vzhledem k nějaké fixně definované startovní adrese (například nule). Při spojovaní modulu jsou pak tyto adresy realokovány, tedy jsou jim přiřazený skutečné adresy z adresového prostoru cílové aplikace (obvykle je změněna právě bázová adresa, relativní adresa vzhledem k této bázové pak zůstane stejná).

2.5 Nástroje pro správu verzí

Nástroje pro správu verzí řeší celou řadu problémů. Primární funkcí je uchování různých verzí zdrojových kódů aplikace. Také řeší celou řadu problémů spojených s vývojem aplikace v týmu. Činnost aplikace by šla definovat takto:

- **Systém pro správu verzí (SCM – Software Configuration Management) uchovává změny projektu, jak probíhaly v čase při jeho vývoji.**

SMC realizuje celou řadu činností.

- **Archivace** – SCM ukládá všechny verze souborů vytvářené aplikace (slouží primárně k uchování zdrojových souborů, ale můžeme chtít uchovávat i další, například konfigurační soubory). Nástroj pro správu verzí potom zajistí, že můžeme libovolně procházet verze těchto zdrojových souborů a získal libovolnou starší verzi. Máme tedy například možnost návratu po nevhodných změnách či zjištění rozdílů mezi různými verzemi.
- **Vývoj různých verzí** – Nástroje pro správu verzí obvykle umožňují rozdělit vývoj aplikace v nějakém bodě do několika větví. Máme potom možnost identifikovat tyto větve a paralelně vyvíjet několik verzí aplikace. Jako příklad může sloužit aplikace pro různá cílová prostředí.
- **Bezpečnost** – Zdrojové kódy aplikace jsou většinou při vývoji komerčního produktu přísně střeženy. Majitel takových zdrojových souborů přirozeně nechce, aby někdo neoprávněně tyto kódy získal. Použití SMC umožňuje uložit soubory standardním způsobem na jednom místě a definuje pravidla a zabezpečení přístupu k nim.
- **Vývoj v týmu** – Použití SCM při vývoji aplikace může zjednodušit její vývoj a předejít řadě problémů, které při vývoji mohou nastat. Použití podobného nástroje se ovšem stává skoro nutností ve chvíli, kdy na vývoji aplikace pracuje více lidí najednou. V takovém případě musíme řešit celou řadu dalších problémů. Mezi takové problémy může patřit například sdílení zdrojových kódů nebo současná modifikace kódů více programátory. Tyto a další problémy můžeme vyřešit použitím vhodného SCM nástroje.

Použití nějakého SCM nástroje přináší řadu výhod. Jeho použití však něco stojí.

- **Velikost uložených dat** – Projekt zabírá mnohonásobně více místa než je nutné. Použili bychom-li jakýsi „naivní přístup“, musíme pro uchování všech verzí při vývoji uložit každou novou verzi do samostatného souboru. Vyvíjená by pak zabírala mnohonásobně více místa. SCM nástroje obvykle používají jiný přístup. Neukládají se celé nové soubory, ale pouze změny. Aplikací těchto změn na původní (respektive aktuální) verzi pak můžeme získat aktuální (respektive původní) verzi.
- **Výkon** – s předchozím bodem úzce souvisí i relativní výpočetní náročnost používání SCM. Získání aktuální verze projektu z SCM může být mnohem náročnější než posté „zkopírování“ z adresáře. Musíme například zkompletovat celou řadu souborů aplikováním uložených změn. Proto může být použití SCM poměrně výpočetně náročné. Zejména pro velké aplikace, na kterých pracuje současně mnoho lidí.
- **Konektivita** – Většina SCM nástrojů ke své činnosti potřebuje konektivitu k síti. Nejčastěji se používají nástroje, které jako úložiště dat používají nějaké centralizované síťové úložiště. Používáme-li takový nástroj, musíme mít k dispozici připojení k síti. Tento problém ovšem není zásadní. Většina SCM nástrojů umožňuje práci off-line, a synchronizaci dat až ve chvíli, kdy je obnoveno připojení.
- **Znalost aplikace** – Lidé používající SCM musí mít nějaké základní znalosti jak se systémem pracovat. To nemusí být jednoduché.

- **Cena za provoz SCM** – Některé komerční SCM nástroje mohou být poměrně drahé a jejich využití při vývoji zvyšuje celkovou cenu. Existuje ovšem i řada volně dostupných nástrojů.
- **Riziko poškození** – Vše uloženo na jednom místě. Díky tomu hrozí ztráta všech dat při poškození tohoto centrálního úložiště. Těmto problémům můžeme předcházet pravidelným zálohováním dat.

Na tomto místě bych chtěl zdůraznit, že žádna z uvedených nevýhod nevyvažuje ohromný přínos použití nějakého SCM nástroje. Využití SCM nástroje při vývoji skutečné aplikace je prakticky nutností.

SCM nástroje můžeme rozdělit dle různých kritérií. První takové rozdělení by mohlo být podle způsobu uložení dat. Můžeme použít:

- **centralizované** – existuje jedno centralizované úložiště dat. V tomto úložišti je aktuální verze vyvíjených souborů. Všechny změny jsou pak ukládány do tohoto úložiště;
- **distribuované** – aktuální verze zdrojových souborů je distribuovaná mezi všemi uživateli, kteří nástroj používají;

Někdy je obtížné striktně rozdělit, do které kategorie konkrétní nástroj spadá. Nástroje mohou například podporovat replikace dat (tím můžeme například urychlit práci).

Další možné dělení je podle způsobu přístupu. Použitý SCM nástroj může implementovat buď:

- **sériový model** – právě jeden uživatel může měnit soubory;
- **konkurenční model** – více uživatelům je povolen přístup k souborům.

Jedním z prvních SCM nástrojů byl RCS (Walter F. Tichy, Purdue University). Umožňoval správu jen pro jednotlivé soubory a byl určen pro použití jedním uživatel. Stále nejpoužívanějším nástrojem pro správu verzí je CVS (Concurrent Version System). Existuje ovšem celá řada dalších nástrojů jako: Subversion, MS Visual SourceSafe, IBM Rational ClearCase, Perforce, BitKeeper, nebo Arch.

Nástroje pro správu verzí jen ukládá a spravují verze kolekcí souborů. Nestarají se například o sestavení (build) projektu!

2.5.1 Concurrent Version System (CVS)

CVS je jeden ze systémů pro správu verzí. Jde o jeden z prvních úspěšných projektů tohoto typu a je pořád jedním z nejpoužívanějších nástrojů pro správu verzí. CVS je postaven na architektuře klient-server. Na CVS Serveru jsou uloženy zdrojové soubory a uživatel jej z tohoto úložiště mohou získat. CVS implementuje konkurenční model práce. Se zdrojovými soubory může pracovat více uživatelů najednou. CVS také neklade žádné nároky na jednotné vývojové prostředí.

V dnešní době existuje řada implementací CVS. Vlastní systém bychom mohli rozdělit na dvě části: server a klient. Server realizuje hlavní funkce spojené se správou verzí a spravuje centralizované úložiště dat. CVS definuje sadu příkazů, pomocí kterých můžeme provádět různé operace a manipulovat s daty. Klient potom může být jednoduchá textová aplikace (podobná například

nástroji *telnet*), která nám umožní připojit se k CVS serveru a zadávat tyto příkazy, a nebo nějaká složitější aplikace, která implementuje například nějaké grafické uživatelské rozhraní.

V dalším textu bude zavedeno několik pojmů, které popisují některé primární části či funkce CVS. Tyto pojmy byly poprvé představeny v souvislosti s CVS, ale staly se standardem prakticky ve všech nástrojích pro správu verzí.

První pojem, který zde bude zaveden je repository.

- **Repository v CVS je místo, kam se ukládají data, jejichž verze chceme uchovávat.**

Typicky bývá repository realizována jako nějaké síťové úložiště, ale v klidně to může být adresář na Vašem počítači. Do repository potom uživatelé ukládají jejich data. Ty jsou rozděleny na moduly.

- **Modul u CVS by se dal ztotožnit s pojmem projekt, je to sada souborů uložených v systému pro správu verzí.**

U dat, která potom do repository, je automaticky uchovávaná historie verzí. V repository nejsou uloženy všechny verze, ale pouze aktuální a změnové soubory. Změnové soubory (označované u CVS jako *diff*) uchovávají změny mezi verzemi, tak jak vystupovaly v čase a aplikací těchto změn jsme schopni získat libovolnou verzi souboru.

Typické schéma práce s CVS by se dalo shrnout do následujících bodů. Předpokládejme, že v CVS je uložena kompletní verze vyvíjeného projektu projekt (CVS modul).

1. **Checkout** – Další z pojmů, které CVS zavádí je checkout. Vývojář se rozhodne pracovat na nějaké části projektu. Operací: `cvs checkout` (jde přímo o příkaz CVS) získá svou osobní pracovní verzi a uloží ji například na disk svého počítače. Při této operaci CVS do souboru automaticky doplní některé údaje. Nejdůležitější z nich je aktuální číslo verze. Součástí získaných souborů jsou a další informace jako kdo či kdy soubor získal.
2. **Edit** – Vývojář pracuje na své lokální verzi (v CVS označována jako *sandbox*). Může přidávat soubory, měnit jejich obsah. K vlastní editaci může použít libovolný editor či vývojové prostředí. Sestavuje a spouští tuto svou lokální verzi.
3. **Diff –** Zjištění změn v pracovní verzi oproti verzi, která je uložená v repository. Výsledek je jaké změna a kde se udály.
4. **Update –** Obsah repository se mohl změnit v průběhu práce programátora. Důležitým údajem je zde číslo pracovní verze a číslo verze v CVS. Pokud jsou stejné, CVS předpokládá, že programátor modifikoval soubor uložený v CVS a operace `update` není nutná a může přejít k následujícímu bodu. Je tady ale možnost, že soubor paralelně modifikovali další aktéři. Potom vývojář získá touto operací aktuální verzi z repository a snaží se jí sladit s jeho pracovní verzí. Musí při tom vyřešit potencionální konflikty.
5. **Commit** - Programátor ukládá změny provedené ve své pracovní verzi zpět do repository (operace: `cvs commit`). Při této operaci je automaticky inkrementováno číslo verze (například z 1.5 na 1.6).

Tímto způsobem je modul uložený v CVS vyvíjen a upravován. Pro pochopení, jak CVS pracuje, jsou nejzajímavější právě potencionální konflikty při ukládání změn (*commit*) do CVS. Nejjednodušší případ, kdy budeme muset řešit potencionální konflikty, prezentuje následující příklad.

1. V repository je uložena verze 1.5 souboru `Pokus.java`.
2. Programátoři Alice a Bob získají soubor `Pokus.java` z CVS v aktuální verzi 1.5 (oprace *checkout*).
3. Oba dva provedou změny v souboru `Pokus.java`.
4. Alice uloží změny do CVS (operace *commit*). Číslo verze je automaticky inkrementováno. Aktuálně je v CVS uložena verze 1.6 souboru `Pokus.java`.
5. Bob chce uložit své změny do CVS. V systému je ale nyní verze 1.6. Bobova verze souboru je 1.5.
6. Bob neměnil aktuální verzi a musí provést *udpade* své verze.

Tento *update* ovšem nemusí uspět! Operace uspěje, když nezáleží na pořadí aplikace změn, které provedli programátoři Alice a Bob. Tento případ popisuje následující vztah.

```
apply(A, apply(B,1.5)) = apply(B, apply(A,1.5))
```

Operace *apply* – označuje aplikaci změn, parametry udávají, kdo změny prováděl a z jaké verze vyšel. Výsledkem *apply* je nová o číslo vyšší verze. Operace *apply* byly použity jen pro vysvětlení pojmů, není to žádná funkce CVS.

Pokud se CVS podařilo verze sloučit, vznikne nova verze souboru `Pokus.java`, která spojuje obě verze změn (jak Alice tak Boba) a má číslo verze 1.6. Bob v této chvíli může soubor uložit do repository operací *commit*.
V případě neúspěchu CVS ohlásí chybu, že verze nelze sloučit. Operace sloužení dvou souborů je v terminologii CVS pojmenovaná *merge*.

V případě, že CVS ohlásí chybu, jsou Bobovi označena místa konfliktů ve zdrojových souborech. Bob tyto konflikty musí vyřešit, pokud chce umístit svou verzi do CVS. CVS rozpoznává změny v souboru na základě změn v textu. Rozpozná tedy například přidaný řádek. Nesnaží se ovšem zjistit, co je obsahem uložených souborů. Nerozpozná například, že v souboru Pokus.java z předchozího souboru je program v Javě, a že ta či ona změna mohla způsobit, že program nepůjde přeložit. Například to, že úspěšně nahrajete soubor do CVS tak ještě nemusí znamenat, že nevznikla ve zdrojových kódech nějaká chyba.

Jak je patrno z předchozího popisu, soubor, který se mění častěji, bude mít vyšší číslo verze. Proto CVS zavádí pojem tag.

- **Tag je pojmenování aktuální verze projektu v určitém době jeho vývoje.**

Můžeme tak například označit první funkční verzi celé aplikace jménem. Čísla verzí konkrétních souborů se v této verzi mohou lišit. Po pojmenování takovéto verze vyvíjené aplikace (označení tagem), jsem potom schopni kdykoliv tuto verzi získat.

Z předchozího popisu je také zřejmé, že normálně nejsou vytvářeny dvě větve projektu, ale změny se „slučují (merging)“ a je udržována jedna hlavní vývojová linie. CVS umožňuje vytvořit další větvi a rozdělit vývoj v jistém

bodě do dvou linií. V terminologii CVS se tyto linie označují jako *branches* (větve).

CVS byl v podstatě prvním prakticky používaným nástrojem pro správu verzí. V průběhu času do něj byla přidávaná celá řada žádaných funkcí, bez změny původní koncepce. To je asi nejčastější příčinou problému, na které u CVS můžete narazit. Typické nevýhody CVS jsou:

- Operace checkout a commit jsou atomické jen na úrovni souborů ne transakcí. Pokud operace commit není korektně dokončena, jsou aktualizovány jen některé soubory. Další problém může vzniknout, pokud někdo čte a zároveň někdo jiný zapisuje, může získat verzi, kde jsou jen některé změny.
- Různé problémy spojené s přejmenováním adresářů, nemožností připojit poznámky k pojmenovaným verzím, pojmenováním souborů a adresářů.

2.5.2 Subversion

Subversion je vydáván v licenci Apace Software Fundation. Jde o přímého nástupce CVS (vytvářeli jej stejní lidé) a realizuje podobný „styl“. Jde o nástroj který, umožňuje konkurenční a centralizovanou správu verzí. Hlavními změnami oproti CVS jsou:

- Číslování verzí souboru – u aplikace Subversion jsou čísla verze celá čísla začínající jedničkou.
- Přejmenování adresářů a souborů – Subversion k uložení dat používá databázi. Oproti repository CVS kde jsou data umístěna v podstatě aplikace. Tento fakt usnadňuje různé přejmenovávání čí přesouvání adresářů či souborů.
- Atomické operace – operace (například získávající či ukládající data) buď uspějí celé a nebo se neprojeví vůbec.
- Metadata jsou také „verzována“. Nejen že máme možnost připojit prakticky k čemukoliv poznámky a další metadata, ale tato metadata jsou vztažená k určitým verzím a můžou se také v čase vyvíjet.
- Plná podpora binárních souboru. U CVS byla verze udržována jen u souborů, které obsahují „text“. Binární soubory nebylo možné v principu porovnávat a tedy ani uchovávat v různých verzích

2.5.3 Arch

Oproti centralizovaným SCM jako je CVS nebo Subversion umožňuje Arch distribuovaný přístup ke správě verzí. Realizuje podobný systém distribuce dat jako BitTorrent. Můžete pracovat se svou „lokální“ repository. V případě že to chcete, je pak tato repository synchronizována s ostatníma. Jde o „open source“ a volně dostupný nástroj. Tento nástroj je stále aktivně vyvíjen.

2.6 Správa projektů

Jednou z hlavních činností při správě projektu je sestavení aplikace. Sestavení aplikace je hlavně kompilace zdrojových kódů do formy, kterou lze provádět na počítači. Sestavení projektu může být poměrně složité a to zejména pokud velikost projektu roste. V takové chvíli je nutností rozdělení zdrojových kódů na části (GUI, interface databáze,…). Jednotlivé části jsou závislé na jiných

částech, ale obvykle ne na všech. Navíc může být složité definovat závislé části (použijeme-li například reflexi v Javě). Díky těmto a jiným problémům může být definováno nějaké netriviální pořadí pro kompilaci těchto částí. Druhým aspektem je, že sestavení celé aplikace (která může obsahovat mnoho megabajtů zdrojových kódů) může být časově náročné. Proto například při změně jednoho zdrojového souboru chceme zkompilovat pouze nezbytně nutné zdrojové soubory a ne všechny. To může být poměrně obtížné. Je to jeden z úkolů, o který se postará vhodně zvolené prostředí pro správu projektů.

Správa projektu ale nezahrnuje jen vlastní sestavení aplikace. Typické činnosti mohou být:

- inicializace prostředí (adresáře, parametry,…)
- překlad a sestavení programů
- systematické testování
- generování dokumentace
- odstranění pracovních souborů
- vytváření archivů a distribucí
- instalace

Některé studie uvádějí 10 % – 30 % času při vývoji komplexních aplikací zabere: práce na skriptech, které sestavují aplikaci; čekání na pomalé sestavování aplikace; hledání chyb, které způsobuje nekonzistentní sestavování aplikace. Z toho je zřejmé, že důsledné použití vhodného nástroje pro správu aplikace může nejen její vývoj značně usnadnit, ale i urychlit a tudíž i zlevnit.

2.6.1 Vlastnosti nástrojů pro správu aplikace

Nástroje pro správu aplikace realizují zejména:

1. sestavení aplikace nezávisle na prostředí (umístění knihoven či programů, různé verze a varianty nástrojů);
2. udržení konzistence při sestavování aplikace;
3. optimalizace budování projektu;

Typická činnost aplikace pro sestavování aplikace by se dala shrnout do této série kroků.

1. Definování cílů. Obvykle předán jako parametr při spuštění. Většina nástrojů umožňuje definovat nějaký „defaultní“.
2. Načtení skriptu pro sestavení aplikace – „build file“. Kromě vlastního načtení se souboru se provede i jeho kontrola.
3. Konfigurace – jeden skript může být použitý na více platformách. Dle konkrétních podmínek při spuštění je odpovídajícím způsobem nastaveno prostředí. Specifické nastavení může být definováno přímo použití programátorem a to jak přímo ve skriptu pro sestavení tak další parametry například z příkazové řádky.
4. Zohlednění závislostí. Zohlední možné chyby, jako jsou například cyklické závislosti.
5. Definice cílů pro sestavení. Sestaví posloupnost kroků, kterou je nutné provést k úspěšnému sestavení aplikace.
6. Vytvoření příkazů, které sestaví aplikaci. Při vytváření aplikace většinou používáme nějaké specifické konstrukce či příkazy použité technologie či nástroje. Tyto konstrukce jsou převedeny na příkazy na cílové platformě. V této fázi se také zohlední další informace

poskytované programátorem. Například specifické vlastnosti cílové platformy.

7. Posledním krokem je provedení vytvořených příkazů

V kterékoliv z těchto fází může nastat nějaká chyba. Různé nástroje na tyto chyby reagují různě. V principu může taková chyba znehodnotit další sestavování celé aplikace. Při práci s nějakým nástrojem pro správu aplikace se můžeme setkat s množinou specifických chyb. Mezi takové chyby patří například použití příliš dlouhého textu v příkazové řádce, různé problémy s absolutními cestami, formát jmen souborů. Takovéto a další chyby při použití nástroje pro správu aplikace je většinou velmi obtížné najít a odstranit. Většina používaných nástrojů nemá (a nebo má nedostatečnou) podporu ladění případně testování.

Nejčastějším problémem (nebo alespoň tím, na který si uživatele nejčastěji stěžují) je pomalé sestavení aplikace. Tento problém v principu nelze vyřešit. Někdy sestavení aplikace nelze urychlit a často jakékoliv zlepšení je pořád menší, než požadavky uživatelů. Můžeme používat různé nástroje pro profilaci. Ty nám umožní odhalit části, které způsobují problémy a na optimalizaci těchto částí se zaměřit. Typicky je překlad prodlužován špatně definovanými závislostmi a z toho plynoucím neoptimálním sestavováním aplikace. Ideální případ je, kdy se činnosti provádějí maximálně jednou a to jen v případě, že je to potřeba. Řešením pomalého sestavování aplikace může být také nějaká „*cache*“. Také se typicky používá nějaký výkonný počítač jako server, na kterém pak programátoři aplikaci sestavují (obdobou by mohla být paralelizace či distribuce sestavování aplikace).

2.6.2 Dávkové zpracování

Nejjednodušší možnost pro správu projektu jsou dávkové soubory. Jako příklad může být *.bat* nebo *.sh* soubor. Příklad takového souboru následuje.

preloz.sh

```
yacc -o synt.cpp -d synt.y
lex   -o lex.cpp lex.l
gcc  -o prekl synt.cpp lex.cpp main.cpp
```

Mezi největší výhody tohoto přístupu je, že napsat takový dávkový soubor je v principu velice jednoduché. Víme také přesně, jaké příkazy se při spuštění provedou. S použitím dávkových souborů souvisí celá řada problémů (z předchozího textu jich řada vyplývá). První nedostatek bude, že proces budování aplikace nebude optimální. Provedou se všechny příkazy, ne jen nutné. Dalším problémem bude detekce chyb. Skript je posloupnost příkazů, které se provede jeden za druhým. Provádění skriptu pokračuje i po chybě. Taková chyba může znehodnotit další sestavování aplikace. Ladění je u dávkových souboru realizováno hlavně textovými výpisy. Někdy je možné provést „dry run“ – příkazy jsou pouze vypsány, ne provedeny. Další z velkých nevýhod správy aplikace pomocí dávkových souborů bude přenositelnost. Obvykle je obtížné (nemožné) přenášet dávkové soubory mezi platformami.

Použití dávkových souborů může být výhodné u specifických jednoduchých aplikací. Pro větší projekty je skoro nutností použít nějaký nástroj pro správu aplikace.

2.6.3 Program `make`

První nástroj pro sestavování aplikací (byl vytvořen v roce 1977 Stuartem Feldmanem). I přes dobu, kterou je k dispozici je to pořád jeden z nejpoužívanějších nástrojů. Zejména při programování v C/C++. Skript pro sestavení se obvykle jmenuje „makefile“. K dnešnímu dni existuje celá řada implementací různých výrobců (make, gmake, nmake,…). Existují také různé produkty postavené na konceptu make (například cake, cook – použitý s CVS Aegis).

Make používá statické definice závislostí. Sestavení cílových objektů pak probíhá na základě předpokladů. Zdrojový soubor pro make obsahuje sadu definic cílů. Každý cíl pak obsahuje sadu předpokladů (které musejí být splněny před použitím) a sadu operací. Operace jsou na dalších řádcích a začínají tabelátorem (v následujícím příkladu je na začátku druhého řádku tabelátor!). Následující příklad ukazuje definici jednoho cíle.

```
prog.o: prog.cpp lex.l synt.cpp
   flex –o lex.cpp lex.l
   gcc –c prog.cpp lex.cpp synt.cpp
```

Výsledkem bude `prog.o` za předpokladu, že existují `prog.cpp`, `lex.l` a `synth.cpp`. *Make* se pokusí tyto předpoklady splnit.

Nástroj *make* poskytuje celou řadu dalších funkcí. Jejich kompletní výčet je mimo možnosti tohoto kurzu. Z nejzajímavějších můžeme uvést:

- implicitní pravidla – v nástroji *make* je vestavěna sada standardních pravidel. Ty například definují jak zkompilovat zdrojový soubor v C++ (ze souboru s příponou .cpp získáme soubor s příponou .o). Tuto množinu pravidel můžeme upravovat a rozšiřovat.
- Makrodefinice – můžeme definovat různá makra. Definici a použití makra demonstruje následující příklad.

```
SRCS = prog.cpp lex.cpp synt.cpp
prog: $(SRCS)
     gcc –o prog $(SRCS)
```

Komplexnější skript pro *make* ukazuje následující příklad. Šlo by o soubor s názvem `makefile`. S pomocí nástroje *make* bychom pak mohli kterýkoliv z definovaných cílů provést. Například příkazem: `make all` bychom provedli definovaný cíl `all`. Nástroj by za nás vyřešil další závislosti.

```
all: p4
par.o: par.c lex.c
p4: par.o
   $(CC) -o p4 par.o
clean:
   $(RM) par.c par.o lex.c
allclean: clean
   $(RM) p4
dist:
   tar -czf p4.tgz Makefile lex.l par.y
```

Nástroj *make* má řadu výhod. Asi hlavní je jeho rozšířenost. Tento nástroj najdete prakticky na všech platformách a máte možnost si vybrat z různých variant nástrojů. Další výhodou je, že velká část programátorů je s tímto nástrojem seznámena a je schopna jej používat.

Má ale také řadu nevýhod. Většina nevýhod vychází z toho, že nástroj byl vytvořen již před skoro třiceti lety. Neobsahuje tedy některé funkce, které bychom v dnešní době od podobného nástroje očekávali. Obvykle je při použití nástroje *make* pro správu nějakého velkého projektu skript rozdělen do několika menších souborů *makefile*. Tyto soubory jsou potom rekurzivně volány. Takový přístup může vest k nekompletní analýze závislostí či k cyklickým závislostem. Závislosti musíme také definovat staticky. Toto je také důvodem, že sestavení aplikace může trvat neúměrně dlouho. Dalším problémem může být přenositelnost. Použití jednoho nástroje (například kompilátoru) se může na jiné platformě lišit. Jiné také může být chování jiné varianty nástroje *make*. Problémem také je ladění. Můžeme použít parametr –n pro „dry run“, ale pořád může být obtížné určit, proč některé soubory byly či nebyly použity. Jako nevýhoda může být také vnímána skutečnost, že nutnost rekompilace je detekována na základě časových razítek souborů. Řešením většiny uvedených problémů může být „další vrstva“ – generátor makefile souborů (nejznámější je Automake).

2.6.4 Další nástroje související s make

Existuje celá řada nástrojů pro správu aplikace, které přímo navazují, či rozšiřují funkcionalitu *make*. Máme k dispozici například tyto nástroje.

- **Autotools** – Jde o nástroj, který tvoří jakousi další vrstvu k *make*. Nejčastěji je používaný pro „open source“ C/C++ projekty. Skládá se z třech částí.
  - *Autoconf* – Vytváří skripty pojmenované `configure`.Tyto skripty zjistí, jak daný systém splňuje požadavky aplikace na něj kladené.
  - Automake – Ze skriptu `Makefile.am` vytváří `Makefile.in`. Ten je potom použit nástrojem *Autoconf* k sestavení zdrojového souboru pro GNU *gmake*.
  - Libtool – Vytváří knihovny pro programy v C.
- **JAM (Just Another Make)** – Vytvořen firmou Perforace. Je volně k dispozici. Jde o jakéhosi nástupce *make*. Je primárně určen pro sestavování aplikací v C/C++. Jedna z hlavních výhod oproti *make* je, že je rychlejší.

2.6.5 ANT

Dalším nástrojem pro správu aplikace je ANT (Another neat tool). Jde o produkt vytvořený v licenci Apache Fundation. Tento produkt byl implementován v Javě a ke svému provozu Javu používá. Díky tomu je platformě nezávislý a můžeme jej používat kdekoliv, kde máme k dispozici Javu. Jde o nejčastěji používaný nástroj pro sestavování aplikací v Javě. V principu v tomto nástroji můžeme provést vše, co „umí“ Java. Nyní implementováno více než 100 funkcí (http://ant.apache.org). Tuto množinu lze rozšiřovat a implementovat nové uživatelské funkce. Podpora ANTu je

integrován do mnoha vývojových prostředí (Eclipse, NetBeans, JBuilder, jEdit,…).

Správa aplikace je řízena skriptem, který potom ANT vykoná. Skript pro ANT je XML soubor. Musí dodržovat všechny běžné konvence pro XML dokument. Tento dokument je obvykle pojmenovaný build.xml, ale v principu můžeme použít libovolné jméno.

Hlavní struktura skriptu pro ANT by se dala shrnou následujícími pravidly.

- Hlavní element je element <project>
- V těle tohoto elementu jsou umístěny elementy <target>. Ty definují jednotlivé cíle.
- Cíle jsou složeny z <task> elementů. Ty definují jednotlivé operace, které se mají provést.

Ve skriptech máme možnost použit proměnné. Ty lze v ANTu definovat pomocí elementu <property>. Použití proměnné test ukazuje tento příklad: `${test}`. Tento text bude nahrazen hodnotou uloženou v *property* test. Pokud by proměnná test nebyla definovaná, zůstane v místě použití řetězec `${test}`.

Následující příklad ukazuje jednoduchý skript pro ANT. Můžete si na něm všimnout jeho základní struktury a také demonstruje použití proměnných.

```
<?xml version="1.0"?>
<project name="Test" default="compile" basedir=".">
  <property name="dir.src" value="src"/>
  <property name="dir.build" value="build"/>
  <target name="prepare">
    <mkdir dir="${dir.build}">
  </target>
  <target name="clean" description="Remove all">
    <delete dir="${dir.build}">
 </target>
  <target name="compile" depends="prepare">
   <javac srcdir="${dir.src}" destdir="${dir.build}">
  </target>
</project>
```

Obvykle je skript pro ANT uložen v souboru s názvem build.xml, ale jméno může být libovolné. Potom pomocí parametru `buildfile` můžeme specifikovat, který soubor chceme spustit. Dalším parametrem potom ne cíl, který se má provést. Pokud není specifikovaný žádný cíl, je proveden cíl specifikovaný v atributu `default` v elementu `project`. Následující příklad ukazuje, jak lze spustit cíl `clean` ve skriptu `MyBuildFile.xml`.

```
ant –buildfile MyBuildFile.xml clean
```

Kromě zmíněných parametrů existuje celá řada dalších. Můžeme například zjistit informace o cílech v projektu.

Proměnné lze také umístit do externího souboru. Definice v externím souboru by vypadala takto: `dir.buid=build`. Ve skriptu pro ANT je potom nutné specifikovat externí soubor, který obsahuje definice proměnných.

```
<property file="local.properties">
```

Následující příklad ukazuje, jak lze vypsat test v ANTu. Také demonstruje, jak můžeme využít možností jazyka XML (použití CDATA).

```
<target name="help">
    <echo message="Toto je nejaky text">
    <echo>
      Tento text bude vypsan taky!
    </echo>
    <echo><![CDATA[
    tento text
    bude na dva radky]]>
    </echo>
</target>
```

Pro každý cíl můžete definovat různé závislosti a předpoklady, které musí být splněny, pokud chceme cíl provést. Každá cíl může záviset na několika dalších cílech (atribut depends). ANT se při spuštění takového cíle snaží tyto závislosti vyřešit. Také můžeme chtít, aby náš cíl byl spuštěn jen v případě, že je nastavena respektive nedefinovaná nějaké proměnná. To demonstruje následující příklad.

```
<!-- Abort if TOMCAT_HOME is not set -->
<target name="checkTomcat" depends="init, compile"
        unless="env.TOMCAT_HOME">
  <fail message="TOMCAT_HOME must be set!">
</target>
```

Při práci s nástrojem pro správu aplikace budeme jistě muset definovat různé množiny souborů. Typicky chceme specifikovat množinu souborů pro kompilaci nebo množinu souborů, které chceme zkopírovat. Popsat tyto množiny souborů může být poměrně obtížné. ANT umožňuje využít následujících konstrukcí.

- Můžeme například specifikovat všechny soubory s příponou java z určitého adresáře.

```
include = "scr/lib/cviceni1/*.java"
```

- Můžeme specifikovat všechny soubory s příponou java z libovolného podadresáře adresáře src.

```
include = "src/**/*.java"
```

- Můžeme specifikovat libovolný soubor, kde v cestě k němu je podadresář a.

```
include = "**/a/**"
```

- Můžeme také specifikovat soubory, kde v názvu je jediný libovolný znak.

```
include = "Test?.java"
```

Předcházející příklady ukazují možnosti ANTu při specifikaci množin souborů. Tyto konstrukce můžeme potom využít. Pro specifikaci množin souborů ANT používá (zejména) element `fileset`. V tomto elementu můžeme pomocí

vnořených elementů a nebo atributů s názvem *include* vložit respektive pomocí atributu *exclude* vyjmout nějaké soubory a adresáře. Pokud specifikujeme nějaký adresář, je standardně do množiny `fileset` vložen i se svým obsahem.

```
<target name="clean" description="Clean project">
  <delete file="uloha3.jar"/>
  <delete>
    <fileset dir=".">
      <include name="**/*.class">
    </fileset>
  </delete>
</target>
```

Také lze v ANTu některé množiny souborů pojmenovat a potom takové pojmenované množiny používat ve více elementech. Ke specifikaci cesty můžeme použít element pojmenovaný `path`. V atributu *id* potom specifikujeme jméno. Toto jméno potom použijeme pro referenci na definovanou množinu souborů. Definici elementu `path` i použití (atribut *refid*) demonstruje následující příklad.

```
<path id="project.classpath">
  <pathelement location="${dir.src}"/>
    <fileset dir="${tomcat}/common/lib">
      <include name="*.jar"/>
    </fileset>
    <fileset location="${dir.src}">
      <include name="*.jar">
    </fileset>
  </path>

<javac destdir="${dir.build}">
  <src path="${dir.src}">
  <classpath refid="project.classpath">
</javac>
```

Důležitou funkcí nástroje pro správu aplikace je zajištění nezávislosti na cílové platformě. ANT nám dává celou řadu možností. Typicky problém může být jiný formát cest na různých platformách. Poslední příklad ukazuje, jak můžeme využít obecně definovanou strukturu `path` z předcházejícího příkladu a uložit ji v proměnné *windowsPath* ve formátu nativním v rodině operačních systémů Windows.

```
<pathconvertor targetos="windows" property="windowsPath"
               refid="projec.classpath">
```

V předcházejícím textu jsem ukázal celou řadu možností, které nám nabízí ANT. Výčet jeho funkcí tímto ovšem není kompletní. Pro další informace doporučuji: http://ant.apache.org/.

ANT kromě jiného umožňuje definovat nějaké podmínky, generovat dokumentaci, spouštět testy, realizuje spolupráci s CVS, FTP, Telnetem. Umí vytvářet archivy, pracovat se soubory (měnit práva, kopírování,…), validovat XML dokumenty a mnoho dalšího.

ANT má řadu výhod, ale i nějaké nevýhody. Mezi jeho nevýhody patří:

- Různá omezení pro XML dokumenty – skript pro ANT bude obsahovat stejná omezení a nedostatky jako jiné XML dokumenty. První takovou nevýhodou bude relativní rozsáhlost souboru. Pak také musíme respektovat omezení jako použití < místo <.
- Složité řetězce závislostí – Skript pro sestavení může být poměrně složitý (zejména pro velké aplikace). Máme různé možnosti, jak rozdělit aplikaci do bloků. Můžeme například rozdělit jediný skript do více zdrojových souborů a volat je pomocí úkolu *antcall*. Tento přístup může značně zpomalit sestavení aplikace. Od verze 1.6 také máme možnost používat úkol *import*.
- Omezené použití `<property>` – Nemají vlastnosti proměnných z programovacích jazyků. Jakmile je jednou nastavena hodnota nemůže už být změněna. Nelze použít *property*, která by obsahovala název další *property* a tak se dostat k její hodnotě. XML editory často neumí pracovat s proměnnými ANTu.
- Paralelní zpracování a „dry run“
- Platformě závislé problémy – Lze jim předcházet, například použitím úkolu *PathConvertor*.

2.6.6 SCons

Hlavní idea za tímto nástrojem je, proč pro nástroj pro správu aplikace nevyužít všech možností komplexního funkcionálního jazyka. V tomto případě je to jazyk Python. Skripty jsou realizovány v jazyce Python. Můžeme použít všech možnosti tohoto jazyka. Mezi hlavní vlastnosti nástroje SCons patří:

- Přenositelné soubory pro sestavení.
- Automatická detekce závislostí.
- K detekci, zda došlo ke změně, používá MD5 signaturu.
- Podpora paralelního sestavování.
- Rozšiřitelnost a modularita.
- Integrace nástrojů jako například nástroje pro správu verzí.

2.7 Ladění programů

Důležitou položkou při vývoji aplikace je ladění programů.

> ➢ **Hledání chyb je proces ověřování mnoha věcí, v jejichž platnost věříme, až po nalezení toho, co není pravda.**

Chceme například určit zda:

- V určitém bodě programu má proměnná x hodnotu v.
- V konkrétním příkazu if-then-else provedeme právě větev else.
- Funkce f se volá se správnými parametry.

Při tomto procesu chceme najít a ověřit tyto vlastnosti.

Pro ladění aplikace můžeme využít celou řadu strategií.

- Binární vyhledávání – chceme najít bod v programu, kde je hledaná chyba. Postupujeme tak, že omezujeme úsek programu, ve kterém se hledaná chyba může vyskytovat. Příklad: Hledáme místo, kde se nastavila nesprávná hodnota nějaké proměnné.
- Ladící výpisy a logovací nástroje – další možností jak hledat chybu ve zdrojových kódech je doplnit do nich nějakou formu výstupu. Tento výstup programátora pak informuje o vykonávání programu a pomocí něj můžeme zjistit informace o běhu programu. Nejjednodušší je umístit přímo do souboru ladící výpisy (například pomocí printf, count, System.out.write). V principu je tento přístup nevhodný. Důvodů je několik. Asi hlavní důvod je, že nakonec musejí být tyto výpisy odstraněny odstranění z odladěné verze (můžeme využít konstrukcí jako komentáře či podmíněného překladu). Lepší možností je využití nějakého logovací nástroje (např. log4j). Výstup je potom ponechán v hotovém programu. Většinou u takových nástrojů můžeme jednoduše definovat, zda se má logovat a nebo ne.
- Sledování stopy programu (trace) – snažíme se získat posloupnosti zpracovaných řádků, a nebo výpis volání podprogramů. Tento výpis je obvykle produktem nějakého nástroje. Neměníme zdrojové kódy, ale při spuštění je sledováno, jaké instrukce běžící aplikace provádí.
- Analýza obsahu paměti po chybě – Další možnou strategii jak najít chybu je analyzovat data v paměti. Snažíme se zjistit, co v ní bylo uloženo v době, kdy došlo k chybě. Důležité je v této chvíli propojení se zdrojovým programem. Díky tomuto propojení jsme schopni interpretovat údaje z paměti.

Existují různé ladící nástroje. V principu je můžeme rozdělit na nástroje, které pracují na úrovni zdrojových kódu a na nástroje, které pracují na instrukční úrovni. Aplikace tohoto typu jsou také obvyklou součástí integrovaných vývojových nástrojů.

Typický scénář použití by potom vypadal následovně.

1. Definice bodů zastavení (breakpoint) – v programu jsou označeny body, kde má být program zastaven.
2. Spuštění programu – program je spuštěn. Jeho běh je normální až do definovaného „breakpointu“.
3. Kontrola stavu v bodech zastavení – po dosažení definovaného bodu zastavení můžeme ověřit, že aplikace probíhala, jak měla. Zjistíme například sekvenci volání podprogramu. Také můžeme zjistit, jaké hodnoty jsou uloženy v paměti.
4. Krokování od bodu zastavení – od bodu zastavení můžeme aplikaci *krokovat*. Tedy procházet jednotlivé příkazy či operace jednu po druhé v pořadí, jak jsou definovány ve zdrojových kódech. Nástroje pro ladění typicky umožňují, aby jeden takový krok byl jeden příkaz. Pokud je příkaz složitější (například obsahuje volání funkce) umožňuje zjemnit krokování a „vnořit“se do takového příkazu a provádět jej krok po kroku.

2.8 Testování aplikace

Další důležitou fází při vývoji aplikace je testování. Testování by mělo odhalit jakýkoliv problém snižující kvalitu programu. S testováním aplikace úzce souvisí následující pojmy:

- Verifikace – ověřování interní konzistence produktu (zda produkt odpovídá návrhu, návrh analýze, analýza požadavkům). Ověřujeme, zda tvoříme správně produkt. Existuje celá řada metod jak provádět verifikaci. Můžeme dokonce použít nějaké formální metody. Tento způsob je ovšem velice obtížný.
- Validace – ověřování (externím nezávislým zdrojem), zda celé řešení splňuje očekávání uživatelů nebo klientů. Ověřujeme, zda tvoříme správnou aplikaci.

Hlavním cílem testování je snížení rizika výskytu chyby. Existuje celá řada scénářů, jak při vývoji aplikace využít testování. V principu zároveň s vývojem aplikace vytváříme testy. Tyto testy ověřují její funkčnost, a pokud neodhalí chybu, můžeme předpokládat, že v aplikaci nejsou (závažné) chyby. Při testování je nutný pesimismus! Výskyt chyby je třeba očekávat. V principu asi platí, že žádná (rozsáhlá) aplikace není bez chyb. Procesem testování také nemůžeme ověřit vše. Proto se snažíme ověřit zejména podstatné části systému, jejichž nefunkčnost by znemožnila používání aplikace. S trochou nadsázky bylo v knize B. Keringham a R. Pike – The Practice of Programming řečeno, že nejdůležitější pravidlo pro testování je dělat ho.

2.8.1 Typy testů

Pokud při testování odhalíme chybu, snažíme se jí odstranit. Po odstranění chyb bychom měli opakovat testování (re-testing) a tak zkontrolovat, zda jsme chybu odstranili. Další pojem, který souvisí s testováním, jsou regresní testy. Je to kontrola, zda jsme úpravou nevnesli nové chyby.

Testy můžeme rozdělit dle různých kritérii. Můžeme je dělit na:

- testy, které jsou součástí výsledného produktu;
- testy, které jsou odděleny od výsledného produktu (tyto budou náplní této sekce).

Další možné dělení je podle množství informací, které máme k dispozici při tvorbě testů. Z tohoto pohledu můžeme testy rozdělit na:

- *black box testing* – osoba, která vytváří testy, nemusí mít informace o tom, jak funguje aplikace na úrovni, na které je test vytvářen.
- *white box testing* – pro vytváření testů je nutné znát informace o fungování testované části.

Další způsob rozdělení testů je podle způsobu jejich vytváření. Můžeme definovat:

- „Ruční“ testování – testy jsou vytvářeny ručně programátory. Tento přístup je nejběžnější a často to je jediný způsob, jak realizovat testování aplikace. Hlavní nevýhodou tohoto přístupu je, že není opakovatelné a často je časově velmi náročné.
- Automatické testování – další možností je automatické testování aplikace. Automaticky testovat vyvíjený software je složité. Automatické metody často používají formální (matematické) metody.

Poslední (a asi i nejzajímavější rozdělení testů) by bylo dle úrovně, na které testy provádíme. Z tohoto pohledu by rozdělení testu vypadalo takto:

- **Jednotkové testy** (Unit tests) – Jsou vytvářeny pro malé části produktu: „jednotky“. Co to je jednotka závisí na konkrétním produktu, programovacím jazyce,… (třída, metoda třídy, funkcionalita tvořené aplikace,…). Při testování testujeme jen konkrétní jednotku. Neočekává se, že k testování bude použit zbytek aplikace. Pokud k testování potřebujeme například databázi, měli bychom její činnost nasimulovat a při testování se bez skutečné databáze obejít. Obvykle jsou jednotkové testy pojmenované jako TestXXX (kde XXX je jméno testované jednotky). Jednotkové testy většinou tvoří přímo vývojář, současně s vlastní aplikaci.
- **Integrační testy** – Testují větší moduly vytvářené aplikace. Testují integraci několika jednotek (tyto jednotky jsme otestovali pomocí jednotkových testů).
- **Systémové testy** – obvykle vytvářeny „testry“ (skupina programátorů, která je oddělena od skupiny vývojářů tvořících aplikaci). Testují systém jako by byl nainstalován zadavateli. Očekává se, že jsou přítomny všechny prostředky nutné pro běh aplikace (databáze, síťové zdroje,…). Systémové testy obsahují ověření funkcionality, uživatelského rozhraní, bezpečnosti,…
- **Zákaznický test** (Customer tests, Acceptance tests) – testují hotový systém. Používají „black-box testing“ celého produktu. Výsledkem testování je, zda je produkt možno předat zákazníkovi.

S procesem testování také souvisí běžně používané označení.

- Alfa – takto je označena verze aplikace před zveřejněním produktu. Provádí se výstupní kontrola. Tyto testy provádí vývojáři aplikace.
- Beta – verze aplikace po interní kontrole. Bývá poskytnuta skupině externích uživatelů, kteří ověřují její kvalitu. Tvůrce aplikace tak získá zpětnou vazbu.
- Gama – verze aplikace, která zcela neprošla interní kontrolou kvality

2.8.2 Prostředí pro testování

V dnešní době existuje celá řada prostředí pro testování. Volba, jaké prostředí pro testování použít závisí na mnoha faktorech. Například na tom, jaký používáme programovací jazyk. Prostředí pro testování by minimálně mělo:

- umět spustit série testů;
- rozhodnout, zda testy proběhly úspěšně. Jednotlivé testy by měly být prováděny nezávisle na ostatních;
- v případě, že test skončil chybou určit proč;
- sumarizovat získané výsledky.

V ideálním případě by prostředí pro testování mělo být nezávislé na vlastních testech.

Vlastní proces testování bychom mohli rozdělit do třech fází.

1. *Příprava před testováním* – Testování předchází jeho příprava. Musíme naplánovat, které testy chceme spustit. Testy můžeme rozdělit do logických skupin. Třeba podle toho, jaké části aplikace testují a nebo tak, aby mohly být testy prováděny paralelně. Také musíme připravit data. Můžeme například vygenerovat data náhodně. Pokud k testování

potřebujeme nasimulovat činnost nějakých externích zdrojů (například databáze), v této fázi je připravíme. Také je připraveno prostředí pro testování. Výsledkem testů mohou být chyby. Důležitou složkou přípravy před testováním je definování zodpovědnosti za určité části aplikace.

2. *Spuštění testů* – Provedení jednoho, nějaké skupiny, všech testů. Jsou-li testy prováděny paralelně a nebo jsou některé činnosti prováděny na pozadí, může být nutné tyto činnosti synchronizovat. Pokud používáme více počítačů a nebo více platforem může nám prostředí pro testování pomoct při řízení těchto strojů a nebo stírá rozdíly mezi platformami. Další funkcí prostředí pro testování by pak bylo uchování výstupu a vstupu testů.
3. *Po skončení testů* – Nejdůležitější je vygenerování zprávy, která shrnuje výsledky testu. Tato zpráva by měla poskytovat přehled o tom, které soubory a jak byly testovány. Měli bychom být schopni rozlišit mezi neúspěšně provedeným testem a chybě při testování. Také bychom měli být schopni uměle vytvořit nalezenou chybu.

2.8.3 JUnit

Jedním z nástrojů pro testování aplikace je JUnit. Tento nástroj je primárně určen pro tvorbu testů v jazyce Java, ale existují varianty i pro jiné jazyky (NUnit, CPPUnit,…). JUnit bychom mohli rozdělit na dvě části. JUnit definuje, jak mají vypadat dobře napsané testy, a poskytuje sadu tříd, které při tvorbě testu můžete využít. Druhou části je prostředí pro testování. JUnit implementuje grafické prostředí, pomocí nějž jsme schopni provádět testy.

Využití tohoto prostředí ale není jedinou možností, jak spouštět JUnit testy. Aktuálně existuje celá řada dalších nástrojů, které jsou schopny s těmito testy pracovat. Podpora spouštění testů v JUnit je součástí standardní sady cílů v ANTu. Také většina běžně používaných IDE integruje prostředí pro spouštění JUnit testů.

JUnit je také průběžně rozšiřován o nové funkce a vlastnosti. Kromě běžných aplikací můžete pomocí něj testovat webové aplikace, servlety, databázových aplikací a řadu dalších.

JUnit využívá reflexe pro vyhledání testů uvnitř tříd. Jednotkový test v prostředí Junit je třída, která musí rozšiřovat třídu **`junit.framework.TestCase`**. Ve vytvořeném testu (třídě) mají dvě metody speciální význam. Jde o metody:

- `void setUp () {…}` – tato metoda slouží k inicializaci. Je zaručeno, že bude spuštěna před každým testováním.
- `void tearDown() {…}` – tato metoda slouží k finalizaci. Bude spuštěna po skončení testů.

Podobnou funkci by v běžné třídě zajišťoval konstruktor respektive destruktor. Použití konstruktoru by ale v případě třídy pro testování bylo nevhodné. Testy nejsou spustitelné třídy a vlastní testování je realizovaná nějakým prostředím pro spouštění testů (může být přímo prostředí implementované JUnit). Takové prostředí typicky umožňuje opakované provádění testů, umožňuje zvolit konkrétní testy a podobně. Nikde není řečeno, že pro opakované spuštění testu bude vytvořená nová instance testu. Máte ovšem zaručeno, že bude spuštěna metoda setUp().

Každá testovací třída potom obsahuje několik vlastních testů. Tyto jednotlivé testy jsou metody začínající *test* – void testX() {…}, kde X je jméno testované vlastnosti. Uvnitř těchto metod potom ověřujeme platnost různých podmínek. V principu by se dal každý test rozdělit na tři části.

1. *Příprava dat* – připravíme si vstupní data a hodnoty, které očekáváme, že budou výsledkem testování.
2. *Příprava podmínek pro test* – připravíme si část aplikace (jednotku), kterou chceme testovat.
3. *Ověření, že testovaná část funguje* – v JUnit máme k dispozici řadu metod *assert.* Máme k dispozici assertTrue, assertFalse, assertEquals, assertNotEquals, assertNull a další. Například metoda assertTrue(podmínka) testuje, zda je uvedená podmínka pravdivá. Metoda assertEquals(…) by ověřila, zda jsou dva argumenty rovny. Pomocí těchto metod ověříme, že se aplikace chová, jak očekáváme. V případě, že některá z podmínek popsaná voláním variant metody *assert* není splněna, test skončí neúspěšně.

Následující třída implementuje jednoduchý test třídy Zlomek.

```
package cviceni3;
import junit.framework.TestCase;

public class TestZlomek extends TestCase {
   protected Zlomek z1 = new Zlomek(1, 3);
   protected Zlomek z2 = new Zlomek(2, 6);
   protected Zlomek z3 = new Zlomek(2, 3);

   protected void setUp()  { }
   protected void tearDown()  { }
   public void testEquals()
   {
      assertEquals(z1, z1);
      assertEquals(z1, z2);
   }
   public void testAdd()
   {
      Zlomek result = Zlomek.plus(z1, z2);
      assertEquals(result, z3);
   }
}
```

Takový to test potom spustíme pomocí prostředí pro testování. Můžeme získat tři typy výsledku.

- Test skončil úspěšně – celý test proběhl a všechny podmínky popsané metodami *assert* byly splněny.
- Test skončil chybou při vykonávání – obvykle jde o výsledek označovaný jako *error*. Některý z testů skončil chybou při vykonávání.

Například se v rámci testu snažíme dělit nulou a nebo došlo k vyvolání nějaké výjimky.

- Test skončil chybou – některá z podmínek popsaných v metodách assert nebyla splněna. Takový test je obvykle prostředím pro spuštění testů označen jako *fail*. V tomto případě jako výstup získáme, jaká podmínka skončila chybou, jakou hodnotu jsme získali a jakou hodnotu jsme očekávali. Tvůrce testu také může připojit zprávu. Pokud tak učinil, bude tato informace také součástí výstupu.

Obecně chceme, aby testy ověřovaly funkcionalitu nějaké části vyvíjeného produktu. V ideálním případě by se napsané testy měly chovat tak, že pokud je v aplikaci chyba, některý test by na ní měl upozornit. Při praktické realizaci často nemůžeme ověřit vše. Informace co testovat získáme spíše z popisu funkcionality než ze zdrojových kódů. Z popisu například získáme, co by daná třída či metoda měla dělat. Test by měl odhalit, pokud testovaná část neimplementuje „popsané“ funkce.

Problematika jak napsat dobré testy je poměrně složitá. Vystačila by na samostatný kurz. Základní ideu ukazuje následujíc příklad.

Testujeme metodu `isEmpty()` třídy `java.util.Vector`. Tato třída reálně existuje a je součástí standardních balíků Javy. Z popisu zjistíme, že metoda vrátí *true* v případě, že vektor je prázdný a *false* v případě že není.
Můžeme test napsat takto:

```
public void testIsEmpty () {
  Vector vector=new Vector();
  assertTrue(vector.isEmpty());
}
```

Jde ovšem o špatné řešení! Takováto implementace metody *isEmpty* by testem prošla a přitom neodpovídá popisu!

```
public boolean isEmpty() { return true;}
```

Lepší řešení by bylo otestovat obě varianty výstupů. Konkrétní řešení závisí na testovaném problému. Jde o tvůrčí činnost a programátor testů je ten, kdo musí jejich logiku vymyslet. Mohli bychom použít několik obecné rad. V principu se snažíme, aby test skončil chybou v případě, že metoda nedělá co má. Pokud testovaná metoda vrací víc „typů“ výsledku (například metoda compareTo() třídy Integer vrací-1, 0, 1) otestujte všechny varianty. Obvykle nejsme schopni otestovat všechny varianty vstupů a výstupu. Proto ověříme nejčastěji používané. Otestujeme například „krajní meze“ vstupů a výstupů. Pokud na vstupu může být nějaký interval hodnot, otestujeme jednu hodnotu z intervalu a jednu vně intervalu.
Testy lze „sdružovat“ pomocí třídy *TestSuite*. To demonstruje následující příklad.

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(cviceni2.TestPredmet.class);
    suite.addTestSuite(cviceni2.TestPredmetIO.class);
    return suite;
}
```

Výsledek testu lze získat pomocí třídy *TestResult*. Pomocí této třídy jsem schopni s výsledky testů přímo v Javě pracovat.

```
TestResult result = new TestResult();
suite.run(result);
result.wasSuccessful();
```

Další informace naleznete na: http://junit.sourceforge.net/javadoc/. Aplikace, související s testy v Junit (například různé prostředí pro spouštění či prezentaci výsledků) jsou ke stažení na: http://xprogramming.com/software.htm.

V souvislosti s novou Javou verze 1.5, která značně rozšířila možnosti tohoto jazyka, vznikla i nová verze Junit. JUnit verze 4.x používá jiné prostředky pro tvorbu testů. Při psaní testů se využívají *anotace Javy*. Pomocí anotací jsou definovány jak samotné testy, tak metody *setUp* a *tearDown*. Obecné principy jak vytvořit a spouštět testy zůstaly stejné. Použití nové verze JUnit demonstrují následující příklady.

- Testy jsou nyní definovány takto:

```
@Test public void testVyhledej() { … }
```

Důležitá je anotace na začátku!

- Inicializace

```
@Before protected void setUp() throws Exception {…}
@After protected void tearDown() throws Exception {…}
```

Následující příklad ukazuje kompletní test v novém JUnit.

```
import static org.junit.Assert.*;
public class PoleTest {
    @Before protected void setUp() throws Exception {
        pole = new Pole();
    }
    @After protected void tearDown() throws Exception {
        pole = null;
    }
    @Test public void vratPocet() {
        …
        assertEquals("return value", expectedReturn,
actualReturn);
    }
}
```

2.8.4 Další nástroje pro testování aplikace

Kromě nástrojů pro testování funkčnosti vytvářené aplikace existuje celá řada dalších aplikací, které slouží k testování aplikace a pomocí nichž můžeme testovat další vlastnosti. Následující přehled ukazuje další nástroje, které bychom mohli použít k testování aplikace.

- **Analyzátor paměti** – Uchovává informace o tom, jak a kolik paměti bylo při běhu aplikace použito – Purify, Electric Fence, …

- **Coverage tools** – Nástroje zjišťující jak velká část aplikace je použita při testování. Výsledek může být zdrojový kód, který nebyl při testování použit. Další možností (branche coverage) je, které části podmínek nebyly provedeny respektive, které větve programu nebyly použity.
- **Testování výkonnosti** – Nástroje označované jako *profilery*. Tyto nástroje sledují počet volání určitých funkcí a zaznamenávají čas, strávený výpočtem. Pomocí tohoto nástroje jsme schopni identifikovat části programu, kde se výpočtem stráví nejvíce času. Tento výstup může sloužit jako podklad pro optimalizaci a nebo změnou algoritmu. Bývají součástí vývojových prostředí (například v NetBeans). Profilaci může podporovat překladač a nebo lze použít samostatný program jako: *gprof*.
- **Analyzátory kódu** (static code analyzers) – testují různé statické vlastnosti zdrojových kódů. Můžeme například testovat:
    - Jak přesně splňují zdrojové kódy normy pro daný programovací jazyk (ANSI C).
    - Další testovanou vlastností může být bezpečnost. Hledáme potencionálně nebezpečné příkazy.
    - Korektnost – některé jazyky umožňují matematicky dokazovat vlastnosti (funkcionální jazyky) nebo hledají vzory častých chyb (FindBug).
    - Velikost případně komplexnost zdrojových kódů. Také můžeme analyzovat dokumentaci, která je součástí zdrojových kódů. Můžeme také testovat „stabilitu“ API. Stabilitou tady myslíme, jak často se v čase mění. Stabilní API bude takové, které se už nevyvíjí.

2.8.5 Vývoj řízený testy

Jedním z možných modelů jak vyvíjet aplikaci je vývoj řízený testy (Test Driven Development - TDD). V tomto modelu vyvíjíme aplikaci v principu takto:

1. napíšeme testy;
2. napíšeme program;
3. spustíme automatizované testování;
4. provedeme *refaktorizaci;*
5. opakujeme až do odstranění všech chyb.

Hlavním cílem je dosáhnout toho, aby všechny testy prošly. V jednotlivých cyklech navíc můžeme upravovat a rozšiřovat funkcionalitu, kterou vyvíjená aplikace obsahuje. Pokud testy popisují všechny funkce, které klient po aplikaci požaduje a aplikace těmito testy úspěšně projde, může být předána klientovi.

Důležitou roli v TDD hraje *refaktorizace*. Refaktorizace je transformace zdrojového kódu, která vede ke zlepšení jeho čitelnosti nebo struktury, avšak bez změny významu nebo chování. Typické činnosti při refaktorizaci jsou přejmenování proměnné, vytvoření podprogramu ze zadaného úseku programu, nahrazení posloupnosti příkazu if polymorfismem a nebo upravení hierarchie dědičností. Nástroje pro refaktorizaci bývají součástí IDE.

Refaktorizace je důležitou součástí nejen TDD, ale je také základním prvkem jiného modelu pro vývoj softwaru Extrémního programování.

2.9 Nástroje pro sledování chyb

Každá větší aplikace při svém vývoji obsahuje chyby. Tyto chyby odhalíme například při testování. Nalezených chyb může být celá řada a jejich odstranění nemusí být triviální. Existuje celá řada nástrojů, které umožňují sledovat výskyt chyb. Pokud najdeme chybu v aplikaci je potřeba zaznamenat celou řadu informací. Je potřeba například zaznamenat jaké kroky k výskytu chyby vedly. Také jaké bylo očekávané chování a jaké bylo skutečné chování. Tedy informace o tom, jak chybu v případě potřeby reprodukovat. Také je obvykle potřeba evidovat jaké jsou důsledky a závažnost chyby.

S odstraňováním chyb potom souvisí seznam chyb, které jsou aktuálně řešeny (nevyřešeny) a informace o tom, kdo je zodpovědný za odstranění chyby.

Při tomto procesu může pomoct nějaký nástroj pro sledování chyb. Jde o nástroj, který umožňuje uložit informace o chybách a rozliší jednotlivé chyby (jednoznačně je identifikuje). Nástroj pro sledování chyb pak využívají všichni členové týmu. Jde o nástroj, který pomáhá celé skupince lidí pracovat na řadě malých problémů (jednotlivých chybách). V dnešní době existuje celá řada nástrojů. Například: Bugzilla, GNATS, FogBugs, JIRA, TestTrack a další.

Většina nástrojů jsou informační systémy, které pro uložení dat používají databázi a nejčastěji komunikují přes webové rozhraní. Dobrý nástroj pro sledování chyb by měl:

- uchovávat informace o chybě, včetně stavu ve kterém je (nevyřešena, řešena, …);
- umět pracovat se skupinami chyb (odstranění komplexnějšího problému);
- vyhledat chyby a sledovat aktuální změny;
- generovat statistiky a komplexnější zprávy o sledovaných chybách;
- podporovat historii jednotlivých chyb a být schopen spojit chyby s příslušnou verzí dané aplikace (integrace s SCM);
- rozlišovat závažnost chyby – nástroje typicky rozlišují několik úrovní chyb. Například chyby, které znemožňují použití aplikace (kategorie A). Méně závažné chyby, které je nutné odstranit, ale ovlivňují jen určité části aplikace (kategorie B). Chyby, které neovlivňují funkčnost celé aplikace (například špatný výpis, kategorie C).
- definovat zodpovědnost za řešení konkrétních chyb.

2.9.1 Bugzilla

Jeden z prvních a nepoužívanější volně dostupný nástroj pro sledování chyb. Je napsána v Perlu. Komunikuje přes internetový prohlížeč a používá email ke komunikaci. Implementuje běžné funkce očekávané od nástroje pro sledování změn. Vyhledávání je realizováno pomocí regulárních výrazů. Pro složitější dotazy můžeme použít boolovské výrazy. Podporuje spolupráci s LDAP, historie změn každé chyby, podpora závislostí mezi chybami. K chybám lze asociovat jejich prioritu. Také lze „hlasovat“ pro určení „otravných“ chyb. Tyto chyby pak budou odstraněny přednostně.

2.10 Generování dokumentace

Existuje celá řada přístupů, jak tvořit dokumentaci k vytvářené aplikaci. Obvykle je dokumentace vytvářena současně s aplikací. Vytvářenou dokumentaci můžeme v principu rozdělit na dvě části podle toho, pro koho je určena. První část bude dokumentace určená pro uživatele aplikace. Další pak dokumentace pro potřeby tvůrců aplikace. Dokumentace určená pro uživatele je typicky vytvářena další skupinou. Tito nemusí být programátory. Dokumentace určená pro programátory je pak vytvářena tvůrci aplikace a je primárně určena ostatním programátorům.

Další možné rozdělení aplikace je na:

- dokumentaci oddělenou od aplikace – uživatelská dokumentace bývá často oddělena od aplikace. Při vytváření musíme řešit problémy s aktualizací. Musíme udržovat konzistenci mezi vytvářenou aplikací a vytvářenou dokumentací.
- dokumentaci jako součást zdrojového kódu – v tomto případě je snadnější údržba (např. včetně verzování). Existuje celá řada přístupů, jak propojit zdrojové kódy s vytvářenou aplikací. Můžeme použít například literární programování (D. Knuth). Při tomto přístupu chceme, aby zdrojový kód aplikace byl sám sobě dokumentací. Další možný přístup je použití dokumentačních značek (Javadoc).

2.10.1 Program Javadoc

Dokumentace je realizovaná ve speciálních poznámkách.

```
/**
 * Dokumentační poznámka
 */
```

V rámci těchto poznámek můžeme použít různé specifické dokumentační značky. Výsledkem bude sada HTML stránek. V dokumentačních značkách proto můžeme využít možností tohoto jazyka.

- @author Joe
- @param x Popis parametru x

Rozšířením použití aplikace Javadoc může být generování zdrojových textů pomocí šablon – XDoclet. Ten můžeme použít například ke generování konfiguračních souborů. Informace pro konfigurační soubor získáme z poznámek ze zdrojových kódů (použití v aplikačním rámci Struts).

Následující příklad ukazuje použití Javadoc.

```
/**
 * Konstruktor zlomku.
 * Naplní čitatele a jmenovatele a převede
 * zlomek do normalizovaného tvaru.
 * @param citatel    Čitatel zlomku.
 * @param jmenovatel Jmenovatel zlomku.
 */
public Zlomek(int citatel, int jmenovatel)
{
```

```
    this.citatel = citatel;
    this.jmenovatel = jmenovatel;
    normalizuj();
}
```

2.11 Nasazení aplikace

Nasazení aplikace může být náročný proces. Musíme řešit různé problémy jako konfigurace okolního prostředí, nastavení parametrů aplikace či propojení s jinými aplikacemi. V tomto procesu nám můžou opět pomoct specifické nástroje. Mezi takové patří generátory instalačních balíčků. Tyto generátory mohou bát platformě závislé i nezávislé. Jak už plyne z názvu tyto nástroje, jsou schopny vygenerovat instalační balíček. Ten potom usnadňuje nasazení aplikace. Příkladem tohoto typu nástroje může být Antigen, Advanced Installer nebo IzPack.

2.12 Tvorba aplikací pro mezinárodní prostředí

Přirozeným požadavkem při vývoji větších aplikace je její přizpůsobení různým jazykovým úpravám a nebo přizpůsobení aplikace různým národním zvyklostem. Úpravu programů pro mezinárodní prostředí bychom mohli rozdělit na dvě části.

- Internacionalizace (i18n) – zajištění takových vlastností aplikace, aby byla potenciálně použitelná kdekoliv. Upravujeme:
    - formát data a času;
    - zápis čísel;
    - měna;
    - jazyk (abeceda, číslice, směr psaní, …);
    - telefonní čísla, adresy, PSČ;
    - míry a váhy.
- Lokalizace (l10n) – přizpůsobení aplikace konkrétnímu jazykovému a kulturnímu prostředí. Upravujeme:
    - jazykové verze, překlady;
    - zvyklosti;
    - symbolika;
    - estetika (barvy, ikony,…);
    - kulturní hodnoty, sociální kontext;

Existují různé metody jak řešit lokalizaci či internacionalizaci. Typický problém je, že tyto specifické problémy například s jazykem musí často řešit jiní odborníci než programátoři. Obvykle řešení tohoto problému je, že tyto informace umístíme vně vlastní aplikace. Pokud například chceme přeložit texty z aplikace umístíme tyto texty do externích souborů (např. soubory .properties v různých jazykových verzích). Tak je oddělíme od vlastního programu. Tyto soubory pak mohou nějací jazykoví odborníci přeložit. Moderní operační systémy pak obvykle podporují získání informaci o lokalizaci. Na úrovni programovacího jazyka pak tyto informace přečteme (například v Javě umístěno od java.util;) a můžeme zvolit vhodnou variantu externího souboru s texty (knihovna *gettext* – pro různé programovací jazyky). Obdobně bychom vyřešili speciální formáty čísel a podobně.

Kontrolní otázky:

1. Jak vypadá proces tvorby zdrojových kódů?
2. Znáte alespoň tři integrovaná vývojová prostředí?
3. Znáte nějakou aplikaci, která je umístěna do PDE SourceForge.net?
4. Zkuste formálně definovat co je to jazyk.
5. Jaký typ překladače je v principu rychlejší, kompilátor a nebo interpret?
6. Lze nějak získat z class souboru Javy původní zdrojový kód?
7. Co je uloženo v systému pro správu verzí?
8. Je CVS centralizovaný jednouživatelský SCM?
9. Soubor se zdrojovým kódem byl šestkrát změněn a pokaždé uložen do CVS. Znamená to, že nyní zabírá asi šestkrát tolik místa na disku?
10. Co se skrývá pod pojmem *sandbox* v CVS?
11. Pokud při ukládání souborů CVS automaticky vyřeší všechny potencionální konflikty. Znamená to, že ve zdrojovém kódu není žádná chyba?
12. Co je to *tag* v CVS?
13. Pokud použijete ANT a dobře sestavíte skript pro sestavení aplikace. Budou při sestavování kompilovány všechny soubory a nebo jen nutné?
14. Jak se obvykle jmenuje skript pro ANT?
15. Lze v ANTu používat proměnné? Pokud ano, umožňují totéž co proměnné v Javě?
16. Lze v ANTu kompilovat i jiné než Javovské aplikace?
17. K čemu slouží operace step-in, která se typicky používá v nástrojích pro ladění?
18. Jaký je rozdíl mezi verifikací a validací?
19. Jaký je rozdíl mezi jednotkovými a systémovými testy?
20. K čemu slouží metoda *setUp* v JUnit a jaký je rozdíl mezi použitím této metody a konstruktorem?
21. Co je to profiler?
22. Co je to refaktorizace?
23. Co je výsledkem činnosti nástroje Javadoc?
24. Jaký je rozdíl mezi internacionalizací a lokalizací?
25. K čemu se používají soubory s příponou .properties při lokalizaci aplikace?

Úkoly k zamyšlení:

1. Vyjmenujte editory, které používáte a zamyslete se nad tím proč.
2. Jak byste zkontrolovali správnost programu v Javě. Které chyby jste schopni odhalit.
3. Zamyslete se nad oblastmi, kde je výhodné použít interpretační překladač.
4. Umožňuje IDE, který používáte k programování, propojení z nějakým SCM?
5. Používáte nějaký SCM při programování. Proč ne?
6. Pokud na projektu pracuje více lidí, jak jim může usnadnit práci nástroj pro sestavení aplikace?
7. Používáte při vývoji aplikací ladící výpisy? Zamysleli jste se někdy nad použitím logovacího nástroje?
8. Které z prezentovaných nástrojů aktivně používáte při programování?
9. Které další nástroje by Vám mohly při programování usnadnit práci?

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními nástroji, které můžete využít při vytváření aplikace. Text nepředstavuje kompletní výčet nástrojů, spíše ukazuje rozšířené představitele jednotlivých kategorií. Jednotlivé nástroje také nejsou vyčerpávajícím způsobem popsány. Častěji jsou na příkladech ukázány jejich možnosti. Většina prezentovaných nástrojů se aktivně vyvíjí a průběžně se objevují nové.

3 Komponentní technologie

V této kapitole se dozvíte:

- Co je to komponentně orientované programování.
- Jak vypadá životní cyklus komponenty.
- Základní pravidla pro tvorbu COM, JavaBeans a komponent v .NETu.

Po jejím prostudování byste měli být schopni:

- Rozumět základním pojmům z oblasti komponentních technologií.
- Definovat fáze při vývoji komponenty.
- Vyjmenovat aktéry při komponentně orientovaném programování.
- Prakticky implementovat jednoduché komponenty JavaBeans, COM a nebo v .NET.
- Použít existující komponenty při vývoji aplikací.

Klíčová slova této kapitoly:

JavaBeans, COM, .NET, C#, aplikační server, webové služby, události, vlastnosti, GUI,…

Doba potřebná ke studiu: 10 hodin

Průvodce studiem

Studium této kapitoly je jednoduché a popisným způsobem zde nastudujete základní principy pro komponentně orientované programování. Prakticky si pak získané informace můžete ověřit na připravených praktických úkolech. Ty nejsou součástí tohoto textu a jsou dostupné prostřednictvím Internetu.
Na studium této části si vyhraďte minimálně 10 hodin. Po celkovém prostudování doporučuji vyřešit praktické úkoly. Na tuto část si vyhraďte dalších 12 hodin. (kapitolu nemusíte studovat najednou).

V této kapitole se dozvíte základní principy komponentních technologií. Po prostudování tohoto modulu získáte představu o tom, co to jsou programové komponenty, jaké vlastnosti by měly mít a jak se těchto vlastností dosahuje. Dále budete schopni porovnat jednotlivé komponentní technologie a vybrat vhodnou technologii pro konkrétní řešený problém. Budou představeny tři komponentní technologie COM, JavaBeans a komponenty v .NETu. Toto není kompletní výčet komponentních technologií. Existují i další (jako CORBA). Také si tento modul neklade za cíl kompletní popis těchto technologií. Spíše na příkladech představuje jejich základní rysy. Vybrané tři komponentní technologie také mají přímou vazbu na jazyky, s kterými se máte možnost během studia seznámit. Jde o Javu, C++ a C#.

3.1 Komponenty

Motivací, proč používat komponenty, je využití komponent v jiných odvětvích. Komponenty se například velice úspěšně používají v automobilovém a nebo stavebním průmyslu. Použití komponent v těchto odvětvích usnadňuje, urychluje a tím i zlevňuje výrobu.

V IT existuje několik oblastí, kde se používají komponenty. Komponenty se používají u technického vybavení. Fyzicky je počítač složen s komponent jako paměti, procesor a podobně. Tyto komponenty ale nebudou součástí tohoto kurzu. Zde se zaměříme na softwarové komponenty a jejich využití při vývoji aplikace.

- **Komponenta je obecně definována jako opakovatelně použitelný stavební blok programu. Jedná se o předem vytvořený a zapouzdřený kus aplikačního programového kódu, který lze kombinovat s jinými komponentami a s ručně psaným programem s cílem rychlého vývoje uživatelské aplikace.**

Komponenty jsou používány v celé řadě situací. Typicky se používají například při tvorbě grafického uživatelského rozhraní. Například ovládací tlačítko může být reprezentováno komponentou, která zapouzdřuje metody pro zobrazování tlačítka a reakci na různé události (jako např. kliknutí myší nebo stisknutí vhodné klávesy). Příkladem nevizuální komponenty je konektor zpřístupňující databázi, časovač nebo FTP server. Mohou to být ale i kompletní aplikace jako třeba textový editor, tabulkový procesor nebo internetový prohlížeč.

Vývoj aplikace postavené na komponentách se liší od běžné aplikace. Komponenta je kus programu, který je určen pro integraci třetí stranou. Autor komponenty neví, kdo a k čemu bude jeho komponenty využívat. Dle komponentní technologie, kterou požívá, specifikuje rozhraní, které jeho komponenta používá. Autor aplikace zase neví jak je komponenta vytvořen. V některých případech nemusí vůbec znát ani tvůrce komponenty. Jediné co o komponentě zná je její rozhraní. Přes toto rozhraní s komponentou komunikuje. Díky tomuto principu je vlastně oddělen vývoj vlastní aplikace a vývoj komponenty. Komponenta může být použita v libovolném kontextu. Je tedy značně zvýšena možnost znuvupoužitelnosti vytvořeného kódu. Díky použití komponent může být vývoj aplikace usnadněn, urychlen a tudíž i zlevněn. Dalším, v čem se komponentně orientované programování liší například od objektově orientovaného je oddělení rozhraní od implementace. Vše co uživatel komponenty a komponentě potřebuje vědět je její rozhraní. Komponenta se tak může vnitřně změnit, ale dokud není ovlivněno toto rozhraní, nemusí na to uživatel komponenty nijak reagovat. Tento uživatel také v principu vůbec nezná strukturu komponenty (například strukturu jejich tříd). Tento způsob programování také někdy nazýváme „black box programming“.

3.1.1 Struktura komponenty

Komponenty jsou charakterizovány rozhraním, které zahrnuje následující prvky: **vlastnosti, události a operace**. Vlastnosti představují z vnějšího pohledu mechanismus pro ukládání a získávání hodnot. Z pohledu implementace komponenty se pak může jednat o hodnoty uložené v proměnných, souborech, databázích nebo o hodnoty vypočtené nebo získané nějakým způsobem z jiných objektů. Vlastnosti reprezentují stav komponenty. Operace realizují funkcionalitu komponenty. Definují, jaké bude chování komponenty. Pro interakci s okolím se u komponentně orientovaného

programování používá mechanismus událostí. Tento model je postaven na následujících principech. Máme dva aktéry: zdroj událostí a posluchače. Zdroj událostí umožňuje, aby se u něj registrovali posluchači. V případě, že nastane situace, kdy by měla být vyvolána událost, zdroj vyvolá událost a o této skutečnosti informuje všechny registrované posluchače. Pomocí tohoto schématu komunikují komponenty s okolními objekty.

3.1.2 Životní cyklus komponenty

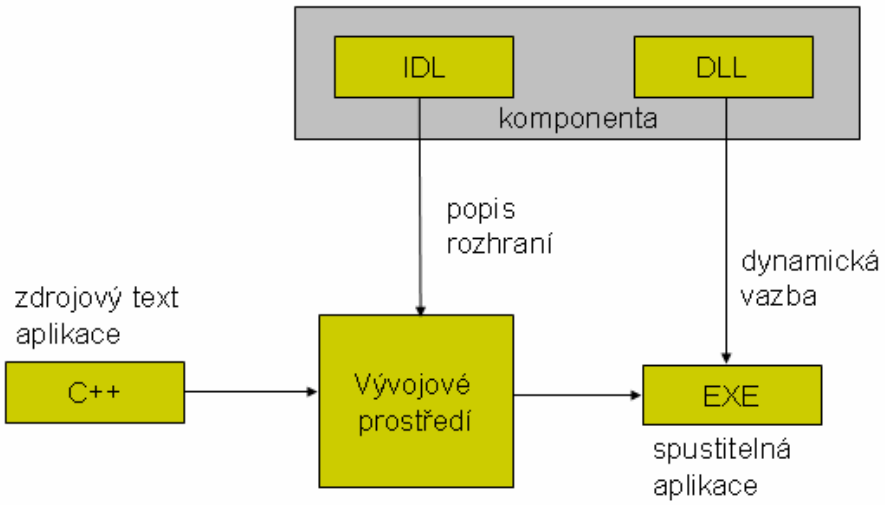
První fází při vývoji komponenty je její tvorba. Tuto část realizuje tvůrce aplikace. Existuje celá řada standardních komponentních technologií. Například můžeme použít standardy jako CORBA, COM+, EJB, .NET. Dalším krokem při vývoji komponenty je publikace rozhraní. Poslední fází je její šíření. Zde můžeme využít služeb jako je LDAP, JNDI, UDDI.

Takto publikovanou komponentu potom vyhledá integrátor (aktér, který integruje komponentu a využívá jejích služeb v budované aplikaci). Komponenta je připojena a dle publikovaného rozhraní. Vhodná komponenta může být vyhledána v době tvorby programu. Například podle nějakého jednoznačného identifikátoru. Další možnost je vyhledat komponentu, která má určité požadované vlastnosti. Také můžeme použít „pozdní vazbu“ a komponentu připojit až d bodě běhu a to ve chvíli, kdy jí opravdu potřebujeme. V připojení komponenty nám může pomoct například IDE. Které z publikovaného rozhraní zjistí potřebné informace a na základě nich realizuje přístup ke komponentě. Programátorovi se pak zdá, jako by přistupoval k interním objektům.

Důležité je, v jakém formátu publikujeme rozhraní komponenty. Jednou z možností by byl nějaký textový popis. Ten je sice snadno srozumitelný pro člověka, ale je přesto nevhodný. Může být nepřesný a nesrozumitelný pro počítač. Abychom mohli využít podpory nějakého nástroje pro propojení s komponentou, musí být její popis strojově čitelný a pokud možno platformě nezávislý. Jednou z možností, jak tuto situaci řešit, je definovat nějaký strukturovaný textový formát, v němž bude uložen deskriptor komponenty. Tato metoda se využívá například v technologii COM nebo CORBA, kde je deskriptor komponenty popsán speciálním jazykem IDL (Interface Definition Language) pro definici rozhraní. Tento popis potom jednoznačně, přesně a strojově čitelně definuje vlastnosti komponenty.

Další variantou je získání dokumentace analýzou vlastností, událostí a metod komponenty – tuto metodu využívá například technologie JavaBeans. Aby byla tato analýza možná, musí být pro pojmenování a strukturu jednotlivých prvků rozhraní komponenty dodrženy určité konvence. Takovou konvencí je např. to, že pro vlastnost x musí být komponentou definovány veřejné metody getX a setX. Dalším vylepšením předchozí metody je zařazení základních prvků komponent přímo do programovacího jazyka, jako to činí např. C#. Překladač tohoto jazyka pak může poskytnout potřebné metainformace o rozhraní komponenty bez nutnosti dodržovat nějaké konvence pro pojmenování. Navíc je schopen zajistit provedení kontrol, které by se jinak mohly provádět až v okamžiku použití komponenty.

Celý tento postup demonstruje následující příklad. V tomto konkrétním případě by šlo o vývoj nějaké COM komponenty.



Při vytváření aplikace z jednotlivých komponent často potřebujeme předem definovat některé jejich vlastnosti. Například můžeme chtít nastavit barvu pozadí nebo typ písma tlačítka, dobu čekání časovače nebo číslo portu pro FTP server. Vývojová prostředí podporující práci s komponentami obvykle nabízejí možnost nastavení hodnot vlastností pomocí formulářů (property sheets). Tyto formuláře obsahují pro každou nastavitelnou vlastnost buď textové pole, do kterého je možné vepsat hodnotu vlastnosti, případně umožňují spuštění speciálního editoru vlastnosti podle jejího typu, například dialogu pro nastavení barvy nebo typu písma.

3.2 JavaBeans

Technologie JavaBeans je komponentní architekturou pro aplikace vytvářené na platformě jazyka Java. Je založena na sadě pravidel umožňujících nastavovat a získávat hodnoty vlastností komponent, předávat události mezi jednotlivými objekty, vytvářet instance objektů a ukládat objekty pomocí serializace. Umožňuje rovněž reprezentovat popisné (meta-)informace o objektech. Komponenty JavaBeans se vytvářejí v souladu s jednotnou specifikací aplikačního programového rozhraní (JavaBeans API) a mohou pracovat v libovolném prostředí podporujícím jazyk Java. Typickou vlastností JavaBeans komponent je možnost s nimi pracovat v rámci vizuálních vývojových prostředí a nástrojů pro vytváření aplikací.

Komponenty JavaBeans mohou také zprostředkovávat přemostění platformě závislých komponentních modelů, jako jsou ActiveX, OpenDoc nebo LiveConnect. Tím lze dosáhnout přenositelnosti komponent mezi různými kontejnery, v rámci nichž tyto komponenty pracují – např. Netscape, Internet Explorer, Visual Basic, Microsoft Word nebo Lotus Notes

3.2.1 Co je to JavaBean komponenta

JavaBean je opakovatelně použitelná programová komponenta, se kterou lze vizuálně manipulovat ve vývojových prostředích. Těmito vývojovými prostředími mohou být nástroje pro tvorbu webových stránek, vizuální prostředí pro tvorbu aplikací, nástroje pro vytváření grafického uživatelského rozhraní nebo pro realizaci serverových aplikací. Ale může to být i obyčejný editor dokumentů, který dokáže zařadit *bean* jako součást dokumentu.

Java Bean komponentou může být například jednoduchý prvek uživatelského rozhraní, jako je tlačítko nebo editační pole ve formuláři, případně složitější programová komponenta jako třeba tabulkový kalkulátor. Některé komponenty dokonce nemusejí být viditelnými součástmi grafického rozhraní, i když je možné je používat ve vizuálních nástrojích – například časovače, datové zdroje apod. Typickou Java Bean komponentu lze charakterizovat následujícími vlastnostmi:

- Introspekce – umožňující vývojovým nástrojům analyzovat to, jak komponenta pracuje.
- Přizpůsobivost – možnost nastavení vzhledu a chování komponenty při vývoji aplikace.
- Události – prostředek pro komunikaci mezi komponentami.
- Vlastnosti – nastavitelné hodnoty určené pro přizpůsobení i pro programovou obsluhu komponenty.
- Perzistence – možnost uložit přizpůsobenou komponentu a později její stav obnovit.

Java Bean komponenta je implementována jako obyčejná třída v Javě, přičemž se nepožaduje, aby tato třída byla odvozena z nějaké konkrétní bázové třídy nebo aby implementovala konkrétní rozhraní. V případě vizuální komponenty je tato třída odvozena od třídy `java.awt.Component`, aby bylo možné ji zařadit do vizuálních kontejnerů. V současné době se s technologií Java Beans setkáváme stále častěji. Zřejmě nejznámějším příkladem aplikace této technologie je samotná knihovna Swing. V oblasti serverových aplikací se dále můžeme setkat s komponentami Enterprise Java Beans, které umožňují tvorbu přenositelných serverových komponent určených zejména pro implementaci informačních systémů.

3.2.2 Struktura Java Bean komponent

> ➢ **Java Bean komponenta je charakterizována množinou vlastností, které nabízí, množinou metod, které mohou být volány z jiných komponent, a množinou událostí, které generuje.**

Vlastnosti jsou pojmenované atributy komponenty, které můžeme číst nebo nastavovat voláním vhodných metod komponenty. Vlastnosti jsou obecně realizovány dvěma přístupovými metodami *set* a *get*. Tyto metody pro nějakou vlastnost pojmenovanou např. `length` mají rozhraní:

```
public void setLength(int length)
public int  getLength()
```

V případě, že je vlastnost typu `boolean`, může být metoda *get* nahrazena metodou *is*, například:

```
public boolean isEmpty()
```

Povšimněte si toho, že v uvedených metodách je první písmeno jména vlastnosti (za předponou get, set nebo is) vždy velké.

Následující příklad ukazuje, komponentu `Counter`, která obsahuje vlastnost `value`.

```
public class Counter {
   private int value;
   public int getValue()
      return this.value;
   }
   public void setValue(int val) {
      this.value=val;
   }
   …
}
```

Vlastnosti JavaBeans komponenty mohou být některého z následujících typů:

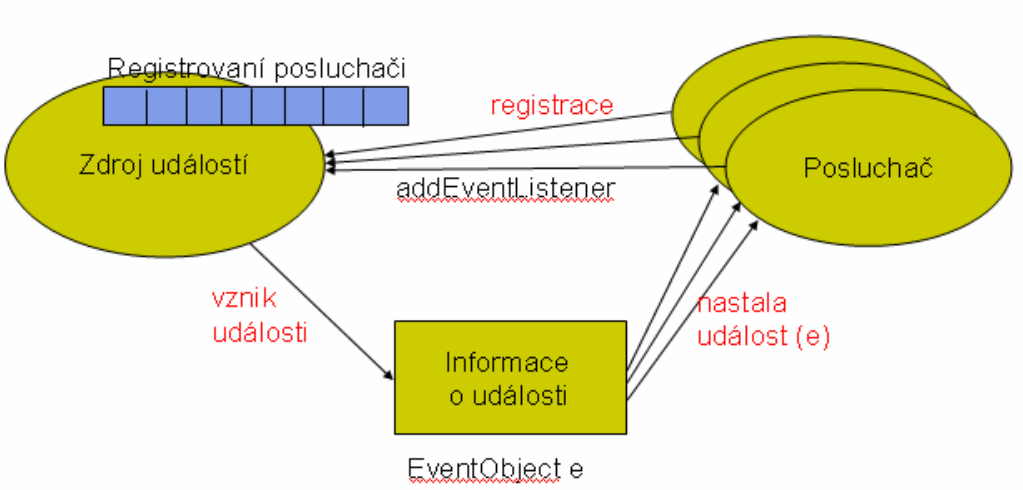
- *jednoduchá vlastnost* – jednoduchá hodnota, která se může měnit nezávisle na hodnotě jiných vlastností. Jednoduchou vlastnost demonstroval předcházející příklad.
- *vázaná vlastnost* – vlastnost, jejíž změna je oznámena jiné komponentě. Toto oznámení je realizováno generováním události `PropertyChange`. Tato třída stejně jako další třídy usnadňující práci s JavaBean komponentami je v balíku `java.bean` ve standardních knihovnách Javy;
- *omezená vlastnost* – vlastnost, jejíž změna je ověřována jinou komponentou, která může změnu odmítnout. Podobně jako u vázaných vlastností i zde je použití omezených vlastností postaveno na principu generování událostí. Při změně hodnoty omezené vlastnosti je vygenerovaná událost `VetoableChange`. Registrovaní posluchači této události zkontrolují, zda je změna přípustná. Pokud není, mohou nastavení nové hodnoty zabránit generováním výjimky;
- *indexovaná vlastnost* – posloupnost vlastností. Pod touto vlastností si můžete představit například nějakou kolekci. Metody set a get potom obsahují další parametr, což je index prvku, který chceme získat případně změnit. Přístupové metody pro indexovanou vlastnost value by mohly mít tuto signaturu:

```
   public int getValue(int index)
   public void setValue(int index,int val)
```

Metody jsou obyčejné javovské instanční metody, implicitně jsou exportovány všechny veřejné metody třídy implementující komponentu.

Události umožňují oznámit jiné komponentě, že se stalo něco zajímavého. Komponenty, které mají o konkrétní událost zájem, se registrují u zdroje události jako posluchači. Nastane-li odpovídající situace, zavolá zdroj události příslušnou metodu na všech registrovaných posluchačích. Model událostí používaný JavaBeans byl převzat z modelu událostí knihovny JDK 1.1 AWT založeného na principu registrace poslouchačů. Jeho hlavními cíli jsou zjednodušení propojení metod s událostmi, zajištění silné typové kontroly a využití standardních „návrhových vzorů“.

Pokud může komponenta generovat nějaké události, musí umožnit registraci posluchačů. V případě, že událost nastane, vyvolá komponenta určitou pojmenovanou a typovanou metodu, která je součástí konkrétního rozhraní implementovaného registrovaným posluchačem. Následující obrázek demonstruje použití událostí.



Při implementaci mechanismu událostí musíte dodržet jisté konvence. Celý postup bychom mohli shrnout takto:

1. Posluchač se zaregistruje u zdroje událostí (např. u tlačítka, na jehož stisknutí čeká). Posluchač musí implementovat rozhraní, které rozšiřuje rozhraní `EventListener`. Zdroj událostí zase musí být schopen takového posluchače zaregistrovat (pokud posluchač implementuje rozhraní `EventListener` provede voláním metody `addEventListener`) a udržuje seznam registrovaných posluchačů.
2. Dojde k vyvolání události. Například uživatel stiskne tlačítko. Zdroj události (tlačítko) projde seznam registrovaných posluchačů a každému z nich oznámí vznik události zavoláním dohodnuté metody. Tuto metodu právě specifikuje rozhraní, které posluchač implementuje (v našem příkladě `EventListener`).
3. Metodě předá informace o události. Tyto informace jsou zapouzdřeny v instanci třídy, která musí rozšiřovat třídu `java.util.EventObject`.
4. Posluchač reaguje na událost.

3.2.3 Komponenta Counter

Popsané vlastnosti Java Bean komponent demonstruje následující příklad. Jde o Jednoduchou komponentu, která realizuje čítač. Tento čítač ve vlastnosti `value` uchovává celočíselnou hodnotu (na začátku nula). Tato hodnota je voláním metody `increment` inkrementována vždy o jedničku. Čítač má také Obsahuje vlastnost limit. Pokud je při inkrementování vlastnosti `value` dosažen tento `limit`, je vygenerovaná událost.

```
public class CounterBean implements Counter {
  private int value = 0;
  private int limit = 10;
  CounterEventListener listener;
```

```
  //vlastnost value
  public int getValue() {
    return value;
  }
  public void setValue(int value) {
    this.value = value;
  }
  //vlastnost limit
  public int getLimit() {
    return limit;
  }
  public void setLimit(int limit) {
    this.limit = limit;
  }
  //metoda inkrementujici hodnotu citace
  public void increment() {
    if( value >= limit ) {
      if( listener != null ) listener.limitReached(new
CounterEvent(this));
    } else
      value++;
  }

  //metody pro registrovani a deregistrovani posluchace(tato
  //trida umoznuje pridat jedineho posluchace
  public void addCounterEventListener(CounterEventListener
listener)
      throws java.util.TooManyListenersException
  {
     if( this.listener != null )
       throw new java.util.TooManyListenersException();
     this.listener = listener;
  }
  public void removeCounterEventListener(CounterEventListener
listener) {
     this.listener = null;
  }
}
```

V komponentě používáme rozhraní `CounterEventListener`. Toto rozhraní specifikuje události, které je schopna komponenta generovat.

Rozhraní pak musí implementovat všichni posluchači. Také musí rozšiřovat rozhraní `EventListener`. Mohlo by vypadat takto:

```
public interface CounterEventListener extends EventListener {
  void limitReached(CounterEvent event);
}
```

Poslední třída, která nám chybí, aby byla implementace komponenty `Counter` funkční je `CounterEvent`. Tuto třídu jsme použili k přenesení informací o události posluchačům. Z instance této třídy by posluchači získali informace potřebné ke zpracování události. Tato třída musí rozšiřovat třídu `EventObject` a proto musí minimálně obsahovat referenci na zdroj událostí.

```
public class CounterEvent extends EventObject {
  public CounterEvent(Counter source) {
    super(source);
  }
  public Counter getCounter() {
    return (Counter)source;
  }
}
```

Následující příklad ukazuje, jak můžeme použít komponentu. Komponenta `Counter` je instanciovaná a je u ní registrován posluchač.

```
class MyListener implements CounterEventListener {
   void run() {
       Counter counter = new Counter();
       counter.addCounterListener(this);
       counter.increment();
   }
   public void limitReached(CounterEvent event){
       Systém.out.println(…);
   }
}
```

3.3 COM

V této kapitole se dozvíte, jak technologie COM vznikala, jaké jsou její hlavní principy.

Komponentní model COM firmy Microsoft byl uveden v roce 1993. V současné době jde o vyspělou technologii pro vývoj aplikací, která se široce používá v operačních systémech Windows a Windows NT. Je podporována množstvím integrovaných služeb a vývojových nástrojů.

Původní COM byl navržen pro použití v aplikacích pracujících na jediném počítači. Jeho rozšířením o možnost přístupu ke komponentám umístěných na jiných počítačích přes počítačovou síť vzniknul model DCOM (Distributed

COM), uvedený v roce 1996. Dalším rozšířením byl model COM+ z roku 2001, který se stal základem pro současnou architekturu .NET.
Na COM je možné se dívat ze dvou pohledů:

- **COM je specifikace pro tvorbu komponent – popisuje jak má vypadat komponenta a jak probíhá komunikace mezi komponentami;**
- **COM je technologie, která implementuje část této specifikace – například implementuje třídy, které pomáhají realizovat komunikaci mezi komponentami.**

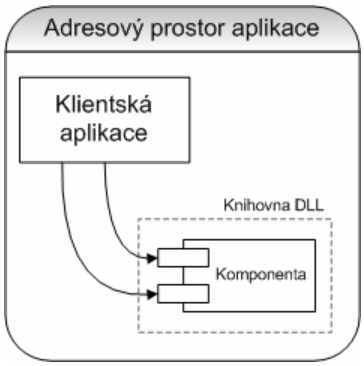
COM je postaven na třech základních principech.

- Při vytváření aplikace programátoři využívají rozhraní.
- Zdrojový kód komponenty není staticky připojen, spíše nahrán na požádání za běhu aplikace.
- Programátoři implementující COM objekty deklarují požadavky a systém zajistí, že tyto požadavky budou splněny.

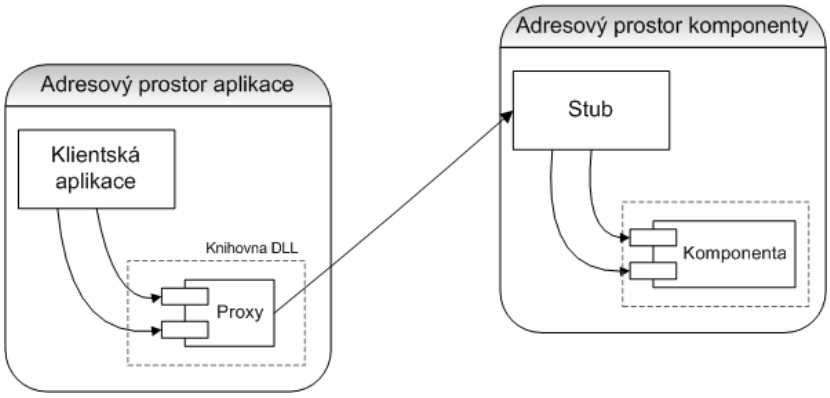
COM rozhraní je kolekce abstraktních operací, které mohou být provedeny objektem. Toto rozhraní o komponentě zveřejňujeme. Z hlediska implementace jde o abstraktní třídu s čistě virtuálními metodami. V technologii COM je toto rozhraní jednoznačně popsáno jazykem pro popis rozhraní (M)IDL (modifikace jazyka IDL). V předchozím testu jsme mluvili o komponentách. V technologii COM je pod pojmem komponenta (**COM komponenta**) rozuměn binární soubor obsahující výkonný kód. Každá COM komponenta pak může zapouzdřovat jednu nebo více definic výkonných jednotek. Tyto jednotky jsou pak označovány jako COM třídy. Každá **COM třída** může implementovat jedno a nebo více COM rozhraní. Tuto COM třídu nelze ztotožňovat například s třídou v jazyce C++ (třída v C++ může být COM třídou, ale jedna COM třída může být složena z celé řady tříd). Analogicky jako v objektově orientovaném programování můžeme definovat vztah mezi COM třídou a COM objektem. **COM objekt** je instancí nějaké COM třídy. Je vytvořen ve chvíli, kdy chceme využít služeb komponenty.

3.3.1 Typy komponent

Architektura COM je založena na binárních komponentách s definovanou sadou rozhraní. Binární komponenta znamená, že komponenta je aplikace zkompilovaná do binární formy spustitelné na dané platformě. Každá komponenta je uložena v nějakém binárním souboru. Protože je technologie COM vyvinuta primárně pro rodinu operačních systému Windows, je to buď dynamicky linkovaná knihovna (DLL), nebo samostatná aplikace (EXE).
Aplikace využívající COM komponenty ve formě dynamických knihoven s komponentou komunikuje přímo – zavede si ji do svého adresového prostoru a volá její služby jako obyčejné funkce. Výhodou tohoto řešení je, že se zjednoduší komunikace mezi aplikací a komponentami. Hovoříme pak o tak zvané „in process“ komponentě.



Pokud ovšem aplikace komunikuje s komponentou uloženou v samostatném spustitelném souboru, je situace složitější. Aplikace i komponenta mají každá svůj vlastní adresový prostor, takže aplikace nemůže komponentu volat přímo. Tato situace se řeší prostřednictvím zástupného (proxy) objektu. Ten na straně aplikace zastupuje komponentu – převezme od aplikace argumenty volání, provede jejich serializaci (marshalling) a zajistí jejich přenos do adresového prostoru komponenty. Tam je připraven tzv. stub, jenž předaná data deserializuje a zajistí volání komponenty. Po provedení operace stub serializuje případné návratové hodnoty, předá je proxy na straně aplikace, provede se opět jejich deserializace a aplikace může pokračovat ve svém běhu. Tuto operaci znázorňuje následující obrázek.



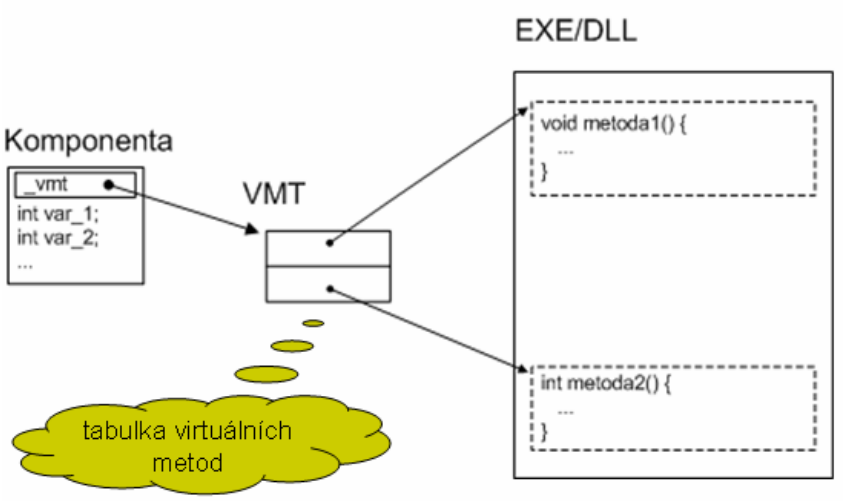
Proxy objekt zastupující komponentu na straně klientské aplikace a stub zajišťující komunikaci s aplikací na straně komponenty není třeba vytvářet ručně. Pro jejich automatické generování postačuje pouze znát rozhraní komponenty. Pro popis rozhraní se ve specifikaci COM používá jazyk MIDL, což je poněkud upravená verze standardizovaného jazyka IDL (Interface Description Language) používaného například v technologii CORBA. Překladem specifikace rozhraní se získá jednak hlavičkový soubor s definicí rozhraní, jednak implementace proxy objektu a stubu. Tím je programátor také osvobozen od nutnosti udržovat tyto tři složky vzájemně konzistentní.

COM umožňuje použití komponent na jednom počítači, ale v rámci různých adresových prostorů. Jedním z rozšíření technologie COM je Distributed COM – DCOM. Ten umožňuje volat komponenty umístěné na jiném počítači.

3.3.2 Tabulka virtuálních metod

Komponentní model COM je založen na využití rozhraní, pomocí nichž komponenta komunikuje se svým okolím. Na rozdíl od jiných modelů (CORBA, JavaBeans) je toto rozhraní definováno až na binární úrovni, takže umožňuje efektivní komunikaci s komponentami psanými v různých programovacích jazycích. Rozhraní COM je tvořeno pouze metodami, takže i přístup k datům je realizován výlučně voláním metod, což umožňuje zcela oddělit implementaci komponent od aplikací, které tyto komponenty využívají. Definice binární struktury rozhraní COM komponent vychází z mechanismu, který se například v jazyce C++ používá pro volání virtuálních metod. Toto volání pro nějaký konkrétní objekt probíhá tak, že se nejprve získá ukazatel na tabulku virtuálních metod, jenž je součástí instance objektu. Tato tabulka je vlastně pole ukazatelů na kódy konkrétních metod, takže známe-li pořadové číslo volané metody, vyzvedneme z tabulky adresu jejího kódu a metodu provedeme. Tento postup je znázorněn na následujícím příkladu.

Předpokládejme, že máme objekt se dvěma virtuálními metodami. Tabulka virtuálních metod tedy bude mít dvě položky, obsahující ukazatele na kód těchto dvou metod.



Tento mechanismus nám umožňuje přístup k metodám nezávisle na programovacím jazyce. Komponenta obsahuje binární kód. Ten je přímo proveditelný na dané platformě. Jak bylo popsáno, k volání používáme tabulku virtuálních metod. COM komponenty tedy můžeme použít v libovolném jazyce, který je schopen tuto strukturu implementovat a použít.

3.3.3 Identifikace komponent

Pokud chceme používat komponenty, musíme je být schopni jednoznačně identifikovat. Ke globální identifikaci komponent a jejich rozhraní v COM se používá *Global component and interface ID* – GUID. Jde o 16 bytový řetězec. Například:

```
{3F2504E0-4F89-11D3-9A0C-0305E82C3301}
```

Základní ideou při generování GUID je, aby dva výrobci komponent nemohli vygenerovat stejný identifikátor. Algoritmus pro generování GUID například původně obsahoval MAC adresu (bylo možné vystopovat autora dokumentu – červ Melissa). GUID jsme schopni vygenerovat pomocí utility přímo v systému (GUIDGEN.EXE – generátor součástí například Visual Studia).

Obvykle se používá jako konstanta:

```
const IID IID_IRandom =
   {0x771853E0,0x78D1,0x11d7,
   {0xBF,0xB4,0xED,0x72,0x61,0xDE,0xA8,0x3D}};
```

Každé COM rozhraní i každá COM třída je identifikovaná nějakým GUID.

3.3.4 Rozhraní IUnknown

Ne každý objekt implementující nějaká rozhraní můžeme považovat za COM objekt. Tím se stane až v okamžiku, kdy implementuje rozhraní IUnknown, definované specifikací COM. Toto rozhraní má mezi všemi ostatními zvláštní postavení, neboť jakékoliv jiné COM rozhraní musí obsahovat také metody rozhraní IUnknown (např. tak, že definici tohoto rozhraní zdědí) a každá COM třída musí rozhraní IUnknown implementovat.

Rozhraní IUnknown obsahuje tři metody: *QueryInterface, AddRef a Release*.

Metoda QueryInterface:

```
virtual HRESULT __stdcall* QueryInterface(
     const IID& iid, void** ppv) = 0;
```

Metoda QueryInterface slouží k získání ukazatele na rozhraní, jehož identifikace je dána parametrem `iid`. Jde o jednoznačný GUID přiděleny tomuto rozhraní. Pokud komponenta nepodporuje rozhraní, na které se dotazujeme, vrátí metoda QueryInterface návratovou hodnotu E_NOINTERFACE. Je-li však toto rozhraní podporováno, vrací hodnotu S_OK a do parametru ppv uloží ukazatel na požadované rozhraní. Tento ukazatel je pak možné použít pro volání metod nalezeného rozhraní. Takto můžeme otestovat, zda daný COM objekt implementuje dané rozhraní.

Metody AddRef a Release:

```
virtual ULONG __stdcall AddRef() = 0;
virtual ULONG __stdcall Release() = 0;
```

Další dvě metody jsou určeny pro sledování, kolik referencí na komponentu je aktivních. Klientská aplikace je povinna zavolat metodu AddRef, kdykoliv vytvoří nový ukazatel na rozhraní, a metodu Release, kdykoliv tento ukazatel zase zruší. Je-li zrušen poslední odkaz na komponentu, může se komponenta z paměti odstranit a uvolnit tak přidělené prostředky.

3.3.5 Příklad komponenty

Následující sekce na příkladu ukazuje, jak funguje tvorba komponent pomocí COM technologie. Příklad ukazuje jednoduchý generátor náhodných čísel.

Každá komponenta musí dědit z rozhraní IUnknown. Rozhraní komponenty pro generování náhodných čísel začíná direktivou pro zabránění opakovaného vkládání tohoto souboru do textu a pro vložení definice rozhraní IUnknown.

```
#pragma once
#include "unknwn.h"  // Definice rozhraní IUnknown
```

Dále následuje definice globálního identifikátoru pro rozhraní IRandom. Tento identifikátor získáme voláním programu GUIDGEN.EXE a můžeme si vybrat z několika nabízených formátů. Zvolíme si formát používající makro.

```
DEFINE_GUID: // {CB8DF8CB-3F6C-4c02-A587-18566C28487B}
DEFINE_GUID(IID_IRandom,
    0xcb8df8cb,
    0x3f6c,0x4c02,0xa5,0x87,0x18,0x56,0x6c,0x28,0x48,0x7b);
```

A nyní již následuje definice rozhraní. Nenechte se zmást klíčovým slovem interface, které samozřejmě není součástí C++, ale jedná se pouze o předefinované klíčové slovo `struct`. V této definici i ve zbývajících částech programu jsou rovněž použita další makra, která zajišťují větší nezávislost na konkrétní verzi překladače. Zde konkrétně makro STDMETHODCALL zajistí použití správných volacích konvencí.

```
interface IRandom : IUnknown {
   virtual HRESULT STDMETHODCALLTYPE Start( int seed) = 0;
   virtual HRESULT STDMETHODCALLTYPE Next( int *val) =0;
};
```

Rozhraní IRandom budeme implementovat třídou CRandomImpl, jejíž rozhraní následuje. Tato třída kromě všech metod rozhraní IUnknown a IRandom obsahuje rovněž konstruktor a dvě instanční proměnné. Proměnná m_refCnt bude sloužit jako počítadlo odkazů, v proměnné m_seed je uložena aktuální zdrojové číslo pro generování dalšího pseudonáhodného čísla. Povšimněte si, že všechny metody rozhraní s výjimkou AddRef a Release vracejí jako návratovou hodnotu HRESULT, zbývající výstupní proměnné se předávají odkazem v argumentech.

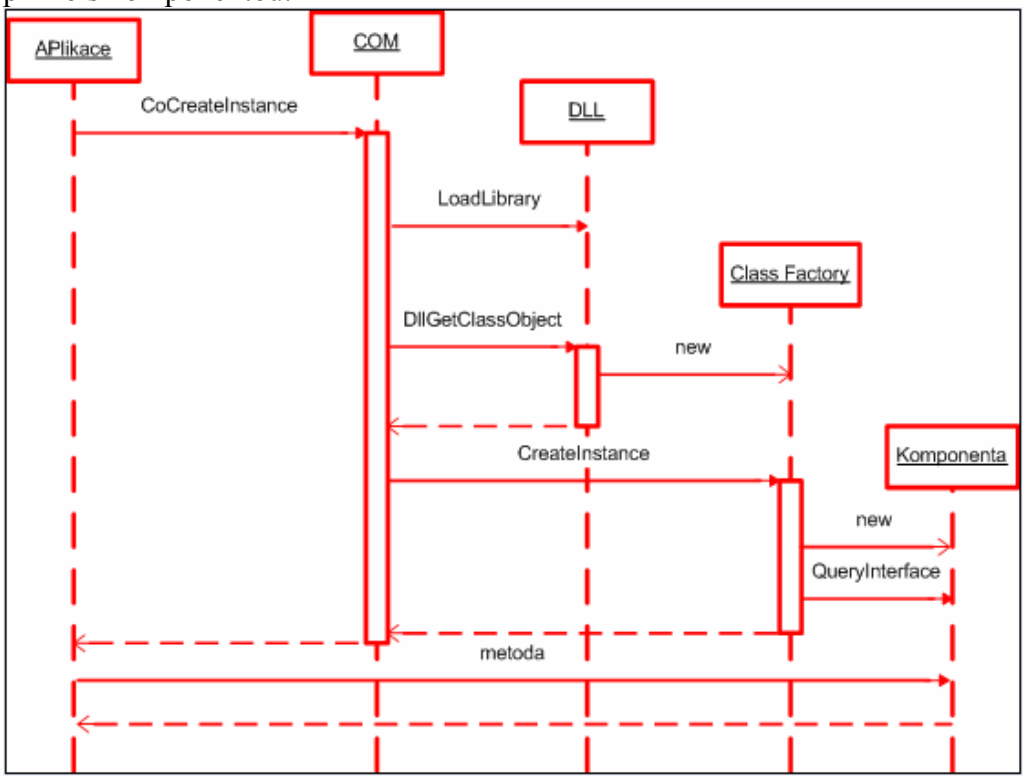
```
#pragma once
#include "IRandom.h"
class CRandomImpl : public IRandom
{
public:
   CRandomImpl();

   // IUnknown
   STDMETHODIMP QueryInterface(REFIID, void **);
   STDMETHODIMP_(ULONG) AddRef(void);
   STDMETHODIMP_(ULONG) Release(void);
   // IRandom
   STDMETHODIMP Start(int seed);
   STDMETHODIMP Next(int* result);
private:
   ULONG m_refCnt;
   ULONG m_seed;
};
```

3.3.6 Použití komponenty v aplikaci

Posledním krokem při využití COM technologie je použití komponent klientskou aplikací. Prvním krokem při použití je vytvoření instance COM objektu v paměti. Zde je využit mechanizmus class factory. Základním principem je, že předáme zodpovědnost za vytváření instancí naší komponenty jiné komponentě, která již bude mít k dispozici dostatek informací o tom, jak instanci vytvořit a jak ji inicializovat. Tato komponenta, nazývaná v terminologii *COM class factory*, má pevně definované rozhraní a může v obecném případě být schopna vyrobit i více typů komponent současně.

Potřebuje-li klientská aplikace ve svém paměťovém prostoru vytvořit instanci komponenty, předá identifikaci požadované komponenty funkci `CoCreateInstance`, která je součástí infrastruktury COM. Na základě identifikace se v systémovém registru zjistí, kde je umístěn odpovídající soubor DLL. Po zavedení knihovny do paměti se provede základní inicializace, při níž se vytvoří instance class factory. Ta pak obdrží požadavek na vytvoření instance komponenty, jenž vyřídí. Dále již klientská aplikace komunikuje přímo s komponentou.



V případě, že je komponenta realizována samostatnou serverovou aplikací, pak je uvedený postup podobný. Vzhledem k tomu, že ale komunikujeme přes více adresových prostorů, dochází při komunikaci klienta s komponentou k serializaci a deserializaci dat uvnitř zastupujících proxy a stubů.

Vlastní použití komponenty je již relativně snadnou záležitostí. Nejprve musíme inicializovat infrastrukturu COM voláním funkce CoInitialize. Instanci nové komponenty vytvoříme voláním funkce CoCreateInstance, které předáme CLSID třídy implementující komponentu a IID rozhraní, přes které chceme s komponentou komunikovat. V případě úspěšného vytvoření obdržíme ukazatel

na požadované rozhraní, který nakonec uvolníme voláním metody Release rozhraní IUnknown.

```
#define INITGUID
#include "IRandom.h"
#include <iostream>
using namespace std;

void main(int argc, char** argv) {
   // inicializace COM
   HRESULT hr = CoInitialize(NULL);
   if (FAILED(hr)) { ... }
   // vytvoření instance komponenty
   IRandom *pRnd = NULL;
   hr = CoCreateInstance(CLSID_RandomImpl, NULL, CLSCTX_ALL,
      IID_IRandom, (void **)&pRnd);
   if (FAILED(hr)) { ... }
   // použití komponenty
   pRnd->Start(1234);
   // uvolnění komponenty
   pRnd->Release();
   // finalizace COM
   CoUninitialize();
}
```

Uvedené ukázky v jazyce C++ představují (pokud neuvažujeme dnes již pro tyto účely téměř nepoužívaný jazyk C) z hlediska programátora ten nejobtížnější postup, jak vytvářet a používat COM komponenty v aplikacích. Další programovací jazyky jako Visual Basic nebo C# již poskytují podstatně větší komfort.

```
Option Explicit On
Module RandomTest
    Sub Main()
        Dim rnd As RandomLib.IRandom
        Dim val As Integer

        rnd = New RandomLib.RandomImpl
        rnd.Start(1234)
        For i As Integer = 0 To 9
            rnd.Next(val)
            Console.WriteLine("{0}: {1}", i, val)
        Next
    End Sub
End Module
```

3.3.7 COM+

Nejnovějším rozšíření technologie COM je COM+. Zachovává vlastnosti COM/DCOM. Navíc přináší:

- **Konfigurované komponenty** - kromě informací v registry přidán i katalog. Komponenta, kterou chceme konfigurovat je přidaná do katalogu a v katalogu můžeme nastavit vlastnosti komponenty. Například můžeme nastavit, které aplikační služby má podporovat. Obdobně bychom mohli říci, že konfigurovaná komponenta je ta, která má záznam v katalogu.
- Aplikační služby – v COM+ byla přidaná možnost nastavit u jednotlivých komponent některé aplikační služby.
    - Automatické transakce- možnost nastavit konfiguraci
    - Just-in-time aktivace – je využit kontextový objekt
    - Object pooling – možnost odložit nepotřebný objekt, uvolnit jej z paměti (stav je uložen do perzistentní paměti)
    - Fronty volání
    - Události

COM+ zavádí také některé nové pojmy. Jedním z takových nových pojmů je **kontext**. Kontext je seskupení objektů se shodně nakonfigurovanými vlastnostmi. V rámci jednoho kontextu spolu mohou komunikovat COM objekty přímo, jinak musí použít proxy. Kontextový objekt je pak objekt reprezentující daný kontext

3.4 Komponenty v .NET

Informace o platformě .NET obecně a o programovacím jazyce C# najdete na adrese http://www.cs.vsb.cz/behalek/vyuka/pcsharp/. V dalším textu se budeme zabývat jen tvorbou komponent v .NET a bude se předpokládat, že jsem obeznámeni minimálně se základy programování v C# na platformě .NET.

Také byste měli nastudovat základy fungování platformy .NET.

Základní principy pro fungování platformy .NET jsou postavena na podobných principech jako u jazyka Java. .NET je multijazykové prostředí. Můžeme použít celou řadu programovacích jazyků. Všechny tyto jazyka jsou kompilovány do instrukcí MSIL (Microsoft Intermediate Language). Ty to instrukce jsou pak prováděny běhovým prostředím (CLR – Common Language Runtime). Při jejich provádění je použit JIT – Just In Time překladač. Instrukce MSIL jsou tímto překladačem transformovány do nativních instrukcí na dané platformě a potom jsou provedeny. Aby tento systém fungoval, musí každý jazyk používaný na platformě .NET dodržovat jistá pravidla. Každý takový jazyk musí implementovat jednotnou specifikaci (CLS – Common Language Specification) a jednotný typový systém (CTS – Common Type System).

Hlavním rozdílem mezi implementací komponent v .NET a jinými komponentními technologiemi jako JavaBeans a nebo COM je, že vývoj komponent je v .NET podporován na nativní úrovni. Můžeme přímo definovat různé položky ve třídě, jako vlastnosti nebo události. Tyto konstrukce jsou definovány společným typovým systémem. Každý jazyk je tedy nějak musí

realizovat. V další části si ukážeme, jak lze implementovat jednoduchou komponentu v jazyce C#.

3.4.1 Vývoj komponent v jazyce C#

Podobně jako u JavaBeans i zde je každá komponenta složena ze tří typů položek. Komponenta je složena z *vlastností, metod a událostí*.

Vlastnosti jsou realizovány dvěma přístupovými metodami *get* a *set*. Jejich realizaci demonstruje následující příklad.

```
class Point {
   private short x, y;    //data
   public short X {    //vlastnost
      get { return x; }
      set { x = value; }
   }
}
```

Speciální význam má klíčové slovo *value*. Příklad demonstruje vlastnost, která jako typ používá `short`. Metoda `get` vrací tento typ a metoda `set` má jediný argument stejného typu. Prezentovaná konstrukce neumožňuje pojmenovat tento parametr. Proto je v rámci metody `set` možné použít klíčové slovo `value`, které reprezentuje tuto hodnotu. K vlastnosti přistupujeme jako v proměnné. Pomocí jména získáme hodnotu a hodnotu do vlastnosti vložíme pomocí operátoru „=“.

Metody jsou běžné veřejné metody třídy.

Před ukázkou, jak jsou v .NET řešeny události si ukážeme jinou specifickou konstrukci této platformy. Jde o *delegáta*. Delegát je typově bezpečný ukazatel na funkci. Delegát má několik použití. Z hlediska komponentního programování je nejzajímavější jeho použití pro mechanismus událostí. Deklarace delegáta má tento tvar:

```
modifiers delegate type delegatsName(parameter1,...);
```

Následující příklad demonstruje, jak můžeme deklarovat delegáta. Následující třída obsahuje definici delegáta. Z definice delegáta vyplývá, jaký je typ funkce, kterou bude zastupovat. Také demonstruje, že můžeme deklarovat položky typu delegáta a tu potom používat jako běžnou metodu.

```
class Text {
   string text;
   public Text(string text) { this.text=text;}
   public delegate void SomePrefix();
   public void WriteIt(SomePrefix prefix)  {
      prefix(); //jako běžná metoda
      Console.WriteLine(text);
   }
}
```

Před použitím je potřeba delegáta instanciovat. Tím jej svážeme s konkrétní funkcí. Následující třída implementuje dvě metody. Ty typově odpovídající definovanému delegátovi z předchozího příkladu.

```
class PrefixBuilder {
   public static void SimplePrefix() {
      Console.Write("## ");
   }
   public void NicePrefix() {
      Console.Write(">-how nice-> ");
   }
}
```

Tyto metody pak použijeme při instanciování delegáta.

```
class RunApp {
   static void Main() {
      Text text=new Text("Hello");

      Text.SomePrefix simplePrefix=new
      Text.SomePrefix(PrefixBuilder.SimplePrefix);

      PrefixBuilder prefixBuilder=new PrefixBuilder();

       Text.SomePrefix nicePrefix=new
          Text.SomePrefix(prefixBuilder.NicePrefix);
      text.WriteIt(simplePrefix);
      text.WriteIt(nicePrefix);
   }
}
```

Delegát nemusí být interně spojen jen s jednou metodou. Můžeme provést kompozici pomocí operátorů + a –.

```
   Text.SomePrefix greatPrefix=simplePrefix +
      nicePrefix + simplePrefix;

   text.WriteIt(greatPrefix);

   greatPrefix-=nicePrefix;
   text.WriteIt(greatPrefix);
```

Výstup programu bude:

```
   ## >-how nice-> ## Hello
   ## ## Hello
```

Zpracování událostí realizováno pomocí delegátů. Pro mechanismus událostí musí mít dva parametry a oba jsou objekty. První udává kdo je zdrojem události. Druhý obsahuje informace spojené s konkrétní událostí. Jde o instanci třídy EventArgs. Definice události je součástí třídy a má následující tvar: `event JmenoDelegata JmenoUdalosti;`

Poslední příklad ukazuje jednoduchou komponentu realizující jednoduchý čítač.

```
public delegate void CounterEvent(object sender, EventArgs
eventArgs);

public class Counter  {
        public event CounterEvent LimitReached;

        private int val;
        private int limit;

        public int Value {
            get { return val; }
            set { val = value; }
        }

        public int Limit {
            get { return limit; }
            set { limit = value; }
        }

        public Counter(int limit) {
            this.limit = limit;
        }
        public void Increment() {
            val++;
            if (val >= limit) {
                if (LimitReached != null)
                    LimitReached(this, EventArgs.Empty);
                val = 0;
            }
        }
        public void Reset()
        {
            val = 0;
        }
    }
```

3.4.2 Distribuce komponenty

.NET je produktem firmy Microsoft. Jako takový je vlastně z hlediska komponentních technologii přímým nástupcem technologie COM. Další

důležitou oblastí, je distribuce komponenty. V této oblasti řeší .NET celou řadu problémů, které byly u technologie COM.

Problém identifikace jednotlivých komponent je v COM řešen pomocí jednoznačného GUID. Prvním problémem může být to, že dva výrobci komponent používají stejné GUID. Tento problém ale není (vzhledem k velikosti GUID) tak závažný. S identifikací COM komponent je ale spojena celá řada jiných problémů, souhrnně označovaných jako „DLL Hell“. Tyto problémy vystihuje následující příklad.

Předpokládejme, že mám dvě aplikace – A a B. Obě používají komponentu X.

1. Nainstalujeme aplikaci A. Tato aplikace nainstaluje do sdíleného prostoru pro komponenty (určité adresáře ve Windows) komponentu X ve verzi 1.0.
2. Nainstalujeme aplikaci B. Tato aplikace přeinstaluje komponentu X novou verzí této komponenty (řekněme ve verzi 1.1). Zatím vše funguje. Výrobce komponenty X se postará o zpětnou kompatibilitu.
3. Uživatel přeinstaluje aplikaci A. V této chvíli je obnovena verze 1.0 komponenty X. Zároveň může přestat fungovat aplikace B. Přitom s ní uživatel vůbec nic nedělal. Uživatel vlastně vůbec neví, jaké komponenty používá.

Tento příklad demonstruje, jaké problémy se snažili vyřešit při realizaci technologie .NET. Je potřeba lépe identifikovat komponenty jednotlivých výrobců a zajistit možnost udržovat více verzí komponenty.

Základní jednotkou distribuce v .NET je assembly. Assembly zapouzdřuje jeden a nebo více DLL souborů nebo samostatnou aplikaci (EXE soubor) do jednoho balíku. DLL soubory uvnitř assembly se obvykle označují jako moduly a i když jich může assembly obsahovat více, obvykle jde o právě jeden soubor. Kromě zmíněných souborů assembly může obsahovat další zdroje, jako jsou ikony, obrázky nebo soubory pro lokalizaci. Každá assembly obsahuje různá metadata. Metadata generuje kompilátor a popisují typy obsažené v assembly. Z hlediska komponent je ale mnohem zajímavější manifest. Podobně, jako metadata popisují typy v assembly, tak manifest poskytuje informace o komponentě jako takové. Manifest obsahuje *jméno assembly, verzi, lokalizaci (*případně *„silné“ jméno“,* o něm bude řeč později). Manifest také obsahuje jména a *hash* hodnotu všech souborů, které jsou v ní uloženy (díky tomu nelze například jednoduše vyměnit některý ze souborů v assembly). Hodnoty uložené v manifestu můžeme měnit pomocí atributů.

Pokud chceme distribuovat nějakou komponentu, musíme ji umístit do assembly. Pak existují dva modely, jak může být naše komponenta používána.

1. Privátní komponenty – distribuovaná assembly je součástí aplikace. Taková to komponenta je instalovaná současně s danou aplikací pouhým „kopírováním“. Výrobce aplikace se postará o správné fungování a nikdo jiný ke komponentě nemá přístup.
2. Sdílené komponenty – komponenty jsou umístěny do GAC (Global Assembly Cache). Tyto komponenty mohou být používány více aplikacemi. Assembly, kterou chceme umístit do GAC musí obsahovat silné jméno. Součástí informací, které assembly obsahuje je i verze a v GAC může být několik verzí stejné komponenty.

Sdílené komponenty jsou rozlišeny pomocí „silného jména“ (strong name). Toto silné jméno je založeno na digitálním podpisu a využívá kódování

s veřejným a privátním klíčem. Díky tomuto mechanismu je zaručeno, že dva výrobci nebudou mít stejné sdílené jméno.

Součástí informací, které obsahuje manifest klientské aplikace pak je i soubor assembly, které daná aplikace používá a to včetně příslušných verzí. Zbytek je plně v režii běhového prostředí .NET. V principu se chová takto:

- Je požadován nějaký typ, který je umístěn v assembly, která ještě nebyla načtena do paměti.
- Nejprve je prohledána GAC a je zjišťováno, zda neobsahuje požadovanou assembly v požadované verzi. Pokud ano, je tato assembly použita.
- Pokud hledaná assembly není v GAC je prohledán „lokální“ privátní prostor aplikace a předpokládá se, že hledaná assembly je v něm.

Kromě zmíněných údajů jsou v rámci manifestu uloženy i další údaje řešící například bezpečnost či přístup k assembly.

Kontrolní otázky:

1. Jaký je životní cyklus komponenty?
2. Co znamená, že v .NET jsou komponenty podporovány na nativní úrovni?
3. Co znamená, že COM komponenty jsou binárně kompatibilní?
4. Co musí splňovat programovací jazyk, aby mohl používat COM komponenty?
5. Byl v technologii JavaBeans rozšířen jazyk Java o nové programové konstrukce?

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy komponentně orientovaného programování. Byly představení někteří představitelé komponentně orientovaných technologii. Specificky byly ukázány základy komponentních technologií COM a JavaBeans. Také bylo ukázáno, jak můžeme implementovat komponenty v prostředí .NET.

4 Správa paměti

V této kapitole se dozvíte:

- Jakým způsobem je přidělovaná paměť za běhu aplikace.
- Jak funguje manuální a automatické přidělování paměti.
- Co je to „garbage collector“.
- Jak je řešena správa paměti v různých programovacích jazycích.

Po jejím prostudování byste měli být schopni:

- Alokovat a dealokovat paměť v různých jazycích.
- Realizovat systém pro správu paměti.

Klíčová slova této kapitoly:

zásobník, hromada, správa paměti, inkrementální regenerace, dvojfázové značkování, počítání referencí

Doba potřebná ke studiu: 2 hodin

Průvodce studiem

Studium této kapitoly je jednoduché a popisným způsobem zde nastudujete různé typy nástrojů. Ke studiu tohoto modulu je nutné mít základní znalosti programování v jazyce C/C++, Java nebo C#, na kterých budou některé algoritmy prezentovány. Pro praktické ověření je pak vhodné mít nainstalováno některé vývojové prostředí.
Na studium této části si vyhraďte minimálně 2 hodiny. Po celkovém prostudování doporučuji vyřešit praktické úkoly. Na tuto část si vyhraďte dalších 4 hodin. (kapitolu nemusíte studovat najednou).

V následující kapitole se budeme zabývat přidělováním paměti. Přidělovat paměť můžeme na několika úrovních. Klíčovou roli zde také hraje operační systém. Dneska typicky používané operační systémy umožňují běh více procesů. S tím je také spojena nutnost rozdělit fyzickou paměť počítače mezi tyto procesy. Obvyklým řešením je, že aplikace používá virtuální adresový prostor a ten je potom nějakým způsobem mapován do paměti fyzické. O tom se ale v této kapitole bavit nebudeme. Cílem kapitoly jsou algoritmy a přístupy pomocí nichž je paměť přidělovaná v rámci adresového prostoru, který je přidělen aplikaci.

Pracujeme-li s daty, jejichž rozsah je předem znám, obvykle vystačíme se statickým přidělováním paměti. To znamená, že již při překladu programu lze určit, kde v paměti budou data umístěna a jakou velikost budou zabírat. Například v jazyce C je paměť pro veškeré proměnné deklarované na globální úrovni nebo označené klíčovým slovem static přidělována staticky. V podstatě to znamená, že pokud v programu používáme nějakou proměnnou, je kompilátorem a nebo běhovým prostředím vyhrazena oblast paměti a ta je této proměnné přidělena.

```
int pole[10]; // 10 * sizeof(int)
```

Pokud však není velikost nebo počet položek konkrétní datové struktury v době překladu známý, musíme použít dynamické přidělování paměti. Paměť je v tomto případě vyhrazena až za běhu programu, obvykle na základě volání speciální funkce nebo použitím operátoru pro přidělení paměti. To demonstruje následující příklad.

```
int* pole = new int[pocet];
```

Proměnná počet může být například zadaná uživatelem z klávesnice.

Existují ale i mnohem méně očividné příklady, kdy je mnohem obtížnější definovat, jakým způsobem musíme přidělit paměť. Zamyslete se například nad způsobem přidělování paměti pro proměnné deklarované uvnitř funkce a pro parametry funkce. Známe-li jejich velikost a počet již v době překladu, je tato podmínka dostatečná pro statické přidělení? Na tuto otázku nejsme schopni odpovědět, pokud neznáme další vlastnosti jazyka. Uvažujeme-li možnost rekurzivního volání funkcí, pak pro lokální proměnné a parametry funkce nemůžeme paměťový prostor přidělit staticky. Nevíme totiž, kolikrát bude v konkrétním okamžiku díky rekurzi konkrétní proměnná v paměti existovat. Každé volání funkce má obvykle své vlastní proměnné, jejichž hodnoty se mohou měnit zcela nezávisle na hodnotách stejně pojmenovaných proměnných z jiných volání téže funkce. Nevíme tedy, kolik současných výskytů jedné proměnné bude existovat – to je závislé na počtu úrovní rekurzivního volání. Paměť pro tyto proměnné tedy musíme přidělit dynamicky, vždy v okamžiku volání podprogramu.

Například při výpočtu hodnoty faktoriálu bude funkce `faktorial` volána rekurzivně až do úrovně dané hodnotou argumentu n. Prostor pro parametr n tedy musí být přidělen dynamicky, i když jeho velikost známe předem.

```
int faktorial(int n) {
   return n == 0 ? 1 : n * faktorial(n-1);
}
```

Pro dynamické přidělování paměti existuje mnoho metod, které se od sebe liší jednak efektivitou využití paměti, která je pro vyhodnocení programu dostupná, jednak velikostí časové a prostorové režie, která se na správu paměti spotřebuje. V této kapitole se těmto algoritmům budeme věnovat podrobněji a ukážeme si, jak se používají v prostředí konkrétních programovacích jazyků a jak se jejich implementace na vlastnostech těchto jazyků projeví.

4.1 Úrovně správy paměti

Správa paměti je obvykle rozdělena do tří úrovní – můžeme ji studovat z pozice technického vybavení, operačního systému nebo aplikací.

V rámci předmětu Programovací techniky se budeme ovšem zabývat pouze aplikační vrstvou správy paměti. Dalším dvěma vrstvám se věnují předměty zaměřené na architektury počítačů a operační systémy.

Na úrovni technického vybavení se správa paměti zabývá elektronickými prvky, v nichž jsou data skutečně uložena. Tato oblast zahrnuje zejména paměťové prvky RAM a paměti typu cache.

Na úrovni operačního systému musí být paměť přidělována uživatelským programům a není-li dále programem vyžadována, pak je znovu použita pro

jiné programy. Operační systém může předstírat, že má počítač mnohem více paměti než odpovídá skutečnosti, případně že každý program má celou dostupnou paměť pouze pro svou potřebu – tyto situace řeší systém virtuální paměti.

Konečně na úrovni aplikačních programů zahrnuje správa paměti přidělování úseků omezené dostupné paměti pro objekty a datové struktury programu a obvykle i opakované použití paměti, která již není obsazena. Správa paměti aplikace řeší dva hlaví úkoly:

- **Přidělování paměti – vyžaduje-li program blok paměti, musí správce vyhledat a přidělit úsek odpovídající délky z většího bloku paměti získané od operačního systému.**
- **Regenerace paměti – není-li úsek paměti přidělené programu dále využíván, může být uvolněn a dán k dispozici pro opakované použití.**

V zásadě zde existují dva možné přístupy: o uvolnění paměti musí rozhodnout programátor (tzv. *manuální správa paměti*), případně musí správa paměti o uvolnění rozhodnout sama (tzv. *automatická správa paměti*).

Během přidělování a uvolňování paměti je třeba dodržovat jisté omezující podmínky, které zahrnují mimo jiné i *časovou režii* – dodatečný čas spotřebovaný správou paměti během činnosti programu, dobu *pozdržení interaktivity* – zpoždění, které pozoruje interaktivní uživatel a paměťovou režii. Paměťová režie je *množství paměti*, které se spotřebuje pro administraci, zaokrouhlování velikosti bloků (tzv. interní fragmentace) a nevhodné využití paměti (tzv. externí fragmentace).

4.2 Problémy správy paměti

Základním problémem správy paměti je správné rozhodnutí o tom, zda je v nějakém úseku paměti třeba ponechat data, která obsahuje, případně zda je možné tato data zahodit a úsek paměti znovu využít pro jiné účely. Ačkoliv to zní jednoduše, jedná se o obtížný problém, jemuž se věnuje samostatná oblast výzkumu. V ideálním případě by se programátor neměl o správu paměti vůbec zajímat. Existuje však naneštěstí mnoho možností, jak může špatné spravování paměti ovlivnit robustnost a rychlost programů, a to jak při manuální, tak i při automatické správě paměti. Mezi typické problémy patří:

- **Předčasné uvolnění paměti** – mnoho programů uvolní paměť, avšak pokouší se k ní přistupovat později, což může být příčinou havárie nebo neočekávaného chování programu. Tento problém nastává obvykle při manuální správě paměti.
- **Únik paměti** – k úniku paměti dochází tehdy, pokud program neustále přiděluje novou paměť, aniž by ji zase uvolňoval. To může vést až k havárii programu v důsledku vyčerpání dostupné volné paměti.
- **Externí fragmentace** – špatně navržená metoda přidělování paměti může vést k tomu, že nelze přidělit dostatečně velký blok volné paměti, i když celkové množství volné paměti je větší. Tato situace vzniká tehdy, pokud je volná paměť rozdělena na mnoho malých bloků, mezi nimiž jsou stále používané bloky, a nazývá se externí fragmentací.

- **Špatná lokalita odkazů** – další problém se strukturou přidělených bloků vychází z toho, jak moderní procesory a operační systémy pracují s pamětí. Přístupy k paměti jsou rychlejší, pokud pracujeme s ne příliš od sebe vzdálenými místy. Pokud správa paměti umístí bloky, k nimž program přistupuje současně, daleko od sebe, může to vést ke zhoršení výkonu programu.
- **Nepřizpůsobivý návrh** – další problémy s výkonem programu mohou nastat tehdy, pokud metoda přidělování paměti předem předpokládá jisté vlastnosti programu, například typickou velikost bloků, posloupnost odkazů nebo dobu života přidělovaných objektů. Nejsou-li tyto předpoklady splněny, může se celková režie správy paměti zvýšit.

Dobře navržená správa paměti může zjednodušit psaní ladicích nástrojů. Takové nástroje mohou zobrazovat objekty, přesouvat se mezi odkazy nebo detekovat neočekávané nahromadění jistých typů bloků nebo jejich velikostí.

4.3 Realizace správy paměti

Správu paměti bychom z hlediska realizace mohli rozdělit do dvou kategorií.

- Manuální správa paměti
- Automatická správa paměti

4.3.1 Manuální správa paměti

Při manuální správě paměti má programátor plnou kontrolu nad tím, zda a ve kterém okamžiku bude paměť uvolněna a případně využita pro opakované přidělení. To obvykle nastává buď explicitním voláním funkcí pro přidělování a uvolňování paměti z hromady (např. malloc/free v jazyce C), nebo jazykovými konstrukcemi ovlivňujícími zásobník (např. pro lokální proměnné). Klíčovou vlastností manuální správy paměti je možnost, aby program sám vrátil část paměti a oznámil, že ji již dále nepotřebuje. Bez tohoto oznámení správa paměti žádný úsek opakovaně nevyužije.

V následující ukázce je uvedena funkce, která vytvoří kopii zadaného řetězce. Nejprve je zjištěna jeho velikost, poté se funkcí malloc přidělí dostatečně velký prostor, do kterého se řetězec okopíruje, a nakonec se vrátí adresa nové kopie řetězce. Po zavolání této funkce je programátor zodpovědný za uvolnění přiděleného prostoru v okamžiku, kdy se kopie řetězce již nebude dále používat.

```
char* strdup(const char* s) {
   int len = strlen(s);
   char* new_s = (char*)malloc(len+1);
   strcpy(new_s, s);
   return new_s;
}
```

Manuální správa paměti přenechává veškerou zodpovědnost za správné uvolňování paměti na programátorovi, což může vést k obtížně odhalitelným chybám, o nichž jsme se již zmínili dříve. Moderní jazyky se proto orientují častěji na automatickou správu, při které lze těmto chybám předejít – samozřejmě za předpokladu, že je zvolena spolehlivá metoda regenerace paměti.

4.3.2 Automatická správa paměti

Automatická správa paměti je služba, která je součástí jazyka nebo jeho rozšířením, a která automaticky regeneruje paměť, kterou by program již znovu nevyužil. Automatická správa programu (zvaná také garbage collector, „sběrač odpadu“) tuto činnost provádí opakovaným použitím těch bloků, které již nejsou živé, tj. program s nimi již dále nepracuje. Pojem živosti bloku se však obvykle nechápe zcela ideálně, většinou se pracuje konzervativněji s bloky, které jsou nedosažitelné z jisté sady programových proměnných (tzv. kořenů dosažitelnosti – např. globální a lokální proměnné, registry apod.) a na které tedy není možné se dostat pomocí ukazatelů. Je-li například blok paměti sice dosažitelný, ale program jej již dále nepoužívá, je tato situace obecně těžko odhalitelná, i když v některých případech může pomoci překladač použitím mnohem přesnějších metod analýzy toku dat.

Častým obratem v některých jazycích s automatickou správou paměti je „vynulování“ odkazu na objekt po jeho posledním použití. Je-li tento odkaz jediným odkazem na objekt, stane se objekt nedostupným a při nejbližší příležitosti je možné jím obsazenou paměť uvolnit.

4.4 Metody přidělování paměti

Při přidělování paměti vycházíme z předpokladu, že máme k dispozici jistým způsobem organizovanou volnou paměť, ze které odebíráme podle požadavků aplikace vždy část paměťového prostoru a přidělujeme ji jednotlivým datovým objektům. Volná paměť je tvořena obvykle seznamem souvislých paměťových bloků, jejichž adresu a délku známe. Úkolem přidělování paměti je pro zadanou velikost požadované paměti vyhledat vhodný úsek volné paměti, tento úsek označit za obsazený a vrátit jeho počáteční adresu.

V této kapitole se budeme věnovat algoritmům, které zajišťují s různou mírou efektivity toto přidělování paměti o zadané velikosti. Existuje celá řada algoritmů pro přidělování paměti. Některé z nich budou dále prezentovány.

4.4.1 Přidělování na zásobníku

Nejjednodušší metodou přidělování paměti je přidělování z jediného souvislého bloku paměti, organizovaného jako zásobník. Pro přidělení potřebujeme mít k dispozici pouze adresu začátku a konce volné paměti. Každému požadavku na přidělení paměti je přiřazena aktuální adresa začátku volné paměti a tato adresa je pak zvýšena o velikost požadavku, přičemž se hlídá překročení limitu volné paměti. Uvolňování paměti je pak implementováno jako prázdná operace, případně je možné bloky paměti uvolňovat vždy v opačném pořadí než v jakém byly přiděleny. Při uvolňování paměti je ukazatel začátku volné paměti opět vrácen zpět o velikost uvolňovaného bloku.

Následující příklad ukazuje implementaci přidělování paměti z jediného souvislého bloku, přičemž operace uvolňování je implementována jako prázdná.

```
class SimpleAllocator {
   // Vytvoří objekt typu SimpleAllocator, přidělující volnou
   // paměť z bloku
```

```
    // na adrese memAddr o velikosti memSize
   public SimpleAllocator(char* memAddr, unsigned memSize) {
      m_addr = memAddr;
      m_size = memSize;
   }
   // Přidělí blok paměti o velikosti size a vrátí jeho adresu.
   // Není-li požadovaný prostor k dispozici, aktivuje výjimku
   // NoMemoryException
   public char* alloc(unsigned size) {
      if( size > m_size ) throw new NoMemoryException();
      char* addr = m_addr;
      m_addr += size;
      return addr;
   }
    // Uvolnění bloku paměti je prázdná operace.
    public void free(char* addr, unsigned size) {}

    // Aktuální začátek volné paměti
    protected char*    m_addr;

    // Aktuální velikost volné paměti
    protected unsigned m_size;
}
```

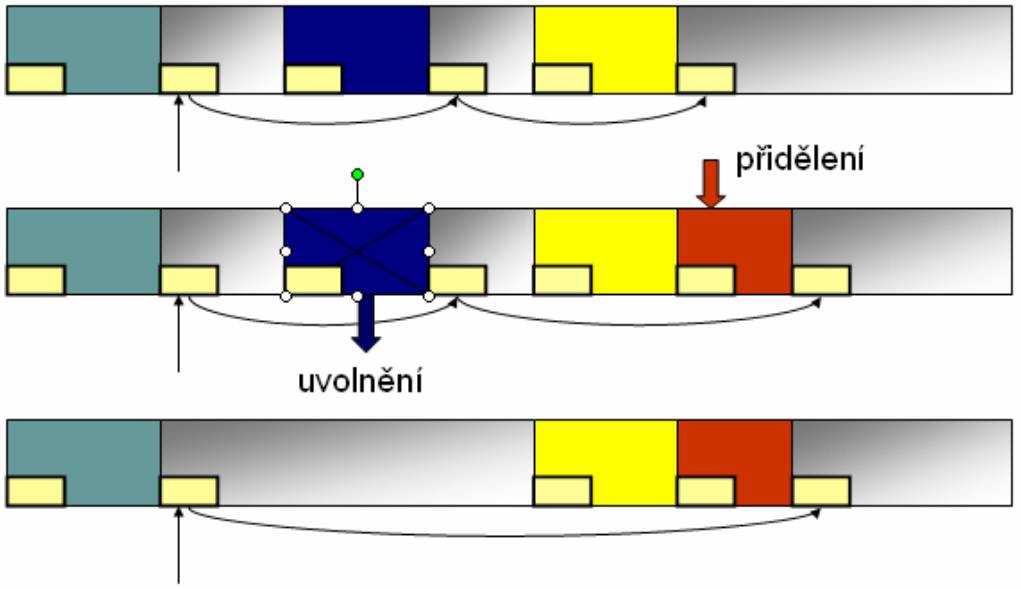
Metoda přidělování ze souvislého bloku paměti je velmi rychlá, a proto se často používá například ve funkci subalokátoru pro přidělování paměti v rámci bloku získaném jinou metodou. Můžeme se s ní také setkat v programovacích jazycích jako je C, C++ nebo Pascal při přidělování paměti pro aktivační záznamy funkcí (obsahující lokální proměnné, návratovou adresu a předávané parametry volání). Právě zde je totiž zajištěno to, že se přidělené bloky uvolňují v obráceném pořadí, a to vždy při návratu z volání funkce.

Jednou z modifikací této metody je zavedení operací mark a release. Operace mark uloží aktuální hodnotu ukazatele začátku volné paměti do proměnné a operace release tento ukazatel z proměnné opět obnoví. Tímto způsobem lze zajistit uvolnění veškeré přidělené paměti od okamžiku, kdy byla provedena odpovídající operace mark. Tyto operace můžeme například velmi efektivně použít například při alokování respektive dealokování lokálních proměnných.

4.4.2 Přidělování paměti ze seznamu volných bloků

Následující metoda prezentuje jiný přístup. U přidělování paměti na zásobník jsme mohli uvolnit paměť, jen v opačném pořadí, než v jakém byla alokovaná. Tento přístup je vhodný například pro lokální proměnné. Pokud ovšem neznáme délku života proměnných, je využívaná následující metoda.

Na následujícím příkladě je znázorněna paměť. Posupně jsou přidělovány bloky paměti určité délky. Protože neznáme délku života jednotlivých bloků, musíme uchovávat nějaký seznam volných bloků. V případě potřeby alokovat paměť je pak vybrán dostatečně velký volný blok. Je-li nějaký blok uvolněn, musí být naopak do seznamu volných bloků zatříděn.



Máme-li volné bloky paměti různé délky seřazené do seznamu, je třeba při požadavku na přidělení paměti určité velikosti vyhledat v seznamu dostatečně velký blok. Obecně může být takových bloků v seznamu více a je tedy třeba zvolit určitou strategii výběru nejvhodnějšího z nich. Vybraný blok je ze seznamu volných bloků odstraněn a je-li delší než je požadovaná velikost, vloží se přebytečná část zpět do seznamu jako další volný blok. Vzhledem k tomu, že pro uložení délky bloku a adresy následujícího bloku v seznamu je nutné rezervovat určitý prostor, nemá někdy vytvoření samostatného bloku z nevyužité části přidělované paměti smysl; v tomto případě je obvykle přidělen celý volný blok.

Existuje celá řada přístupů, jak zvolit vhodný velký blok. Nejpoužívanější metoda je metoda *výběru prvního vhodného bloku (**First fit**)*. Metoda jednoduše prochází seznam volných bloků a vybere z něj první blok, jehož velikost je větší nebo rovna požadované velikosti. Je-li blok delší, rozdělí se a zbývající část je vložena zpět do seznamu.

To však vede k situaci, že dělením velkých bloků na začátku seznamu vznikne mnoho malých bloků, jejichž sekvenční procházení může podstatně zpomalit operaci přidělování paměti. Jedním z možných řešení je využití složitějších datových struktur pro ukládání volných bloků, např. stromů.

Při uvolňování paměti je třeba volný blok zařadit zpět do seznamu. Nejjednodušší a nejrychlejší metodou je zařazení volného bloku na začátek seznamu. Jinou variantou je zařazení do uspořádaného seznamu podle adres bloku – to zjednodušuje slévání sousedních volných bloků do větších souvislých bloků volné paměti, ovšem za cenu sekvenčního vyhledávání při uvolňování paměti. Tato varianta zajišťuje menší fragmentaci paměti a je nejčastější používaná. Další možností je umístění volného bloku na konec seznamu.

Modifikací metody je *výběr dalšího volného bloku (**next fit**)*, kdy vyhledávání vhodného bloku začíná vždy na pozici, kde předchozí vyhledávání skončilo. Prohledávání tedy nezačíná vždy na stejném místě, ale postupně prochází celým seznamem, což zabrání hromadění menších bloků na začátku seznamu. Velkou nevýhodou této metody je však to, že bloky přidělované v téže fázi výpočtu mohou být od sebe značně vzdáleny, z čehož pak vyplývá menší lokalita odkazů a s ní spojený pomalejší přístup do paměti. Naopak bloky s různou dobou života mohou ležet vedle sebe, a to zase způsobuje větší fragmentaci paměti poté, co jsou bloky s kratší dobou života uvolněny. Celkově lze tedy říci, že i přes některé výhody vede výběr dalšího volného bloku k menší efektivitě přidělování paměti.

Další modifikací této metody jsou například metody:

- **Worst fit –** ta vychází z ideje, že pokud použijeme největší volný blok, zbude dost volného místa a zbylé volné místi bude ještě použitelné.
- **Best fit** – snažíme se najít volný blok, jehož velikost je větší nebo rovna požadované.

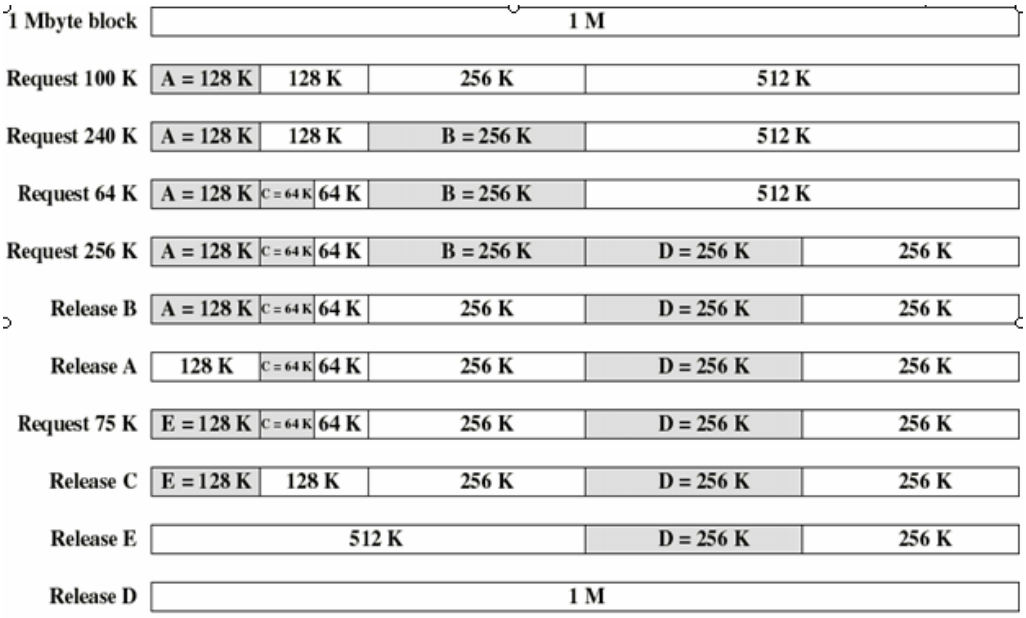
4.4.3 Přidělování s omezenou velikostí bloku (buddy system)

Alternativou k předcházející metodě je přidělování paměti s omezenou velikostí bloků. Tyto metody přidělování paměti jsou založeny na hierarchickém dělení volného paměťového prostoru na části. V nejjednodušším případě je paměť rozdělena na dvě velké části, ty se dále dělí na menší části atd. Toto dělení paměti definuje omezující podmínky na to, kde jsou bloky paměti alokovány, jaká je jejich možná velikost a jak mohou být volné bloky znovu spojovány do větších celků. Pro každou možnou velikost bloku se udržuje samostatný seznam volných bloků, takže jde v podstatě o variantu přidělování výběrem nejlepšího vhodného bloku, i když s nejrůznějšími variacemi týkajícími se způsobu rozdělování a spojování bloků. Zásadou spojování bloků je, že se mohou spojit pouze ty bloky, které spolu sousedí na stejné úrovni hierarchie (buddies). Výsledný blok pak patří do bezprostředně vyšší úrovně hierarchie.

Cílem tohoto systému je, aby bylo možné při uvolnění bloku najít jeho souseda pouze jednoduchým adresovým výpočtem. Tímto sousedem může být buď celý volný blok, nebo blok, jenž je přidělen jako celek nebo je rozdělen na další bloky. Další výhodou je nízká paměťová režie, postačuje pouze jediný bit obsahující informaci o tom, zda je blok volný nebo ne. Žádné další ukazatele nebo pomocné informace nejsou potřeba. Cenou za tuto výhodu je, že při uvolňování bloku musíme znát jeho velikost – ta je však obvykle v typovaných jazycích známa.

Nejznámější variantou je **binární přidělování**, kdy jsou velikosti bloků vždy mocninami dvou. Při dělení se blok vždy rozdělí na dvě stejné části. Všechny bloky leží na adrese zarovnané na mocninu dvou, každý bit relativní adresy bloku vzhledem k začátku volné paměti představuje jednu úroveň hierarchie – má-li hodnotu 0, jde o první polovinu, má-li hodnotu 1, jde o druhou polovinu nadřazeného bloku. Přístup k blokům je tedy založen na jednoduchých výpočtech zvládnutelných pomocí bitových operací.

Tento přístup demonstruje následující obrázek.



Další variantou je **Fibonacciho přidělování**. Varianta Fibonacciho přidělování se snaží snížit vnitřní fragmentaci využitím kompaktnější sady možných velikostí bloků. Vzhledem k tomu, že každý prvek Fibonacciho řady je součtem dvou předcházejících prvků, lze blok vždy beze zbytku rozdělit na dva bloky, jejichž velikosti jsou opět prvky řady. Jistým problémem této metody je, že pokud přidělíme blok určité velikosti, má zbytek po dělení jinou velikost a pravděpodobně nebude příliš užitečný, pokud bude program požadovat přidělení většího počtu bloků téže velikosti.

4.5 Metody regenerace paměti

Regenerace paměti se používá v případech, kdy chceme již nepoužívané bloky paměti dát k dispozici pro další přidělování. V této kapitole se budeme zabývat metodami automatické regenerace paměti, kdy jsou nepoužívané bloky paměti vyhledány automaticky systémem správy paměti.

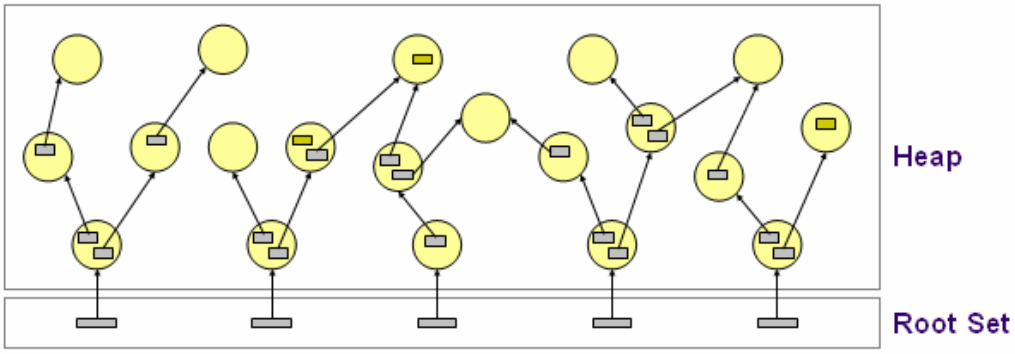
Pro zjištění toho, které úseky paměti se již nepoužívají, je k dispozici mnoho algoritmů. Většinou spoléhá automatická regenerace paměti na informace o tom, na které bloky paměti neukazuje žádná programová proměnná. V zásadě existují dvě skupiny metod – metody založené na sledování odkazů a metody založené na čítačích odkazů.

Mezi metodami regenerace paměti mají zvláštní význam metody inkrementální, při kterých probíhá regenerace paměti po částech a střídá se s prováděním vlastního programu. Je-li zaručen vhodný poměr mezi dobou věnovanou čistění paměti a činnosti aplikace (například pokud během každých dvou milisekund je nejvýše jedna věnována správě paměti a zbytek připadá na „užitečnou“ činnost, lze tyto inkrementální metody využít i pro systémy pracující v reálném čase, kdy musí být zajištěna rychlá odezva systému.

4.5.1 Dvoufázové značkování

Metoda dvoufázového značkování („mark and sweep“) vychází z toho, že jsme schopni najít všechny „kořenové ukazatele“ (globální proměnné, registry, lokální proměnné) a víme, kde jsou v datech umístěny další ukazatele. Rekurzivním průchodem jsme pak schopni označit postupně všechny bloky

paměti, dostupné z kořenových ukazatelů, a zbývající neoznačené bloky uvolnit a znovu využít pro další požadavky na přidělení paměti. Činnost demonstruje následující obrázek.



Tato metoda vyžaduje znalost struktury objektů za běhu aplikace. Uplatní se tedy nejlépe v interpretovaných jazycích, kde v době běhu máme tyto informace stále k dispozici. U ostatních jazyků pak vyžaduje jistou součinnost překladače, např. generování metadat o rozmístění ukazatelů do cílového programu.

4.5.2 Regenerace s kopírováním

Víme-li, které bloky paměti jsou dostupné a kde leží ukazatele na ně se odkazující, můžeme všechny obsazené bloky zkopírovat z jedné oblasti paměti do jiné souvislé oblasti a uvolněnou oblast využít znovu pro přidělování dalších bloků. Tímto postupem snížíme externí fragmentaci, neboť uvolněná paměť vytvoří souvislý celek. Při kopírování musíme odkazy na přesunuté objekty přesměrovat.

4.5.3 Inkrementální regenerace

Inkrementální regenerace paměti probíhá po částech a střídá se s prováděním samotné aplikace. Obvykle nedochází k současné modifikaci dat aplikací i správou paměti – pokud běží aplikace a správa paměti v různých vláknech, probíhají jako koprogramy. V případě souběžné regenerace může docházet k současnému čtení i modifikaci dat a je tedy nutná synchronizace.

4.5.4 Regenerace s počítáním odkazů

Pro každý přidělený blok paměti můžeme udržovat informaci o počtu odkazů, které na tento blok ukazují. Při každém kopírování odkazu typu L: = R se čítač odkazů bloku, na který ukazuje R, zvýší, zatímco čítač odkazů bloku, na který odkazoval původní ukazatel L, se sníží. Paměť pak může být uvolněna v případě, že se čítač odkazů sníží na nulu a na blok paměti tedy neukazuje žádný ukazatel.

Je-li blok z paměti uvolněn, sníží se čítače odkazů i u všech objektů, na které z uvolněného bloku vede nějaký odkaz. To může způsobit kaskádu dalších uvolňování objektů – například pokud se uvolní objekt, odkazující se jako jediný na složitou datovou strukturu, mohou se všechny čítače odkazů na objekty v této struktuře dostat na nulu a tyto objekty se pak mohou uvolnit. Toto tranzitivní uvolňování objektů může být časově značně náročné a může vést ke zhoršení odezvy programu. Je však možné požadavky na uvolňování řadit do fronty a provádět je pak souběžně s činností aplikace. To činí metodu

počítání odkazů zvlášť vhodnou pro aplikace pracující v reálném čase, kdy je třeba zajistit dostatečně rychlou odezvu systému.

Nevýhodou počítání odkazů je, že tuto metodu nelze použít v případě cyklických odkazů mezi objekty. Tehdy na sebe mohou bloky ukazovat navzájem a jejich čítač odkazů bude vždy nenulový, a to i v případě, že na tyto bloky již žádný další odkaz existovat nebude. Není tedy zaručena úplná regenerace nedostupné paměti. Problému cyklických odkazů je možné se vyhnout například tak, že tuto metodu omezíme pouze na struktury, u nichž k cyklickým odkazům dojít nemůže – např. na řetězce nebo jiná data neobsahující ukazatele. Pro zbývající struktury pak můžeme použít jinou metodu. Případně lze použít čítače odkazů na všechny struktury a jinou metodu využít až při nahromadění většího počtu neodstranitelných objektů s cyklickými odkazy.

Metoda regenerace s počítáním odkazů má velmi malou paměťovou režii, která spočívá v připojení čítače ke každému přidělenému bloku paměti. Velikost čítače určuje nejvyšší počet odkazů, které mohou na blok ukazovat; je-li tento počet překročen, je možné například ponechat hodnotu čítače stále na maximální hodnotě, což v důsledku vede k tomu, že tento blok již nelze regenerovat.

Větší režii však tato metoda přináší v okamžiku, kdy dochází k častému vzniku a zániku objektů nebo při častých přesunech odkazů. Je-li například odkaz na objekt předán jako parametr funkce, zvýší se při volání funkce čítač odkazů a krátce na to se při návratu čítač opět vrací na původní hodnotu. Ke vzniku a okamžitému zániku dočasných objektů také dochází při vyhodnocování některých typů výrazů. Tato režie se může redukovat tzv. odloženým počítáním odkazů, kdy se při operacích nad lokálními proměnnými čítače neaktualizují okamžitě, ale pouze v jistých časových intervalech.

4.5.5 Generační regenerace paměti

Většina objektů má krátkou životnost. Pokud tedy objekt v paměti zůstane nějaký čas je pravděpodobné, že v ní zůstane i nadále. Tato metoda tedy definuje generace objektů v paměti. Objekty rozděleny do několika oblastí dle jejich „stáří“ a pokud objekt „přežije“ regeneraci paměti je zvětšena hodnota určující jeho stáří. Takto jsou definovány generace. Nultá generace je ta, která ještě nepřežila žádnou regeneraci paměti. První je ta, která přežila jednu regeneraci paměti atd. K regeneraci paměti můžeme použít jiné metody, například dvoufázové značkování. Hlavní ideou této metody je, že objekty starších generací není nutné tak často testovat při regeneraci paměti.

4.6 Správa paměti v programovacích jazycích

V předchozích kapitolách jsme si představili jednotlivé metody přidělování a regenerace paměti. Nyní si ukážeme, jak jsou tyto metody standardně využity v některých programovacích jazycích na úrovni jazykových konstrukcí nebo knihovních funkcí.

4.6.1 Programovací jazyk C

Programovací jazyk C řeší uživatelské přidělování paměti pomocí standardních knihovních funkcí. K dispozici jsou funkce malloc() pro přidělení prostoru

určité velikosti a free() pro uvolnění přidělené paměti. Paměťový prostor přidělený funkcí malloc() není spojen s žádným datovým typem a není inicializovaný, vrácený ukazatel musí být přetypován na požadovaný typ a o jeho správnou inicializaci se musí postarat uživatel.

Následující funkce alokuje pole velikosti `size`, všechny prvky pole inicializuje na hodnotu `init` a vrátí ukazatel na začátek pole.

```
# include <stdlib.h>
int* allocIntArray(unsigned size, int init)
{
   int i;
   int* p = (int*)malloc(size * sizeof(int));
   for(i = 0; i < size; i++)
      p[i] = init;
   return p;
}
```

4.6.2 Programovací jazyk C++

V C++ je přidělování paměti již součástí syntaxe jazyka. Pro přidělení paměti je k dispozici operátor `new` a pro uvolnění operátor `delete`. Tyto operátory jsou vždy svázány s hodnotou určitého datového typu (standardní i uživatelské typy, pole) a přidělený paměťový prostor je vždy správně inicializován.

Následující funkce je totožná s ukázkou v jazyce C, alokuje pole velikosti `size`, všechny prvky pole inicializuje na hodnotu `init` a vrátí ukazatel na začátek pole.

```
int* allocIntArray(unsigned size, int init)
{
   int* p = new int[size];
   for(int i = 0; i < size; i++)
      p[i] = init;
   return p;
}
```

4.6.3 Správa paměti v jazyce Java

Java používá automatickou regeneraci paměti. Využívá *garbage collector*. To jak je automatická regenerace paměti realizovaná často záleží na výrobci. Nejrozšířenější distribuce od firem jako SUN či IBM používají metodu mark & sweep. Prázdné bloky jsou spojovány kopírováním a je využito mechanismu generací. Využívá také inkrementální regenerace. Garbage collector je realizován jako samostatné vlákno s nízkou prioritou.

Oproti C++ objekty nemají destruktor. Před uvolněním paměti volá metoda: `protected void finalize() throws Throwable`.Tato metoda je definovaná v třídě `Object`. Každý objekt ji tedy implementuje a nebo z této třídy zdědí. Metoda je volána při uvolňování objektu. To kdy je objekt uvolněn ovšem závisí na garbage collectoru. Můžeme přímo volat

garbage collector: `System.gc()`. V této chvíli jsou uvolněny všechny objekty a až potom pokračuje vykonávání programu. Pro vyhledání referencí se využívají metadata

4.6.4 Programovací jazyk C#

Správa paměti je podobná jako v Javě. Opět je využita automatická regenerace paměti. Používá algoritmus next fit, regeneraci s kopírováním, a také využívá generací. Udržuje pointer na další volné místo: `NextObjPointer`. Při regeneraci paměti jsou objekty na hromadě „setříděny“ dle jejich vzdálenosti od kořenů.
Udržuje tři generace.

- Vytvořené objekty
- Objekty, které přežily jeden průchod GC.
- Objekty, které přežily více průchodů GC.

Využívá inkrementální regenerace. Správa paměti je realizována pomocí dvou vláken běžících na pozadí. První používá metodu dvoufázového značkování a identifikuje „garbage“. Druhé volá `finalize` a uvolňuje paměť.

Také zde můžeme explicitně uvolnit paměť voláním `System.GC.Collect`. Pro jednotlivé instance nutno použít rozhraní `System.IDispose`.

Kontrolní otázky:

1. Jaký je rozdíl mezi živými a dostupnými bloky paměti?
2. Jaká je režie při regeneraci paměti s počítáním odkazů?
3. Pokud potřebuje získat paměť pro lokální proměnné je vhodná metoda přidělování paměti na zásobníku?
4. Existují destruktory v Javě?
5. Můžeme implementovat automatickou správu paměti v jazyce C (zamyslete se nad použitím poitrů)?

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy využívanými při správě paměti uvnitř jednotlivých procesů. Byly prezentovány různé metody pro přidělování paměti a různé přístupy při její automatické regeneraci. Také bylo stručně ukázáno, jak je správa paměti řešena v některých jazycích.

5 Programování aplikací s použitím vláken

V této kapitole se dozvíte:

- Základy programovaní s vláky v Javě.
- Jak lze implementovat aplikaci používající více vláken v Javě.
- Jak lze synchronizovat běh více vláken.
- Jak vypadá životní cyklus vlákna.

Po jejím prostudování byste měli být schopni:

- Naimplementovat aplikaci, která používá více vláken.
- Synchronizovat činnost více vláken.

Klíčová slova této kapitoly:

vlákno, konkurenční provádění, Thread, synchronizace, Monitor, …

Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Studium této kapitoly je jednoduché a popisným způsobem zde nastudujete jak naimplementovat aplikaci využívající vláken. Prakticky jsou pak tyto informace demonstrovány na příkladech.
Na studium této části si vyhraďte minimálně 2 hodiny.

Programy vyžadující interakci s uživatelem musejí reagovat na jeho aktivitu pokud možno okamžitě. Zároveň však musejí provádět výpočty nutné k co nejrychlejší prezentaci dat uživateli. Souběžné provádění více činností je zajišťováno prostředky operačního systému počítače, v mnoha aplikacích však s ním musí programátor počítat a zajistit, aby byly prostředky počítače efektivně využity.

V této kapitole si uvedeme základní pojmy a nástroje, které souvisejí se souběžným zpracováním, a ukážeme si, ve kterých situacích se tyto prostředky používají a jaké problémy jsou s nimi spojené.

5.1 Procesy a vlákna

Operační systémy používají pro oddělení různých běžících aplikací procesy. Vlákna jsou základní jednotkou, které operační systém přiděluje čas procesoru, přičemž v rámci jednoho procesu může běžet i více vláken. Každému vláknu přísluší vlastní priorita a řada systémových struktur, v nichž je uložen kontext výpočtu v době, kdy vlákno neběží. Tento kontext obsahuje veškeré informace, které jsou nutné pro obnovení výpočtu, včetně uloženého obsahu registrů, zásobníku.

Operační systémy s preemptivním multitaskingem vytvářejí dojem souběžného provádění více vláken ve více procesech. To je zajištěno rozdělením času procesoru mezi jednotlivá vlákna po malých časových intervalech. Pokud časový interval vyprší, je běžící vlákno pozastaveno, uloží se jeho kontext a obnoví se kontext dalšího vlákna ve frontě, jemuž je pak předáno řízení. Délka přiděleného časového intervalu závisí na konkrétním

algoritmu, jenž je v operačním systému implementován. Vzhledem k tomu, že tyto úseky jsou ale z pohledu uživatele velmi krátké, je výsledný dojem i na počítači s jediným procesorem takový, jako by pracovalo více vláken současně. V případě, že máme k dispozici více procesorů, jsou mezi ně vlákna přidělována ke zpracování a k současnému běhu pak skutečně dochází.

Souběžné zpracování více aplikací je již delší dobu standardní možností, kterou nám operační systémy nabízejí. Například i v operačním systému MS-DOS bylo možné těchto postupů v omezené míře využít pro tisk na pozadí běhu dalších aplikací. Ve víceuživatelských systémech pak zcela logicky očekáváme, že se čas procesoru bude nějakým spravedlivým způsobem rozdělovat mezi aplikace spuštěné jednotlivými uživateli tak, aby každý z nich měl pocit, že pracuje s počítačem sám (i když se mu ten počítač pak může jevit pomalejší, než kdyby na něm pracoval opravdu sám).

Pro další zjemnění souběžného zpracování na jednotlivá vlákna téže aplikace jsou však ještě další důvody. Některé aplikace mohou pracovat v roli serverů, které mohou současně obsluhovat více požadavků. Například WWW server dostává současně mnoho požadavků k zaslání prohlížených stránek a je rozumné, aby byl schopen během zpracování jednoho požadavku přijímat požadavky další, případně aby mohl zpracovávat i několik požadavků současně.

Dalším důvodem pro použití více vláken v jedné aplikaci je zajištění dostatečné interaktivity aplikací s grafickým uživatelským rozhraním, kdy po spuštění náročnější operace nemůžeme nechat aplikaci „zamrznout“ až do jejího ukončení. Je-li jedno vlákno zaměstnáno výpočtem, mohou další vlákna zajišťovat animaci dialogu ukazujícího, jak daleko výpočet pokročil, případně reagovat na další příkazy uživatele. Nebo v tabulkovém kalkulátoru se může během komunikace s uživatelem provádět současně přepočítávání obsahu buněk tak, aby vždy odrážely aktuální data vyplňovaná uživatelem.

- **Vlákno je samostatně plánovatelný tok řízení programu. Představuje tedy jistou posloupnost instrukcí, jejíž provádění může být přerušeno např. po vypršení určitého časového limitu nebo čekáním na nějakou událost. Po ukončení důvodu přerušení může vlákno dále pokračovat v činnosti.**
- **Proces je tvořen paměťovým prostorem a jedním nebo více vlákny. Tento paměťový prostor jednotlivá vlákna sdílejí. V rámci operačního systému pak může běžet více procesů, ovšem každý proces již má svůj vlastní paměťový prostor.**

5.1.1 Výhody a nevýhody práce s více vlákny

I když existují i další metody jak zajistit souběžné provádění více činností, například pomocí asynchronního programování s využitím služeb dalších počítačů, představuje použití více vláken nejvýkonnější dostupnou techniku pro zvýšení rychlosti odezvy aplikací ve vztahu k uživateli při zajištění současného zpracování potřebných dat téměř ve stejném čase.

Aplikace využívající více vláken jsou schopny bez další modifikace dramaticky zlepšit svou odezvu už jen tím, že je spustíme na počítači s více procesory. Souběžná vlákna lze použít typicky k řešení následujících úloh:

- Komunikace po síti s webovým serverem a databází.
- Provádění operací, které vyžaduje velký objem času.
- Rozlišení mezi úlohami s různou prioritou. Například vlákno s vysokou prioritou obsluhuje časově kritické úlohy, zatímco vlákno s nízkou prioritou provádí další činnosti.
- Zajištění rychlé odezvy uživatelského rozhraní se současným během úloh na pozadí.

Samotným zvyšováním počtu vláken však obvykle odpovídajícího zvýšení výkonu aplikace nedosáhneme. Naopak se doporučuje, abychom používali co nejméně vláken a tím omezili spotřebu systémových prostředků a nárůst režie. Použití vláken může také vést při nevhodném návrhu aplikace k nejrůznějším konfliktům při soutěžení o některé systémové prostředky. Typické problémy jsou následující:

- Pro ukládání kontextových informací se spotřebovává dodatečná paměť, a tedy celkový počet procesů a vláken, které mohou v systému současně existovat, je omezený.
- Obsluha velkého počtu vláken spotřebovává významnou část času procesoru. Existuje-li tedy příliš mnoho vláken, většina z nich příliš významně nepostupuje. Navíc pokud je většina vláken v jednom procesu, dostávají se vlákna jiných procesů na řadu méně často.
- Organizace programu s mnoha vlákny je složitá a může být zdrojem mnoha chyb. Zejména je obtížné zajistit jejich správnou synchronizaci.
- Rušení vláken vyžaduje dobrou znalost toho, co by se mohlo stát a jak vzniklé problémy řešit.

5.1.2 Synchronizace

Komunikace mezi vlákny v rámci jednoho procesu je jednodušší než komunikace mezi různými procesy, a to právě díky sdílené paměti, pomocí které mohou vlákna komunikovat. Na druhé straně je třeba zajistit, aby vlákna k této sdílené paměti přistupovala synchronizovaně, aby nedocházelo např. k přepisu jedné informace několika vlákny současně. Problematika synchronizace procesů a vláken se studuje zejména v oblasti operačních systémů, i když v současné době je aktuální i při programování uživatelských aplikací.

Pokud nezajistíme synchronizovaný přístup ke sdíleným zdrojům (v rámci téže aplikace nebo i mezi více aplikacemi současně), může to vést k situacím jako je uváznutí nebo časový souběh. Při uváznutí (deadlock) přestanou dvě vlákna reagovat, neboť na sebe vzájemně čekají. Časový souběh nastává tehdy, pokud může k nějaké zvláštní situaci dojít v závislosti na kritickém načasování dvou událostí.

Nutnost synchronizace demonstruje následující příklad.

Uvažujme situaci, kdy si majitel účtu ukládá v bance 500 Kč, zatímco jeho syn vybírá z účtu 9500 Kč platební kartou. To znamená, že bankovní server obdrží oba požadavky na data o účtu v témže čase. Je zřejmé, že by se měl stav účtu

snížit celkem o 9000 Kč. K tomu, aby obě transakce proběhly správně, je však nutné obě činnosti synchronizovat.

Předpokládejme, že žádný synchronizační mechanismus nevyužijeme. Pak vlákno realizující vložení na účet (označme ho jako vlákno A) nejprve zjistí stav účtu. Poté může nastat přepnutí kontextu a stav účtu si zjistí také vlákno B, které vypočte novou výši bankovního konta po provedení výběru a tu zapíše zpět. Vlákno A pak udělá totéž, ovšem bude zcela ignorovat předchozí činnost vlákna B a stav účtu přepíše svým výsledkem výpočtu. Banka tak bude ochuzena o celou vybranou částku. Pokud ovšem v podobné situaci zapíše svůj výsledek dříve vlákno B, přijde o vloženou částku pro změnu majitel účtu. Tento příklad ilustruje typický časový souběh dvou operací bez synchronizace.

Abychom se tomuto problému vyhnuli, musíme zajistit, aby v době, kdy jedno vlákno čte nebo zapisuje sdílená data, k nim nemohlo přistupovat žádné jiné vlákno.

K synchronizaci přístupu ke sdíleným zdrojům se používají synchronizační objekty. Synchronizační objekt dává svému vlastníkovi právo přístupu ke sdílenému zdroji. Vlákno tedy musí na obdržení synchronizačního objektu čekat a až poté teprve může ke sdílenému zdroji přistupovat. Po ukončení operace objekt uvolní a tím umožní jeho přidělení případnému dalšímu vláknům, které čeká ve frontě spojené se synchronizačním objektem.

Typické sdílené zdroje vyžadující synchronizaci souběžného přístupu lze rozdělit do následujících skupin:

- Systémové zdroje – např. komunikační porty.
- Zdroje sdílené více procesy – např. deskriptory souborů.
- Zdroje v rámci jedné aplikace, k nimž přistupuje více vláken – např. globální a statické proměnné, instanční proměnné v objektech apod.

5.2 Vlákna v jazyce Java

Každé vlákno v Javě je instancí třídy `java.lang.Thread`. Tato třída zajišťuje spuštění, zastavení a ukončení vlákna. Vlákno musí implementovat metodu run, která definuje činnost vlákna. Této metodě je předáno řízení po spuštění vlákna metodou `start`. Instrukce jsou prováděny sekvenčně podle jejich pořadí ve zdrojovém programu. Vlákna spolu mohou spolupracovat, ale jinak je jejich kód prováděn nezávisle na ostatních. Každé vlákno může přistupovat k datům programu. Musí dodržet standardní přístupová práva v Javě. Můžeme použít:

- *lokální proměnné* – jsou přístupné pouze uvnitř metody. Nejsou sdílené mezi vlákny. Spouští-li vlákna stejnou metodu dostanou vždy vlastní kopii lokálních proměnných.
- *instanční a třídní proměnné* – mohou být sdílené mezi vlákny.

Jakákoliv třída Javy může být použita jako „startovní bod“ nového vlákna. Musí:

- buď přímo implementovat rozhraní `java.lang.Runnable;`
- nebo musí rozšiřovat třídu `java.lang.Thread`.

V obou případech musí být požadovaná funkčnost vlákna implementována v metodě `run()`.

Následující příklad demonstruje první přístup.

```
class MyThread extends Thread {  // vlákno
   public void run() {
       System.out.println(" this thread is running ... ");
   }
} // end class MyThread
class ThreadExample {  // program, který vytvoří vlákno
   public static void main(String [] args  ) {
       MyThread t = new MyThread();
       //metoda start spusti předefinovanou metodu run
        t.start();
   }
}
```

Nutnost dědit z třídy Thread může být omezující. V Javě můžeme dědit pouze z jedné třídy. Implementovat můžeme ale více rozhraní. K vytvoření vlákna můžeme použít rozhraní Runnable.

```
class MyThread implements Runnable  {
   public void run() {
       System.out.println(" this thread is running ... ");
   }
}
class ThreadEx2 {
   public static void main(String [] args  ) {
       Thread t = new Thread(new MyThread());
       t.start();
   }
}
```

Vytvoření vlákna by se dalo shrnout v následujících bodech. Po zavolání metody `start` (se ať už použijeme jakýkoliv přístup) provede:

1. Volání metody `start` vytvoří nový souběžný tok řízení v běhovém systému Javy.
2. Volání metody `start` vrátí řízení ihned po vytvoření nového toku řízení.
3. V novém toku řízení je vykonávána metoda `run`.
4. Po skončení vykonávání metody `run` je ukončeno i nový tok řízení.

Vlákno v průběhu svého života prochází posloupností následujících stavů. Vlákno se může nacházet v těchto hlavních stavech: I*nitial, Runnable, Not Runnable a Dead.*

- New – bezprostředně po vytvoření ještě nejsou vláknu přiděleny žádné systémové prostředky, vlákno neběží.

- Runnable – po provedené metody `start` je vlákno připraveno k běhu. V tomto stavu se může nacházet více vláken, ovšem jen jedno z nich (na počítači s jedním procesorem) je ve stavu „běžící“.
- Not runnable – do tohoto stavu se vlákno dostane, je-li pozastaveno voláním jedné s metod `sleep`, `wait` nebo `suspend`, případně čekáním na dokončení operace vstupu/výstupu.
- Dead – do tohoto stavu se vlákno dostane ukončením metody `run` nebo voláním metody `stop`.

Následující metody třídy `Thread` ovlivňují životní cyklus vlákna

- `public void start()`
- `public void run()`
- `public static void sleep(long milisekund)` – volání této metody uspí provádění vlákna na přesně specifikovanou dobu.
- `public void join()` – voláním této metody můžeme slučovat vlákna. Metoda je volána na nějakou instanci vlákna a pozdrží provádění vlákna, ze kterého je volána až do doby, kdy druhé vlákno skončí.
- `public void interrupt ()` – každé vlákno může být přerušeno.
- `public static void yield()` – v případě konkurenčního provádění několika vláken se tyto vlákna střídají a *konkurují* si při snaze získat čas procesoru. Každé vlákno má nastavenu prioritu. Vlákna s vyšší prioritou častěji získají přístup k procesoru. Voláním metody `yield` vlákno předá řízení a dá možnost ostatním vláknům získat čas procesoru.

Stav vlákna jsme schopni zjistit voláním metod `public Thread.state getState()`, `public boolean isAlive()`, `public boolean isInterrupted()`.

Každé vlákno obsahuje tyto informace:

- jméno vlákna – standardně jsou vlákna pojmenovaná jako: „Thread-0“, „Thread-1“, … Jméno lze nastavit voláním metody `setName`, nebo nastavit přímo parametrem konstruktoru třídy `Thread`.
- ID - jde o proměnnou typu `long`. Je vygenerovaná a jednoznačně identifikuje vlákno. Lze získat pomocí metody `geoid`.

O spuštěných vláknech můžeme získat celou řádu dalších informací. Můžeme využít například metod třídy `Thread`. Běhové prostředí Javy rozlišuje dva typy vláken: obyčejná vlákna a démoni Oba typy vlákna jsou si podobná. Rozdíl je, že pokud zůstanou jen démoni, je běh virtuálního stroje ukončen. Typický příklad démona je *garbage collector*. Tuto vlastnost vlákna můžeme nastavit voláním metody `setDeamon(boolean on)`.

5.2.1 Synchronizace vláken v Javě

V Javě máme možnost spustit několik vláken. Jak již bylo uvedeno, muže být nutné synchronizovat běh více vláken. Sekce kódu, které nesmějí být vykonávány paralelně, se nazývají kritické sekce. Ty to kritické sekce například modifikují stejná data. V Javě je kritická sekce svázaná s určitým

objektem nebo polem. Provedení kritické sekce pak je možné pouze v případě, že vlákno získá exkluzivní zámek na tento objekt. V Javě je kritická sekce realizovaná příkazem `synchronized` a může být definovaná následující konstrukcí:

- `synchronized (expresion) { … }`

Kde výraz musí být odkaz na objekt (nebo pole). V těle pak následuje kritická sekce. Klíčové slovo `synchronized` lze použít i jako modifikátor u metody. Jako zámek je pak použit objekt, o jehož metodu se jedná. Pro statické metody použije Java zámek na danou třídu.

S každým objektem je asociován monitor. Ten slouží pro synchronizaci jako zámek na daný objekt. Pokud vlákno získá monitor na nějaký objekt, žádné jiné vlákno nemůže získat přístup k tomuto objektu. Díky tomu může kritickou sekci provádět maximálně právě jedno vlákno. Ostatní vlákna jsou pozdržena až do doby, dokud toto vlákno monitor nevrátí. Abychom předešli problémům, které souhrnně označujeme jako „*deadlock*“, Java umožňuje vícenásobný přístup (reentrant) k monitoru – umožňuje vláknu „získat“ monitor, který už má. Pokud o něj znovu požádá.
Další možnosti synchronizace realizují metody třídy `Object`.

- `wait()` - zajistí, že vlákno počká, než nastanou určité podmínky.
- `notify()` – probudí jedno z čekajících vláken. Je volána v případě že došlo ke změně sledovaných podmínek.
- `notifyAll()` – probudí všechny čekající vlákna

5.2.2 Příklad aplikace Producer – Customer

Následující příklad demonstruje programování aplikace s použitím vláken. Také ukazuje, jak lze synchronizovat běh více vláken. Jde o jednoduchou aplikaci, kdy instance třídy Producer „vyrábí“ nějaké „zboží“. To je pak konzumováni instancemi třídy Customer. Výrobce i tito spotřebitelé jsou realizování jako samostatná vlákna. Pro synchronizaci je použit společný „Pool“.

```
class Producer extends Thread{
  private Pool pool;
  public Producer(Pool pool) {
    this.pool=pool;
  }
  public void run() {
    for(int i=0;i<10;i++) {
      System.out.println("Produced item: "+i);
      pool.putItem(i);
      try{
      ´ Thread.sleep(new java.util.Random().nextInt(1000));
      }catch (InterruptedException e) {}
    }
  }
}
```

```
class Customer extends Thread{
  private Pool pool;
  private String name;
  public Customer(Pool pool,String name) {
    this.pool=pool;
    this.name=name;
  }
  public void run() {
    for(int i=0;i<5;i++) {
      int tmp=pool.getItem();
      System.out.println(name+": Consumed item: "+tmp);
    }
  }
}
```

U implementace třídy Pool jsme využili synchronizační možnosti Javy. Všimněte si, že metody jsou synchronizované. Tedy žádná dvě vlákna k instancím této třídy nemůžou přistupovat najednou. Pokud je „pool“ plný nelze do něj nic vložit a vlákno, které se o to pokusí, je uspáno. Pokračovat může až ve chvíli, kdy se situace změní. To je řešeno metodami třídy Object. Obdobně je řešena i vybírání položek.

```
class Pool {
    private int item;
    private boolean full = false;

    public synchronized void putItem(int item) {
        while(full) {
            try{
                wait();
            }catch(InterruptedException e){ }
        }
        this.item=item;
        full=true;
        notifyAll();
    }
    public synchronized int getItem() {
        while(!full) {
            try{
                wait();
            }catch(InterruptedException e) {}
        }
        int tmp= this.item;
        this.full=false;
```

```
        notifyAll();
        return tmp;
    }
}
```

Nyní můžeme implementaci vyzkoušet. Spuštění by mohlo vypadat například takto.

```
public static void main(String[] args) {
        Pool pool = new Pool();
        Producer producer=new Producer(pool);
        Customer consumer1=new Customer(pool,"A");
        Customer consumer2=new Customer(pool,"B");
        consumer1.start();
        consumer2.start();
        producer.start();
}
```

Výstup není jednoznačně určen. Závisí na tom, které vlákno získá čas procesoru a tak se může pořadí, v jakém instance třídy Customer vyberou položky „produkované“ instancí třídy Producer měnit. Výstup může vypadat například takto.

```
Produced item: 0
A: Consumed item: 0
Produced item: 1
B: Consumed item: 1
Produced item: 2
A: Consumed item: 2
Produced item: 3
B: Consumed item: 3
Produced item: 4
A: Consumed item: 4
Produced item: 5
A: Consumed item: 5
Produced item: 6
B: Consumed item: 6
Produced item: 7
A: Consumed item: 7
Produced item: 8
B: Consumed item: 8
Produced item: 9
B: Consumed item: 9
```

5.2.3 Závěr

Prezentovány byly pouze základy, jak implementovat více vláknové aplikace. Existuje celá řada dalších funkcí a možností, které zde nebyly prezentovány. Navíc se tato část JDK neustále aktivně vyvíjí a je to jedna z částí, které se poměrně hodně mění. Použití vláken v různých verzích JDK může být odlišné. V aktuální verzi 1.5 byla přidána celá řada nových tříd. Ty jsou například v balících: java.concurrent; java.concurrent.atomic; java.concurrent.locks;

Kontrolní otázky:

1. Jak je vykonávaná aplikace obsahující více vláken na počítači, který má pouze jediný procesor?
2. Co je to monitor v Javě?
3. Co je to kritická sekce?
4. Jak vznikne „deadlock“?
5. Můžou vlákna používat jednu společnou proměnnou?

Úkoly k zamyšlení:

1. Zamyslete se nad možnostmi využití aplikací s více vlákny. Například se zamyslete, jak byste mohli využít tuto technologii při návrhu GUI.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základy programování aplikací s využitím vláken. Bylo demonstrováno, jak lze takovéto aplikace vytvářet v programovacím jazyce Java.

6 Komprese dat

V této kapitole se dozvíte:

- Proč používáme kompresi či kódování dat.
- Jaké typy komprese používáme.
- Jaké informace odstraníme při kompresi.
- Jaké základní algoritmy se pro kompresi používají.
- Jak můžeme realizovat kompresi dat v Javě.

Po jejím prostudování byste měli být schopni:

- Rozumět podstatě kódování a komprese.
- Implementovat některé základní algoritmy kro kompresi dat.
- Rozumět jakým způsobem se ukládá hudby, obrázky a video.
- Vytvořit v Javě ZIP archív.

Klíčová slova této kapitoly:

informace, entropie, kódování, komprese, Hufmannovo kódování, LZ77, LZ78, LZW, GIF, JPEG, MP3, JPEG, ZIP,…

Doba potřebná ke studiu: 2 hodin

Průvodce studiem

Studium této kapitoly je jednoduché a popisným způsobem zde nastudujete základní informace z oblasti teorie kódování.
Na studium této části si vyhraďte minimálně 2 hodiny. Po celkovém prostudování doporučuji získané informace prakticky vyzkoušet a naimplementovat některý z prezentovaných algoritmů.

Pracujeme-li s velkými objemy dat, brzy narazíme na potřebu tato data někam přenést nebo je někde archivovat. Při přenosu dat jsme omezeni šířkou přenosového pásma, a tedy i objemem dat, který jsme schopni za časovou jednotku přenést. To může vést ke značnému zpomalení přenosu a v některých případech i zcela znemožnit zamýšlené využití přenášených dat, například pokud chceme přenášet živé vysílání hudby nebo videa, případně zprostředkovat současné telefonické spojení velkého počtu účastníků. Při ukládání dat zase může jejich objem převyšovat kapacitu prostoru, který máme k dispozici.

Jednou z možností, jak uvedené situace řešit, je použití metod komprese dat. To znamená, že se pokusíme konkrétní data (např. textový soubor, obrázek, zvukový záznam nebo video) reprezentovat na menším prostoru než kolik zabírají v původní podobě. Tato data často obsahují mnohem menší množství informace, než odpovídá velikosti zabraného prostoru – například často se vyskytující slova v textovém souboru nebo opakující se sekvence v hudbě. Případně můžeme využít některých nedokonalostí lidského vnímání a odstranit z dat to, co člověk není schopen svými smysly zachytit (např. vysoké frekvence zvuku, rychlé změny v obrazu nebo podobné odstíny barev) nebo co neovlivní výsledný dojem natolik, aby odchylky od původního stavu nebylo možné tolerovat.

Metody komprese dat jsou charakterizované svou rychlostí (komprese i dekomprese dat), kvalitou a efektivitou. Kvalita komprese je dána tím, jak moc se liší rekonstruovaná data od původních, počínaje úplnou rekonstrukcí (bezeztrátová komprese), až po omezení určitých vlastností u ztrátové komprese, např. omezení přenášeného frekvenčního pásma (zvuk), počtu zobrazovaných barev (obraz, video) nebo počtu rozlišitelných prvků informace (obraz, video, dynamika zvuku).

V dalších kapitolách tohoto modulu se budeme věnovat metodám používaným pro kompresi jednotlivých druhů informace. Ukážeme si obecné metody používané v běžných archivačních programech i metody, se kterými se můžeme setkat v oblasti komprese zvuku, obrazu či videa. Vybrané jednodušší algoritmy si vyzkoušíme implementovat přímo v některém programovacím jazyce, u těch složitějších se seznámíme alespoň s jejich principem.

6.1 Základy teorie informace a kódování

Začneme definicí několika základních pojmů a několika základních poznatků teorie informace a kódování. První pojem, který si zde představíme, bude zpráva. Zpráva je souhrnem nějakého množství informace. Tyto informace bychom si mohli rozdělit na relevantní a irelevantní. To co jsou relevantní a irelevantní informace závisí na obsahu zprávy a na informacích, které obsahuje. Další dělení by mohlo být na entropickou část zprávy a redundantní část zprávy. Entropická část nese informaci. Redundantní část „opakuje“ některé již sdělené informace. Pokud odstraníme redundantní část zpráv, jsme schopní původní zprávu rekonstruovat z entropické části.

Pod pojmem komprese zprávy rozumíme její zestručnění, které zachovává informační obsah zprávy. Přitom nás nejvíce zajímá relevantní a entropická část. Podstatou komprese je že použitím nějakého algoritmu zestručníme zprávu. Typicky při tom odstraníme redundantní část zprávy. Tu jsem schopni rekonstruovat. Další možností je vypustit i část irelevantních informací. Jejich vypuštění nezmění informační hodnotu zprávy, ale nevratně změní zprávu. Nejsme už nadále schopni původní zprávu obnovit. Vypuštění irelevantních informací může být ovšem jedinou možností, jak zprávu zestručnit. Důležitou roli zde hraje entropie zprávy. Tato veličina definuje jakousi mezní hodnotu, za kterou se už nemůžeme při zestručňování zprávy dostat. Vypuštění irelevantních informací zmenší objem informací ve zprávě, ale ze své podstaty nezmění její celkovou informační hodnotu.

Typickým příkladem metody, která vypouští irelevantní informace je formát MP3. Ten vychází z omezení lidského ucha a vypouští ty části zvuku, které by lidské ucho stejně nebylo schopno slyšet. Díky tomu dosahuje mnohem lepších kompresních poměrů než algoritmy, které tyto informace zachovávají.

Při kompresi se snažíme maximálně zestručnit zprávu a dosáhnout maximálního poměru mezi informačním obsahem a délkou zprávy (mohou existovat i další omezení pro kompresi, například dalším omezujícím kritériem by mohla být rychlost).

Vedlejším efektem komprese zprávy je její menší odolnost vůči chybám. Pokud například ze správy odstraníme redundantní informace, zvětšíme riziko chyby. Proto se na jedné straně snažíme zprávu co nejvíce komprimovat, na straně druhé pak k ní přidáváme redundantní informace, které jí zabezpečí vůči

chybám. Zde bychom mohli efektivně použít bezpečnostní kódování, které ke zprávě přidává redundantní informace, které zajistí její odolnost vůči chybám. Existují například Hammingový kód. U těchto kódu jsme schopni přesně určit, kolik chybných bitů ve zprávě jsme schopni detekovat či opravit.

6.2 Komprese textu

Algoritmy pro kompresi textových dat patří k nejdéle používaným kompresním algoritmům. Začaly se objevovat v šedesátých letech minulého století a byly původně určeny zejména pro zvyšování kapacity paměťových médií. Později se začaly využívat také v oblasti komunikací. Komprese dat také umožnila zvýšit spolehlivost datových přenosů, u kterých může být pravděpodobnost vzniku chyby úměrná délce přenášeného bloku. Některé zde prezentované algoritmy se úspěšně používají i u jiných než textových dat.

6.2.1 Kódování

Pokud chceme reprezentovat nějaký text, s velkou pravděpodobností začneme nějakým kódováním vstupní abecedy. Jistě znáte celou řadu příkladu různých kódu pro reprezentaci textu a to nejen z informatiky. Asi nejznámějším příkladem je Morseova abeceda. Zde byla každému symbolu přiřazena sekvence teček a čárek. V dnešních počítačích je ovšem tento způsob kódování nevhodný. Dnešní počítače používají binární soustavu, a i když se Morseova abeceda tváří jako binární, k reprezentaci informace potřebujeme tři znaky. Kromě tečky a čárky ještě potřebujeme nějaký oddělovač, který by rozděloval jednotlivá písmena.

Kódování bychom mohli rozdělit na dvě skupiny.

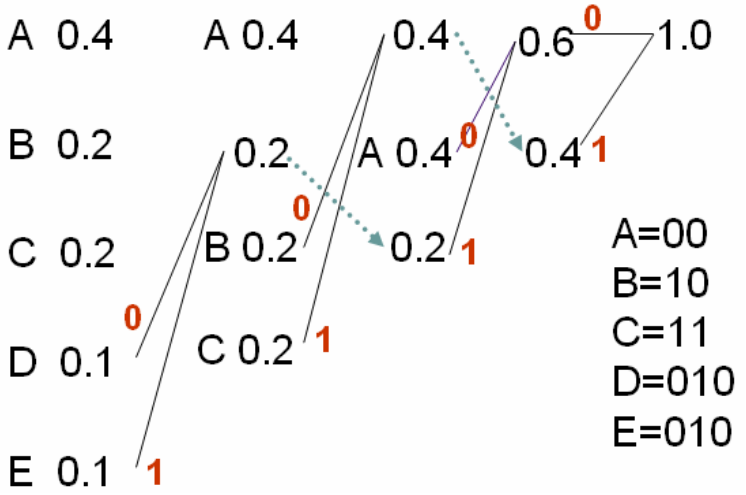
- Rovnoměrné kódy – každému symbolu vstupní abecedy je přiřazena stejně dlouhá sekvence nul a jedniček. Každý symbol je zakódován jako n bitové slovo. Příkladem takovéhoto kódu je například ASCI. Zde je každý znak kódován jako osmi botové slovo. Rovnoměrné kódy se dají velice jednoduše realizovat. Proces kódování a dekódování je velice jednoduchý a rychlý. Nevýhodou pak je, že zakódovaná informace v mnoha případech může být značně zestručněna.
- Nerovnoměrné kódy – symboly zdroje jsou kódovány symboly binární abecedy. Přitom jsou jednotlivé symboly kódovány nestejně dlouhými řetězci. Takto můžeme využít například toho, že známe pravděpodobnost výskytu jednotlivých symbolů. Znaky, které se vyskytují častěji, jsou kódovány kratší sekvencí. Příkladem (a nejčastěji používaným) nerovnoměrnými kódy jsou prefixové kódy. Základní myšlenkou těchto kódů je, že žádné slovo není prefixem jiného slova. Takto můžeme číst vstup bit po bitu a jednoznačně identifikovat zakódované znaky na vstupu.

6.2.2 Huffmanovo kódování

Nejznámějším příkladem prefixového kódu je Huffmanovo kódování. Tato metoda je pojmenována podle svého objevitele, D. A. Huffnana. Je založena na znalosti pravděpodobnosti výskytu jednotlivých znaků. Princip metody spočívá ve vytvoření binárního stromu, jehož koncové uzly odpovídají symbolům původní abecedy, hrany jsou ohodnoceny symboly 0 a 1 a uzly jsou

ohodnoceny pravděpodobností výskytu. Pravděpodobnost vnitřního uzlu je přitom rovna součtu pravděpodobností jeho následníků. Uzly řadíme do posloupnosti podle rostoucí pravděpodobnosti, v každém kroku z ní odstraníme dva uzly s nejnižší prioritou, vytvoříme z nich následníky nového uzlu a ten opět zařadíme do seznamu.

Proces kódování krok po kroku znázorňuje následující obrázek.



Algoritmus bychom mohli popsat takto:

1. Do fronty si připravíme dle pravděpodobnosti výskytu setříděné listy budoucího stromu (obsahují kódovaný symbol a pravděpodobnost jeho výskytu).
2. Dokud je ve frontě víc než jeden uzel stromu opakuj:
   a. Vyjmi dva uzly s nejmenší pravděpodobností.
   b. Vytvoř nový uzel, který bude mít tyto dva jako potomky a jehož pravděpodobnost bude součet jejich pravděpodobností.
   c. Zatřiď uzel do fronty.
3. Zbylý uzel je kořen vytvořeného stromu. Projdi strom od kořene k listům a přiřaď vždy levé a právě větvi každého uzlu jedničku respektive nulu.

Kód určitého symbolu získáme tak, že projdeme stromem k listu, který obsahuje tento symbol a vypíšeme přiřazené jedničky a nuly.

Huffmanův kód má dvě důležité vlastnosti. Jednak je kódem s minimální délkou, jednak je to prefixový kód a je tedy jednoznačně dekódovatelný. Jeho problémem je to, že musíme znát rozdělení pravděpodobnosti výskytu jednotlivých symbolů. To lze nahradit odhadem, případně je možné tento odhad v průběhu komprese upřesňovat.

Použití Huffmanova kódu je časté v kombinaci s jinými kompresními algoritmy, například při kompresi obrazu a videa ve standardech JPEG a MPEG. Samostatně se s ním můžeme setkat v programu `compress` pod OS Unix.

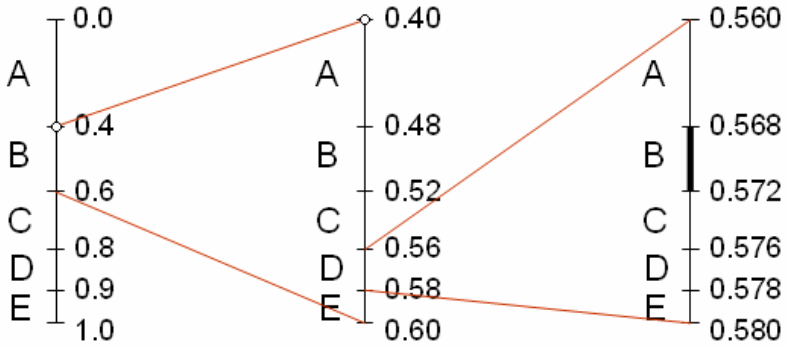
6.2.3 Aritmetické kódování

Dalším používaným kódováním je aritmetické. Aritmetické kódování na rozdíl od jiných metod nepracuje na principu nahrazování vstupního znaku specifickým kódem. Místo toho se kódovaný vstupní proud znaků nahradí jediným reálným číslem z intervalu <0,1).

Na základě pravděpodobnosti výskytu jednotlivých symbolů vstupní abecedy je každému symbolu přiřazena odpovídající poměrná část intervalu <0,1). Při

kódování je pak celý interval <0,1) postupně omezován z obou stran na základě postupně přicházejících symbolů. Každý symbol vybere z aktuálního intervalu odpovídající poměrnou část a ta se stane novým základem pro následující symbol. Kódovaná hodnota se reprezentuje libovolným reálným číslem, které leží ve výsledném intervalu získaném po přečtení všech vstupních symbolů. Vzhledem k tomu, že z takto reprezentované hodnoty nelze při dekódování určit konec zprávy, je třeba navíc ke zprávě přidat speciální znak označující konec, případně musí být uložena i délka původní posloupnosti.

Činnost demonstruje následující obrázek.

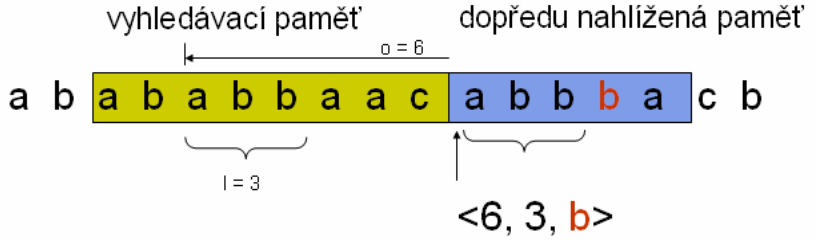


6.2.4 Slovníkové metody

Základním principem slovníkových metod komprese je vyhledávání shodných, opakujících se řetězců ve vstupním souboru. Těmto řetězcům jsou pak přidělovány kódy, které se ukládají (často s využitím dalších kompresních metod) na výstup. Dva nejstarší algoritmy implementující slovníkové metody jsou LZ77 a LZ78.

Algoritmus LZ77 kóduje vstup jako posloupnost trojic. Základní myšlenkou je, že pokud kóduje vstup od nějakého místa, pak můžeme využít předcházející vstup. Pokud najdeme nějakou shodnou sekvenci, je dán pouze „odkaz“ na tuto sekvenci.

Činnost demonstruje následující obrázek. Posloupnost rozdělíme na dvě poloviny. Posloupnost, kterou jsme již zakódovali a posloupnost, kterou ještě chceme zakódovat. Poslední kus zakódované sekvence bereme jako vyhledávací paměť a snažíme se najít co nejdelší shodu mezi řetězcem z vyhledávací paměti a dopředu nahlíženým řetězcem. Pokud takovou najdeme, vygenerujeme trojici obsahující posunutí v rámci vyhledávácí paměti, velikost společného řetězce a následující symbol. Následující symbol je tam proto, abychom ošetřili případ, kdy nenajdeme žádnou shodu.



Jedná se o bezeztrátovou komprimační metodu vhodnou pro kompresi textových i binárních souborů. Existují různé modifikace této metody. Například LZ78 a nebo LZW. Tyto algoritmy se používají v běžných komprimačních programech jako WinZIP ve Windows nebo compress v

Unixu, ale také ve formátech určených pro počítačovou grafiku jako GIF, TIFF nebo PostScript.

LZ78 realizuje trochu jiný přístup. Zde už opravdu tvoříme nějaký slovník. V předchozím případě byl „skrytý“ při prohledávání vyrovnávací paměti. Tato metoda vstup (posloupnost symbolu) transformuje na posloupnost dvojic. V těchto dvojicích je uložen odkaz do slovníku a následující symbol. Tento algoritmus také pracuje se slovníkem, který se adaptivně přizpůsobuje kódovaným datům. Během komprese se dynamicky vytváří slovník, který lze při dekompresi na základě přijímaných dat obnovit a není tedy třeba jej ke komprimovaným datům přidávat. Algoritmus pro zakódování bychom mohli popsat takto.

1. Začneme s prázdným slovníkem.
2. Začne s prázdným prefixem a prvním znakem (dále *znak*) nezakódované zprávy.
3. Pokusí se najít ve slovníku slovo = prefix + *znak*
4. Pokud najde, přidá do prefixu *znak* ze zprávy a jako nový znak vezme další symbol. Opakuje krok 3.
5. V případě že prefix + *znak* nejsou ne slovníku, vygeneruje výstupní dvojici. Prefix je slovo ve slovníku a jeho pozice bude první prvek dvojice. Druhým je pak *znak*. Posune se na další prvek vstupu za *znak* a přidá slovo do slovníku slovo vzniklé spojením prefixu a *znaku*.

Činnost algoritmu demonstruje následující příklad. Na vstupu máme sekvenci

- A B B C B C A B A

Pokud bychom aplikovali předcházející algoritmus, tak provedeme následující kroky:

1. krok, do slovníku přidáme A a vygenerujeme dvojici (0, A)
2. do slovníku přidáme B a vygenerujeme dvojici (0, B)
3. B C , (2, C)
4. B C A, (3, A)
5. B A, (2, A)

Výstupem je tedy sekvence pěti dvojic. Pro dekódování můžeme použít tento postup:

1. Začne s prázdným slovníkem.
2. Dokud jsou další dvojice na vstupu, opakuje:
    a. Vytiskne řetězec ze slovníku specifikovaný jako první element (může být prázdný)
    b. Vytiskne znak specifikovaný jako druhý element.
    c. Přidá „vytištěnou“ sekvenci na konec slovníku.

Takto dostaneme původní nezakódovanou sekvenci.

Další modifikace těchto algoritmů je algoritmus označovaný jako LZW (je pojmenovaný podle svých objevitelů A. Lempela, J. Ziva a Terryho A. Welche, ten modifikoval původní algoritmus LZ77). Tento algoritmus také pracuje se slovníkem, který se adaptivně přizpůsobuje kódovaným datům. Během komprese se dynamicky vytváří slovník, který lze při dekompresi na základě přijímaných dat obnovit a není tedy třeba jej ke komprimovaným datům přidávat. Výhody algoritmu LZW jsou podobné jako u předchozích. Je to zejména jednoduchá implementace, rychlá komprese a dekomprese (vhodná zejména pro časově kritické aplikace), malé nároky na paměť a možnost kontinuálního vysílání zkomprimovaných dat bez nutnosti čekání na dokončení komprese datového bloku – např. při posílání modemem. Hlavním rozdílem

v činnosti je, že výsledkem není seznam dvojic, ale jen seznam odkazů do slovníku. Důsledkem je, že příjemce musí znát všechny možné symboly na vstupu. Slovník pak na začátku obsahuje jednoznaková slova, která korespondují se všemi možnými symboly na vstupu.

Jinou variantou je metoda LZ77-deflate, implementovaná v programech zip, gzip, pkzip apod. Tato metoda kombinuje slovníkový algoritmus LZ77 s Huffmanovým kódováním. Zdrojový soubor je komprimován po blocích, přičemž opakující se řetězce jsou charakterizovány pozicí svého prvního výskytu v bloku a délkou. Pozice i délka mohou být kódovány Huffmanovou metodou, přičemž lze využít fixních kódovacích tabulek definovaných ve specifikaci metody a není třeba je vkládat do bloku komprimovaných dat pro potřebu dekomprese.

6.2.5 Prediktivní metody

Předchozí metody vycházejí z posloupnosti nezávislých symbolů. Často jsou symboly závislé. V určitém kontextu se vyskytují více nebo méně často. Vezměme v úvahu následující příklad: „V Á N O C“, je velice pravděpodobné, že následující symbol bude „E“. Na druhou stranu může být tato velice obtížné určit, že právě tento symbol je pravděpodobnější než jiné. Existují ovšem i jiné, mnohem lépe zpracovatelné kontextové vazby. Například ve faxové zprávě, po černém nebo bílém bodu lze očekávat opět bod stejné barvy. Totéž platí i u obrázku (po bodu nějaké barvy následuje pravděpodobně bod stejné barvy). Jedním ze základních algoritmů, který bychom v takovém případě mohli efektivně použít je algoritmus Run Length Encoding.

Tato metoda se snaží v datovém toku objevit a redukovat posloupnosti opakujících se znaků. Místo této posloupnosti je uložen pouze speciální znak představující indikátor, opakovaný znak a počet opakování. Například posloupnost xyzzzzyyyyyyxwwww můžeme uložit jako xy#z4#y5x#w4, kde znak '#' představuje indikátor komprese. V tomto případě jsme tedy řetězec délky 16 znaků nahradili komprimovaným řetězcem délky 12 znaků.

Úspěšnost komprese můžeme popsat parametrem faktor komprese jako 12/16=0,75, jenž udává, jakou část původního prostoru zabírají údaje po kompresi. V tomto případě je faktor komprese 75 %. Jinou možností je jeho převrácená hodnota – **kompresní poměr**, který v tomto případě činí 16/12=1,33. Čím větší je kompresní poměr, tím úspěšnější je komprese.

Je třeba si uvědomit, že každou posloupnost opakujících se znaků zakódujeme do tří znaků (indikátor, opakující se znak a počet opakování). Je-li tedy tato posloupnost kratší než tří znaky, spotřebujeme na její zakódování více prostoru, než kolik zabrala původně. Při právě třech znacích ke zkrácení ani prodloužení nedojde, ovšem ztratíme čas, nutný k dekompresi textu. Proto se tato komprese používá až od jisté minimální délky posloupnosti, v tomto případě například až od délky čtyři.

Dalším problém nastává v situaci, kdy délka posloupnosti opakujících se znaků je větší než číslo, kterým jsme schopni tuto délku reprezentovat. Například pokud ukládáme délku posloupnosti do jedné slabiky, můžeme uložit pouze hodnoty 0 až 255. Vzhledem k tomu, že komprimujeme pouze posloupnosti délky alespoň čtyři znaky, jsme schopni ukládat hodnotu délky v kódu s posunutou nulou (číslo 0 znamená délku 4, číslo 255 délku 259 znaků). Je-li posloupnost delší než 259 znaků, můžeme ji však rozdělit na kratší úseky a ty pak kódovat samostatně za cenu jisté ztráty kompresního poměru. Případně

můžeme jiným indikátorem specifikovat, že délka posloupnosti bude uložena např. na dvou slabikách.

Pokud víme, že komprimujeme pouze textové soubory obsahující tiskutelné znaky, mezery, tabulátory a konce řádků, stačí zvolit jako indikátor znak, jenž se v textu nemůže vyskytnout. Připustíme-li však možnost výskytu znaků s libovolným kódem, je třeba zavést další speciální znak, jenž ruší speciální význam následujícího znaku. Příklady tohoto řešení najdete ve většině programovacích jazyků u speciálních znaků v řetězcových konstantách. V jazyce C je zpětné lomítko v řetězcové konstantě považováno právě za takový únikový znak (escape character), definující nebo měnící speciální význam následujícího znaku nebo posloupnosti znaků. Při použití zpětného lomítka jako únikového znaku bychom tedy posloupnost a#cccc\d zakódovali jako a\##c4\\d. Je zřejmé, že výskyt speciálních znaků v komprimovaném textu opět vede ke zhoršení efektivity komprese.

Jinou variantou je do výstupu vždy po výskytu tří stejných znaků vložit počet dalších opakování. Potom nemusíme rezervovat žádné speciální znaky, neboť začátek komprimovaného úseku je indikován výskytem tří stejných znaků po sobě. Například řetězec aabbbccccddddddse zakóduje jako aabbb0ccc1ddd2. Tato varianta metody RLE se používá v kombinaci s dalšími v protokolech určených pro modemy.

Metoda RLE je bezeztrátová, symetrická a velmi rychlá, ovšem často za cenu nižšího kompresního poměru. Vzhledem k tomu, že v běžných textových souborech se posloupnosti opakujících se znaků příliš nevyskytují (snad s výjimkou mezer), je tato metoda efektivní spíše pro kompresi grafických a zvukových souborů, kde nedochází k velkým změnám, například pro kompresi obrázků s malou hloubkou barev nebo u zvukových souborů zachycujících řeč. Konkrétně se tato metoda používá například v grafických formátech PCX nebo BMP.

6.3 Komprese zvukových souborů

V této kapitole popíšeme metody komprese zvukových souborů, které hrají důležitou úlohu v současných multimediálních aplikacích. Uplatňuje se zejména v oblasti komprese hlasového signálu, kde můžeme pro digitální záznam použít nižších vzorkovacích frekvencí (např. 800 vzorků za sekundu), a v oblasti hudebního signálu, vyžadující pro zachování kvality použití vysokých frekvencí (např. 44000 vzorků za sekundu).

K tomu, aby bylo možné definovat různé úrovně kvality komprese, zavádí standard MPEG pojem vrstvy (layer), které jsou označeny podle vzrůstající složitosti počínajíce 1. Je-li dekodér určen pro jistou vrstvu, musí dokázat dekódovat i data všech vrstev nižších. V současnosti je pro záznam zvuku nejatraktivnější formát MPEG-1 vrstva 3, známý také pod zkratkou MP3. Ten umožňuje dosáhnout kvalitní reprodukce stereofonního signálu i při kompresním poměru 1:12, přičemž pro záznam telefonního signálu lze dosáhnout komprese až 1:100.

6.3.1 Reprezentace zvukového signálu metodou PCM

Zvuk se šíří prostředím ve formě tlakových změn. Tyto změny můžeme zachytit ve formě analogového signálu a ten pak převést na signál digitální. Jak během snímání zvuku, tak i při jeho dalším zpracování dochází ke ztrátám a

zkreslením, která mají vliv na kvalitu zpětné reprodukce signálu nebo na možnost jeho analýzy, například pro účely rozpoznávání řeči.

Pro digitální reprezentaci zvuku lze využít metody pulsní kódové modulace (PCM). Touto metodou získáme ze spojitého vstupního signálu (získaného například mikrofonem) periodickým vzorkováním posloupnost hodnot amplitud signálu, kterou převedeme na celočíselnou hodnotu – tento proces se nazývá analogově-digitální (A/D) konverze. Vzhledem k tomu, že jsme obvykle omezeni na určitý počet bitů pro uložení každé hodnoty amplitudy (typicky 8 nebo 16 bitů), je třeba provést kvantizaci signálu, tj. nahradit spojitě se měnící hodnoty redukovaným počtem diskrétních úrovní. Chyba vzniklá kvantizací způsobuje vysokofrekvenční šum, jenž zhoršuje kvalitu reprodukce uloženého signálu.

Reprodukce signálu zakódovaného metodou PCM probíhá prostřednictvím analogově-digitálního (A/D) převodníku, jenž pro konkrétní hodnotu amplitudy vytvoří na výstupu odpovídající napěťovou úroveň. V nejjednodušším případě, pokud není použita některá interpolační metoda, zůstává úroveň výstupního signálu po dobu jednoho vzorku konstantní, což vytváří další zkreslení.

6.3.2 Metoda DPCM

Metoda rozdílové PCM (Differential Pulse Code Modulation, DPCM) je založena na předpokladu, že v typickém průběhu řečového signálu dochází pouze k relativně malým změnám. Tyto změny můžeme efektivně reprezentovat pomocí rozdílů po sobě jdoucích hodnot PCM místo hodnot samotných.

Výpočet změn lze rovněž provést až po provedení kvantizace, kdy můžeme dosáhnout lepší komprese. To však do výpočtu přináší kvantizační chybu, která se může postupně akumulovat. Pro její omezení používáme rámcování (framing) nebo rozptyl (dithering).

Metoda rámcování kombinuje ukládání diferencí s ukládáním vzorkované hodnoty do rámců pevné délky – první vzorek rámce je zakódován přímo, zbývající vzorky pomocí diferencí. Tím je proces akumulace kvantizační chyby redukován na malý počet kroků uvnitř jediného rámce. S podobnou metodou se ještě setkáme znovu při kompresi videosignálu.

Metoda rozptylu se pokouší omezovat chybu tím, že akumulovanou kvantizační chybu postupně přidává k hodnotě reprezentace vzorku a tu teprve kvantizuje.

V praxi se setkáme častěji s adaptivní variantou (ADPCM), která využívá více kvantizátorů (typicky 4), z nichž vybírá ten, jenž nejlépe odpovídá dynamickému rozsahu vzorků v rámci jednoho rámce. Zvolený kvantizátor se zakóduje do rámce. Dále se zakóduje první vzorek v plném rozlišení, za nímž následují indexy do kvantizační tabulky zvoleného kvantizátoru. Tím se dosahuje dalšího zlepšení kompresního poměru.

6.3.3 Metoda MPEG Audio (MP3)

Při kompresi hudebního signálu obvykle s jednoduchými metodami typu ADPCM nevystačíme, jejich účinnost je při požadavku na zachování vysoké kvality reprodukce velmi malá. Expertní skupina MPEG (Moving Pictures Experts Group) připravila skupinu mezinárodně uznávaných standardů pro kompresi zvuku a videa. Jsou z komerčního hlediska velmi důležité, neboť se

používají nejen v počítačovém průmyslu, ale i ve spotřební elektronice – např. v MP3 přehrávačích a autorádiích, systémech domácího kina nebo digitálních fotoaparátech a kamerách.

Standardy MPEG jsou určeny pro široké spektrum aplikací využívajících zvuk, od mluvené řeči až po kvalitní hudbu nebo speciální zvukové efekty. Vysoké komprese se dosahuje zejména využitím vlastností lidského sluchu, jež umožňují odstranění redundantních dat bez znatelné újmy na kvalitě reprodukce. Využívá se omezení frekvenčního rozsahu přenášeného pásma (20 Hz až 20 kHz) a rozdělení dynamického rozsahu (rozdíl hlasitosti 96 dB). Metoda maskování frekvencí využívá nelinearity citlivosti lidského sluchu, kdy v přítomnosti silnějšího signálu nedokážeme vnímat slabší signál, jenž tedy není třeba při kompresi ani uvažovat. Metoda časového maskování využívá setrvačnosti lidského sluchu, kdy po zániku silnějšího signálu začneme současně působící slabší signál vnímat až s jistým časovým odstupem. Tuto prodlevu můžeme opět využít ke snížení objemu komprimovaných dat.

6.4 Komprese obrazu

Existuje celá řada formátů pro kompresi obrázků. Mezi nejznámější patří:

- GIF (Graphics Interchange Format) – Implementuje bezztrátovou kompresi. Implementuje LZW algoritmus s omezenou délkou slovníku (po naplnění je statický).
- JPEG (Joint Photographic Experts Group) - V roce 1986 vznikla expertní skupina JPEG (Joint Photographics Experts Group), která si kladla za cíl vytvoření mezinárodního standardu pro kompresi a dekompresi spojitého vícebarevného statického obrazu. Výsledkem je soubor algoritmů, přizpůsobitelných požadavkům uživatele (např. na úroveň komprese). Podstatou těchto algoritmů je využití transformačních metod diskrétní kosinové transformace (DCT), jejímž výsledkem je soustava koeficientů, které jsou dále kvantizovány a efektivně uloženy ve výsledné reprezentaci. JPEG představuje ztrátovou metodu komprese dat, přičemž ke ztrátám dochází zejména u těch informací, které lidské oko nedokáže snadno rozlišit. Například malé změny barev jsou rozeznatelné obtížněji než změny v intenzitě a v případě barevných předloh se spojitými přechody barev lze dosáhnout touto metodou kompresního poměru 15:1 až 25:1 bez výrazné ztráty kvality zpětného zobrazení předlohy.
- TIFF (Tagged Image File Format) Pro přenos rastrových dat. Velmi flexibilní, ale obtížný na zpracování.
- PNG (Portable Network Graphics) – Důraz na přenositelnost. Vylepšuje vlastnosti GIF (průhlednost, prokládání, lepší komprese).

Obdobné metody jako pro kompresi obrázků můžeme použít ke kompresi videa. Základní ideou při kompresi videa je, že video je posloupnost obrazovek, dvě obrazovky bezprostředně po sobě se příliš neliší. Proto se často používá přístup kdy je z nějako periodou uložená celá informace o aktuálním snímku a mezi těmito snímky se ukládají jen informace o změnách.

Kontrolní otázky:

1. Pokud je entropie zdroje 3 bity na symbol, znamená to, že žádný symbol nemůže být kódován sekvencí kratší než 3 bity?
2. Huffmanův kód je minimální, co musí platit, aby správa byla zakódovaná na „minimální“ počet bitů?
3. Huffmanův kód je minimální přesto můžeme použitím některých slovníkových metod dosáhnout lepšího kompresního poměru, proč?
4. Přenáší se příjemci slovník u metody LZ78?
5. Pokud zkomprimujeme záznam zvuku do formátu MP3, lze jej bez ztráty kvality převést zpět do původního formátu? Pokud ne, jaké informace v záznamu zvuku chybí?

Úkoly k zamyšlení:

1. Zamyslete se nad tím, jak efektivně realizovat slovník u algoritmů LZW.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními pojmy z oblasti komprese dat. Byly ukázány některé základní kompresní algoritmy. Tyto algoritmy jsou pořád aktivně používány (často existuje celá řada různých variací těchto algoritmů). Byly také demonstrovány oblasti, kde můžeme tyto algoritmy využít.

7 Literatura

1. Beneš M., *Překladače*, Skripta k předmětu Programovací jazyka a překladače.
2. Lowy J.: .NET Componets, O`Reilly, April 2003, ISBN 0-596-000347-1