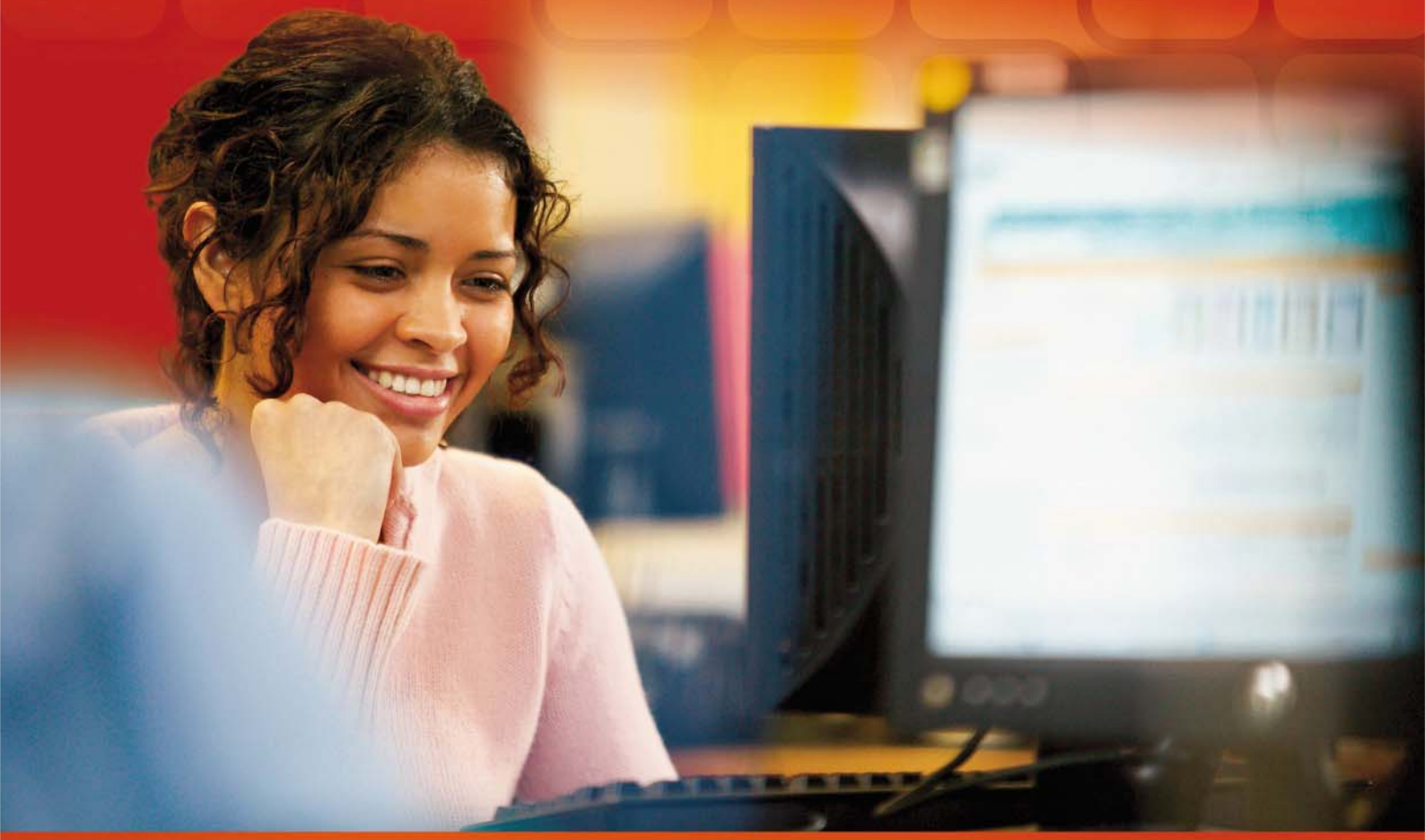
Microsoft



Ján Hanák

Objektovo orientované programovanie v jazyku C# 3.0

Príručka pre vývojárov, programátorov
a softvérových expertov

Microsoft Visual Studio 2008

Microsoft Visual Studio Express Editions

Ján Hanák

Objektovo orientované programovanie v jazyku C# 3.0

(Príručka pre vývojárov, programátorov a softvérových expertov)

Microsoft
2008

Obsah

Úvod

Počas uplynulého polstoročia sa v informatických vedách uskutočnil taký progres, aký sa v iných oblastiach ľudských činností zaznamenáva spravidla za niekoľko storočí. Z pôvodne sálových počítačov sa stali mikropočítače s ohromným výkonom, ktoré sú kedykoľvek a kdekoľvek pripravené pomáhať ľuďom s riešením ich každodenných problémov. Vedno s prevratným technologickým tempom držala krok aj teoretická informatika, ktorá definovala metodiky, postupy a paradigmy vývoja počítačového softvéru. Je to totiž práve softvér, ktorý dokáže vdýchnuť dušu rozmanitým hardvérovým strojom a zariadeniam. Bez sofistikovaného softvéru by nemohol byť úspešne realizovaný ani zlomok súčasnej technologicko-informačnej revolúcie.

Hoci z dnešného pohľadu je to snáď až nepredstaviteľné, paradigmy vývoja počítačového softvéru neboli vždy také vyspelé ako v súčasnosti. Začiatky sa spájajú s priamym programovaním počítačov pomocou rýdzo technických úkonov, ktoré boli neskôr nahradené tvorbou programov v strojovom kóde. Explicitné programovanie číslicových počítačov na nízkej úrovni sa vyznačovalo nielen veľmi malou mierou pracovnej produktivity, ale aj náročnosťou na abstraktné myslenie a technickú precíznosť. Presun od strojového kódu smerom k jazykom symbolických adries priniesol vo svojej dobe signifikantnú úroveň abstrakcie. Hoci aj dnes sa časti jadier mnohých operačných a informačných systémov programujú v jazyku symbolických adries, v akademickej a komerčnej sfére sa udomácnili paradigmy vývoja počítačového softvéru, ktoré sú typické vyššou mierou abstrakcie od hardvérovej infraštruktúry. Štruktúrované programovanie umožnilo naplno rozvinúť myšlienku problémovej dekompozície, kedy sa problém rozložil na viacero algoritmicky riešiteľných podproblémov, ktorých spracovanie mali na starosti viaceré funkcie jedného programu. Štruktúrované programovanie sa stalo prvou vývojovou vetvou, ktorej sa podarilo zapojiť do programovania počítačov široké cieľové publikum. O perspektívnosti štruktúrovaného programovania svedčí jeho stále veľká praktická obľúbenosť, a to aj napriek tomu, že od uvedenia tohto štýlu vývoja softvéru uplynulo už niekoľko desaťročí.

Veľký posun vpred zaznamenala objektová paradigma, ktorá sa stala základným pilierom objektovo orientovaného programovania. Dáta a manipulačné operácie, ktoré sú s dátami uskutočňované, sú zapuzdrené do logických jednotiek, ktorým vravíme objekty. Virtuálne objekty vznikajú v procese objektového modelovania, pričom reflektujú vlastnosti a schopnosti skutočných objektov tvoriacich reálny svet. Virtuálne objekty sú navyše inteligentné a vedia kooperovať tak, aby spoločne vyriešili mnohé aj z tých najnáročnejších problémov, s akými sa teoretická informatika stretla.

Netrvalo dlho a vývoj softvéru sa z pôvodne akademických a výskumných centier preniesol do rýdzo komerčnej sféry. A v nej, ako je známe, sa cení najmä veľká pracovná

produktivita, ktorej cieľom je kreácia produktov s vysokou pridanou hodnotou. Proces vývoja softvéru sa začal podobať výrobe iných zložitých strojov a zariadení, pričom sa doň implementovali prvky hromadnej výroby. Komponentové programovanie predstavilo komponentovú technológiu vývoja softvéru, podľa ktorej sú aplikácie tvorené kolekciou vhodne nakonfigurovaných počítačových súčiastok, teda komponentov. Komponentové programovanie využíva všetky silné stránky objektovo orientovaného programovania, veď koniec koncov, komponenty ako také sú tvorené súpravou spolupracujúcich objektov. Komponentové programovanie umožnilo naplno rozvinúť ideu rýchleho vývoja počítačových aplikácií.

Výzvou súčasných a určite aj budúcich rokov je paralelné programovanie, ktoré možno realizovať v súčinnosti s objektovo orientovaným a komponentovým programovaním. Paralelné programovanie sa stáva hitom, pričom tento stav je podmienený predovšetkým príchodom počítačov, ktoré sú osadené viacjadrovými procesormi. Práve takéto stroje umožňujú dosiahnuť skutočný paralelizmus, kedy sú toky programových inštrukcií spracúvané paralelne na jednotlivých jadrách procesora. Keďže môžeme s istotou predpovedať, že v budúcnosti sa budú objavovať procesory so stále väčším počtom exekučných jadier, je nutné v záujme optimálneho využitia celkovej výpočtovej kapacity systému venovať zvýšenú pozornosť masívnej paralelizácii.

Cieľom tejto knihy je vysvetliť všetky princípy objektovo orientovaného programovania v jazyku C# 3.0. V publikácii sme sa snažili vytvoriť robustné teoreticko-praktické zázemie, ktoré pomôže vývojárom pripraviť sa na novo prichádzajúcu paradigmu tvorby počítačového softvéru, ktorou je paralelné objektovo orientované programovanie (POOP). Tak budú môcť vývojári, programátori a softvéroví experti čeliť novým apelom, ktoré prinesie paralelná epocha počítačovej evolúcie.

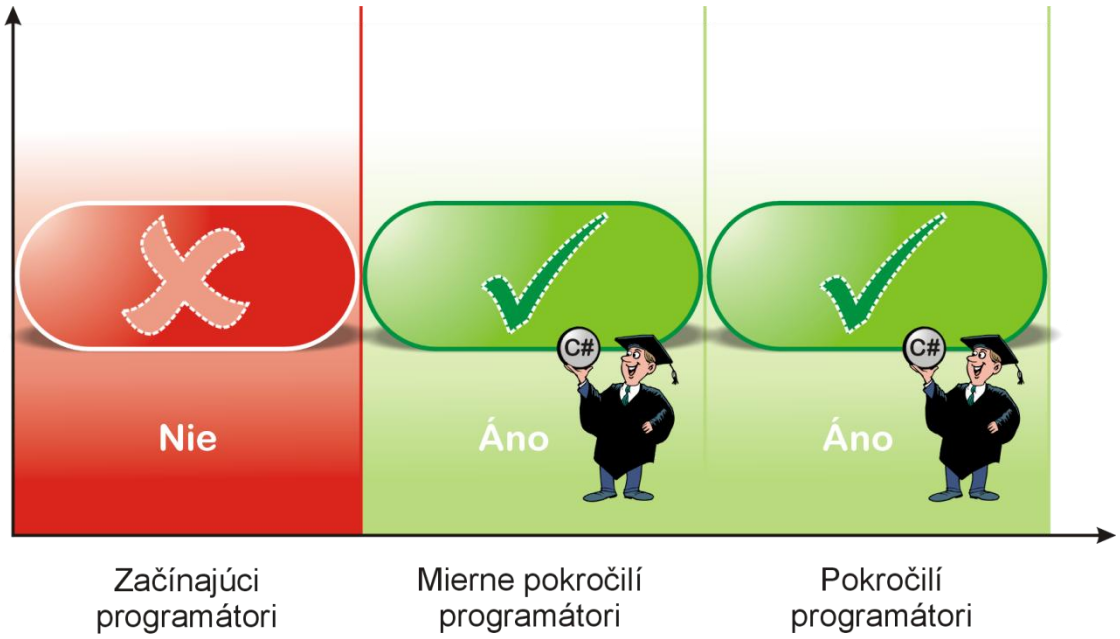
Ján Hanák

Praha, september 2008

Pre koho je táto kniha určená

Kniha **Objektovo orientované programovanie v jazyku C# 3.0** sa sústreďuje na výklad základných pilierov všeobecnej teórie objektovo orientovaného programovania s ich praktickou aplikáciou v programovacom jazyku C# 3.0, ktorý je implementovaný v produktoch Microsoft Visual C# 2008 a Microsoft Visual C# 2008 Express. Primárnym cieľovým segmentom publikácie sú mierne pokročilí programátori a vývojári v jazyku C#, ktorí ovládajú základy tohto programovacieho jazyka.

Kniha predpokladá, že čitatelia vedia, ako funguje program v jazyku C# 3.0 a rovnako sú oboznámení s elementárnymi programovými konštrukciami, ku ktorým patria premenné, operátory, rozhodovacie príkazy a cykly. Ak ste začiatočníkom v algoritmizácii alebo v programovaní vo všeobecnosti, odporúčame vám, aby ste si pred štúdiom tejto publikácie najskôr prečítali niektorú z množstva kníh, ktoré podávajú základný výučbový kurz programovania v jazyku C# 3.0. Výklad kladie mimoriadny dôraz na vysokú technickú kvalitu, terminologickú exaktnosť a vedecké spracovanie problematiky, čím napĺňa koncepciu primeraného študijného zaťaženia čitateľov.



Obr. A: Vhodnosť knihy pre rôzne segmenty používateľov

Publikácia je vhodná pre poslucháčov vysokých škôl univerzitného typu so zameraním na výučbu informatických a počítačových vied. Svoje uplatnenie nachádza v špecializovaných predmetoch 1., resp. 2. stupňa vysokoškolského štúdia.

Obsahová štruktúra knihy

Kniha **Objektovo orientované programovanie v jazyku C# 3.0** sa skladá z dvoch hlavných tematických celkov:

1. Teoretické základy objektovo orientovaného programovania.
2. Objektovo orientované programovanie v jazyku C# 3.0.

1. tematický celok oboznamuje čitateľov s tromi hlavnými paradigmami vývoja počítačového softvéru, ktoré sa vyvinuli za posledné desaťročia. Ide o štruktúrované, objektovo orientované a komponentové programovanie. Dôraz je kladený predovšetkým na komplexné ozrejmenie princípov všeobecnej teórie objektovo orientovaného programovania.

2. tematický celok nadväzuje na teoretické základy položené v 1. tematickom celku. Hlavný výkladový prúd tvoria kapitoly, ktoré sa venujú praktickému objektovo orientovanému programovaniu v jazyku C# 3.0. Komplementárnou pridanou hodnotou je výklad syntakticko-sémantických inovácií jazyka C# 3.0, ktoré sa viažu s filozofiou objektovo orientovaného programovania.

Typografické konvencie

Aby sme vám čítanie tejto knihy spríjemnili v čo možno najväčšej miere, bol prijatý kódex typografických konvencií, pomocou ktorých došlo k štandardizácii a unifikácii použitých textových štýlov a grafických symbolov. Veríme, že prijaté konvencie zvýšia prehľadnosť a používateľskú prívetivosť výkladu. Prehľad použitých typografických konvencií uvádzame v tab. A.

Tab. A: Prehľad použitých typografických konvencií

Typografická konvencia	**Ukážka použitia typografickej konvencie**
Štandardný text výkladu, ktorý neoznačuje zdrojový kód, identifikátory, modifikátory a kľúčové slová jazyka C# 3.0, ani názvy iných syntaktických elementov a entít, je formátovaný týmto typom písma.	Vývojovo-exekučná platforma Microsoft .NET Framework 3.5 kreuje spoločne s jazykom C# 3.0 jednotnú technologickú bázu na vytváranie moderných riadených aplikácií pre Windows, web a inteligentné mobilné zariadenia.

Tab. A: Prehľad použitých typografických konvencií (pokračovanie)	
Typografická konvencia	**Ukážka použitia typografickej konvencie**
Názvy ponúk, položiek ponúk, ovládacích prvkov, komponentov, dialógových okien, podporných softvérových nástrojov, typov projektov ako aj názvy ďalších súčastí grafického používateľského rozhrania sú formátované **tučným** písmom. **Tučným** písmom sú rovnako formátované aj identifikátory programových entít, ktoré sú uvádzané priamo vo výkladovom texte.	V záujme založenia nového projektu štandardnej aplikácie pre systém Windows (**Windows Forms Application**) v prostredí produktu Visual C# 2008 postupujeme takto: 1. Otvoríme ponuku **File**, ukážeme na položku **New** a klikneme na príkaz **Project**. 2. V dialógovom okne **New Project** klikneme v stromovej štruktúre **Project Types** na položku **Visual C#**. 3. Zo súpravy projektových šablón (**Templates**) vyberieme ikonu šablóny **Windows Forms Application**. 4. Do textového poľa **Name** zapíšeme názov pre novú aplikáciu a stlačíme tlačidlo **OK**.

Fragmenty zdrojového kódu jazyka C# 3.0, prípadne akýchkoľvek iných programovacích jazykov, sú formátované neproporcionálnym písmom `Courier New`.	`// Definícia premennej typu string.`
	`string správa;`
	`// Inicializácia premennej.`
	`správa = "Vitajte v jazyku " +`
	`"C# 3.0!";`
	`// Zobrazenie okna so správou`
	`// pomocou metódy Show triedy`
	`// MessageBox.`
	`MessageBox.Show(správa);`

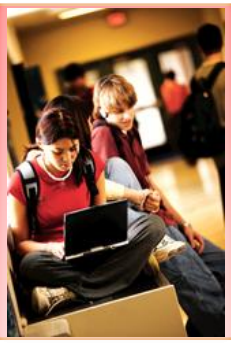
Poďakovanie

Na tomto mieste by autor knihy rád vyjadril svoje poďakovanie pánovi Jiřímu Burianovi, ktorý zastáva pozíciu produktového manažéra vo vývojárskej divízii českej pobočky spoločnosti Microsoft za výbornú niekoľkoročnú spoluprácu, ktorá priniesla vývojárskej komunite mnoho hodnotných produktov. Rovnako srdečne ďakuje autor za skvelú spoluprácu, ochotu a podporu aj pánovi Miroslavovi Kubovčíkovi, ktorý je kmeňovým členom vývojárskeho tímu slovenskej pobočky spoločnosti Microsoft. Bez ich podpory

by nemohla vzniknúť nielen táto kniha, ale ani mnohé ďalšie diela, ktoré permanentne zlepšujú stav vývojárskeho ekosystému. Takže páni, ešte raz, úprimná vďaka!

1. tematický celok

Teoretické základy objektovo orientovaného programovania



Microsoft® Visual Studio® 2008

1. Teoretické základy objektovo orientovaného programovania

1.1 Vývojové vetvy a paradigmy programovania

Počas uplynulých päťdesiatich rokov zaznamenali prístupy k vývoju počítačového softvéru nebývalý rozmach. Pri bližšej analýze môžeme determinovať tri základné vetvy programovania:

1. Štruktúrované programovanie.
2. Objektovo orientované programovanie.
3. Komponentové programovanie.

Každá z uvedených vetiev prichádza s vlastnou paradigmou vývoja počítačového softvéru.

1.1.1 Štruktúrované programovanie

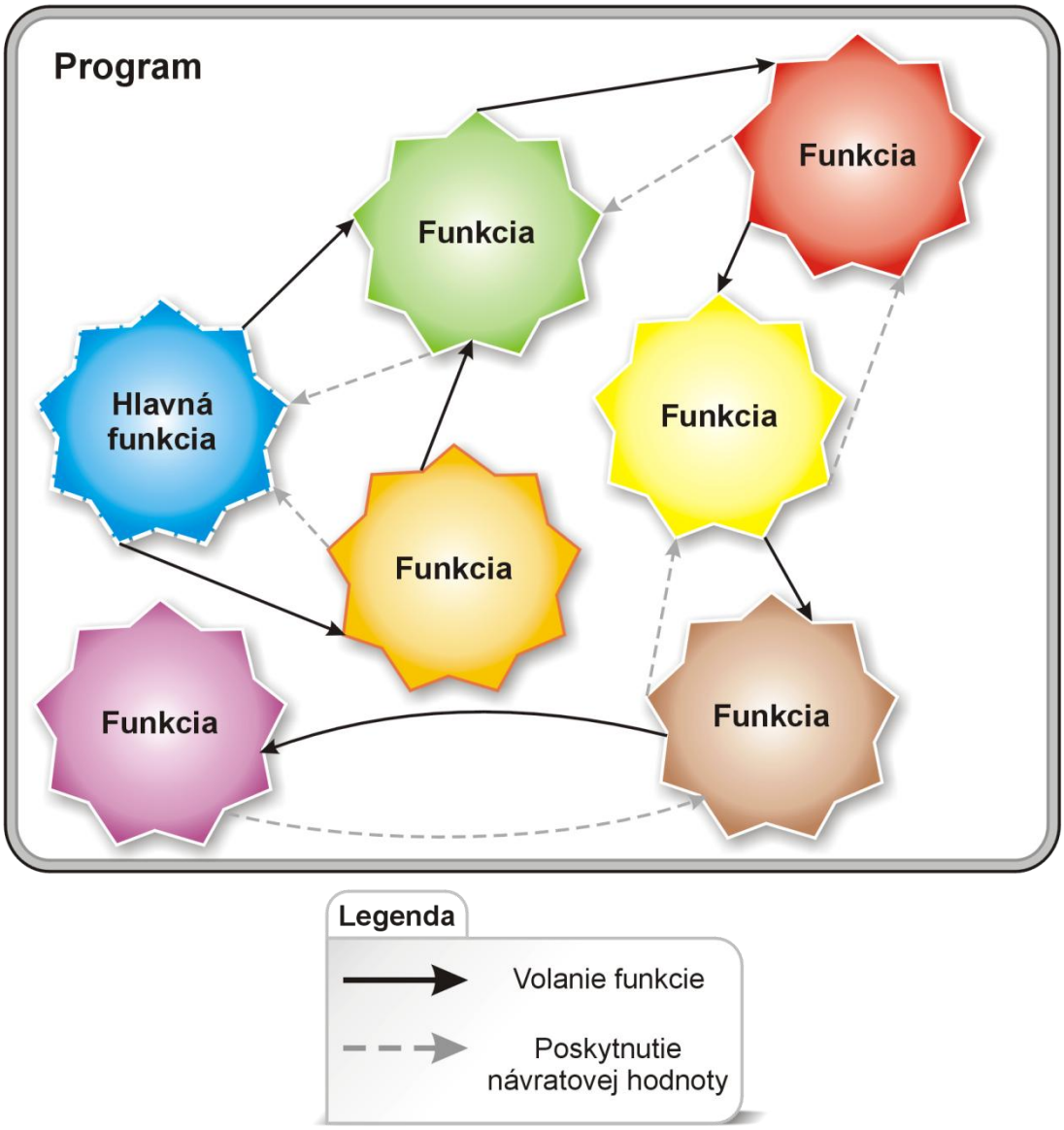
Štruktúrované (procedurálne) programovanie sa sústreďuje na návrh imperatívnych programov, ktoré implementujú algoritmy prostredníctvom troch základných grafov riadenia, resp. riadiacich konštrukcií, ktorými sú: sekvencia, selekcia a iterácia. Návrh štruktúrovaných programov spravidla prebieha postupným zjemňovaním zložitosti, kedy sa pôvodne riešený komplexný problém rozkladá na menšie podproblémy, ktoré disponujú nižšou zložitosťou. Naznačená problémová dekompozícia je spojená s intenzívnym využitím základných riadiacich konštrukcií, ktoré je možné spoľahlivo definovať syntaktickými príkazmi aplikovaného programovacieho jazyka. Štruktúrované programovanie využíva štruktúrované algoritmy. Štruktúrovaný algoritmus môžeme charakterizovať induktívno-rekurzívnym spôsobom:

1. Každý jednotlivý krok je štruktúrovaným algoritmom.
2. Základné riadiace konštrukcie (sekvencia, selekcia a iterácia) štruktúrovaných algoritmov predstavujú taktiež štruktúrovaný algoritmus.
3. Štruktúrovaným algoritmom je všetko to, čo získame opakovaným použitím premís položených v 1. a 2. kroku tohto postupu.

Z praktického hľadiska je štruktúrovaný program tvorený množinou funkcií, ktoré medzi sebou vedú prospešný informačný dialóg. Funkcia je základnou stavebnou jednotkou štruktúrovaného programu a ako taká sa často stotožňuje s podprogramom. Z uvedeného vyplýva, že štruktúrovaný program P je množinou spriaznených funkcií F, čo môžeme matematicky formalizovať takto:

$$P = F = \{f_1, f_2, \dots, f_n\}$$

Jedna z funkcií štruktúrovaného programu má výsadné postavenie, pričom sa klasifikuje ako hlavná funkcia programu. Hlavná funkcia predstavuje vstupný bod programu, čo znamená, že exekúcia programu sa začína volaním a spracovaním tejto funkcie. Hlavná funkcia je automaticky aktivovaná operačným systémom vždy, keď je zaregistrovaná požiadavka na spustenie štruktúrovaného programu. V rámci implementácie štruktúrovaných algoritmov je ideálne, ak jedna funkcia programu rieši práve jeden algoritmus. Tak je možné dosiahnuť efektívnu úlohovú dekompozíciu. Úlohová dekompozícia pracuje v súčinnosti s technikou postupného zjemňovania zložitosti riešeného problému. Hlavná funkcia sa preto nekoncentruje na samostatné riešenie problému, jej úlohou je optimálna dekompozícia problému na jednotlivé podproblémy, ktorých spracovanie je delegované rôznym funkciám (podprogramom). Vizuálna kompozícia štruktúrovaného programu je znázornená na obr. 1.



Obr. 1: Štruktúrovaný program

Medzi hlavnou funkciou a ostatnými, tzv. používateľsky definovanými, funkciami existujú tesné dátové väzby. Pri volaní používateľsky definovanej funkcie, ktorá zabezpečuje spracovanie istého parciálneho algoritmu, je možné tejto funkcii odovzdať kvantá vstupných dát. Vstupné dáta špecifikované pri volaní funkcie sa nazývajú argumenty. V úlohe argumentov môžu vystupovať dátové objekty, ktoré sú na syntaktickej úrovni reprezentované hodnotami primitívnych dátových typov, resp. inštanciami používateľsky deklarovaných dátových typov. Je pochopiteľné, že ak sú funkcii odovzdávané určité argumenty, musí funkcia disponovať prostriedkom, pomocou ktorého bude schopná tieto argumenty prijať a uchovať. Týmto prostriedkom sú v štruktúrovaných programoch formálne parametre, ktoré z technického hľadiska predstavujú lokálne premenné funkcie určené len na absorpciu argumentov. Ak funkcia štruktúrovaného programu definuje vo svojej signatúre aspoň jeden formálny parameter, charakterizujeme ju ako parametrickú funkciu. Naproti tomu, funkcie s prázdnou signatúrou sa označujú termínom bezparametrické funkcie.

Štruktúrované programovacie jazyky zvyčajne implementujú dva základné mechanizmy odovzdávania argumentov formálnym parametrom:

1. Mechanizmus odovzdávania argumentov hodnotou.
2. Mechanizmus odovzdávania argumentov odkazom.

Pri aplikácii mechanizmu odovzdávania argumentov hodnotou je vždy vytvorená kópia argumentu (pôvodného dátového objektu), ktorá je následne poskytnutá volanej funkcii. Volaná funkcia získa identický duplikát argumentu, ktorý môže použiť pri svojej práci. Dokonca môže získaný duplikát aj modifikovať, no tieto modifikácie nijakým spôsobom neovplyvnia pôvodne odovzdávaný argument. Na druhú stranu, pri mechanizme odovzdávania argumentov odkazom je volanej funkcii poskytnutá pamäťová adresa, na ktorej je argument (dátový objekt) situovaný. Keďže volaná funkcia pozná pamäťovú adresu argumentu, má k nemu priamy prístup a je schopná ho ľubovoľne modifikovať (aj potenciálne nebezpečným spôsobom, ktorý by mohol spôsobiť porušenie dátovej integrity argumentu). Ak uskutočníme dôkladnú komparáciu oboch mechanizmov odovzdávania argumentov, dospejeme k nasledujúcim záverom:

1. Mechanizmus odovzdávania argumentov hodnotou je bezpečnejším mechanizmom odovzdávania dátových objektov volaným funkciám. Existuje totiž garancia, že volané funkcie nie sú schopné vykonať potenciálne nebezpečné programové operácie, ktoré by vyústili do porušenia dátovej integrity. Keďže jednotlivé programovacie jazyky preferujú ochranu dátovej integrity, mechanizmus odovzdávania argumentov hodnotou je v nich realizovaný spravidla implicitne.

2. Za istých okolností sa použitie mechanizmu odovzdávania argumentov hodnotou viaže s nežiaducou postrannou réžiou. Spomenuté režijné náklady sú alokované

predovšetkým vtedy, ak volajúca funkcia poskytuje volanej funkcii kapacitne náročné dátové objekty. Kapacitne náročné dátové objekty sú objekty, ktoré alokujú v operačnej pamäti počítača pomerne veľké miesto, zvyčajne presahujúce niekoľko stovák kilobajtov. Keby boli determinované dátové objekty odovzdávané hodnotou, pre každú z volaných funkcií by bolo nutné vytvoriť samostatný duplikát pôvodného dátového objektu. Pochopiteľne, vyššie požiadavky na dostupný alokačný pamäťový priestor by sa vypuklejšie prejavili najmä pri frekventovanom výkone mechanizmu odovzdávania argumentov hodnotou. To sa deje napríklad pri odovzdávaní dát volaným funkciám v iteratívnych príkazoch. Možnosťou, ako eliminovať pamäťové zaťaženie systému, je uprednostniť mechanizmus odovzdávania argumentov odkazom. Toto riešenie je oveľa efektívnejšie, pretože nech už budeme pracovať s ľubovoľným počtom volaných funkcií, v pamäti bude alokovaný vždy len jeden dátový objekt. Všetky aktivované funkcie získajú pamäťovú adresu, prostredníctvom ktorej je daný objekt dosiahnuteľný[1].

Volaná funkcia môže po vyriešení problému, resp. po dokončení svojej činnosti, podať o jej výsledku správu hlavnej funkcii. Tento komunikačný model je realizovaný pomocou odovzdávania tzv. návratovej hodnoty volanej funkcie. Za bežných okolností môžeme kategorizovať všetky používateľsky definované funkcie v štruktúrovanom programe podľa toho, či istú návratovú hodnotu vracajú, alebo nie. Funkcie bez návratovej hodnoty sú zo sémantického hľadiska totožné s procedúrami, teda blokmi zdrojového kódu, ktoré sú spracované bez toho, aby podávali správu o výsledku uskutočnených aktivít.

Podobne, ako môže komunikovať hlavná funkcia so všetkými používateľsky definovanými funkciami, môžu kooperovať aj jednotlivé používateľsky definované funkcie medzi sebou. Pritom platia všetky pravidlá, ktoré sme rozobrali vyššie.

Manipulácia s argumentmi a návratovými hodnotami kreuje základný rámec pre riadenie tokov dát v štruktúrovaných programoch. Pre úplnosť dodajme, že existujú aj iné mechanizmy, ktoré umožňujú transport dát medzi viacerými funkciami navzájom. Typickým príkladom je použitie globálnych dátových objektov, ktoré sú zapuzdrené v globálnych premenných. K takýmto premenným majú prístup všetky funkcie, ktoré sa nachádzajú v príslušnej prekladovej jednotke zdrojového kódu.

Ak dôjde k vyvolaniu funkcie, riadenie programu je presmerované na kód vstupného bodu tela príslušnej funkcie. Ak sú funkcii odovzdávané argumenty, alokuje sa priestor pre formálne parametre, do ktorých sú argumenty uložené. V ďalšej etape je zahájená

[1] V tejto súvislosti je potrebné upozorniť na synchronizovaný prístup k dátovému objektu, a to najmä pri paralelnom programovaní. Ak by štruktúrovaný program obsahoval funkcie, ktoré by mohli byť vykonávané paralelne, bolo by nutné naprogramovať aplikačnú logiku programu tak, aby nevznikali nežiaduce interferencie pri prístupe k zdieľanému dátovému objektu z viacerých programových vlákien súčasne.

exekúcia programových príkazov, ktoré sú umiestnené v tele volanej funkcie. Z technického hľadiska nízkoúrovňovej interpretácie funkcie sú všetky programové príkazy vyjadrené v strojovom (natívnom) kóde triedy x86. Vo všeobecnosti, štandardný prenositeľný súbor PE (Portable Executable) štruktúrovaného programu obsahuje natívny kód, ktorý je po načítaní rozložený na mikroinštrukcie priamo spracovateľné inštrukčnou súpravou procesora. Po spracovaní všetkých programových príkazov uložených v tele funkcie je vrátená návratová hodnota funkcie a riadenie programu je presmerované späť do volajúcej funkcie.

Používateľsky definované funkcie štruktúrovaného programu môžu byť okrem priamej aktivácie vyvolávané aj rekurzívne. Teoretická informatika rozlišuje viacero variantov rekurzie, pre potreby tejto publikácie sa budeme sústrediť len na priamu a nepriamu rekurziu. Pri priamej rekurzii je vytvorená nová inštancia identickej funkcie predtým, ako sa ukončí spracovanie všetkých príkazov v tele tejto funkcie. Pri priamej rekurzii volaná funkcia aktivuje „samú seba“ v určitom okamihu svojho spracovania. Hoci empiricky môže rekurzívne volanie funkcie pôsobiť najprv netradične, z technického hľadiska nie je žiaden rozdiel medzi volaním nerekurzívnej a rekurzívnej funkcie.

Priama rekurzia sa uskutočňuje opakovane, pričom pri každom opakovaní je volaná nová inštancia funkcie s modifikovanou súpravou argumentov (rekurzívne funkcie budú vždy parametrické). Keďže pri priamej rekurzii jedna inštancia funkcie zrodí novú inštanciu funkcie, môžeme sledovať celý reťazec funkcií, ktoré vznikajú v priebehu rekurzívne riešeného problému. Reťazec jednotlivých inštancií volanej funkcie musí byť vždy konečný (bez ohľadu na to, či pracujeme s priamou, alebo nepriamou rekurziou)[2].

Pri každom zrode novej inštancie funkcie sa táto funkcia musí rozhodnúť, či parciálna časť problému, ktorú rieši, predstavuje rekurzívny, alebo základný prípad. Ak funkcia identifikuje, že ide o rekurzívny prípad, tak v procese dekompozície pôvodne komplexného problému ešte stále nebola dosiahnutá taká úroveň problému, ktorú by bolo možné okamžite vyriešiť. Podproblém na príslušnej úrovni je preto nutné opäť dekomponovať na jednoduchšie problémy. Rekurzívne prípady generujú rekurzívne volania funkcie, ktoré znamenajú zakladanie nových a nových inštancií funkcie v operačnej pamäti. Proces rekurzívneho volania funkcie sa musí podľa prijatých predpokladov ukončiť vtedy, keď istá inštancia v reťazci rekurzívneho volania funkcií narazí na základný prípad.

Základný prípad je asociovaný s takou úrovňou analyzovaného podproblému, ktorý dokáže funkcia v danej inštancii vyriešiť. Samozrejme, otázkou zostáva, za ako dlho bude

[2] Je zrejmé, že nekonečná rekurzia nemôže existovať. Tento stav je v skutočnosti veľmi nebezpečný, pretože by sa v praktických podmienkach skončil vyčerpaním systémových zdrojov, ktoré operačný systém dedikuje fyzickému procesu štruktúrovaného programu. Moderné operačné systémy umožňujúce viacúlohové a viacvláknové spracovanie štruktúrovaných programov však nedovolia, aby jeden „havarovaný“ program nepriaznivo ovplyvnil iné, aktuálne spracúvané programy.

schopná rekurzívne volaná funkcia dosiahnuť základný prípad. Všetky inštancie funkcie vytvorené v procese priamej rekurzie možno vizuálne reprezentovať ako úrovne vnárania sa rekurzívne volanej funkcie. Po dosiahnutí základného prípadu sa rekurzívne volaná funkcia „vynára späť", pričom každá z vytvorených inštancií poskytuje svojmu nadradenému náprotivku výsledok svojej činnosti.

Nepriama rekurzia je len variáciou priamej rekurzie: pri nej rekurzívne volaná funkcia aktivuje inú používateľsky definovanú funkciu a táto zase spätne volá pôvodnú funkciu. Rekurzívne volané funkcie sa uplatňujú pri riešení problémov, ktoré majú rekurzívnu povahu. Hoci je syntakticko-sémantické vyjadrenie rekurzívne volanej funkcie prirodzené a v mnohých prípadoch aj elegantné, je potrebné poukázať na vyššiu priestorovú zložitosť rekurzívne implementovaných algoritmov. Kapacitná náročnosť je spôsobená generovaním inštancií rekurzívne volanej funkcie, ktoré musia byť alokované v operačnej pamäti počítača. Vyššia záťaž je viazaná tiež s výkonom komunikačného mechanizmu medzi jednotlivými inštanciami rekurzívne volanej funkcie.

1.1.2 Objektovo orientované programovanie

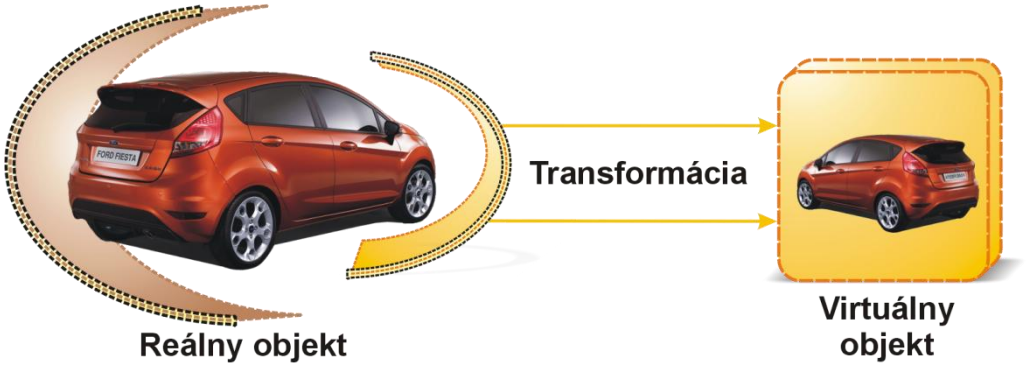
Hoci základy paradigmy objektovo orientovaného vývoja počítačového softvéru boli teoreticky položené už v 60. rokoch 20. storočia, praktický rozmach zaznamenal tento model tvorby softvéru približne o tridsať rokov neskôr. Objektovo orientované programovanie sa snaží riešiť jeden zo základných problémov štruktúrovaného programovania, ktorým je separácia dátových objektov a funkcií (podprogramov), ktoré s týmito dátovými objektmi manipulujú. Všeobecná teória objektovo orientovaného programovania vychádza z nasledujúcich axióm:

1. Objekt je základná logická entita, ktorá je charakterizovaná svojím identifikátorom, vlastnosťami a činnosťami. Identifikátor objektu je jednoznačné určenie objektu v systéme, resp. v objektovo orientovanom prostredí. Vlastnosti objektu sú dané jeho atribútmi a predstavujú znaky, ktorými objekt disponuje. Činnosti objektu sú reprezentované jeho metódami a tvoria behaviorálnu zložku objektu. Podľa množiny metód môžeme diagnostikovať štýl správania sa objektu voči klientom, resp. voči iným objektom v jeho prostredí.

2. Objekt je výsledkom objektového modelovania, teda procesu, prostredníctvom ktorého sa vytvárajú virtuálne ekvivalenty fyzických objektov, s ktorými pracujeme v reálnom svete.

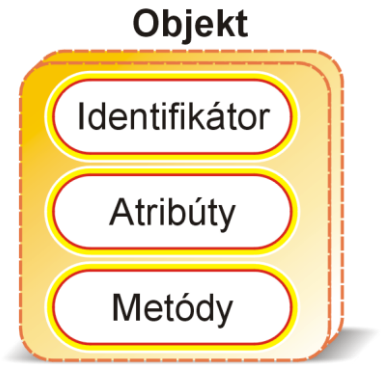
3. Keďže fyzické objekty sú spravidla veľmi zložité, počas objektového modelovania sa uplatňuje princíp objektovej abstrakcie, v rámci ktorej abstrahujeme od tých atribútov a činností fyzických objektov, ktoré pre nás nie sú dôležité (obr. 2).

Objektová abstrakcia nám dovoľuje sústrediť sa len na tie vlastnosti a činnosti objektu, ktoré sú významné z hľadiska vytváraného systému, resp. aplikácie.



Obr. 2: Objektové modelovanie a objektová abstrakcia

4. Každý objekt je svojprávnou jednotkou, ktorá obsahuje vlastné atribúty a metódy. Atribúty sú reprezentované dátovými položkami, ktoré sú určené na úschovu dát objektu. Metódy stelesňujú činnosti vykonávané objektom. Metódy môžu priamo pracovať s atribútmi objektu. Nahliadanie na objekt ako na monolitný kontajner, v ktorom sa nachádzajú atribúty a metódy, je princípom zapuzdrenia (obr. 3).



Obr. 3: Atribúty a metódy sú zapuzdrené do objektu

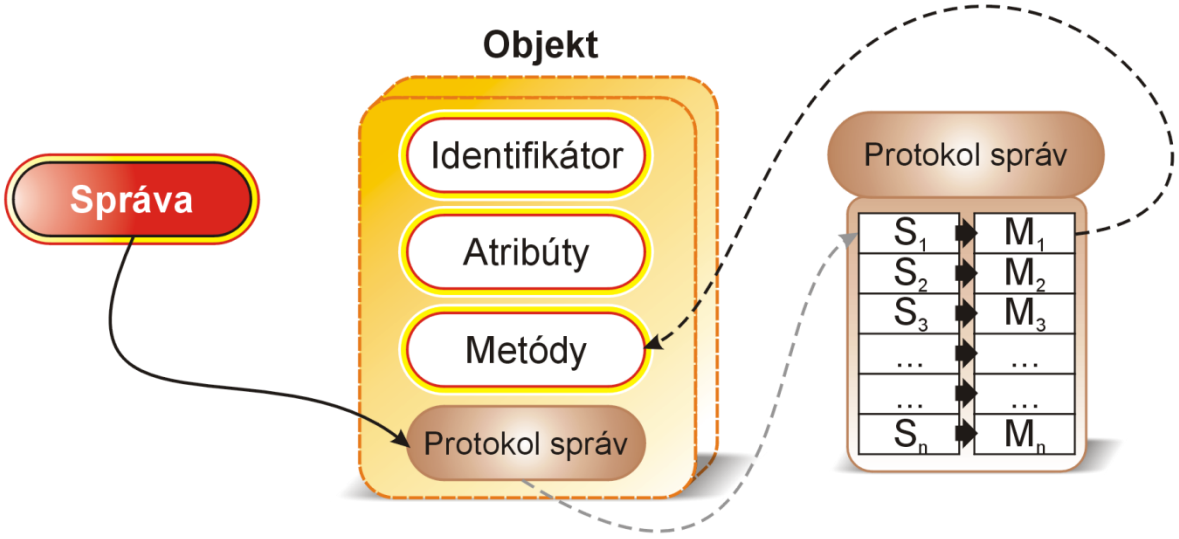
5. Objekt je z vonkajšieho pohľadu „čiernou skrinkou“, pretože iné objekty nevidia jeho atribúty, ani metódy. Tým, že objekt ukrýva svoje atribúty, zabezpečuje ochranu dát, ktoré obsahuje. Nie je teda možné, aby boli tieto dáta modifikované potenciálne nebezpečným spôsobom iným objektom, resp. inou entitou vyskytujúcou sa vo vonkajšom prostredí objektu. Vo všeobecnej teórii objektovo orientovaného programovania je uvedený princíp známy ako ukrývanie dát.

6. S dátami objektu môžu priamo narábať len metódy objektu. Tie sú naprojektované tak, aby sa dátová časť objektu nikdy nedostala do nekonzistentného stavu. Podobne ako atribúty, aj metódy sú v objekte ukryté, a teda nikto (okrem objektu samotného) nevie, ako je v skutočnosti vykonávaná činnosť, ktorej realizácia je príslušnou metódou garantovaná. Tento princíp sa

nazýva ukrývanie implementácie a umožňuje používateľovi využívať služby objektu aj bez znalosti toho, ako sú tieto služby implementované.

7. Atribúty objektu formujú jeho vnútornú pamäť. Objekt teda vie o svojej existencii, pričom jeho aktuálny stav vždy reflektujú hodnoty jeho atribútov. Stav objektu môže byť kedykoľvek diagnostikovaný prostredníctvom metód na to určených.

8. Komunikácia vo vzťahoch „používateľ → objekt“ a „objekt → iný objekt“ sa uskutočňuje pomocou mechanizmu správ. Každý objekt je schopný prijať a spracovať istú množinu správ. Kolekcia správ, na ktoré dokáže objekt reagovať, je zaznamená v protokole správ. Protokol správ determinuje vzájomné relácie medzi jednotlivými správami a metódami objektu. Pomocou protokolu správ je možné vždy jednoznačne určiť, ktorá metóda bude aktivovaná ako reakcia na príjem istej správy. Medzi správami a metódami uvedenými v protokole správ existuje relácia typu 1:1. Každá správa je teda mapovaná na práve jednu metódu. Ak príde správa od používateľa, resp. iného objektu, objekt ju zachytí a spracuje. Spracovanie správy znamená vyhľadanie správy v zozname správ, ktorý je uvedený v protokole správ (obr. 4). Proces ďalej pokračuje aktivovaním metódy, ktorá je s prijatou správou asociovaná. Objekt vykoná požadovanú činnosť a s jej výsledkom oboznámi používateľa.



Obr. 4: Komunikácia s objektom pomocou protokolu správ

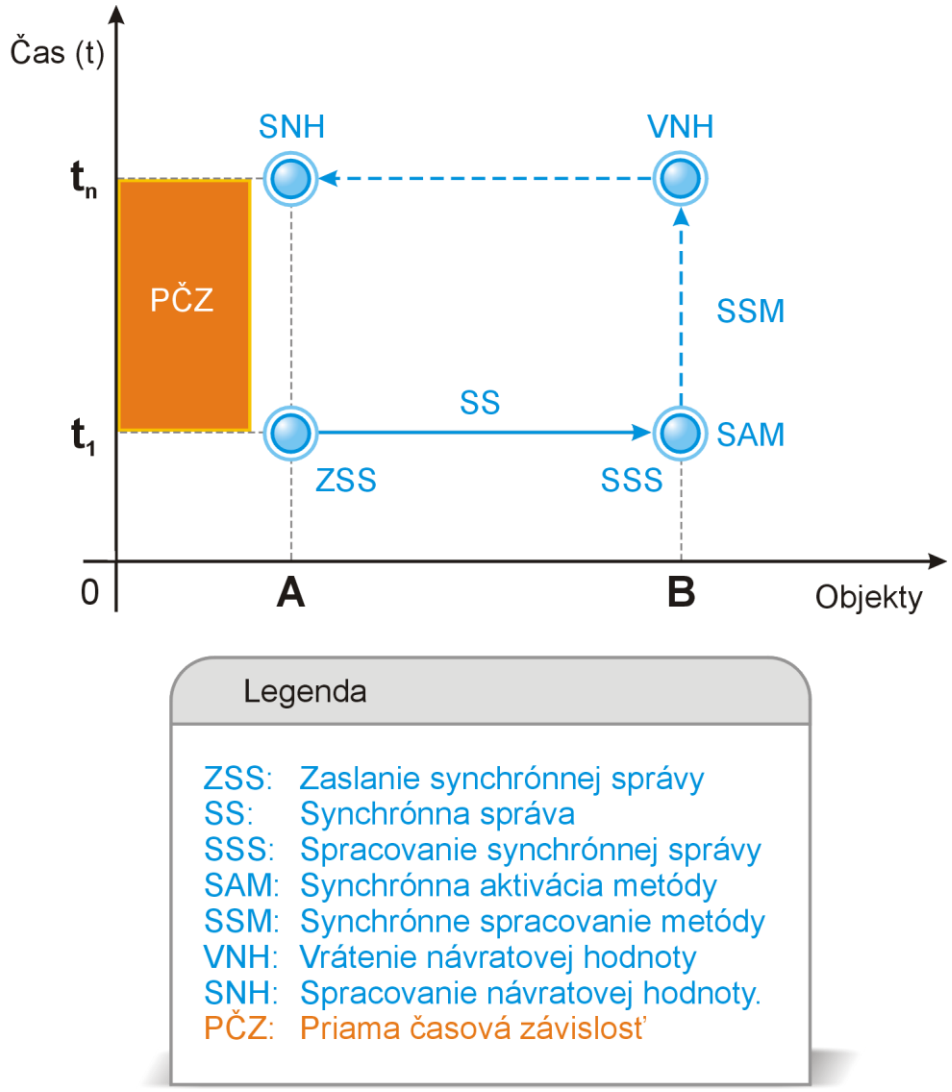
9. Správy, ktoré sú objektu zasielané môžeme rozdeliť na bezparametrické a parametrické. Bezparametrické správy so sebou nenesú žiadne dáta a ako také sú určené len na zistenie aktuálneho stavu objektu. Naopak, parametrické správy integrujú dáta, ktoré modifikujú aktuálny stav objektu. Kým parametrické správy menia vnútornú pamäť objektu, bezparametrické správy sa žiadnej priamej modifikácie nedopúšťajú. Medzi správami a metódami objektu existuje priama

korelácia: bezparametrické správy aktivujú bezparametrické metódy a parametrické správy spúšťajú parametrické metódy.

10. Z hľadiska časovej latencie pri čakaní na výsledok zaslania správy môžeme správy klasifikovať na synchrónne a asynchrónne. Aby sme mohli lepšie vysvetliť rozdiely medzi synchrónnymi a asynchrónnymi správami, budeme uvažovať abstraktné komunikačné modely s viacerými objektmi. Prvý objekt označíme identifikátorom **A**, druhému objektu priradíme identifikátor **B** a tretí objekt pomenujeme identifikátorom **C**. Zameriame sa na skúmanie dvoch situácií:

    1. Analýza synchrónneho komunikačného modelu.
    2. Analýza asynchrónneho komunikačného modelu.

    V synchrónnom komunikačnom modeli pracujeme s dvomi objektmi: **A** a **B**. Komunikácia medzi spomenutými objektmi sa začína tým, že objekt **A** zašle synchrónnu správu objektu **B**. Objekt **B** správu pomocou protokolu správ spracuje a synchrónne spustí metódu, ktorá je s danou správou prepojená. Objekt **B** teda začne vykonávať operácie, ktoré sú naprogramované v tele metódy, ktorú si objekt **A** želal spustiť. Dôležité je poukázať na skutočnosť, že objekt **A** nemôže realizovať žiadne ďalšie aktivity (napr. rozposielať iné správy), pokiaľ metóda objektu **B** nedokončí svoju činnosť. Povedané inak, objekt **A** musí čakať na návrat synchrónne aktivovanej metódy objektu **B**. Je zrejmé, že synchrónny komunikačný model vytvára pevnú väzbu medzi komunikujúcimi objektmi. Časová latencia, ktorá vzniká po doručení správy, je signifikantná. Vizuálne zobrazenie synchrónneho komunikačného modelu medzi objektmi môžeme vidieť na obr. 5.

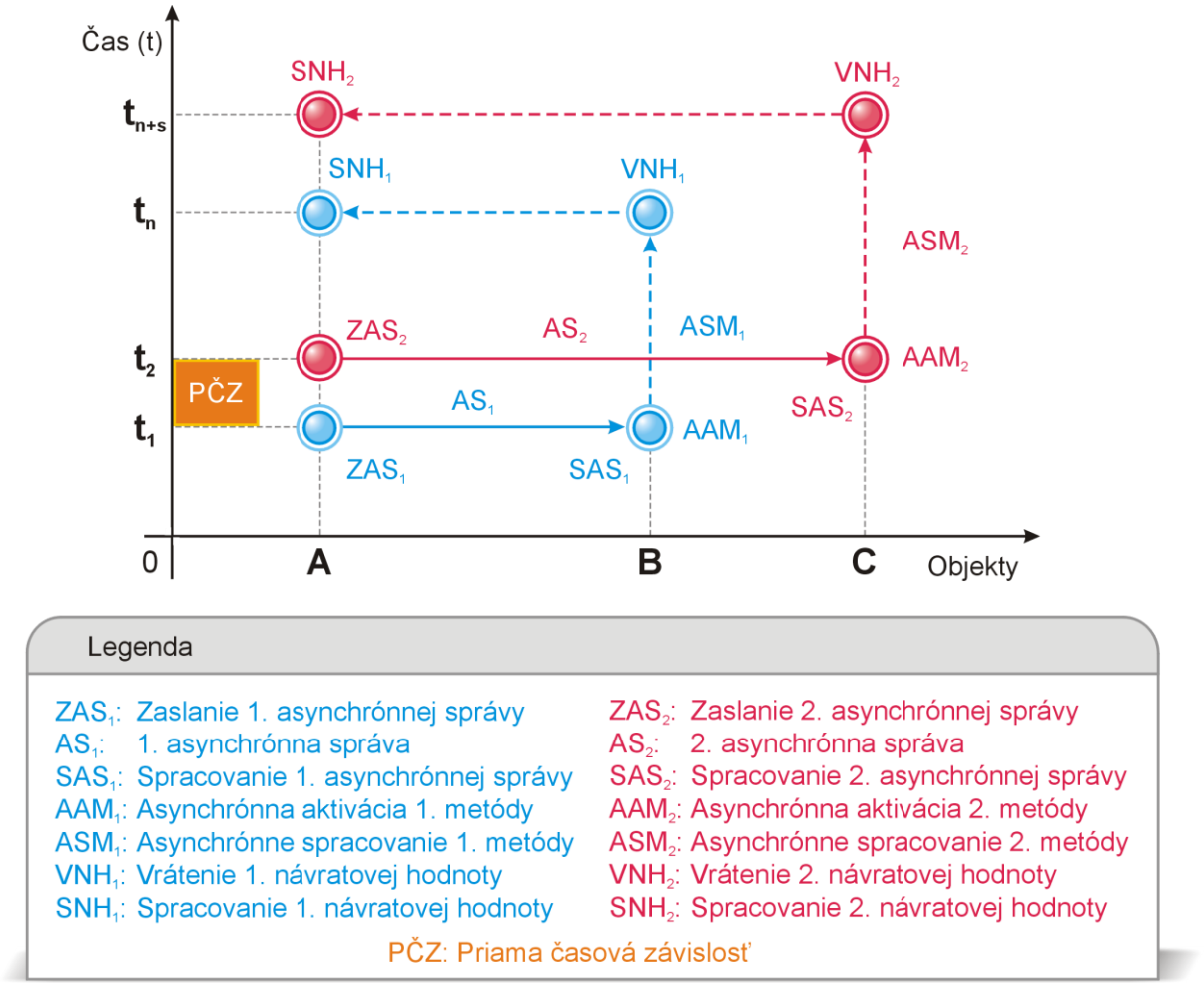


Obr. 5: Synchrónny komunikačný model medzi objektmi

Komentár k synchrónnemu komunikačnému modelu (obr. 5): V čase t_1 zasiela objekt **A** synchrónnu správu (SS) objektu **B**. Objekt **B** zabezpečuje spracovanie synchrónnej správy (SSS) protokolom správ. Po diagnostike správy objekt **B** zahajuje synchrónnu aktiváciu asociovanej metódy (SAM). Volaná metóda je spracúvaná synchrónne (SSM), čo znamená, že objekt **A** je znehybnený do okamihu, kedy táto metóda ukončí svoju činnosť. Po vrátení návratovej hodnoty (VNH) a jej spracovaní (SNH) v čase t_n sa riadenie vracia späť objektu **A**. Priama časová závislosť (PČZ), vyjadrujúca inoperabilitu objektu **A**, je daná intervalom $<t_1, t_n>$. Objekt **A** nemôže zaslať ďalšiu správu objektu **B** (alebo inému objektu) skôr, než v čase t_{n+1}.

Asynchrónny komunikačný model eliminuje citeľnú časovú latenciu, ktorá vzniká v synchrónnom komunikačnom modeli. V tomto modeli sa komunikácia medzi zúčastnenými objektmi odohráva nasledujúcim spôsobom: Objekt **A** zašle asynchrónnu správu objektu **B**. Riadenie sa v tejto chvíli okamžite vracia späť objektu **A**, ktorý môže zahájiť uskutočnenie ďalších operácií. Objekt **B** zaslanú

správu prijme, diagnostikuje ju pomocou protokolu správ a asynchrónne spustí metódu asociovanú s príslušnou správou. Keď metóda objektu **B** skončí svoju činnosť, bude o jej výsledku informovaný objekt **A**. Ako môžeme zaregistrovať, asynchrónny komunikačný model je paralelizácii úloh naklonený oveľa viac ako synchrónny model. Keďže objekt zasielajúci asynchrónnu správu nie je znehybnený počas výkonu metódy dopytovaného objektu, smie pokračovať vo svojej práci. Vizuálnu interpretáciu asynchrónneho komunikačného modelu medzi objektmi znázorňuje obr. 6.



Obr. 6: Asynchrónny komunikačný model medzi objektmi

Komentár k asynchrónnemu komunikačnému modelu (obr. 6): V záujme skúmania asynchrónneho komunikačného modelu musíme analyzovať viac ako dva objekty. V našom modeli uskutočňujeme výskum na troch objektoch s identifikátormi **A**, **B** a **C**. V čase t_1 zasiela objekt **A** objektu **B** 1. asynchrónnu správu (AS_1). Objekt **B** spracuje asynchrónnu správu (SAS_1) a asynchrónne spustí metódu spojenú s touto správou (AAM_1). Keďže objekt **A** nečaká na dokončenie asynchrónne aktivovanej metódy objektu **B**, môže v čase t_2 zaslať ďalšiu asynchrónnu správu (AS_2) objektu **C**. Objekt **C** doručenú asynchrónnu správu spracuje (SAS_2) a asynchrónne spustí zviazanú metódu (AAM_2). Obe asynchrónne

aktivované metódy objektov **B** a **C** pracujú paralelne. Ich finálny exekučný čas je variabilný: v našom modeli predpokladáme, že asynchrónne aktivovaná metóda objektu **B** vráti svoju návratovú hodnotu v čase t_n, zatiaľ čo asynchrónne aktivovaná metóda objektu **C** nám poskytne návratovú hodnotu v čase t_{n+s}. Všimnime si, že v asynchrónnom komunikačnom modeli je priama časová závislosť (PČZ) oveľa menej signifikantná ako v synchrónnom komunikačnom modeli. Keďže objekt **A** po zaslaní správy objektu **B** nečaká na spracovanie príslušnej metódy, môže už v čase t_2 (pričom platí, že $t_2 < t_n$) vytvoriť a zaslať inú asynchrónnu správu.

11. Protokol správ predstavuje verejne prístupné rozhranie objektu a je potrebné upozorniť na skutočnosť, že používatelia objektu majú prístup len k tomuto verejne prístupnému rozhraniu (atribúty i metódy objektu sú skryté). To však postačuje, pretože rozhranie ponúka kompletný aparát, prostredníctvom ktorého môžu používatelia využívať všetky služby objektu.

12. Objekty s funkčne spriaznenými atribútmi a metódami patria do rovnakej triedy objektov. Triedu objektov môžeme charakterizovať ako množinu funkčne ekvivalentných objektov. Objekty sú inštanciami rovnakej triedy[3]. Každý objekt triedy má svoje vlastné atribúty a metódy[4].

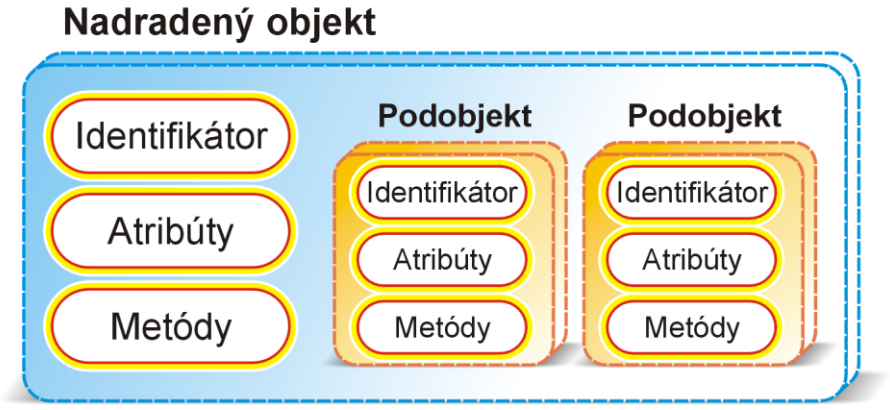
13. Objekty môžu dediť vlastnosti a činnosti od iných objektov. To sa deje v procese dedičnosti, kedy potomkovia preberajú charakteristické črty svojich rodičov. Triedy objektov môžu vytvárať rôzne varianty dedičnosti. Ak potomok dedí svoje schopnosti len od jedného rodiča, ide o jednoduchú dedičnosť. Naopak, keď potomok zdedil svoje schopnosti od viacerých rodičov súčasne, vravíme o viacnásobnej dedičnosti. Potomkovia však nie sú odkázaní len na tie schopnosti, ktoré zdedili po svojich rodičoch. K už nadobudnutým (zdedeným) vlastnostiam a činnostiam môžu pridávať nové vlastnosti a činnosti. Z hierarchie objektov, ktoré vzniknú na báze dedičnosti, je zrejmé, že potomkovia sú vždy prinajmenšom tak funkčne vyspelí ako ich rodičia. Spravidla sú však potomkovia vyspelejší než ich predkovia, pretože majú možnosť rozšíriť aparát svojich schopností. To nás privádza k zaujímavej a vskutku jedinečnej vlastnosti objektovo orientovaného programovania, z ktorej vyplýva, že potomok sa môže vyskytnúť všade tam, kde je očakávaný jeho rodič. Je teda možné uskutočniť substitúciu rodiča potomkom bez vzniku kolíznych stavov. Podotknime, že

[3] Aj keď všeobecná teória objektovo orientovaného programovania nešpecifikuje triedu ako základný predpis na vytváranie objektov, hybridné programovacie jazyky s najväčšou penetráciou na trhu takúto špecifikáciu uplatňujú. V ich prostredí je nutné najskôr deklarovať triedu, ktorá determinuje fyzickú reprezentáciu a možnosti použitia budúcich objektov (inštancií danej triedy).

[4] V konkrétnej implementácii objektovo orientovaného programovania v hybridnom programovacom jazyku je situácia odlišná. Hoci každý objekt disponuje svojou vlastnou súpravou atribútov, metódy objektov totožnej triedy sú v operačnej pamäti uložené práve jedenkrát. Samotná činnosť metódy potom prebieha tak, že metóda si po svojej aktivácii vyhľadá objekt, v súvislosti s ktorým bude spracovaná.

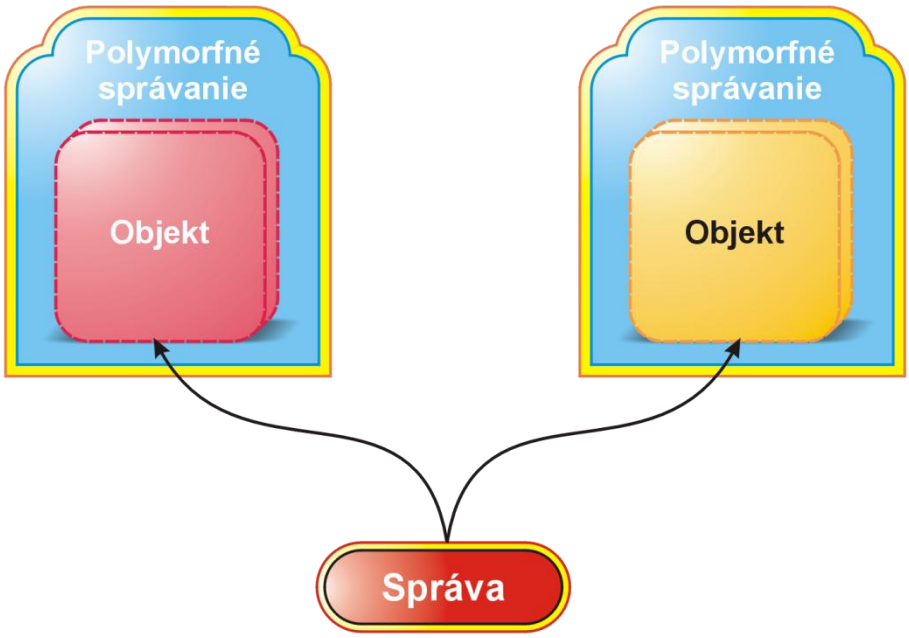
naopak tento proces nefunguje. Totiž tam, kde je očakávaný potomok, nemôže byť dosadený rodič tak, aby nedošlo k vzniku chybových stavov. Je to preto, že rodič nie je schopný v plnej miere zastúpiť svojho potomka, pretože ten môže byť funkčne vyspelejší ako on.

14. Vzťahy medzi objektmi nemusia byť generované len na základe dedičnosti. Odhliadnuc od dedičnosti, je možné modelovať väzby medzi objektmi aj pomocou asociatívnych relácií. K týmto reláciám patrí agregácia a kompozícia. Ich podstata spočíva v tom, že objekt môže vzniknúť zložením z iných objektov. V tomto kontexte rozlišujeme hlavný (nadradený) objekt a množinu podobjektov, ktoré nadradený objekt obsahuje. Agregačno-kompozičné vzťahy sa využívajú predovšetkým pri konštrukcii zložitých objektov, ktoré vykonávajú širokú paletu činností. Nadradený objekt potom môže delegovať právomoci na vykonávanie istých činností na rôzny počet vnorených objektov. Podľa sily väzby, aká panuje medzi nadradeným objektom a podobjektmi, sú ich vzťahy definované pomocou agregácie alebo kompozície. Vo všeobecnosti, kompozícia reprezentuje silnejšiu asociatívnu väzbu, pričom vraví, že nadradený objekt nemôže poskytovať svojim používateľom všetky služby v zodpovedajúcej kvalite, ak je aspoň jeden vnorený objekt nefunkčný, resp. absentujúci. Kým pri kompozícii nemôžu podobjekty pracovať samostatne bez nadradeného objektu, agregácia tento spôsob použitia podobjektov umožňuje. Analogicky, ako pri už zmienených komunikačných modeloch, aj nadradený objekt komunikuje s vnorenými objektmi pomocou mechanizmu správ. Všetky vnorené objekty disponujú svojimi protokolmi správ, ktoré formujú ich verejne prístupné rozhrania.



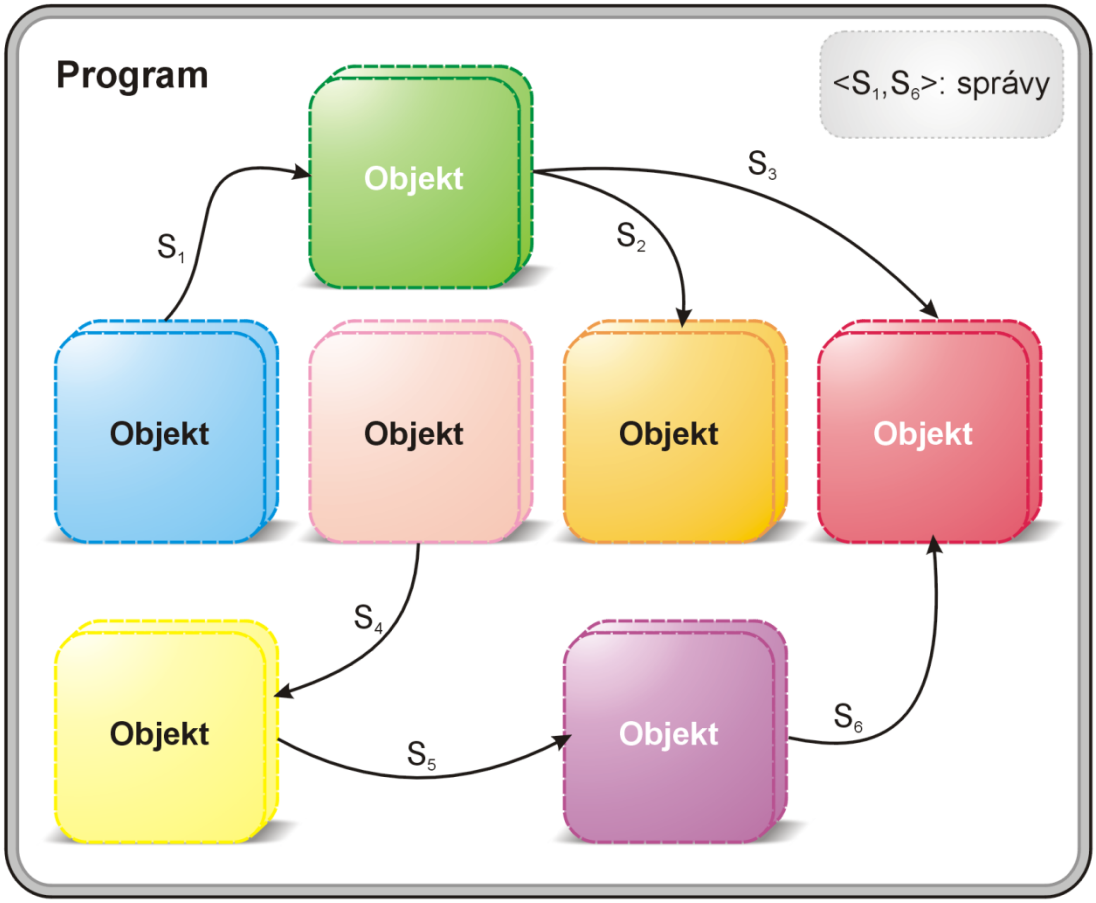
Obr. 7: Agregačno-kompozičné vzťahy medzi objektmi

15. Ak dva objekty reagujú na zaslanie totožnej správy odlišným spôsobom, hovoríme, že sa správajú polymorfne. Polymorfizmus je aspekt objektovo orientovaného programovania, ktorý úzko súvisí s dedičnosťou a umožňuje vzájomnú substitúciu objektov (potomkovia sú schopní zastúpiť svojich predkov).



Obr. 8: Polymorfné správanie inštancií tried

Vizualizáciu programu, ktorý bol vytvorený v rýdzo objektovo orientovanom programovacom jazyku, uvádza obr. 9.



Obr. 9: Rýdzo objektovo orientovaný program

1.1.2.1 Hybridné a objektovo orientované programovacie jazyky

Väčšina v praxi rozšírených programovacích jazykov patrí do kategórie hybridných programovacích jazykov, pretože v sebe kombinuje možnosti pre vývoj štruktúrovaného a objektovo orientovaného počítačového softvéru. Pri niektorých jazykoch je ich zaradenie do tejto kategórie dané požiadavkou na zachovanie spätnej kompatibility. K hybridným programovacím jazykom patria C#, C++, Visual Basic, Delphi a Java. Popri hybridných programovacích jazykoch existujú aj rýdzo objektovo orientované jazyky, ku ktorým patria najmä dynamické jazyky riešiace úlohy pre potreby umelej inteligencie, simulácie, počítačovej grafiky atď. Množinu rýdzo objektovo orientovaných programovacích jazykov tvorí Smalltalk, CLOS, Eiffel, ESP a Object-Prolog.

Program vytvorený v rýdzo objektovo orientovaných jazykoch nie je závislý od hlavnej funkcie, resp. metódy, ktorá zahajuje spracovanie programov vytvorených v hybridných programovacích jazykoch. V rýdzo objektových programoch sa exekúcia začína zaslaním správy z rozhrania programu objektom, ktoré na túto správu príslušným spôsobom reagujú.

1.1.3 Komponentové programovanie

Komponentové programovanie je modulárnou nadstavbou objektovo orientovaného programovania. Komponentové programovanie využíva celý aparát objektovo orientovaného programovania, pričom pomocou objektov vytvára komponenty, ktoré charakterizujeme ako softvérové jednotky vyššieho kvalitatívneho stupňa. Základný princíp komponentového programovania spočíva vo vytváraní komponentov, ktoré sú natoľko inteligentné, aby dokázali na seba prebrať zodpovednosť za realizáciu množiny príbuzných činností aplikácie, resp. počítačového systému.

Komponentové programovanie vychádza z premisy, že softvér by mal byť vytváraný presne tak, ako produkty priemyselnej výroby, ktoré sa vyrábajú hromadne v pásovej výrobe. Veľmi obľúbená je analógia najmä s automobilovým priemyslom, kde sa autá vyrábajú na výrobnej linke. Výrobný proces je značne komplikovaný, no je dobre štruktúrovaný do jednotlivých výrobných štádií. V každom štádiu sa podoba finálneho výrobku mení v závislosti od komponentov, ktoré sú do neho inštalované. Ak je možné vyrobiť auto ako umne navrhnutú a implementovanú kombináciu niekoľkých tisícov rôznych technických komponentov, prečo by podobným štýlom nemohol byť produkovaný aj počítačový softvér? Komponentové programovanie vraví, že aj v softvérovom svete možno zhotoviť samostatné programové súčiastky, z ktorých sa potom skonštruuje finálny program. Aby bolo možné naplniť túto ideu, musia byť prijaté nasledujúce postuláty:

1. Aplikácia (A) je konečnou neprázdnou množinou komponentov (K), ktorú môžeme matematicky definovať takto:

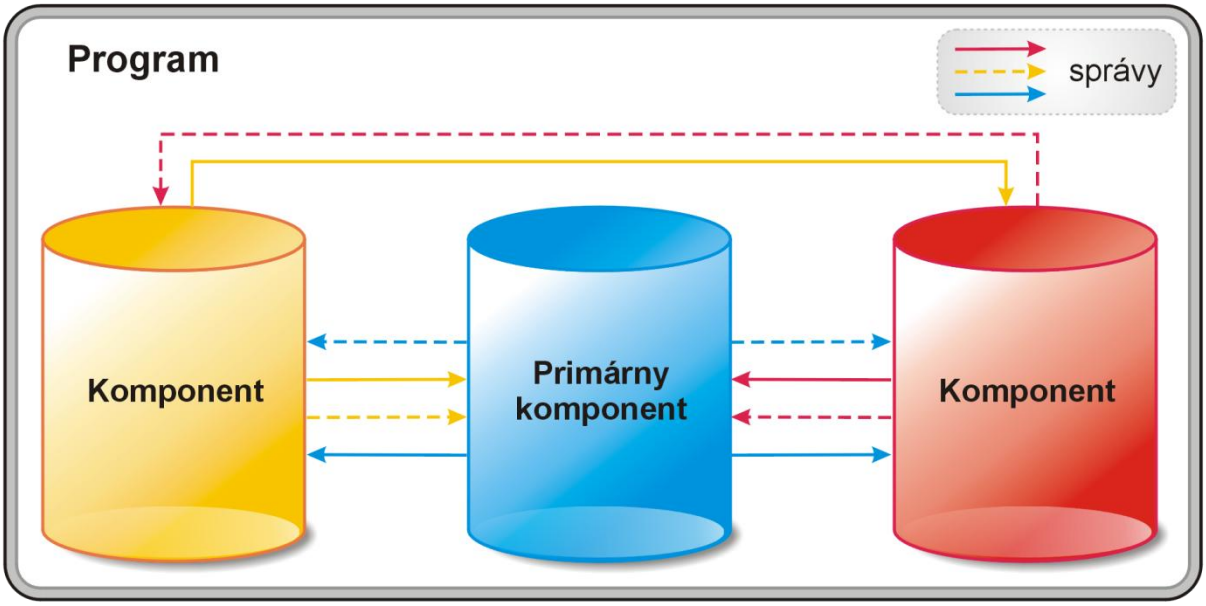
$$A = K = \{k_1, k_{2,} \dots, k_n\}$$

2. Jeden z komponentov aplikácie zaujíma stanovisko primárneho komponentu. Primárny komponent je bázový komponent, ktorý zabezpečuje jadrovú funkcionalitu komponentovej aplikácie. Ak by aplikácia nedisponovala primárnym komponentom, nemohla by fungovať.

3. Okrem primárneho komponentu môže byť aplikácia tvorená ľubovoľnou prípustnou množinou ďalších (komplementárnych) komponentov. Každý z komplementárnych komponentov je zodpovedný za vykonávanie určitej množiny činností aplikácie. Ak odhliadneme od nutnej prítomnosti primárneho komponentu, tak môžeme konštatovať, že finálna podoba aplikácie smie byť značne variabilná, a teda konfigurovateľná. Keďže komponenty sú projektované na modulárnej báze, môžeme vytvoriť rozmanitú kolekciu komplementárnych komponentov. Modularita komponentov je výhodná nielen z pohľadu vývojárov, ale aj cieľových používateľov. V závislosti od svojich potrieb môžu používatelia priamo ovplyvniť konečnú komponentovú skladbu aplikácie.

   Softvérové firmy často produkujú rôzne verzie komponentových aplikácií, ktoré sa odlišujú množstvom inštalovaných komponentov. Uvažujme trebárs aplikáciu na spracovanie grafických obrazov, ktorá v základnej verzii obsahuje dva komponenty: primárny komponent vykonávajúci načítavanie a zobrazovanie bitových máp rôznych grafických formátov a jeden komplementárny komponent realizujúci grafické transformácie, akými sú inverzia bitovej mapy, prevod bitovej mapy do sivotónu a prahovanie bitovej mapy. Tvorca komponentovej aplikácie však môže dodávať aj iné komplementárne komponenty, ktoré si používateľ môže zakúpiť (napr. komponent umožňujúci vytvárať z kolekcií bitových máp videosúbory). Používateľ sa môže rozhodnúť, ktoré komponenty zakúpi a ktoré nie. Softvérová firma môže zasa kategorizovať komponentové aplikácie podľa počtu implementovaných komponentov (napr. základná, pokročilá či luxusná verzia komponentovej aplikácie).

   Vizuálnu podobu komponentovej aplikácie môžeme vidieť na obr. 10.

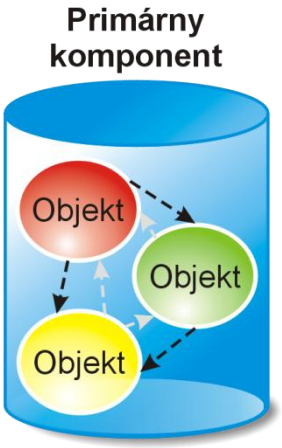


Obr. 10: Komponentová aplikácia

4. Každý komponent je charakterizovaný týmito vlastnosťami:

- **Identifikácia**. Komponent musí byť jednoznačne identifikovateľný (a to tak v aplikácii, ako aj v informačnom a operačnom systéme). Vďaka jednoznačnej identifikácii môžu existovať dva rovnaké komponenty, ktoré sa líšia len svojimi identifikačnými číslami. Podotknime, že v praktických podmienkach je identifikačné číslo komponentu často zhodné s číslom verzie komponentu.

- **Zapuzdrenie.** Komponent zapuzdruje všetku požadovanú funkcionalitu v sebe. Hoci z vonkajšieho pohľadu to nie je zrejmé, vo svojom vnútri obsahuje komponent súpravu objektov s delegovanými právomocami pre vykonanie špecifikovaných činností (obr. 11). S objektmi komunikuje komponent pomocou mechanizmu zasielania a prijímania správ.



Obr. 11: Interná kompozícia primárneho komponentu

- **Rozhranie**. Podobne ako objekty, aj komponenty majú svoje verejne prístupné rozhrania, pomocou ktorých využívajú používatelia ich služby. Pri funkčne spriaznených komponentoch je rozhranie štandardizované, unifikované a typizované. Komponent rovnako ukrýva dáta a implementáciu činností, ktoré vykonáva. Všetky tieto aspekty sú pred vonkajším svetom vhodne ukryté.

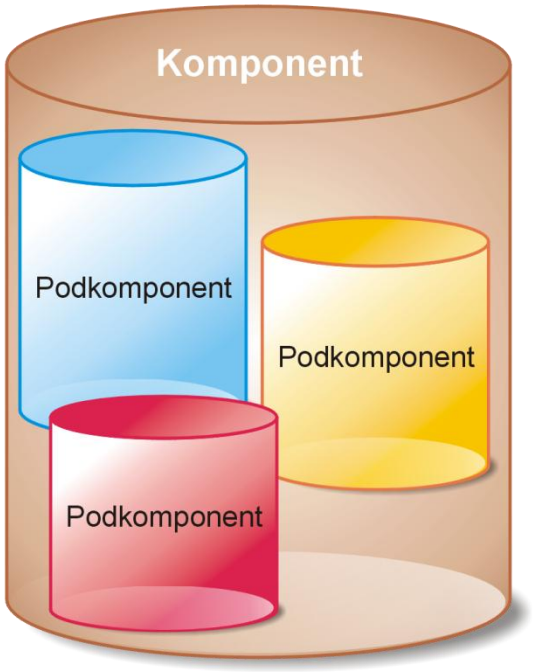
- **Pripravenosť na použitie.** Komponent pôsobí za každých okolností ako hotová softvérová súčiastka, ktorá poskytuje komplexnú automatizáciu vybraných činností. Za predpokladu, že klient má ku komponentu prístup, môže okamžite začať využívať jeho služby prostredníctvom verejne prístupného rozhrania.

- **Opätovná použiteľnosť.** Ak je komponent vyvinutý, odladený, otestovaný a optimalizovaný, môžeme ho použiť toľkokrát, koľkokrát potrebujeme. Komponentové programovanie v tomto smere ďalej rozvíja myšlienku objektového programovania, ktorej zmyslom je značný nárast pracovnej produktivity vývojárov pomocou opätovnej použiteľnosti raz vytvorenej softvérovej entity.

- **Anonymita používateľov.** Funkcionalita komponentu je úplne oddelená od jeho aplikácie. Komponent musí poskytovať služby v zodpovedajúcom čase a kvalite akémukoľvek používateľovi, ktorý má záujem komponent použiť. Keďže neexistuje žiadna fixácia na používateľov, báza praktického využitia komponentu je veľmi široká.

- **Interoperabilita.** Komponenty musia byť schopné medzi sebou spolupracovať aj vtedy, keď boli vytvorené v rôznych integrovaných vývojových prostrediach a programovacích jazykoch. V záujme uplatnenia bezproblémovej komunikácie medzi komponentmi musí byť interoperabilita zabezpečená na nízkej, spravidla binárnej úrovni.

5. Vybrané komponenty smú vytvárať celky, ktoré sa sami osebe môžu stať komponentmi. Proces skladania komponentov pracuje na podobných princípoch ako agregácia, resp. kompozícia objektov v objektovo orientovanom programovaní. Identifikujeme teda nadradený komponent a súpravu podkomponentov (obr. 12).



Obr. 12: Skladanie komponentov

2. tematický celok

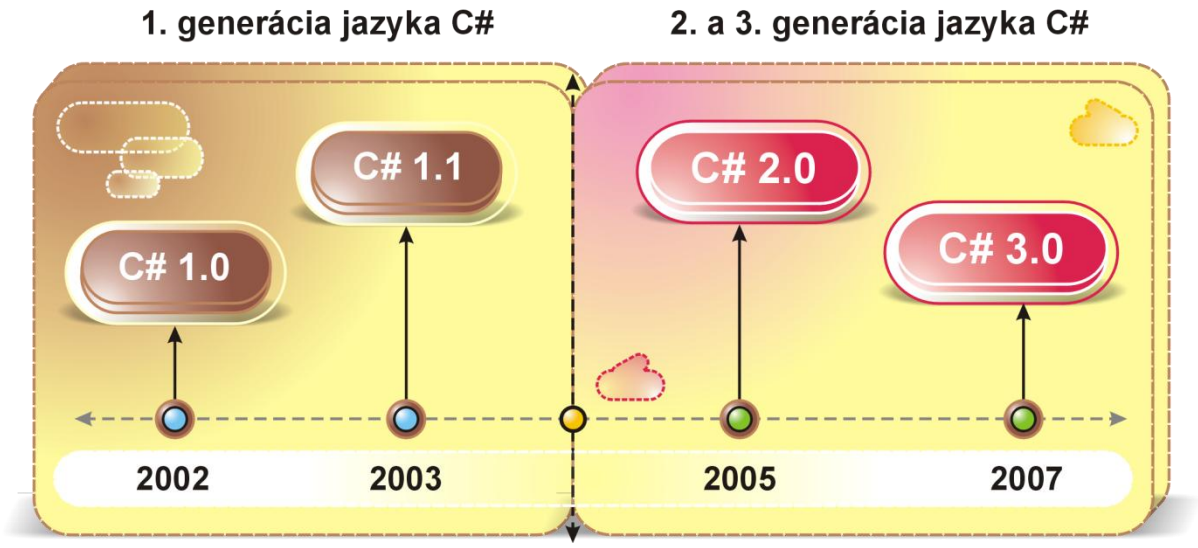
Objektovo orientované programovanie v jazyku C# 3.0

Microsoft® Visual Studio® 2008

2. Objektovo orientované programovanie v jazyku C# 3.0

Programovací jazyk C# bol vytvorený spoločnosťou Microsoft, ktorá jeho prvú verziu uviedla na softvérový trh v roku 2002. C# je hybridným programovacím jazykom, pretože implementuje syntakticko-sémantické konštrukcie štruktúrovaného i objektovo orientovaného programovania. Jazyk C# bol vytvorený špeciálne pre potreby vývojovo-exekučnej platformy Microsoft .NET Framework, ktorá ponúka vývojárom metodiky, nástroje a technológie pre vytváranie rozmanitých typov počítačových aplikácií určených pre operačné systémy Windows a web. Prvá verzia programovacieho jazyka C# bola implementovaná v produkte Microsoft Visual Studio .NET. Po nej nasledovali ďalšie verzie s číselným označením C# 1.1 (Visual Studio .NET 2003), C# 2.0 (Visual Studio 2005) a zatiaľ najaktuálnejšia verzia C# 3.0 (Visual Studio 2008). Programovací jazyk C# bol štandardizovaný organizáciami ISO a ECMA[5], podobne ako aj základné komponenty vývojovo-exekučnej platformy Microsoft .NET Framework.

Genézu programovacieho jazyka C# zobrazuje obr. 13.



Obr. 13: Vývoj programovacieho jazyka C#

2.1 Trieda ako abstraktný objektový dátový typ

Všetky hybridné programovacie jazyky vytvárajú objekty z tried, ktoré sú v týchto prostrediach považované za abstraktné objektové dátové typy. Z hľadiska klasifikácie dátových typov v jazyku C# 3.0 (a rovnako aj v spoločnom typovom systéme vývojovo-exekučnej platformy Microsoft .NET Framework 3.5), sú triedy zaraďované k odkazovým objektovým dátovým typom. Trieda predstavuje abstraktný dátový typ,

[5] Štandard ISO: ISO/IEC 23270:2006 Information Technology - Programming Languages -- C#
Štandard ECMA: ECMA - 334 C# - Language Specification

ktorého deklarácia špecifikuje vlastnosti a činnosti objektov, ktoré z tejto triedy vzniknú. Triedy sú rovnako používateľsky deklarovanými dátovými typmi, čo znamená, že vývojári musia najskôr zaviesť deklarácie tried a až potom ich môžu inštanciovať. V tomto smere sa triedy odlišujú od primitívnych dátových typov, ktoré sú priamo vstavané do jazykovej špecifikácie C# 3.0. Primitívne dátové typy sú pre kompilátor známe a okamžite použiteľné.

2.1.1 Deklarácia triedy

Trieda ako abstraktný, objektový a odkazový používateľsky deklarovaný dátový typ sa v jazyku C# 3.0 deklaruje podľa tohto generického syntaktického modelu:

```
class T
{
   ...
}
```

kde:

- **T** je identifikátor triedy[6].

Deklarácia triedy je zložená z hlavičky triedy a jej tela. V hlavičke sa musí nachádzať kľúčové slovo **class**, podľa ktorého dávame kompilátoru na známosť, že sa chystáme deklarovať triedu ako nový používateľsky deklarovaný odkazový dátový typ. Za kľúčovým slovom **class** je umiestnený identifikátor triedy, ktorý predstavuje jednoznačné pomenovanie triedy v danej oblasti platnosti. Aby sme predišli potenciálnym menným konfliktom, môžeme deklaráciu triedy vložiť do vopred pripraveného menného priestoru. Pokiaľ nie je určené inak, deklarovaná trieda má interné prístupné práva. V jazyku C# 3.0 je triede implicitne priradený prístupový modifikátor **internal**, ktorý vymedzuje viditeľnosť a oblasť platnosti triedy. Pri aplikovanom modifikátore **internal** je trieda prístupná pre všetky ostatné dátové typy (či už hodnotové, alebo odkazové), ktoré sú situované v danom zostavení aplikácie .NET.

Aj keď sme sa v našom generickom syntaktickom modeli spoľahli na implicitne internú viditeľnosť deklarovanej triedy, je potrebné dodať, že prístupový modifikátor môže byť zadaný aj explicitne. Okrem modifikátora **internal** môžu vývojári deklarovať aj verejne prístupné triedy (modifikátor **public**). Verejne prístupná trieda je viditeľná pre akýkoľvek externý, a teda klientsky zdrojový kód.

[6] Identifikátor triedy, podobne ako aj identifikátory iných programových entít, musia spĺňať základné nomenklatúrne pravidlá pre pomenovanie. Identifikátor sa nesmie začínať číslicou, nemôže obsahovať medzery a taktiež sa nemôže zhodovať s niektorým z kľúčových slov jazyka C# 3.0.

Telo triedy je ohraničené zloženými zátvorkami ({}), v ktorých sa nachádzajú definície členov triedy. Jazyk C# 3.0 podporuje mnoho programových entít, ktoré môžu vystupovať ako členovia triedy. K základným členom triedy patria atribúty (dátové položky) a metódy. Okrem nich však smú členmi triedy byť tiež vlastnosti či delegáti. Podľa predstaveného generického syntaktického modelu môžeme v jazyku C# 3.0 navrhnúť nasledujúcu deklaráciu triedy:

```
// Deklarácia triedy.
class Zamestnanec
{
    // Definície atribútov (dátových položiek).
    private string meno;
    private string priezvisko;
    private int mzda;
    private int odpracovanéRoky;
    // Definícia konštruktora.
    public Zamestnanec(string meno, string priezvisko, int mzda, int roky)
    {
        this.meno = meno;
        this.priezvisko = priezvisko;
        this.mzda = mzda;
        odpracovanéRoky = roky;
    }
    // Definície prístupových metód.
    public string PrečítaťMeno()
    {
        return meno;
    }
    public void NastaviťMeno(string meno)
    {
        this.meno = meno;
    }
    public string PrečítaťPriezvisko()
    {
        return priezvisko;
    }
    public void NastaviťPriezvisko(string priezvisko)
    {
        this.priezvisko = priezvisko;
    }
    public int ZískaťMzdu()
    {
        return mzda;
    }
    public void NastaviťMzdu(int mzda)
    {
        this.mzda = mzda;
    }
    public int ZískaťRoky()
    {
        return odpracovanéRoky;
    }
    public void NastaviťRoky(int roky)
    {
        odpracovanéRoky = roky;
    }
}
```

Trieda deklaruje abstraktný typ **Zamestnanec**, ktorý je charakterizovaný svojimi atribútmi a metódami. Atribúty sú implementované definíciami súkromných dátových položiek. Dátové položky musia byť súkromné, pretože musíme dodržať jeden zo základných princípov objektovo orientovaného programovania, ktorým je ukrývanie dát. Pre naše potreby sledujeme u zamestnanca nasledujúce atribúty: meno, priezvisko, mzdu a počet odpracovaných rokov. Všetky uvedené atribúty sú v relácii typu 1:1 namapované na príslušné dátové položky. Z definičných príkazov je zrejmé, že meno a priezvisko zamestnanca budeme uchovávať v podobe textových reťazcov, ktoré budú uskladnené v inštanciách systémovej triedy **System.String**. Na uloženie mzdy a počtu odpracovaných rokov nám poslúžia dátové položky primitívneho integrálneho hodnotového typu **int**.

Po definícii dátových položiek, ktoré tvoria dátovú sekciu triedy, prichádza na rad konštruktor. Konštruktor je špeciálna metóda triedy, ktorá je zodpovedná za korektnú inicializáciu jej zakladaných inštancií. Zo syntaktického hľadiska je konštruktor verejne prístupnou metódou a v našom prípade aj parametrickou metódou. Identifikátor konštruktora sa zhoduje s identifikátorom triedy, v tele ktorej je konštruktor definovaný. Konštruktor nevracia žiadnu návratovú hodnotu (v hlavičke konštruktora nesmie byť použité ani kľúčové slovo **void**, návratovú hodnotu jednoducho vôbec nešpecifikujeme). Podotknime, že analyzovaný konštruktor je inštančný. Adjektívum „inštančný“ znamená, že úlohou konštruktora je uviesť do východiskového (a teda okamžite použiteľného) stavu dátové položky každej jednej zrodenej inštancie triedy. Považujeme za vhodné poukázať na inštančnú povahu vyššie definovaného konštruktora, pretože jazyk C# 3.0 nám dovoľuje okrem inštančného konštruktora definovať aj statický konštruktor[7].

Ak by v tele triedy nebol explicitne definovaný žiaden inštančný konštruktor, kompilátor jazyka C# 3.0 by vygeneroval implicitný inštančný konštruktor. Implicitný inštančný konštruktor je verejne prístupný a bezparametrický. Kompilátor jazyka C# 3.0 garantuje, že implicitný inštančný konštruktor zabezpečí implicitnú inicializáciu všetkých dátových položiek inštancie triedy. Pokiaľ do tela triedy istý inštančný konštruktor explicitne zavedieme, kompilátor sa nebude zaoberať zostavovaním jeho implicitnej verzie. Za týchto okolností sme my zodpovední za správnu inicializáciu príslušných dátových položiek inštancie triedy.

V tele verejne prístupného parametrického inštančného konštruktora dochádza k inicializácii inštančných dátových položiek, ktoré sme definovali v dátovej sekcii

[7] V hlavičke statického konštruktora sa musí nachádzať modifikátor **static**. Zatiaľ čo inštančný konštruktor sa sústreďuje na inicializáciu dátových položiek inštancií triedy, jeho statický náprotivok sa venuje inicializácii statických dátových položiek triedy. Statické dátové položky nie sú asociované s jednotlivými inštanciami triedy, ale s triedou samotnou. Medzi inštančným a statickým konštruktorom existujú aj ďalšie rozdiely. Napríklad, hoci inštančný konštruktor sa môže stať predmetom preťaženia, pri statickom konštruktore to nie je možné. V skutočnosti musí byť statický konštruktor vždy bezparametrický a nesmie uvádzať žiaden prístupový modifikátor.

triedy. Ktorákoľvek z dátových položiek je priamo dosiahnuteľná prostredníctvom svojho identifikátora. Na dátovú položku sa môžeme odvolať rovnako použitím kľúčového slova **this**, za ktorým nasleduje operátor priameho prístupu (.) a identifikátor požadovanej dátovej položky. Týmto spôsobom vieme vždy jednoznačne určiť, ktorá entita predstavuje dátovú položku a ktorá formálny parameter s inicializačnou hodnotou. Technicky povedané, kľúčové slovo **this** je identifikátorom špeciálnej odkazovej premennej, v ktorej je uložená objektová referencia determinujúca aktuálnu inštanciu triedy. Aktuálna inštancia triedy je za každých okolností práve jedna inštancia triedy, v súvislosti s ktorou je vykonávaná istá činnosť (napr. inicializácia jej dátových položiek, volanie metódy a pod.). Ako si môžeme všimnúť, v tele konštruktora sú inicializované všetky dátové položky.

Keďže objekt je logická jednotka zachovávajúca konzistenciu a integritu svojich dát, nemôže umožniť externému zdrojovému kódu svojvoľný prístup k dátam objektu. Pritom však chceme navrhnúť spôsob, akým by používateľ mohol získať o objekte požadované informácie. Týmto variantom sú verejné prístupové metódy, ktoré umožňujú vopred naprogramovaný, overený a najmä bezpečný prístup k dátam objektu. Ku každej inštančnej dátovej položke preto navrhujeme súpravu dvoch prístupových metód. Zatiaľ čo jedna z prístupových metód sa sústreďuje na získanie hodnoty dátovej položky, druhá sa venuje jej modifikácii.

2.1.2 Vizualizácia deklarácie triedy a Návrhár tried (Class Designer)

Integrované vývojové prostredie (Integrated Development Environment, IDE) Microsoft Visual Studio 2008 umožňuje vizualizovať akúkoľvek správne deklarovanú triedu. Pre automatické vygenerovanie súboru s grafickým modelom triedy musíme urobiť toto:

1. V podokne **Solution Explorer** klikneme pravým tlačidlom myši na zdrojový súbor jazyka C# 3.0 s deklaráciou triedy (*.cs), ktorej vizuálny model chceme zostaviť.

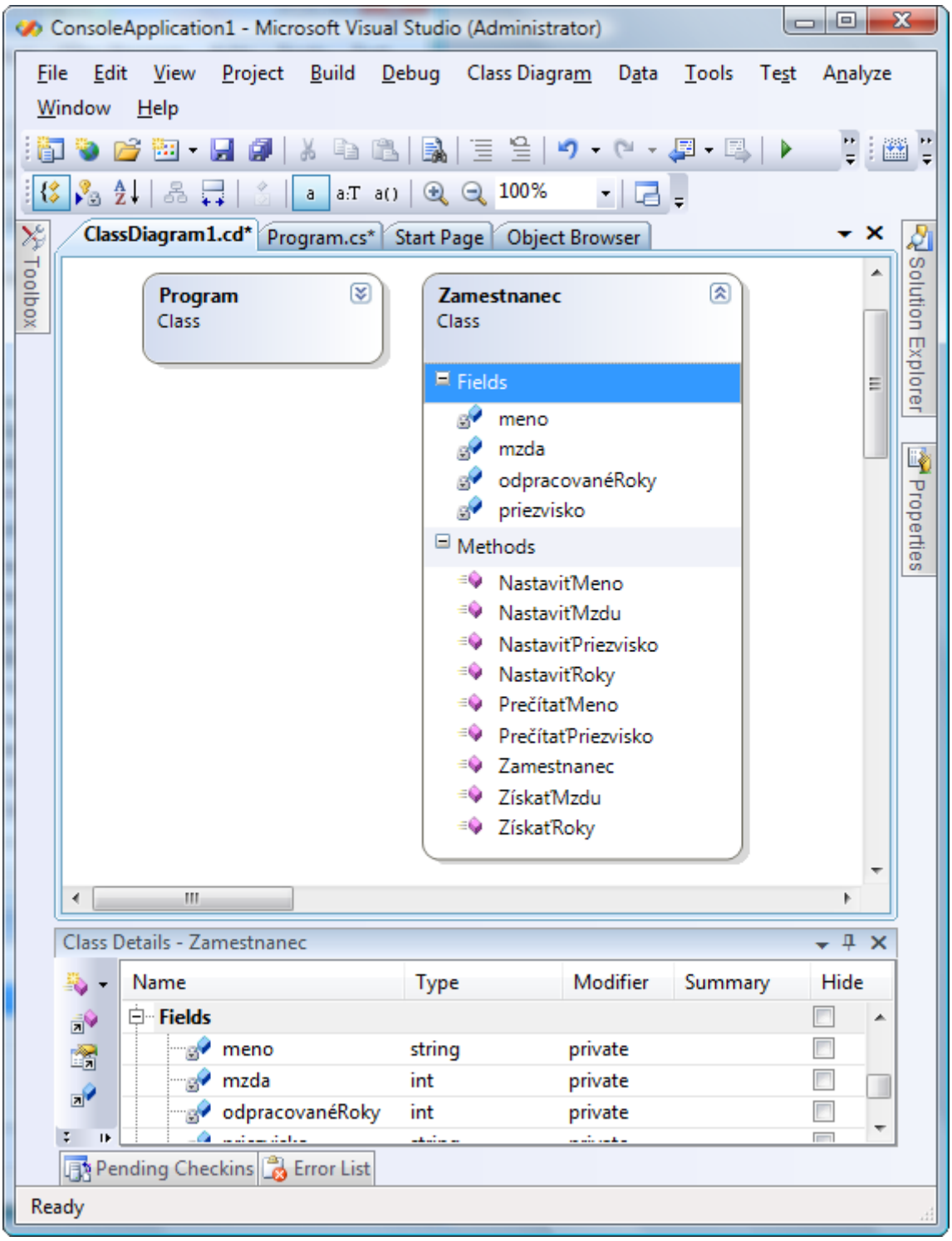
2. Z kontextovej ponuky vyberieme príkaz **View Class Diagram**.

3. Prostredie IDE vytvorí nový súbor s extenziou *.cd a vloží doň grafický model, ktorý vizuálne charakterizuje deklarovanú triedu a všetky jej členy.[8]

Grafická podoba predstavuje tzv. diagram triedy (Class Diagram). Diagramy tried zostavuje špeciálny nástroj integrovaného vývojového prostredia Visual Studio 2008, ktorý sa volá Návrhár tried (Class Designer).

[8] V automaticky zostavenom diagrame tried budú prítomné modely všetkých hodnotových a odkazových dátových typov, ktoré sa v príslušnom zdrojovom súbore jazyka C# 3.0 nachádzajú.

Diagram triedy **Zamestnanec** je uvedený na obr. 14.



Obr. 14: Diagram triedy reprezentuje vizuálny model deklarovanej triedy

Diagram triedy **Zamestnanec** je na obr. 14 zobrazený v tzv. rozšírenom režime, kedy vidíme kompletné zloženie členov deklarovanej triedy. Na druhú stranu, diagram tried smie byť zobrazený aj v kompaktnom režime (podotknime, že diagram primárnej triedy Program je zobrazený v kompaktnom režime). Medzi oboma zobrazeniami sa môžeme

prepínať pomocou tlačidla s ikonou dvojitej šípky (spomínané tlačidlo sa nachádza v pravom hornom rohu diagramu tried).

Z grafického hľadiska sa diagram tried veľmi ponáša na diagram tried unifikovaného modelovacieho jazyka UML. Podobne ako v jazyku UML, aj v prostredí produktu Visual Studio 2008 je diagram tried zložený z troch základných častí:

1. Identifikačná časť (podáva informácie o názve triedy).
2. Dátová časť (podáva informácie o dátových členoch triedy).
3. Procedurálna časť (podáva informácie o metódach triedy).

Návrhár tried umožňuje dvojcestné modelovanie. Tak, ako je možné z deklarácie triedy vytvoriť grafický diagram, môžeme postupovať aj opačným smerom. Najskôr navrhneme pomocou jednoduchých grafických nástrojov a pridružených konfiguračných prostriedkov diagram triedy, a potom necháme Návrhára tried, aby vygeneroval syntakticko-sémanticky korektnú deklaráciu triedy podľa navrhnutého grafického modelu. Samozrejme, je na vývojárovi, aby do tela triedy doplnil zdrojový kód, ktorý bude zavádzať požadovanú aplikačnú logiku.

Okrem deklarácií tried smieme vizualizovať aj vzťahy medzi triedami, ktoré sú založené na agregácii, kompozícii a dedičnosti.

2.1.3 Inštanciácia triedy a použitie zrodenej inštancie

Deklarovaním triedy vyjadrujeme naše požiadavky, ktoré sa týkajú vlastností a schopností inštancií, ktoré z tejto triedy vzniknú. Proces zrodu inštancie triedy (objektu) sa volá inštanciácia triedy. Inštanciácia sa člení na niekoľko štádií, s ktorými sa v tejto kapitole zoznámime. Predovšetkým ale musíme vyhlásiť, že trieda je v jazyku C# 3.0 inštanciovaná operátorom **new**. Generická syntaktická podoba inštanciácie triedy vyzerá takto:

```
T obj = new T();
```

kde:

- **T** je identifikátor triedy.
- **obj** je identifikátor odkazovej premennej.

Z akademických skúseností plynie, že uvedený inštanciačný príkaz je lepšie pochopiteľný, ak naň nahliadame ako na definičnú inicializáciu odkazovej premennej s identifikátorom **obj**. Keďže ide o priraďovací príkaz, môžeme ho vizuálne rozdeliť na tri časti:

1. Výraz nachádzajúci sa na ľavej strane od priraďovacieho operátora (=).
2. Priraďovací operátor.
3. Výraz nachádzajúci sa na pravej strane od priraďovacieho operátora.

Začnime analýzou výrazu naľavo od operátora =. Výraz **T obj** predstavuje definíciu odkazovej premennej s identifikátorom **obj**. Takto definovaná premenná bude alokovaná na zásobníku programového vlákna. Keďže typom definovanej odkazovej premennej je trieda, vieme už v tejto chvíli povedať, čo môže byť do odkazovej premennej uložené: bude to objektová referencia (alebo odkaz), ktorá je nasmerovaná na inštanciu triedy **T**.

Vzhľadom na to, že ide o definičnú inicializáciu odkazovej premennej **obj**, je táto premenná po svojej definícii okamžite inicializovaná. Inicializačnou hodnotou odkazovej premennej je hodnota výrazu, ktorý sa nachádza napravo od priraďovacieho operátora. Výraz **new T()** znamená použitie operátora **new** v súvislosti s identifikátorom deklarovanej triedy **T**. Operátor **new** realizuje inštanciáciu špecifikovanej triedy, pričom vykoná tieto činnosti:

1. Na riadenej halde vyhľadá a alokuje dostatočný pamäťový priestor pre novo zakladanú inštanciu triedy **T**.[9]

2. Vytvorí inštanciu triedy, umiestni ju do vopred pripraveného alokačného priestoru a inicializuje ju. Inicializáciu realizuje verejne prístupný inštančný konštruktor, ktorý je operátorom **new** automaticky aktivovaný. Je zrejmé, že v generickom syntaktickom inštanciačnom príkaze je za inicializáciu založenej inštancie triedy zodpovedný verejne prístupný bezparametrický inštančný konštruktor. Inicializáciu však samozrejme môže uskutočňovať aj parametrický inštančný konštruktor. Potom by však výraz na pravej strane priraďovacieho príkazu vyzeral takto: **new T(a_1, a_2, ..., a_n)**, kde v zátvorkách sú zoskupené argumenty, ktoré budú odovzdané formálnym parametrom parametrického inštančného konštruktora[10]. Po spracovaní konštruktora sa inštancia triedy nachádza v aktívnom, a teda priamo použiteľnom stave.

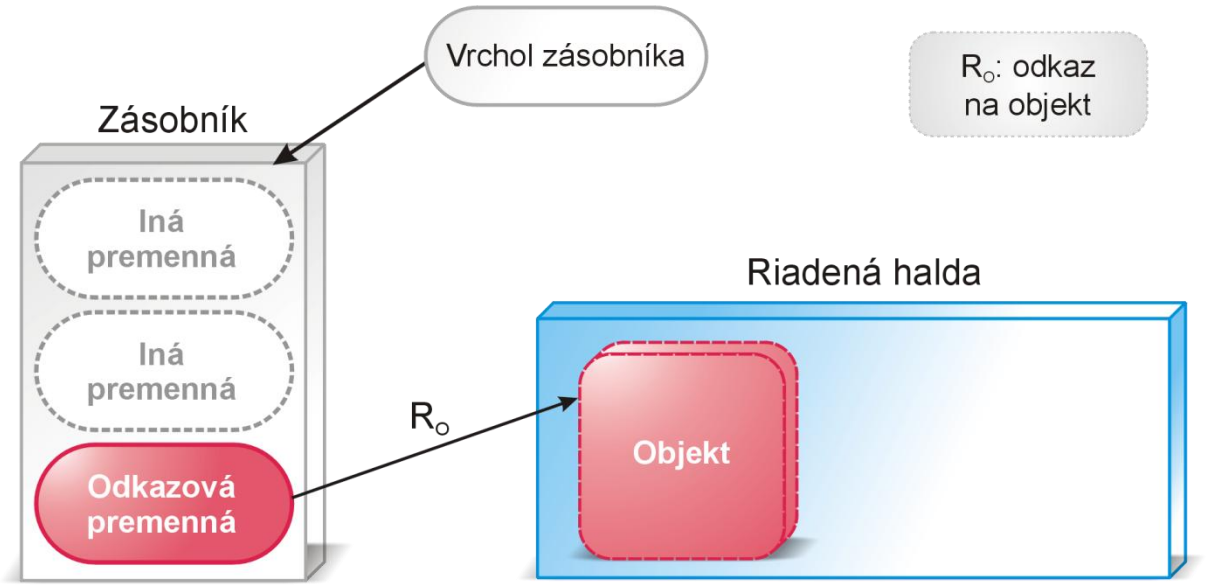
3. Vráti typovo silnú objektovú referenciu na vytvorenú inštanciu triedy. V tejto súvislosti je nutné poukázať na skutočnosť, že založená inštancia triedy nachádzajúca sa na riadenej halde je anonymná (nedisponuje žiadnym symbolickým pomenovaním). Jediným spôsobom, ako s inštanciou triedy

[9] Riadená halda je vyhradený dynamický pamäťový priestor fyzického procesu aplikácie .NET, ktorý je určený na alokáciu inštancií používateľsky deklarovaných odkazových dátových typov. Riadená halda je interne členená na tri objektové generácie (0, 1 a 2), v ktorých sa nachádzajú objekty v závislosti od dĺžky svojich životných cyklov. Objekty umiestnené na riadenej halde sú pod kontrolou automatického správcu pamäte.

[10] Zapísaný inštanciačný výraz predpokladá existenciu relácie typu 1:1 medzi argumentom a formálnym parametrom. Povedané inak, parametrický inštančný konštruktor prevezme práve toľko argumentov, koľko formálnych parametrov definuje.

pracovať, je použiť objektovú referenciu, ktorú nám poskytne operátor **new**. Objektová referencia jednoznačne identifikuje pozíciu inštancie triedy na riadenej halde.

Objektová referencia, ktorá je návratovou hodnotou operátora **new**, je teda hodnotou celého výrazu **new T()**, resp. **new T(a_1, a_2, ..., a_n)**. Je samozrejme veľmi dôležité, aby sme poskytnutú objektovú referenciu uchovali, pretože len prostredníctvom nej môžeme využívať služby vytvorenej inštancie triedy. To sa koniec koncov aj deje: navrátenú objektovú referenciu ukladáme do definovanej odkazovej premennej **obj**. Vravíme, že táto odkazová premenná je inicializovaná objektovou referenciou identifikujúcou inštanciu triedy na riadenej halde.



Obr. 15: Vizualizácia inštanciácie triedy

Vždy, keď budeme chcieť pracovať s vytvoreným objektom, musíme použiť odkazovú premennú, v ktorej je umiestnená objektová referencia ukazujúca na daný objekt. K verejne prístupným inštančným metódam objektu pristupujeme pomocou bodkového operátora (.). Nasledujúci fragment zdrojového kódu jazyka C# 3.0 demonštruje inštanciáciu deklarovanej triedy **Zamestnanec** a jej praktické použitie:

```
static void Main(string[] args)
{
    // Inštanciácia triedy Zamestnanec.
    Zamestnanec zam1 = new Zamestnanec("Emil", "Malý", 30000, 10);
    // Využitie služieb založenej inštancie.
    Console.WriteLine("Údaje o zamestnancovi: \n" +
       "Meno: " + zam1.PrečítaťMeno() + "\n" +
       "Priezvisko: " + zam1.PrečítaťPriezvisko() + "\n" +
       "Mzda: " + zam1.ZískaťMzdu() + "\n" +
       "Odpracované roky: " + zam1.ZískaťRoky()
    );
    Console.ReadLine();
}
```

Na rozdiel od jazyka C++, vývojári v jazyku C# 3.0 nie sú povinní dynamicky založené objekty z riadenej haldy explicitne uvoľňovať. Odstraňovanie nepotrebných objektov má na starosti automatický správca pamäte (Garbage Collector, GC), ktorý dokáže rozpoznať tie objekty, ktoré sa už nepoužívajú. Ak automatický správca pamäte zistí, že objekt je nedosiahnuteľný z programového kódu, uskutoční jeho uvoľnenie[11].

Generický syntaktický inštanciačný príkaz sa dá rozdeliť do dvoch samostatných príkazov, samozrejme so zachovaním pôvodnej funkcionality:

```
T obj;
obj = new T();
```

Prvý príkaz vytvára novú odkazovú premennú s identifikátorom **obj**. Ako sme už uviedli, do takto definovanej odkazovej premennej môžeme uložiť objektovú referenciu na inštanciu triedy **T**. Keďže v tomto prípade nejde o definičnú inicializáciu odkazovej premennej, je namieste otázka, či bude táto premenná implicitne inicializovaná. Aby sme mohli na zmienenú otázku kvalifikovane odpovedať, musíme zistiť viac informácií o oblasti platnosti a charaktere odkazovej premennej. Pokiaľ by táto premenná vystupovala ako lokálna premenná (definovaná napríklad v tele hlavnej metódy **Main**), tak potom by nebola implicitne inicializovaná. V tomto smere sa jazyk C# 3.0 správa podobne ako C++, jeho ISO štandard vraví, že hodnota definovaných lokálnych premenných nie je determinovaná. Keby sa ale definičný príkaz vytvárajúci odkazovú premennú nachádzal v tele triedy, implicitný inštančný konštruktor by zabezpečil jej implicitnú inicializáciu. Implicitnou inicializačnou hodnotou takejto odkazovej premennej by potom bola nulová referencia, reprezentovaná kľúčovým slovom **null** jazyka C# 3.0.

Odkazová premenná nemusí byť inicializovaná okamžite počas svojej definície. Pochopiteľne, k inicializácii premennej musí dôjsť niekde v oblasti platnosti, v ktorej je definovaná odkazová premenná viditeľná. Ak by premenná nebola vôbec inicializovaná, kompilátor jazyka C# 3.0 by nás na túto skutočnosť upozornil. V každom prípade je neprípustné používať neinicializované premenné. Kompilátor jazyka C# 3.0 je však veľmi citlivý a dokáže správne diagnostikovať všetky neinicializované premenné.

Inicializácia definovanej odkazovej premennej sa odohráva v druhom príkaze, ktorý je spojený s inštanciáciou deklarovanej triedy **T**. Samotná inštanciácia je realizovaná presne tak, ako sme ju opísali.

[11] Proces uvoľňovania objektov z riadenej haldy je známy ako finalizácia objektov. V závislosti od internej kompozície objektu môže byť jeho finalizácia buď implicitná, alebo explicitná.

2.1.4 Konštruktory

Konštruktory, s ktorými sa môžeme v telách tried programovacieho jazyka C# 3.0 stretnúť, smieme rozdeliť do viacerých skupín:

1. Inštančný konštruktor.
    - Implicitný inštančný konštruktor.
    - Explicitný inštančný konštruktor.
        - Bezparametrický explicitný inštančný konštruktor.
        - Parametrický explicitný inštančný konštruktor.
2. Statický konštruktor.

Ako už bolo spomenuté, ak do tela inštančnej (nestatickej) triedy nevložíme žiaden konštruktor, kompilátor automaticky vygeneruje implicitný inštančný konštruktor. Implicitný inštančný konštruktor je verejne prístupný a bezparametrický. Jeho úlohou je uložiť do všetkých inštančných dátových položiek východiskové inicializačné hodnoty.

Trieda s implicitným inštančným konštruktorom:

```
class A
{
    // Telo triedy je prázdne. Kompilátor do neho automaticky
    // vloží implicitný bezparametrický inštančný konštruktor.
}
```

Keď programátor umiestni definíciu inštančného konštruktora do tela triedy, hovoríme o takomto konštruktore ako o explicitnom. Explicitný inštančný konštruktor môže byť buď bezparametrický (s prázdnou signatúrou), alebo parametrický (signatúra je naplnená množinou formálnych parametrov).

Trieda s explicitným bezparametrickým inštančným konštruktorom:

```
class B
{
    public B()
    {
        // Telo bezparametrického konštruktora.
    }
}
```

Trieda s explicitným parametrickým inštančným konštruktorom:

```
class C
{
    public C(int x)
    {
        // Telo parametrického konštruktora.
    }
}
```

V prípade dodania len parametrického explicitného inštančného konštruktora nie je kompilátorom samočinne vytváraný žiaden bezparametrický inštančný konštruktor. Ak takýto konštruktor potrebujeme, musíme ho zostrojiť sami. V jazyku MSIL je inštančný konštruktor (či už implicitný, alebo explicitný) predstavovaný metódou s identifikátorom **.ctor**.

Je povolené, aby sa v triede nachádzalo viacero definícií explicitného inštančného konštruktora. Pritom platí, že bezparametrická verzia explicitného inštančného konštruktora sa v triede smie objaviť práve jedenkrát. Naopak, parametrická verzia explicitného inštančného konštruktora sa môže vyskytovať vo viacerých verziách, avšak len vtedy, ak sa tieto verzie odlišujú zoznamami svojich formálnych parametrov. To konkrétne znamená, že rôzne definície explicitného parametrického inštančného konštruktora sa môžu líšiť počtom formálnych parametrov, dátovými typmi formálnych parametrov alebo poradím formálnych parametrov.

Ak v tele triedy pozorujeme jednu definíciu explicitného bezparametrického inštančného konštruktora a viacero definícií explicitného parametrického inštančného konštruktora, tak vravíme, že inštančný konštruktor triedy je preťažený. Klient triedy si teda môže vybrať z viacerých exemplárov konštruktora a inicializovať tak zakladané inštancie triedy rôznymi spôsobmi. O tom, ktorá z rôznych definícií inštančného konštruktora bude v procese inštanciácie triedy zavolaná, rozhoduje početnosť a postupnosť argumentov (atomických kvánt vstupných dát). Ak je trieda navrhnutá korektne, kompilátor dokáže vždy určiť cieľový inštančný konštruktor, ktorý bude zavolaný v záujme inicializácie inštancie triedy[12].

[12] Kompilátor jazyka C# 3.0 je schopný detegovať nejednoznačnosti v signatúrach jednotlivých definícií preťaženého inštančného konštruktora. Ak budú takéto nejednoznačnosti zistené, kompilátor preruší preklad generovaním chybového hlásenia.

```
class D
{
    public D()
    {
        // Telo bezparametrického konštruktora.
    }
    public D(int x)
    {
        // Telo 1. verzie preťaženého parametrického konštruktora.
    }
    public D(int x, E y)
    {
        // Telo 2. verzie preťaženého parametrického konštruktora.
    }
}
```

V tele triedy **D** vidíme jednu definíciu explicitného bezparametrického inštančného konštruktora a dve verzie explicitne definovaného parametrického inštančného konštruktora. Explicitný inštančný konštruktor triedy **D** je teda preťažený. Prvá parametrická verzia konštruktora pracuje len s jedným formálnym parametrom, ktorý očakáva 32-bitový celočíselný argument. Druhá parametrická verzia konštruktora má zložitejšiu signatúru: okrem celočíselného formálneho parametra sa v nej nachádza aj druhý formálny parameter s typom, ktorého identifikátor je **E**. Samozrejme, z deklarácie triedy **D** nevieme vyčítať, čo typ **E** predstavuje. Môže ísť o názov štruktúry, enumeračného typu, triedy, delegáta, alebo rozhrania.

Ak **E** reprezentuje štruktúru, tak formálny parameter bude inicializovaný inštanciou tejto štruktúry. Pokiaľ je **E** identifikátorom enumeračného typu, tak formálny parameter bude inicializovaný enumerátorom.

Ak bude **E** trieda, tak formálny parameter získa odkaz na inštanciu tejto triedy. Keď bude identifikátor **E** predstavovať delegáta, formálny parameter bude schopný akceptovať odkaz na inštanciu tohto delegáta (inštancia delegáta pritom obsahuje odkaz na cieľovú metódu, ktorá bude pomocou delegáta nepriamo aktivovaná). V prípade, ak bude **E** rozhraním, tak formálny parameter bude môcť prijať odkaz na inštanciu akejkoľvek triedy, ktorá rozhranie **E** implementuje.

2.1.5 Statický konštruktor a statické členy inštančných tried

Význam statického konštruktora spočíva v inicializácii statických dátových členov triedy. Statické dátové členy triedy (tiež statické dátové položky triedy) sú definované pomocou modifikátora **static**. Statické dátové položky a statický konštruktor sa smú vyskytovať v inštančných a statických triedach. V tejto kapitole budeme skúmať pôsobnosť statických dátových položiek a statického konštruktora v inštančných triedach. Rozpravu o statických triedach budeme viesť v kapitole *2.8 Statické triedy*.

Keď sú dátové položky triedy statické, existujú vždy v práve jednom vyhotovení, a to bez ohľadu na to, koľko inštancií (inštančnej) triedy existuje[13]. Naopak, všetky inštancie triedy môžu využívať statické dátové položky triedy. Vravíme, že statické dátové položky triedy sú inštanciami triedy zdieľané. Aj pri statických dátových položkách triedy sa zachováva princíp ukrývania dát, a preto sú (implicitne) definované ako súkromné. Popri dátových položkách sa môžu v telách inštančných tried vyskytovať aj iné statické členy, napríklad metódy alebo vlastnosti.

Všeobecná definícia statického konštruktora v tele inštančnej (a aj statickej) triedy je nasledujúca:

```
static T()
{
    // Telo statického konštruktora.
}
```

kde:

- **T** je názov inštančnej, resp. statickej triedy, v ktorej deklarácii je statický konštruktor uložený.

Pri práci so statickým konštruktorom musíme vedieť toto:

1. Statický konštruktor je implicitne súkromný (explicitne nesmie obsahovať žiadne prístupové modifikátory).

2. Statický konštruktor je vždy bezparametrický.

3. Statický konštruktor nemôže byť preťažený. Okrem bezparametrickej verzie, nie sú v tele triedy prípustné žiadne iné verzie statického konštruktora.

4. Statický konštruktor nie je nikdy explicitne aktivovaný. Je to preto, že statický konštruktor je automaticky volaný virtuálnym exekučným systémom, a to v týchto prípadoch:

    - Pred vytvorením prvej inštancie triedy (platí len pre inštančné triedy so statickým konštruktorom).

    - Pred prvým pokusom o prístup k statickému členu triedy.

[13] Naopak, každá inštancia (inštančnej) triedy obsahuje svoje vlastné inštančné dátové položky. Ak teda vytvoríme 100 inštancií triedy, každá z nich bude zapuzdrovať vlastnú kolekciu dátových položiek. Dátové položky ľubovoľných dvoch rôznych inštancií zo 100-člennej súpravy sú úplne samostatné a na sebe absolútne nezávislé.

5. V jazyku MSIL je statický konštruktor reprezentovaný metódou s identifikátorom **.cctor**.

6. Statický konštruktor sa môže vyskytovať iba v telách tried, nie štruktúr.

Nasleduje deklarácia inštančnej triedy, v ktorej tele sa stretávame so statickou dátovou položkou, statickým konštruktorom a statickou metódou. Trieda dokáže pomocou statickej dátovej položky sledovať, koľkokrát bola aktivovaná jej inštančná metóda.

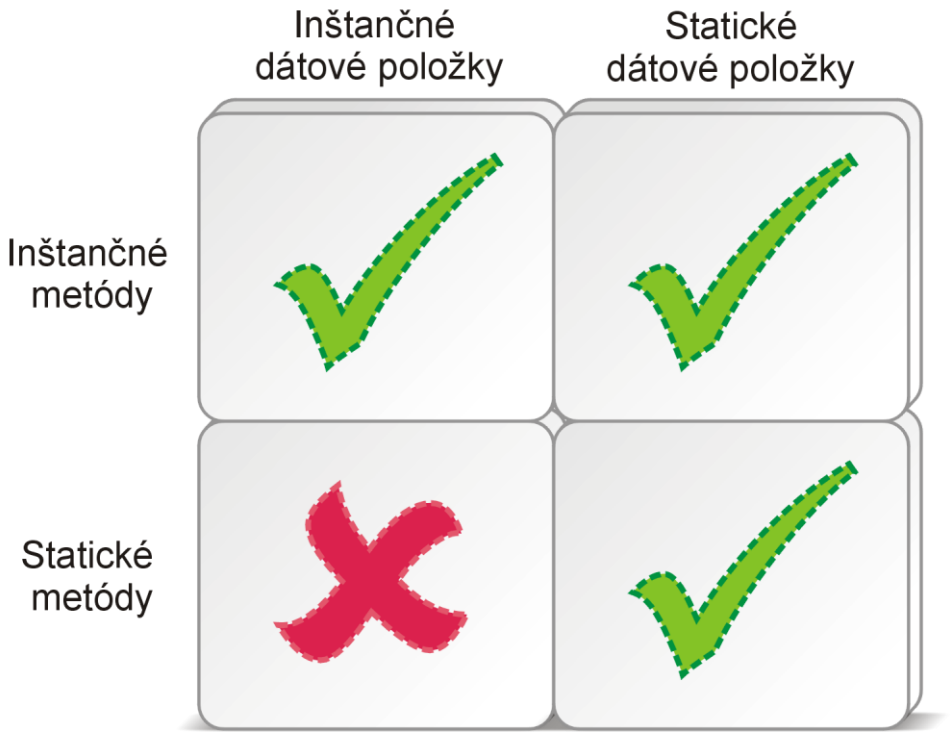
```
class K
{
    // Definícia statickej dátovej položky.
    static int početVolaní;
    // Definícia statického konštruktora.
    static K()
    {
        početVolaní = 0;
    }
    // Definícia inštančnej metódy.
    public void M()
    {
        početVolaní++;
    }
    // Definícia statickej metódy.
    public static int ZistiťPočetVolaní()
    {
        return početVolaní;
    }
}
```

Všetky statické členy triedy disponujú modifikátorom **static**. Statický konštruktor inicializuje statickú dátovú položku. Z charakteru ukážky vyplýva, že zmysluplná inicializačná hodnota je nula, pretože v čase inicializácie statickej dátovej položky ešte nebola inštančná metóda ani raz zavolaná (a ani nemohla byť, keďže na jej aktiváciu potrebujeme zostrojiť aspoň jednu inštanciu triedy). Vždy, keď bude spomínaná metóda zavolaná, inkrementuje sa hodnota statickej dátovej položky triedy. Na zistenie aktuálneho počtu spustení metódy využijeme statickú metódu, ktorá nám na požiadanie poskytne aktuálnu hodnotu statickej dátovej položky.

Deklarovanú triedu použijeme v ďalšom fragmente zdrojového kódu programovacieho jazyka C# 3.0:

```
static void Main(string[] args)
{
    K obj = new K();
    for (int i = 0; i < 20; i++)
    {
        obj.M();
    }
    Console.WriteLine("Metóda bola zavolaná " +
        K.ZistiťPočetVolaní() + "krát.");
}
```

Zaujímavá je interakcia medzi inštančnými dátovými položkami, inštančnými metódami, statickými dátovými položkami a statickými metódami. Vizualizáciu tejto interakcie ponúka obr. 16.



Obr. 16: Interakcia medzi rozličnými členmi tried

Inštančné metódy triedy smú pristupovať k inštančným a statickým dátovým položkám triedy. To je evidentné, pretože sme povedali, že statické dátové položky zdieľajú všetky inštancie triedy. Na druhú stranu, statické metódy môžu pristupovať k statickým, no už nie k inštančným dátovým položkám triedy. Dôvod, prečo nie je povolené, aby statické metódy pristupovali k inštančným dátovým položkám, je logický. Vzhľadom na to, že statická metóda môže byť zavolaná bez nutnosti inštanciácie triedy, pokus o prístup k inštančným dátovým položkám by sa skočil abnormálnym ukončením programu. V čase aktivácie statickej metódy totiž neexistuje žiadna inštancia, ktorej dátové položky by mohli byť predmetom programových operácií.

2.1.6 Mechanizmus typovej inferencie (MTI)

Kompilátor jazyka C# 3.0 dokáže automaticky určiť dátový typ lokálnej hodnotovej alebo odkazovej premennej podľa jej inicializačného výrazu. Implicitné stanovenie dátového typu lokálnej premennej uskutočňuje mechanizmus typovej inferencie (MTI). Všeobecný zápis inštanciačného príkazu triedy, ktorý pracuje s MTI, je nasledujúci:

```
var obj = new T();
```

alebo

```
var obj = new T(a1, a2, ..., an);
```

kde:

- **var** je kľúčové slovo indikujúce, že typ lokálnej premennej bude implicitne inferovaný.
- **obj** je identifikátor lokálnej premennej.
- **T** je názov triedy, ktorej inštancia je vytváraná.
- **a_1, a_2, ..., a_n** sú argumenty, ktoré sú poskytované formálnym parametrom parametrického inštančného konštruktora triedy **T**.

Syntaktickou modifikáciou je použitie kľúčového slova **var** namiesto špecifikácie dátového typu lokálnej premennej. Len čo MTI vyhodnotí výraz na pravej strane od priraďovacieho operátora, determinuje typ lokálnej premennej. V tejto súvislosti je dôležité uviesť, že dátový typ lokálnej premennej je pomocou MTI určený vždy v čase prekladu zdrojového kódu. Použitie MTI teda nemožno chápať ako dynamickú identifikáciu dátového typu lokálnej premennej, resp. ako vytvorenie dynamickej väzby medzi lokálnou premennou a jej dátovým typom.

MTI smie byť aplikovaný aj pri vytváraní lokálnych polí inštancií tried. Generický zápis zdrojového kódu sa potom zmení takto:

```
var poleInštancií = new [] {
    new T(), new T()
};
```

alebo

```
var poleInštancií = new [] {
    new T(a1, a2, ..., an),
    new T(a1, a2, ..., an)
};
```

V 1. prípade zhotovujeme pole dvoch inštancií triedy **T**, ktoré sú inicializované bezparametrickým inštančným konštruktorom. Hodnotou výrazu na pravej strane od priraďovacieho operátora je odkaz na jednorozmerné pole, ktoré je alokované na riadenej halde.

V 2. prípade konštruujeme pole dvoch inštancií triedy **T**, pričom obe inicializujeme parametrickými inštančnými konštruktormi. Hodnota výrazu napravo od operátora = je rovnaká ako v predchádzajúcom prípade.

```
// Deklarácia triedy.
class T
{
    // Definícia statickej dátovej položky triedy.
    static int t;
    public T()
    {
        Console.WriteLine("Inštancia č. {0}", ++t);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Implicitné určenie dátového typu
        // lokálnej odkazovej premennej pomocou MTI.
        var poleInštancií = new [] { new T(), new T() };
        Console.Read();
    }
}
```

Hoci pri inštanciácii pomenovaných tried je použitie MTI iba voliteľnou pomôckou, zásadne iná je situácia pri inštanciácii anonymných tried. Vzhľadom na to, že identifikátor anonymnej triedy nám nie je známy, nevieme explicitne určiť dátový typ lokálnej odkazovej premennej, do ktorej bude uložený odkaz na zrodenú inštanciu anonymnej triedy. Za týchto okolností sa musíme v plnej miere spoľahnúť na MTI, ktorý zabezpečí priradenie príslušného dátového typu lokálnej premennej v čase prekladu zdrojového kódu.

O anonymných triedach budeme hovoriť bližšie v kapitole *2.9 Anonymné triedy a inicializátory inštancií anonymných a pomenovaných tried*.

2.1.7 Skalárne inštančné vlastnosti triedy

Hoci v jazyku C++ boli prístupové metódy typu **get**/**set** jediným mechanizmom na prácu s dátami objektu, jazyk C# 3.0 poskytuje elegantnejšie riešenie, ktoré je implementované pomocou skalárnych inštančných vlastností. Skalárna inštančná vlastnosť je programovou konštrukciou, ktorá zapuzdruje dve prístupové metódy **get** a **set**, ktorých úlohou je získanie, resp. úprava hodnoty požadovanej inštančnej dátovej položky. Pozrime sa, ako by vyzerala deklarácia triedy **Zamestnanec**, ak aby sme množinu prístupových metód nahradili skalárnymi inštančnými vlastnosťami:

```
class Zamestnanec
{
    private string meno;
    private string priezvisko;
    private int mzda;
    private int odpracovanéRoky;
    public Zamestnanec(string meno, string priezvisko, int mzda, int roky)
```

```
    {
        this.meno = meno;
        this.priezvisko = priezvisko;
        this.mzda = mzda;
        odpracovanéRoky = roky;
    }
    // Definície skalárnych inštančných vlastností.
    public string Meno
    {
        get { return meno; }
        set { meno = value; }
    }
    public string Priezvisko
    {
        get { return priezvisko; }
        set { priezvisko = value; }
    }
    public int Mzda
    {
        get { return mzda; }
        set { mzda = value; }
    }
    public int OdpracovanéRoky
    {
        get { return odpracovanéRoky; }
        set { odpracovanéRoky = value; }
    }
}
```

V tele triedy sa nachádzajú definície štyroch skalárnych inštančných vlastností: **Meno**, **Priezvisko**, **Mzda** a **OdpracovanéRoky**. Všetky skalárne inštančné vlastnosti sú verejne prístupné a zdieľajú podobnú syntaktickú formu. Každá z týchto skalárnych inštančných vlastností je previazaná s jednou dátovou položkou. Za účelom bližšej inšpekcie si vyberieme prvú skalárnu inštančnú vlastnosť **Meno**, na ktorej vysvetlíme jej syntaktickú reprezentáciu. Ostatné vlastnosti budú pracovať podľa vysvetleného modelu.

```
public string Meno
{
    get { return meno; }
    set { meno = value; }
}
```

Skalárna inštančná vlastnosť má svoj dátový typ, ktorý je zhodný s dátovým typom dátovej položky, s ktorou je táto vlastnosť prepojená. V prípade skalárnej inštančnej vlastnosti **Meno** je to primitívny odkazový dátový typ **string**. Už vieme, že skalárne inštančné vlastnosti môžu získavať, resp. modifikovať hodnoty asociovaných dátových položiek. Ak je vlastnosť schopná nielen získavať, ale aj upravovať hodnoty dátových položiek, označuje sa ako „vlastnosť určená na čítanie a zápis“. Skalárna inštančná vlastnosť **Meno** je vlastnosťou určenou na čítanie aj zápis, pretože vo svojom tele obsahuje dve prístupové metódy **get** a **set** (presne také isté zloženie majú aj všetky ostatné skalárne inštančné vlastnosti analyzovanej triedy). Zatiaľ čo prístupová metóda **get** obsahuje zdrojový kód, ktorý poskytne hodnotu dátovej položky používateľovi, prístupová metóda **set** sa k slovu dostáva pri potrebe zmeniť hodnotu dátovej položky.

V tele prístupovej metódy **set** je zapísaný priraďovací príkaz, ktorý inicializuje dátovú položku. Samotná inicializačná hodnota je uložená v premennej **value**. Táto premenná je špeciálnou premennou jazyka C# 3.0, takže vývojár ju nikdy nevytvára. Typ premennej **value** je implicitne inferovaný tak, aby bol totožný s typom skalárnej inštančnej vlastnosti.

Dodajme, že definíciu prístupovej metódy **get**, resp. prístupovej metódy **set**, môžeme v tele skalárnej inštančnej vlastnosti vynechať (nie však obe súčasne). Ak sa bude v tele vlastnosti nachádzať len prístupová metóda **get**, vytvoríme vlastnosť, ktorá bude určená len „na čítanie“ hodnoty cieľovej dátovej položky. Analogicky, ak v tele vlastnosti bude definovaná len prístupová metóda **set**, pôjde o vlastnosť určenú „len za zápis“ hodnoty do cieľovej dátovej položky. V praktickom nasadení sa často stretávame so skalárnymi inštančnými vlastnosťami, ktoré nám umožňujú len získať, no už nie modifikovať dáta objektu uložené v istej dátovej položke. S oveľa menšou frekvenciou sa vyskytujú skalárne inštančné vlastnosti určené len na zápis dát do dátových položiek.

Fragment zdrojového kódu jazyka C# 3.0, ktorý inštanciuje triedu **Zamestnanec** a využíva množinu skalárnych inštančných vlastností, vyzerá takto:

```
static void Main(string[] args)
{
    Zamestnanec zam1 = new Zamestnanec("Emil", "Malý", 30000, 10);
    Console.WriteLine("Údaje o zamestnancovi: \n" +
       "Meno: " + zam1.Meno + "\n" +
       "Priezvisko: " + zam1.Priezvisko + "\n" +
       "Mzda: " + zam1.Mzda + "\n" +
       "Odpracované roky: " + zam1.OdpracovanéRoky
    );
    Console.ReadLine();
}
```

Nepochybne, práca s inštanciou triedy pomocou skalárnych inštančných vlastností je elegantnejšia a prirodzenejšia. Z pohľadu používateľa sa vlastnosť javí ako inteligentná dátová položka, aj keď v skutočnosti ide o programovú konštrukciu, ktorá zapuzdruje dvojicu prístupových metód.

2.1.8 Automaticky implementované skalárne inštančné vlastnosti triedy

Štandardné správanie prístupových metód **get** a **set** skalárnych inštančných vlastností triedy môžeme charakterizovať takto:

1. Prístupová metóda **get** získava hodnotu súkromnej dátovej položky inštancie triedy.

2. Prístupová metóda **set** modifikuje hodnotu súkromnej dátovej položky inštancie triedy.

Ak nemáme záujem o pokročilú konfiguráciu prístupových metód **get** a **set** (napr. za účelom zapracovania pokročilej aplikačnej logiky), môžeme prenechať kompletné zostavenie tiel prístupových metód vlastností a vytvorenie definícií spriaznených dátových položiek kompilátoru. Tak vytvoríme automaticky implementované skalárne inštančné vlastnosti triedy. Ďalej uvádzame praktickú ukážku triedy, ktorej vlastnosti sú automaticky implementované.

```
// Deklarácia triedy, ktorá využíva automaticky implementované
// skalárne inštančné vlastnosti.
class MP3Prehrávač
{
    public string Značka { get; set; }
    public byte KapacitaPamäteVGB { get; set; }
    public ushort PočetPiesní { get; set; }
}
```

Inštancie tejto triedy budú reprezentovať obľúbené MP3 prehrávače. Pri každom prehrávači budeme sledovať tri atribúty: značku, kapacitu vstavanej (flash) pamäte v GB a maximálny počet piesní, ktoré dokáže prehrávač uchovať.

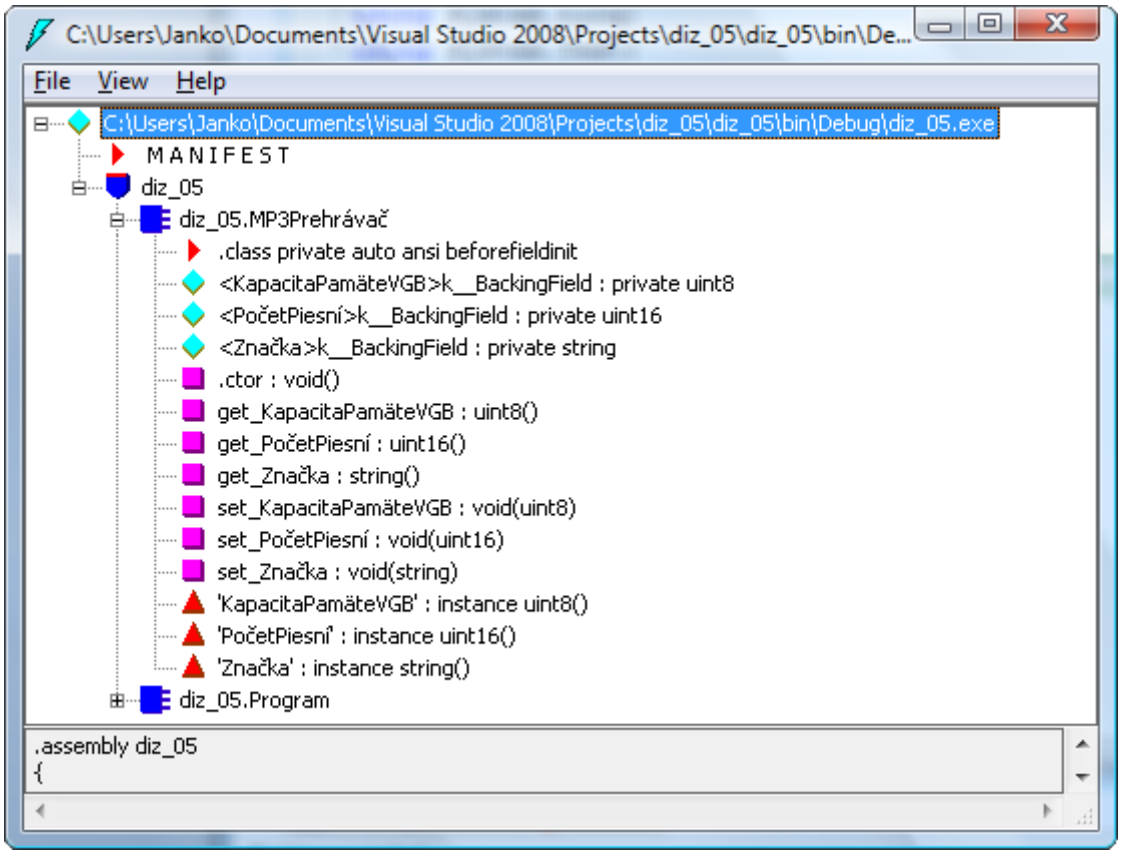
V tele triedy sú definované tri skalárne inštančné vlastnosti, ktoré budú kompilátorom automaticky implementované. Ak chceme dať najavo našu intenciu o začlenenie automaticky implementovaných skalárnych inštančných vlastností do tela triedy, do tiel príslušných vlastností umiestnime len deklarácie (nie definície) prístupových metód **get** a **set**. Pri doručení požiadavky na vytvorenie automaticky implementovaných vlastností uskutoční kompilátor nasledujúce operácie:

- Telo triedy rozšíri o definíciu množiny súkromných dátových položiek. Vzťah medzi dátovými položkami a vlastnosťami ostáva nezmenený a opisuje ho relácia typu 1:1.

- Vytvorí definície pôvodne iba deklarovaných prístupových metód **get** a **set**. Vygenerované definície budú fungovať podľa štandardného modelu: metóda **get** bude získavať hodnotu dátovej položky, zatiaľ čo metóda **set** bude túto hodnotu meniť.

Okrem spomenutého vytvára kompilátor v našej ukážke samozrejme aj verejne prístupný implicitný inštančný konštruktor.

O tom, že kompilátor vykonal konštatované operácie, sa môžeme presvedčiť pri pohľade na MSIL kód predmetnej triedy (obr. 17).



Obr. 17: MSIL obraz triedy s automaticky implementovanými vlastnosťami

Deklarácia triedy s automaticky implementovanými vlastnosťami je funkčne ekvivalentná kompletnej deklarácii rovnomennej triedy:

```
class MP3Prehrávač
{
    private string značka;
    private byte kapacitaPamäteVGB;
    private ushort početPiesní;
    public string Značka
    {
        get { return značka; }
        set { značka = value; }
    }
    public byte KapacitaPamäteVGB
    {
        get { return kapacitaPamäteVGB; }
        set { kapacitaPamäteVGB = value; }
    }
    public ushort PočetPiesní
    {
        get { return početPiesní; }
        set {početPiesní = value; }
    }
}
```

Kompletná deklarácia triedy sa odlišuje explicitným definovaním súkromných dátových položiek, ktoré sú pre členy triedy vždy prístupné. Dátové položky, ktoré kompilátor

samočinne generuje pri automatickej implementácii skalárnych inštančných vlastností však nie sú nijakým spôsobom dosiahnuteľné. Programátor teda nemá možnosť s nimi priamo narábať.

Použitie inštancie je potom už jednoduché:

```
static void Main(string[] args)
{
    MP3Prehrávač zen = new MP3Prehrávač();
    zen.Značka = "Creative ZEN";
    zen.KapacitaPamäteVGB = 4;
    zen.PočetPiesní = 2000;
    Console.WriteLine("Produktové informácie o " +
        "MP3 prehrávači:\n" +
        "Značka: " + zen.Značka + "\n" +
        "Kapacita pamäte (GB): " + zen.KapacitaPamäteVGB + "\n" +
        "Počet piesní: " + zen.PočetPiesní);
    Console.Read();
}
```

2.1.9 Indexované inštančné vlastnosti triedy

Okrem skalárnych inštančných vlastností smieme v jazyku C# 3.0 rozšíriť telo triedy tiež o definície indexovaných inštančných vlastností. Indexovaná vlastnosť[14] nám umožňuje inicializovať inštanciu triedy podobne, ako keby bola táto inštancia poľom.

Indexovaná inštančná vlastnosť sa definuje nasledujúcim spôsobom:

```
public T1 this[T2 i]
{
    get { /* Zdrojový kód prístupovej metódy get. */ }
    set { /* Zdrojový kód prístupovej metódy set. */ }
}
```

kde:

- **T_1** je dátový typ indexovanej vlastnosti a súčasne aj dátový typ množiny dátových položiek, s ktorými je indexovaná vlastnosť asociovaná.
- **this** je identifikátor indexovanej vlastnosti.
- **T_2** je dátový typ formálneho parametra indexovanej vlastnosti.
- **i** je identifikátor formálneho parametra indexovanej vlastnosti. Tento formálny parameter uchováva index, ktorý determinuje požadovanú dátovú položku inštancie triedy.

[14] Indexovaná vlastnosť sa niekedy nazýva aj „indexer“. Keďže v tejto publikácii dbáme na terminologickú čistotu výkladu, budeme uprednostňovať výstižnejší termín „indexovaná vlastnosť“.

Pre indexované inštančné vlastnosti platia tieto pravidlá:

1. Definície indexovaných vlastností sa môžu nachádzať v deklaráciách tried a štruktúr.

2. Indexované vlastnosti môžu byť preťažené. Je teda možné vytvoriť množinu definícií rovnomennej indexovanej vlastnosti. Pochopiteľne, jednotlivé verzie preťaženej indexovanej vlastnosti sa musia odlišovať svojou signatúrou (zoznamom formálnych parametrov).

Trieda **Vektor**, ktorej deklaráciu približuje nasledujúci fragment zdrojového kódu jazyka C# 3.0, disponuje jednou indexovanou inštančnou vlastnosťou.

```
class Vektor
{
    private int x, y, z;
    // Definícia indexovanej inštančnej vlastnosti.
    public int this[int i]
    {
        get
        {
            switch (i)
            {
                case 0:
                    return x;
                case 1:
                    return y;
                case 2:
                    return z;
                default:
                    throw new System.ArgumentOutOfRangeException();
            }
        }
        set
        {
            switch (i)
            {
                case 0:
                    x = value;
                    break;
                case 1:
                    y = value;
                    break;
                case 2:
                    z = value;
                    break;
                default:
                    throw new System.ArgumentOutOfRangeException();
            }
        }
    }
}
```

Komentár k triede: Inštancia triedy stelesňuje trojrozmerný (3D) vektor. Vzhľadom na to, že dátovým obsahom inštancie je trojica vektorových zložiek, ako rozumné riešenie

sa javí zapracovať indexovanú vlastnosť, ktorá nám dovolí inicializovať inštanciu triedy (vektor) ako jednorozmerné pole. Indexovaná vlastnosť je verejne prístupná a parametrická, pretože v jej signatúre je umiestnená definícia jedného formálneho parametra. Podľa hodnoty tohto formálneho parametra bude indexovaná vlastnosť buď získavať hodnoty vektorových zložiek, alebo ich upravovať. V tele indexovanej vlastnosti môžeme pozorovať definície dvoch prístupových metód **get** a **set**. Venujme sa najskôr prístupovej metóde **get**.

Programový kód metódy **get** indexovanej vlastnosti bude vykonaný vtedy, keď bude doručená požiadavka na získanie hodnoty istej vektorovej zložky. Keďže 3D vektor obsahuje 3 zložky, môžeme ich indexovať hodnotami z intervalu <0, 2>. Pre prístup prostredníctvom indexovanej vlastnosti je dôležitá hodnota indexu, pretože len na základe nej vieme určiť, s ktorou zložkou vektora budeme pracovať. Toto viaccestné rozhodovanie je riešené príkazom **switch**, ktorý operuje s hodnotou formálneho parametra **i**. Ak **i == 0**, tak prístupová metóda **get** vráti hodnotu dátovej položky **x** (teda hodnotu 1. zložky vektora). V prípade zvyšujúcej sa hodnoty indexu bude metóda **get** vracať vždy zložku vektora s vyšším poradovým číslom. Na tomto mieste si dovolíme pripojiť poznámku k syntaktickej implementácii: Ako môžeme postrehnúť, po vrátení hodnoty dátovej položky príkazom **return** už nezadávame v príslušnej vetve **case** ukončovací príkaz **break**. Tento príkaz nie je v tomto kontexte potrebný, pretože už samotné vrátenie hodnoty príkazom **return** znamená ukončenie rozhodovania v príkaze **switch**.

Ak sa nedopatrením stane, že hodnota formálneho parametra **i** bude mimo hraníc intervalu <0, 2>, bude spustený kód vo vetve **default**. Za týchto okolností iniciujeme generovanie chybovej výnimky, pretože nevieme mapovať index na dátovú položku.

Účelom existencie prístupovej metódy **set** je nastavenie hodnoty dátovej položky, ktorá bude určená svojím číselným indexom. Viaccestné rozhodovanie prebieha podľa známeho algoritmu. Rozdiel spočíva v tom, že vybratú dátovú položku inicializujeme v rámci priraďovacieho príkazu. Keďže priraďovací príkaz neukončuje spracovanie príkazov vo vetve **case**, musíme zaň pridať samostatný terminačný príkaz **break**.

Nasledujúci program ukazuje, aké praktické implikácie so sebou prináša použitie indexovaných inštančných vlastností triedy.

```
static void Main(string[] args)
{
    Vektor vektor = new Vektor();
    // Inicializácia inštancie triedy pomocou indexovanej vlastnosti.
    vektor[0] = 1;
    vektor[1] = 1;
    vektor[2] = 1;
    Console.WriteLine("Vektor [{0}, {1}, {2}]", vektor[0],
       vektor[1], vektor[2]);
}
```

Všimnime si, ako je inicializovaná založená inštancia triedy **Vektor**. Za identifikátorom odkazovej premennej zapisujeme hranaté zátvorky a špecifikujeme index dátovej položky (zložky vektora), s ktorou si prajeme manipulovať. Syntakticky použitie indexovanej vlastnosti naozaj vyzerá, ako keby sme s inštanciou triedy **Vektor** pracovali ako s jednorozmerným poľom. V závislosti od toho, kde sa výraz **vektor[i]** nachádza, bude spracovaná buď prístupová metóda **get**, alebo prístupová metóda **set** indexovanej vlastnosti.

Indexované vlastnosti preukazujú svoje silné stránky aj pri práci s poľami vektorov:

```
static void Main(string[] args)
{
    Vektor[] poleVektorov = new Vektor[10];
    Random generátor = new Random();
    for (int i = 0; i < 10; i++)
    {
        poleVektorov[i] = new Vektor();
        Console.Write("Vektor č. {0} má súradnice ", i + 1);
        for (int m = 0; m < 3; m++)
        {
            poleVektorov[i][m] = generátor.Next(1, 101);
            Console.Write("\t{0}\t", poleVektorov[i][m]);
        }
        Console.Write("\n");
    }
    Console.Read();
}
```

Tento zdrojový kód vytvára pole desiatich inštancií triedy **Vektor**. Naším zámerom je všetky zostrojené vektory inicializovať pomocou ich indexovaných vlastností. Inicializačnými hodnotami budú pseudonáhodné celé čísla z intervalu <1, 100>. V ďalšom texte sa budeme venovať len kritickým miestam programu:

1. Prvý príkaz vytvára jednorozmerné pole, ktorého prvky budú môcť byť inicializované odkazmi na inštancie triedy **Vektor**. Je dôležité si uvedomiť, že po spracovaní tohto príkazu existuje pole, no nie je naplnené žiadnymi objektmi (vektormi).

2. Vektory sú zhotovované až v tele nadradeného cyklu **for**. Vo vnorenom cykle **for** používame indexované vlastnosti vektorov na ich inicializáciu pseudonáhodnými celými číslami. Výraz **poleVektorov[i][m]** znamená spustenie indexovanej vlastnosti i-tej inštancie triedy **Vektor**[15].

[15] Výraz **poleVektorov[i][m]** môže, najmä pre vývojárov prichádzajúcich z jazykov C a C++, indikovať, že pracujeme s dvojrozmerným poľom. To je však chyba, ktorá vzniká aplikáciou pravidiel jazykov C/C++ pri práci s dvojrozmernými poľami v prostredí C# 3.0. Ak by sme v jazyku C# 3.0 pracovali s dvojrozmerným poľom, tak indexácia by vyzerala inak: **poleVektorov[i, m]** a nie **poleVektorov[i][m]**. V tomto prípade ide o aktiváciu indexovanej vlastnosti požadovanej inštancie triedy **Vektor**.

2.1.10 Finalizér

Finalizér je špeciálna metóda triedy, ktorej úlohou je vykonať finalizačné práce predtým, ako bude inštancia triedy uvoľnená z riadenej haldy[16].

Všeobecná syntaktická podoba finalizéra je nasledujúca:

```
~T()
{
    // Telo finalizéra.
}
```

kde:

- **T** je identifikátor triedy, v ktorej tele sa finalizér nachádza.

Pre finalizér platia tieto zásady:

- Finalizér sa môže vyskytovať iba v telách tried (odkazových dátových typov), nikdy nie v telách štruktúr (hodnotových dátových typov).

- Finalizér nemá žiadnu návratovú hodnotu (podobne ako pri konštruktore nemožno použiť ani kľúčové slovo **void**).

- Finalizér musí byť vždy bezparametrický. Nemôže teda definovať žiadne formálne parametre.

- Finalizér sa nesmie stať predmetom preťaženia, čo znamená, že jedna trieda môže definovať práve jeden finalizér.

- Ak bázová trieda explicitne definuje svoj finalizér, tento nie je dedený odvodenou triedou.

- Finalizér nesmie byť volaný priamo klientskym zdrojovým kódom. Je to preto, že finalizér je v prípade potreby implicitne aktivovaný automatickým správcom pamäte virtuálneho exekučného systému.

- Ak do tela triedy umiestnime definíciu finalizéra, kompilátor automaticky vloží do jeho tela kód, ktorý bude volať finalizér primárnej systémovej bázovej triedy **System.Object**.

[16] V predchádzajúcich verziách jazyka C# 3.0 bol finalizér často označovaný ako deštruktor, čo však nie je podľa nášho názoru správne. Vyplýva to z odlišných pracovných modelov deštruktorov v jazyku C++ a finalizačných metód (finalizérov) v jazyku C# 3.0. Preto sme uprednostnili pojem finalizér, ktorý nahrádza skôr uplatňovaný termín deštruktor.

- Ak je finalizovaná inštancia odvodenej triedy, najskôr je volaný jej finalizér. Po spracovaní zdrojového kódu, ktorý je uložený v tele finalizéra odvodenej triedy dochádza k aktivácii finalizéra bázovej triedy a exekúcii v ňom uložených programových príkazov.

- Keďže finalizácia inštancií tried na riadenej halde je nedeterministická, nemožno presne predikovať okamih, kedy bude finalizér spustený.

- Na rozdiel od konštruktora, finalizér nesmie byť statický.

Podľa toho, či inštancia vznikla z triedy, ktorá explicitne definuje svoj finalizér alebo nie, rozlišujeme dva základné modely finalizácie inštancií tried: implicitná a explicitná finalizácia. Implicitná finalizácia sa týka len tých inštancií, ktoré vznikli z tried, ktoré explicitne nedefinujú svoje finalizéry. Ak sa takáto inštancia triedy stane nepotrebnou, automatický správca pamäte uskutoční jej okamžitú dealokáciu. Na druhú stranu, ak inštancia vzišla z triedy s explicitne definovaným finalizérom, bude vyžadovať explicitnú finalizáciu. Proces explicitnej finalizácie je bližšie ozrejmený v kapitole *2.2 Analýza životných cyklov inštancií tried*.

2.1.11 Deštruktory v jazyku C++ a finalizéry v jazyku C# 3.0

Napriek tomu, že syntaktický obraz finalizéra jazyka C# 3.0 je taký istý ako vzhľad deštruktora v jazyku C++, medzi oboma entitami existujú významné sémantické rozdiely. Vôbec najdôležitejším rozdielom je skutočnosť, že kým v jazyku C++ je finalizácia inštancií tried deterministická, v prostredí jazyka C# 3.0 pracujeme s nedeterministickou finalizáciou. Ak v jazyku C++ založíme dynamicky inštanciu triedy (pomocou operátora **new**), môžeme na ňu kedykoľvek aplikovať operátor **delete**[17]. Použitie operátora **delete** implikuje okamžitú aktiváciu deštruktora, ktorý je zodpovedný za realizáciu množiny finalizačných operácií predtým, ako bude inštancia triedy dealokovaná. Je teda zrejmé, že programátor v jazyku C++ je schopný explicitne vyvolať deštruktor na požiadanie kedykoľvek, keď to uzná za vhodné. V jazyku C# 3.0 nemôžeme finalizér aktivovať priamo, pretože túto kompetenciu za vývojárov preberá virtuálny exekučný systém. Aj z tohto dôvodu sa v jazykovej špecifikácii C# 3.0 nenachádza žiaden operátor **delete**, alebo iný, funkčne príbuzný operátor.

V jazyku C++ môžeme vytvoriť inštanciu triedy aj automaticky, a to príkazom

```
X x;
```

alebo

[17] Presnejšie povedané, operátor **delete** je aplikovaný na smerníkovú premennú, v ktorej je uložený smerník identifikujúci pozíciu inštancie triedy v dynamickej pamäti.

```
X x(a1, a2, ..., an);
```

Oba uvedené príkazy zakladajú automaticky inštanciu triedy **X**. V prvom príkaze je vytvorená inštancia inicializovaná bezparametrickým konštruktorom (či už implicitným, alebo explicitným). V druhom príkaze je inštancia po svojej alokácii inicializovaná explicitným parametrickým konštruktorom s vhodne zvolenou súpravou vstupných argumentov. Bez ohľadu na syntaktickú reprezentáciu automatickej inštanciácie triedy **X** je nutné spomenúť, že zostrojená inštancia bude situovaná na zásobníku programového vlákna a nie v halde (dynamickej pamäťovej oblasti).

Za predpokladu, že sa príkaz uskutočňujúci automatickú inštanciáciu triedy **X** nachádza v tele funkcie (metódy), vieme presne predpovedať okamih zániku automatickej inštancie. Tento okamih je ohraničený posledným programovým príkazom, ktorý sa v tele funkcie nachádza. Vzápätí dochádza k odstráneniu všetkých dátových objektov zo zásobníka. Pri tejto príležitosti bude zlikvidovaná aj automaticky skonštruovaná inštancia triedy **X**. Ešte predtým však dôjde k implicitnej aktivácii deštruktora inštancie, ktorý dostane príležitosť vykonať požadované finalizačné aktivity (súvisiace najmä s uvoľnením dynamicky prideleného pamäťového priestoru, uzatvorením vstupno-výstupných dátových prúdov atď.).

V jazyku C# 3.0 neexistuje spôsob, ktorý by viedol k automatickej inštanciácii triedy a k alokácii automatickej inštancie na zásobníku programového vlákna. Inštancie tried budú v tomto prostredí vždy alokované v riadenej halde (dynamickej pamäťovej oblasti).

Programy napísané v jazyku C++ nemôžu na rozdiel od aplikácií jazyka C# 3.0 ťažiť z prítomnosti automatického správcu pamäte[18]. Správca pamäte eliminuje množinu chýb, ktoré vznikali najmä z dôvodu nekorektnej dealokácie inštancií tried. V prostredí jazyka C++ sa veľmi často stávalo, že vývojár zabudol dynamicky alokovanú inštanciu triedy v príhodnej chvíli uvoľniť, alebo sa pokúsil dealokovať už raz uvoľnenú inštanciu. Ešte závažnejšie chyby vznikali pri tzv. kruhových referenciách, kedy existovala hierarchia navzájom prepojených inštancií, no absentovala algoritmicky presná distribúcia zodpovednosti za ich dealokáciu. V aplikáciách .NET naprogramovaných v jazyku C# 3.0 sú inštancie tried pod stálou kontrolou automatického správcu pamäte, ktorý analyzuje objekty a ak diagnostikuje ich nedosiahnuteľnosť, tak zabezpečí ich automatickú dealokáciu.

[18] Hoci na softvérovom trhu existujú automatickí správcovia pamäte pre jazyk C++, vo všetkých prípadoch ide o produkty softvérových firiem tretích strán. ISO štandard jazyka C++ nešpecifikuje žiaden vstavaný stroj, ktorý by automaticky uvoľňoval nepotrebné inštancie tried.

2.2 Analýza životných cyklov inštancií tried

V tejto kapitole podáme exaktné vysvetlenie analýzy životných cyklov inštancií tried ako používateľsky deklarovaných odkazových dátových typov.

Životný cyklus inštancie triedy sa skladá z týchto štádií:

1. Vytvorenie inštancie triedy.
2. Využívanie služieb inštancie triedy.
3. Ukončenie života inštancie triedy.

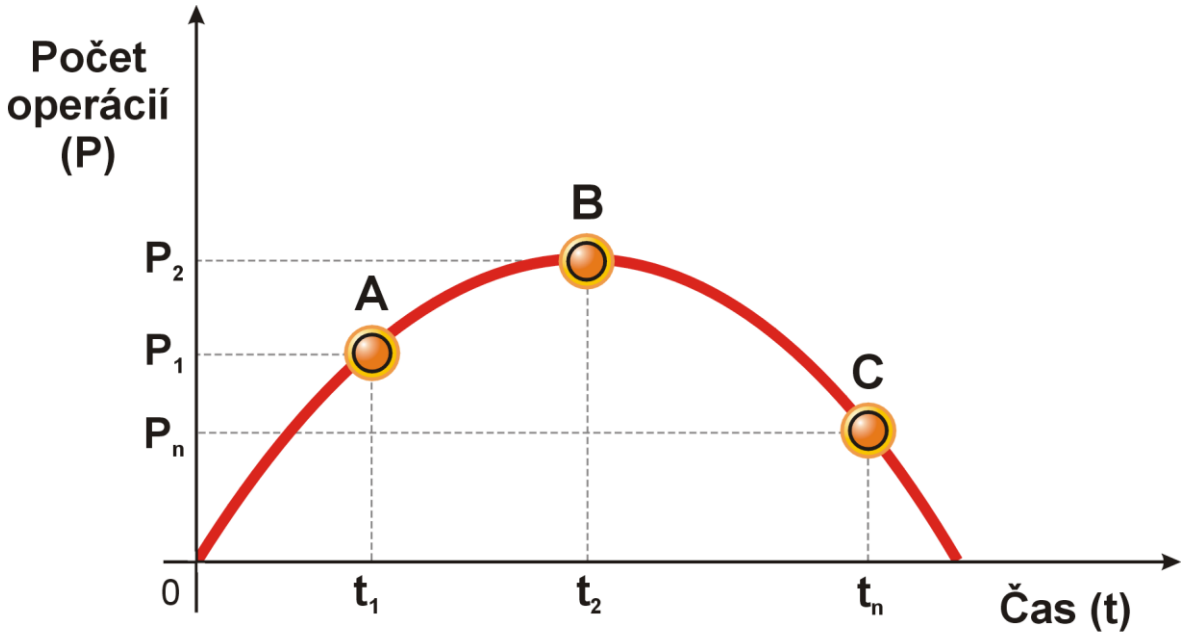
Inštancia triedy je výsledkom inštanciačného procesu, ktorý bol podrobne rozobratý v časti *2.1.3 Inštanciácia triedy a použitie zrodenej inštancie*. V 1. štádiu životného cyklu sú z pohľadu ďalšieho života inštancie triedy dôležité predovšetkým dve operácie: alokácia inštancie a inicializácia inštancie. Len čo je inštancia triedy alokovaná v riadenej halde a náležite inicializovaná, môže začať poskytovať služby svojim klientom. Počnúc týmto okamihom vstupuje inštancia triedy do 2. štádia svojho životného cyklu, ktoré je z pohľadu celkovej analýzy najvýznamnejšie. Inštancia triedy vystupuje ako server, ktorý na požiadanie plní požiadavky konečnej a neprázdnej množiny klientov. Štýl správania sa inštancie triedy je plne determinovaný komunikačným modelom klient – server. 3. etapa životného cyklu inštancie triedy je charakterizovaná vykonaním finalizačných prác[19] a dealokáciou inštancie z riadenej haldy. Dealokácia pamäťového priestoru je spojená s likvidáciou inštancie, čím sa končí jej životný cyklus.

2. etapu životného cyklu inštancie triedy možno charakterizovať funkciou (f), ktorá analyzuje počet operácií (P) realizovaných inštanciou v priebehu určitého časového intervalu (t):

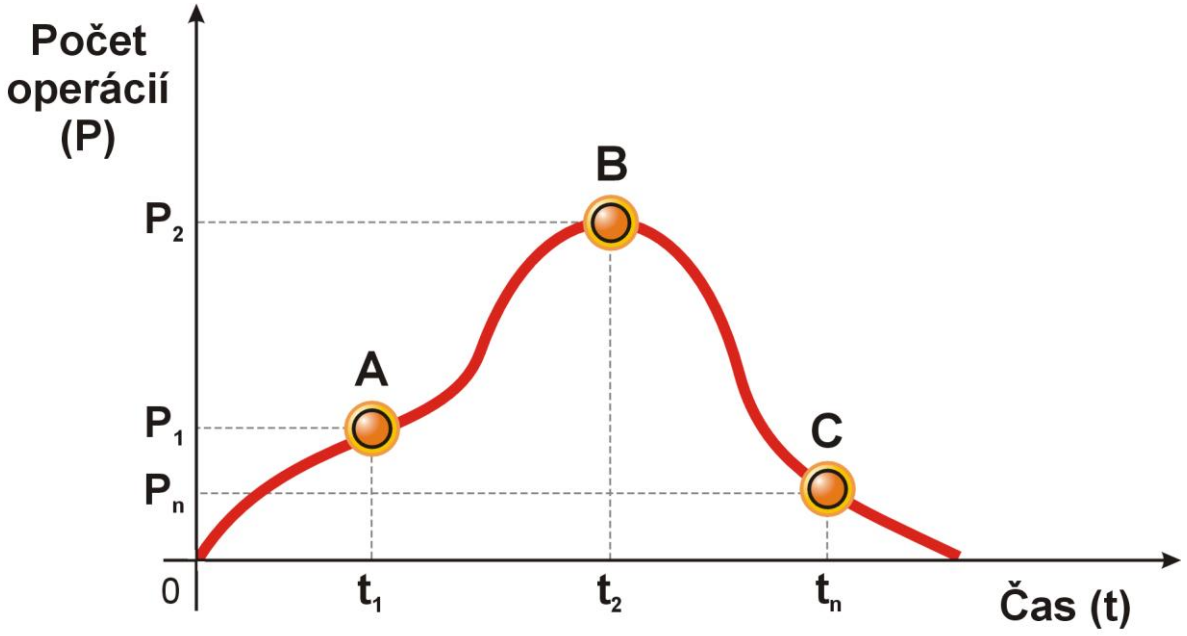
$$f(t) = P, \forall t \in < 1, 2, \dots, n >$$

Počet operácií vykonávaných inštanciou triedy je priamo závislý od počtu správ, ktoré inštancia triedy prijíma od klientov. Vizuálnu interpretáciu priebehu 2. etáp vybraných inštancií tried uvádzajú obr. 18 – 20.

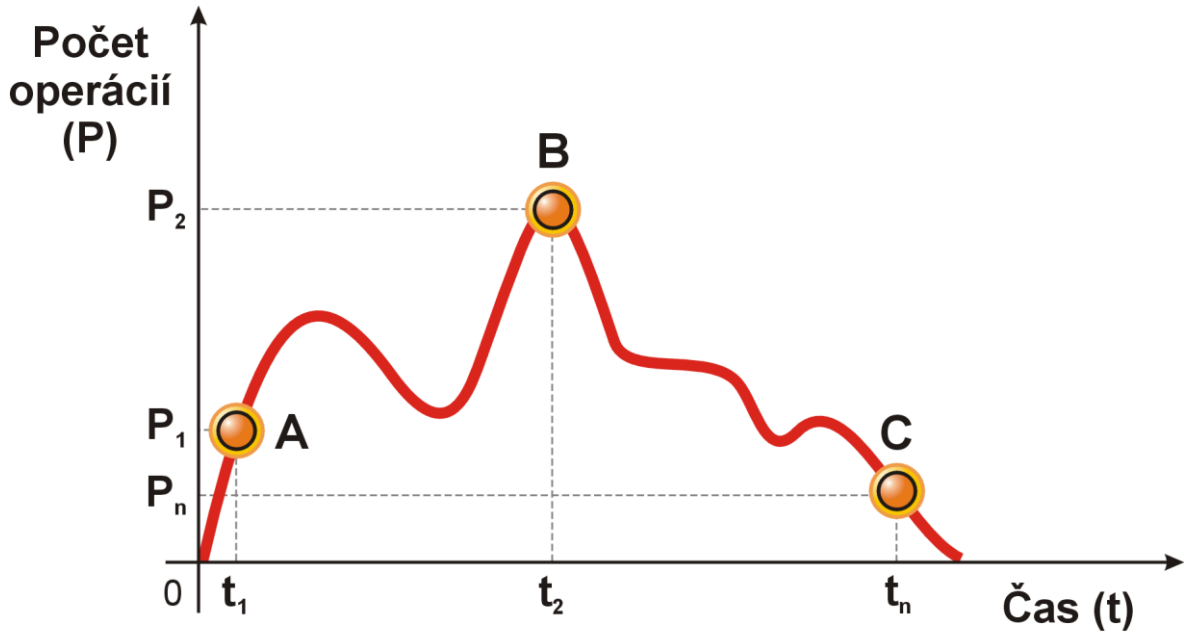
[19] Spektrum finalizačných prác realizovaných pred deštrukciou inštancie triedy závisí od finalizačného modelu. Pri implicitnej finalizácii nie je réžia automatického správcu pamäte nijako citeľná, no pri explicitnej finalizácii musí automatický správca pamäte spustiť finalizér inštancie a vykonať všetky programové príkazy, ktoré finalizér obsahuje. Počet akcií, ktoré musí finalizér spracovať, priamo ovplyvňujú časovú náročnosť explicitnej finalizácie inštancie triedy.



Obr. 18: Vizualizácia 2. štádia životného cyklu inštancie triedy (1. ukážka)



Obr. 19: Vizualizácia 2. štádia životného cyklu inštancie triedy (2. ukážka)



Obr. 20: Vizualizácia 2. štádia životného cyklu inštancie triedy (3. ukážka)

Napriek tomu, že krivka grafu monitorujúca činnosť inštancie triedy môže mať variabilný priebeh, vždy dokážeme spoľahlivo identifikovať tri základné body:

1. Bod **A** so súradnicami [t_1, P_1]. V tomto bode inštancia vstupuje do 2. etapy svojho životného cyklu. Môžeme teda povedať, že inštancia úspešne prekonala 1. štádium, v ktorom došlo k jej alokácii a inicializácii. Podľa charakteru inštanciačného procesu môže existovať medzi 1. a 2. štádiom životného cyklu inštancie rôzne dlhý časový interval. V niektorých prípadoch je vzniknutá časová latencia minimálna – tento jav zaznamenávame najmä pri inštanciách, ktoré nevyžadujú náročnú inicializáciu. Naopak, ak inštancia vytvára v inicializačnej fáze nové dátové objekty, alebo alokuje pamäť pre dynamické dátové štruktúry, do 2. etapy svojho životného cyklu vstupuje s citeľnejším oneskorením.

2. Bod **B** so súradnicami [t_2, P_2]. Toto je bod maximálneho pracovného vyťaženia inštancie triedy. Bod maximálnej aktivity dosahuje inštancia triedy v momente, keď poskytuje svoje služby maximálnemu počtu klientov. Na rozdiel od bodu **A** je možné, aby inštancia triedy dosiahla bod maximálneho pracovného vyťaženia aj viackrát počas 2. etapy svojho životného cyklu. Podotknime, že inštancia triedy môže byť naprogramovaná tak, aby bola schopná sama flexibilne modifikovať množstvo maximálneho pracovného vyťaženia, resp. maximálny počet obsluhujúcich klientov.

3. Bod **C** so súradnicami [t_n, P_n]. Tento bod predstavuje hraničný bod, v ktorom sa končí 2. etapa životného cyklu inštancie triedy a začína sa 3. etapa – finalizácia.

2.2.1 Finalizácia inštancií tried

3. štádiom životného cyklu inštancie triedy je finalizácia. Ako sme uviedli, v prostredí jazyka C# 3.0 rozlišujeme medzi implicitnou a explicitnou finalizáciou. Ak inštancia triedy neplánuje v 3. štádiu svojho životného cyklu vykonať rozsiahle finalizačné práce, jej finalizácia bude implicitná. S implicitnou finalizáciou sa stretávame pri všetkých inštanciách, ktoré vznikli z tried, v telách ktorých sa nenachádzali definície finalizérov. Implicitná finalizácia je výkonnostne priaznivejšia, pretože vyžaduje realizáciu len niekoľkých pracovných cyklov automatického správcu pamäte. Pri implicitnej finalizácii je inštancia triedy uvoľnená z riadenej haldy bez nutnosti spustenia jej finalizéra.

Ak bola na vytvorenie inštancie použitá trieda, ktorá definuje svoj finalizér, tak takáto inštancia bude vyžadovať explicitnú finalizáciu. To znamená, že skôr ako bude môcť byť inštancia triedy dealokovaná z riadenej haldy, bude musieť byť spustený jej finalizér a rovnako budú musieť byť spracované všetky programové príkazy, ktoré sa v tele finalizéra nachádzajú. Skôr, ako budeme môcť presnejšie objasniť proces explicitnej finalizácie inštancií tried v jazyku C# 3.0, musíme sa zoznámiť so stavbou riadenej haldy a riadiacimi algoritmami automatického správcu pamäte.

2.2.2 Stavba riadenej haldy a riadiace algoritmy automatického správcu pamäte

Vo fyzickom procese aplikácie .NET je oblasť dynamickej pamäte (riadená halda) kontrolovaná automatickým správcom pamäte. Riadená halda je segmentovaná do troch objektových generácií, ktoré sú identifikovateľné podľa svojho poradového čísla. Ide o nasledujúce objektové generácie:

1. Objektová generácia č. 0 (1. objektová generácia).
2. Objektová generácia č. 1 (2. objektová generácia).
3. Objektová generácia č. 2 (3. objektová generácia).

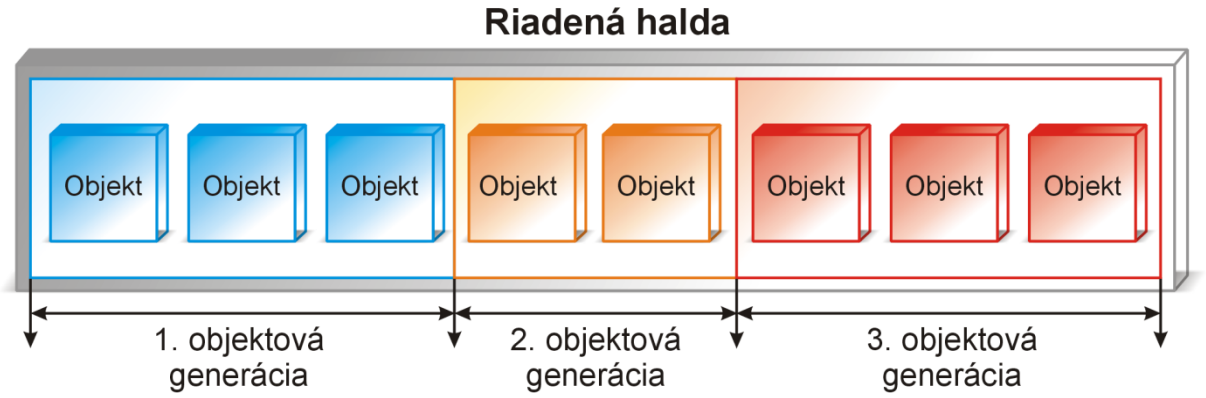
Každá objektová generácia disponuje alokačnou kapacitou, určujúcou počet objektov, ktoré je možné v príslušnej objektovej generácii uskladniť.[20] Objekty sú z hľadiska času, počas ktorého pracujú podľa modelu klient – server, klasifikované do troch generácií:

1. Najmladšie objekty (objekty len nedávno vytvorené).
2. Objekty so stredne dlhým životným cyklom.

[20] Alokačná kapacita jednotlivých objektových generácií je rôzna: pohybuje sa od stoviek kilobajtov až po niekoľko megabajtov. Virtuálny exekučný systém je schopný dynamicky meniť alokačné kapacity objektových generácií podľa aktuálneho aplikačného kontextu. Ak napríklad aplikácia generuje veľa objektov s krátkodobou životnosťou, virtuálny exekučný systém zvýši alokačnú kapacitu 1. objektovej generácie. Naopak, ak sa aplikácia spolieha najmä na objekty s dlhou životnosťou, najväčšiu alokačnú kapacitu bude mať 3. objektová generácia.

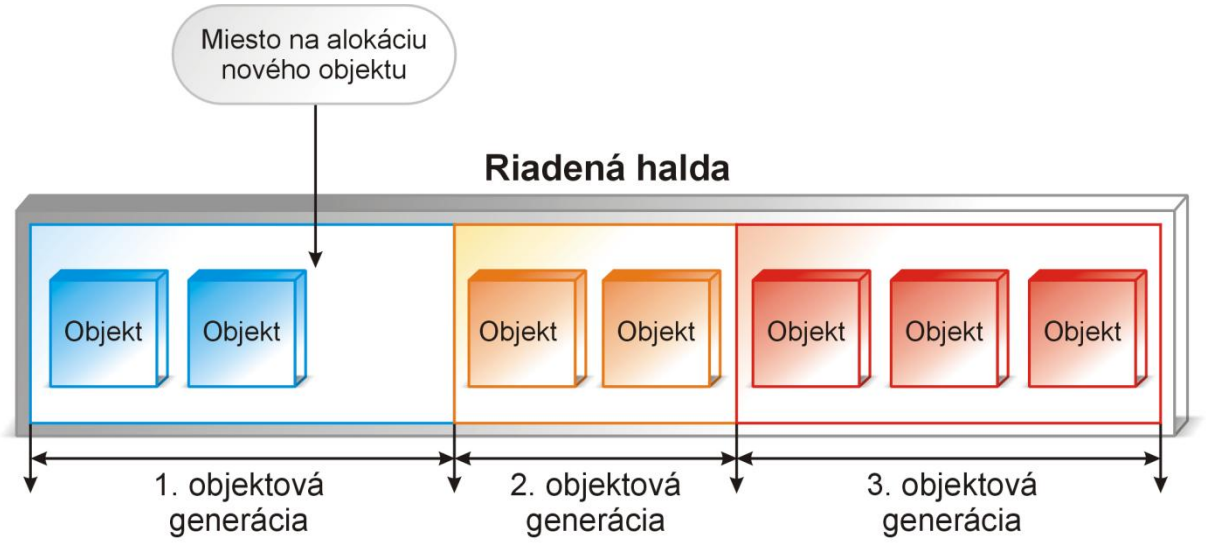
3. Najstaršie objekty (objekty s dlhou dobou životnosti).

Riadená halda s objektovými generáciami je znázornená na obr. 21.



Obr. 21: Generačné zloženie riadenej haldy

Ako si môžeme všimnúť, objekty v objektových generáciách sú umiestnené v sekvenčnej postupnosti, čím vypĺňajú súvislý pamäťový blok. Takéto usporiadanie je efektívne, pretože umožňuje veľmi rýchlo vyhľadať objekt, s ktorým chceme pracovať[21]. Riadiace algoritmy automatického správcu pamäte garantujú, že medzi objektmi nebudú vznikať žiadne prázdne miesta, ktorých prítomnosť by znižovala efektivitu pri práci s objektmi. Automatický správca pamäte vždy uchováva smerník identifikujúci pamäťovú pozíciu, na ktorej bude môcť byť alokovaný nový objekt (obr. 22).



Obr. 22: Smerník determinujúci pozíciu novo alokovaného objektu v 1. objektovej generácii

[21] V jazyku C++ sú objekty na halde uchovávané inak. V tomto prostredí je halda dynamickou dátovou štruktúrou, ktorá pracuje ako zreťazený zoznam. Každá bunka zoznamu obsahuje objekt a rovnako aj smerník, ktorý odkazuje na ďalší dosiahnuteľný objekt. Prechádzanie zreťazeného zoznamu za účelom prístupu k požadovanému objektu je tak časovo náročnejšie. Správa objektov v jazyku C# 3.0 sa skôr podobá na správu inštancií hodnotových dátových typov na zásobníku.

Všetky objekty, ktoré vzniknú, sa z pohľadu pamäťového manažmentu najskôr ocitajú v 1. objektovej generácii. 1. objektová generácia je kapacitne projektovaná tak, aby dokázala absorbovať novo vznikajúce objekty. Keď sa kapacita 1. objektovej generácie zaplní a napriek tomu príde od aplikácie požiadavka na zostrojenie nového objektu, automatický správca pamäte uskutoční tzv. kolekciu 1. objektovej generácie. V procese kolekcie dochádza k preskenovaniu 1. objektovej generácie, pričom cieľom tejto akcie je určenie množiny nedosiahnuteľných objektov. Nedosiahnuteľné objekty sú nepoužívané objekty, na ktoré nie sú nasmerované žiadne odkazy. Takéto objekty môžu byť podrobené finalizácii.

Keď automatický správca pamäte zahájí kolekciu 1. objektovej generácie, predpokladá, že všetky prítomné objekty sú nedosiahnuteľné. Činnosť správcu pamäte pokračuje prechádzaním stromu koreňov a generovaním grafu všetkých dosiahnuteľných objektov. Strom koreňov je dátová štruktúra, ktorá obsahuje odkazy (objektové referencie). Tieto odkazy môžu byť uložené na rôznych miestach, no najčastejšie sú uskladnené v premenných odkazových dátových typov. Graf dosiahnuteľných objektov zoskupuje odkazy na objekty, ktoré sú „živé“ – ide teda o objekty, ktoré aplikácia .NET stále využíva. Naopak, všetky ostatné objekty, na ktoré sa odkazy v grafe nenachádzajú, môže automatický správca pamäte považovať za nedosiahnuteľné objekty.

Predpokladajme, že automatický správca pamäte zistí, že v 1. objektovej generácii sa nachádzajú 3 nedosiahnuteľné objekty. Keďže tieto objekty už nie sú viac potrebné, môžu byť finalizované. Rozviňme naše úvahy ďalej a povedzme, že 2 zo spomenutej trojice objektov smú byť implicitne finalizované. Ak je to tak, tieto objekty budú podrobené deštrukcii a ich alokačná kapacita bude prístupná na budúce použitie. Zostáva nám však jeden objekt, ktorý vyžaduje explicitnú finalizáciu.

Ak založený objekt vzišiel z triedy, ktorá explicitne definuje svoj finalizér, tak ešte pred inicializáciou tohto objektu (pred aktiváciou konštruktora objektu) bol smerník naň pridaný do zoznamu objektov vyžadujúcich explicitnú finalizáciu (ZOVEF). Keď automatický správca pamäte diagnostikuje nedosiahnuteľný objekt ako objekt, ktorý vyžaduje explicitnú finalizáciu, vie, že je nutné aktivovať jeho finalizér. Preto správca pamäte presunie smerník na objekt zo zoznamu objektov vyžadujúcich explicitnú finalizáciu do zoznamu dosiahnuteľných objektov (ZDO). Po tejto operácii dochádza k obnoveniu stavu objektu, pretože z pôvodne nedosiahnuteľného objektu spravil automatický správca pamäte dosiahnuteľný objekt. V tejto súvislosti je veľmi dôležité upozorniť na jednu zásadnú skutočnosť: správca pamäte vykonáva obnovenie stavu objektu len za účelom jeho explicitnej finalizácie. Zmyslom obnovenia stavu objektu je „oživenie“ pôvodne nedosiahnuteľného objektu. Tak získavame „živý“ objekt, na ktorom môže byť spustený finalizér. (Z uvedeného jasne vyplýva, že aktivácia finalizéra nemôže byť uskutočnená v spojení s nedosiahnuteľným objektom.)

Finalizácia objektu, resp. objektov, na ktoré sa odkazuje ZDO, je realizovaná asynchrónne na samostatnom, tzv. finalizačnom programovom vlákne. Hoci poradie volaní finalizérov jednotlivých objektov nie je determinované, automatický správca pamäte garantuje, že budú spracované finalizéry všetkých objektov, ktoré požadujú svoju explicitnú finalizáciu.

Po vykonaní všetkých príkazov finalizéra bude objekt opäť označený ako nedosiahnuteľný, avšak tento raz bude opatrený príznakom, ktorý vraví, že objekt už bol explicitne finalizovaný. Pri nasledujúcej kolekcii bude nedosiahnuteľný a už explicitne finalizovaný objekt jednoducho zlikvidovaný, a to bez akýchkoľvek dodatočných akcií. Pri druhej kolekcii je teda analyzovaný objekt okamžite uvoľnený, podobne ako implicitne finalizované objekty.

Samozrejme, môže sa stať, že automatický správca pamäte objaví v 1. objektovej generácii množstvo dosiahnuteľných objektov, ktoré nemožno finalizovať (proste z toho dôvodu, že stále existujú klienti, ktorí využívajú služby týchto objektov). Vzhľadom na to, že cieľom automatického správcu pamäte je uvoľniť pamäťovú kapacitu na alokáciu nových objektov, budú objekty, ktoré prežijú kolekciu, transportované do objektovej generácie s vyšším poradovým číslom. Dosiahnuteľné objekty z 1. objektovej generácie sa dostanú do 2. objektovej generácie. Presun objektov medzi objektovými generáciami má významné praktické implikácie:

1. V objektovej generácii s nižším poradovým číslom vznikne po premiestnení objektu prázdne miesto. Toto prázdne miesto predstavuje pamäťovú medzeru, pričom je ohraničené pamäťovým priestorom, ktorý pred svojím premiestnením do objektovej generácie s vyšším poradovým číslom okupovala inštancia triedy. Je evidentné, že čím viac objektov bude z nižšej objektovej generácie propagovaných do vyššej objektovej generácie, s tým väčšou pravdepodobnosťou budú vznikať pamäťové medzery. Intenzívna kreácia pamäťových medzier zapríčiňuje fragmentáciu objektovej generácie riadenej haldy. V záujme optimalizovaného prístupu k objektom je nutné pamäťové medzery odstrániť a zabezpečiť sekvenčné usporiadanie objektov. Eliminácia pamäťových medzier sa uskutočňuje v procese defragmentácie pamäte objektovej generácie. Po defragmentácii budú objekty v objektovej generácii usporiadané sekvenčne a prístup k nim bude rovnako efektívny ako predtým.

2. Pri presune objektov z 1. do 2. objektovej generácie sa pôvodné odkazy na tieto objekty (smerujúce do 1. objektovej generácie) stávajú neplatnými. Keďže objekty sa po transporte ocitnú v 2. objektovej generácii, je potrebné aktualizovať pôvodné odkazy na ne. Aktualizáciu odkazov na objekty vykonáva automatický správca pamäte vo svojej vlastnej réžii. Len čo sú odkazy aktualizované, objekty sú aj naďalej priamo dosiahnuteľné, a to aj napriek tomu, že sa zmenila ich fyzická pozícia na riadenej halde.

Kolekcia 1. objektovej generácie riadenej haldy je najčastejšie uskutočnená pri zaplnení tejto objektovej generácie. Ak sa automatickému správcovi pamäte podarí získať dostatok pamäťového priestoru na alokáciu nových objektov, bude 1. objektová generácia jedinou, v ktorej sa kolekcia vykoná. Hoci za bežných okolností možno získať dostatok pamäťového priestoru kolekciou 1. objektovej generácie, existujú situácie, kedy nie je automatický správca pamäte schopný uvoľniť dostatočný alokačný priestor. V týchto prípadoch musí správca pamäte realizovať kolekciu aj v generácii s vyšším poradovým číslom, teda v 2. objektovej generácii. Počas kolekcie dochádza k nasledujúcim akciám:

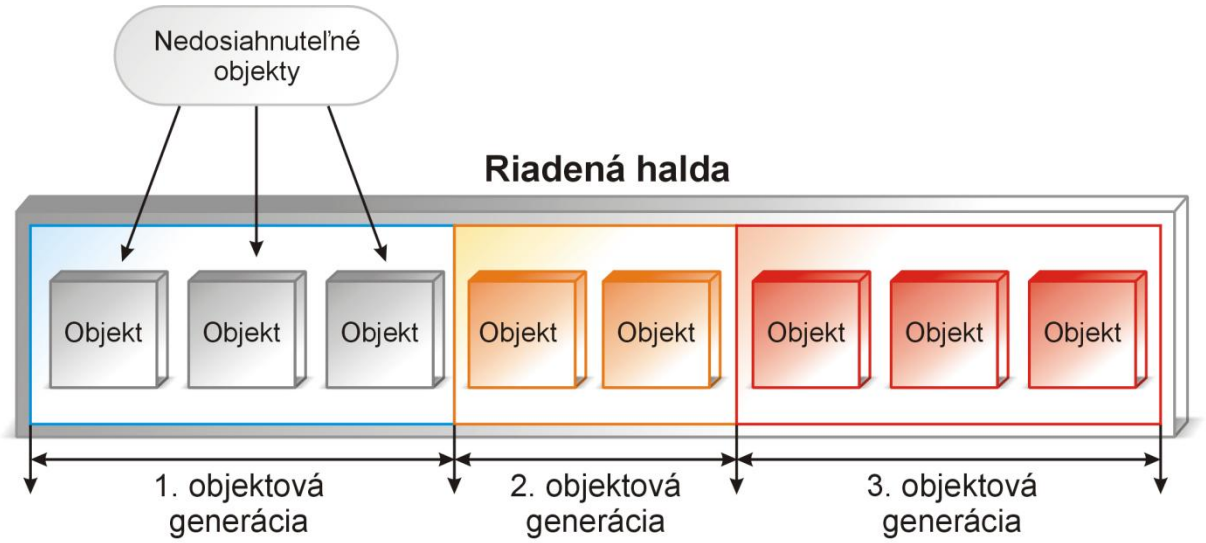
- Diagnostikuje sa kolekcia nedosiahnuteľných objektov 2. objektovej generácie.

- Nedosiahnuteľné objekty sú finalizované. Ak objekt nevyžaduje explicitnú finalizáciu, bude okamžite uvoľnený. Objekty vyžadujúce explicitnú finalizáciu budú explicitne finalizované tak, ako sme už vysvetlili.

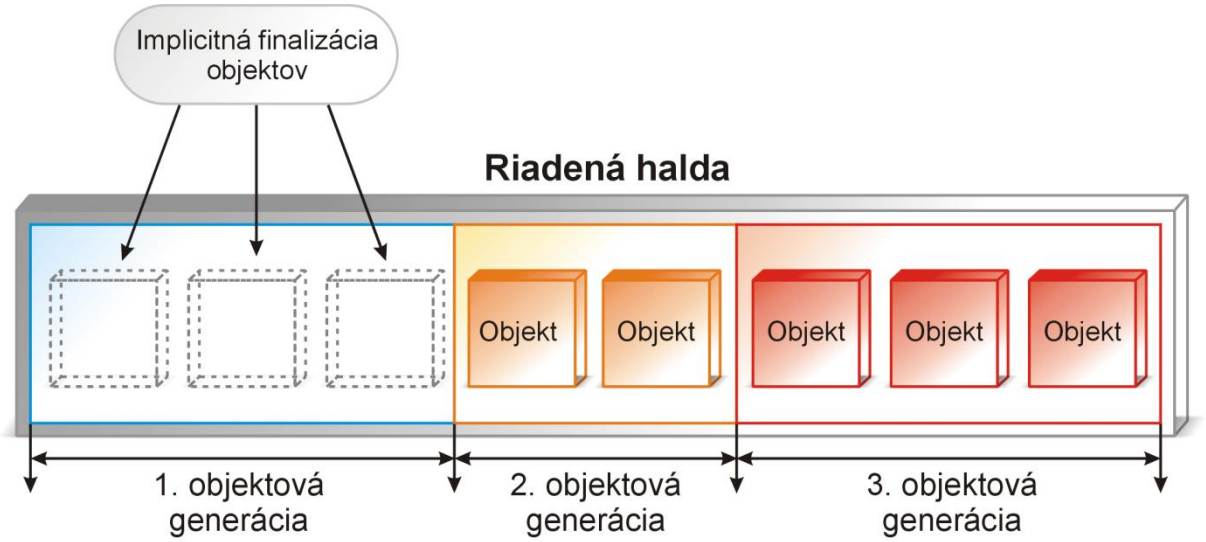
- Dosiahnuteľné objekty, ktoré prežijú kolekciu, patria k objektom s dlhou dobou životnosti, a preto budú prenesené do objektovej generácie s vyšším poradovým číslom, teda do 3. objektovej generácie.

- Objekty z 1. objektovej generácie, ktoré prežili kolekciu, budú presunuté do 2. objektovej generácie.

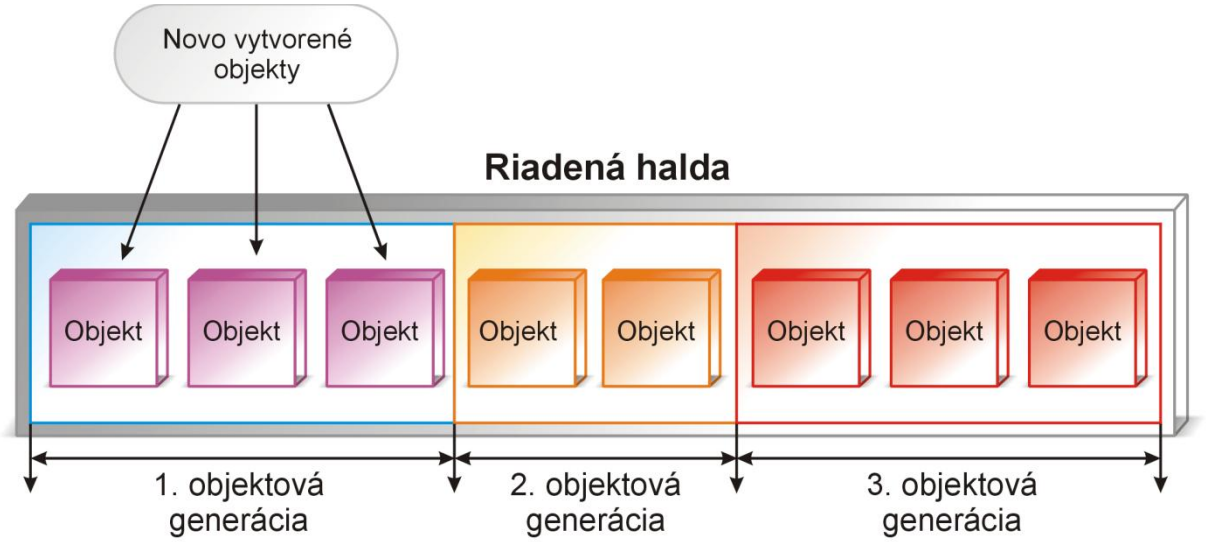
Vo výnimočných prípadoch sa môže stať, že vyhovujúci pamäťový priestor nebude nájdený ani v 1. objektovej generácii a ani v 2. objektovej generácii. Vtedy automatický správca pamäte vykoná kolekcie všetkých troch objektových generácií a uvoľní nedosiahnuteľné objekty. Dosiahnuteľné objekty 3. objektovej generácie v tejto generácii aj naďalej zostávajú, pretože už žiadna ďalšia objektová generácia neexistuje. Ak by sa ani po 3-fázovej kolekcii nepodarilo automatickému správcovi pamäte dealokovať potrebné množstvo pamäťového priestoru, virtuálny exekučný systém ohlási chybu pri pokuse o alokáciu a exekúcia aplikácie .NET bude ukončená. Riadiace algoritmy automatického správcu pamäte pri kolekcii 1. objektovej generácie, implicitnej finalizácii nedosiahnuteľných objektov a alokácii novo vytvorených objektov predstavujú obr. 23 – 25.



Obr. 23: Výsledkom kolekcie 1. objektovej generácie sú 3 nedosiahnuteľné objekty



Obr. 24: Implicitná finalizácia nedosiahnuteľných objektov



Obr. 25: Umiestnenie nových objektov do 1. objektovej generácie

Segmentácia riadenej haldy do troch objektových generácií prináša tieto pozitíva:

1. Klasifikuje objekty podľa času ich životnosti.

2. Umožňuje spravovať objekty samostatne podľa ich príslušnosti k určitej objektovej generácii.

3. Napomáha produktívnejšej správe riadenej haldy. Je totiž vždy jednoduchšie vykonať kolekciu jednej, alebo dvoch objektových generácií, ako uskutočňovať kolekciu celej riadenej haldy.

4. Hoci transporty objektov medzi rôznymi objektovými generáciami sú náročné na prácu procesora, automatický správca pamäte eliminuje pamäťovú fragmentáciu a vždy pozicuje objekty v sekvenčnej postupnosti, čo produkuje minimálne časové latencie pri práci s nimi.

Objekty, ktorých alokačná kapacita je väčšia ako 85 000 bajtov, sú z pohľadu automatického správcu pamäte považované za tzv. kapacitne náročné objekty. Objekty tohto typu sú alokované a manažované na samostatnej riadenej halde. Ide o riadenú haldu pre kapacitne náročné objekty. V tejto riadenej halde sú objekty uskladnené v súvislých pamäťových segmentoch. Na rozdiel od 3-generačnej riadenej haldy, riadená halda pre kapacitne náročné objekty nie je členená na objektové generácie. Namiesto toho vystupuje ako monolitický celok, ktorý je poskladaný z viacerých pamäťových segmentov. Aj životné cykly kapacitne náročných objektov sú riadené automaticky, no s tým rozdielom, že sa neuskutočňujú žiadne presuny objektov. Vzhľadom na to, že objekty sú kapacitne náročné, ich transporty by znižovali výkon a v konečnom dôsledku by pôsobili kontraproduktívne.

2.3 Agregačno-kompozičné vzťahy medzi triedami

V programovacom jazyku C# 3.0 existuje niekoľko možností, ako na syntaktickej úrovni vybudovať medzi viacerými triedami agregačné, resp. kompozičné vzťahy. Jedným riešením je aplikovať techniku vnárania tried, kedy vložíme deklaráciu jednej triedy do tela inej triedy. V tele nadradenej triedy potom definujeme dátovú položku, ktorej typom bude vnorená trieda.

V nasledujúcej praktickej ukážke vytvoríme model počítača, ktorý bude obsahovať procesor určitého typu. Príklad demonštruje vytvorenie kompozičnej väzby medzi triedou **Počítač** a vnorenou triedou **Procesor**. Ide teda o silnejší typ agregácie (kompozíciu), čo je však pochopiteľné, pretože počítač bez procesora nemôže fungovať.

Najskôr uvedieme deklaráciu triedy **Procesor**:

```
// Deklarácia triedy Procesor.
class Procesor
{
    // Definície dátových položiek triedy.
    string názov;
    string viacJadier;
    float  celkováFrekvencia;
    string technológiaHT;

    // Definícia verejne prístupného parametrického inštančného
    // konštruktora triedy.
    public Procesor(string názov, string viacJadier,
       float celkováFrekvencia, string technológiaHT)
    {
        this.názov = názov;
        this.viacJadier = viacJadier;
        this.celkováFrekvencia = celkováFrekvencia;
        this.technológiaHT = technológiaHT;
    }

    // Definície verejne prístupných inštančných skalárnych
    // vlastností triedy.
    public string Názov
    {
        get { return this.názov; }
        set { this.názov = value; }
    }
    public string ViacJadier
    {
        get { return this.viacJadier; }
        set { this.viacJadier = value; }
    }
    public float CelkováFrekvencia
    {
        get { return this.celkováFrekvencia; }
        set { this.celkováFrekvencia = value; }
    }
    public string TechnológiaHT
    {
        get { return this.technológiaHT; }
        set { this.technológiaHT = value; }
    }
}
```

Pokračujeme zostrojením triedy **Počítač**, do tela ktorej vnoríme práve deklarovanú triedu **Procesor**:

```
class Počítač
{
    // Tu sa nachádza deklarácia vnorenej triedy Procesor.
    class Procesor { ... }

    // Definícia dátovej položky, do ktorej budeme môcť uložiť odkaz
    // na inštanciu triedy Procesor.
    Procesor procesor;

    // Definícia verejne prístupného bezparametrického inštančného
    // konštruktora triedy.
    public Počítač()
    {
```

```
            procesor = new Procesor("AMD Turion 64 X2", "Áno", 4.0f, "Nie");
        }

        // Definícia verejne prístupnej inštančnej metódy triedy.
        public void InformácieOProcesorePočítača()
        {
            Console.WriteLine("Informácie o procesore počítača:");
            Console.WriteLine("Názov: {0}\nViac jadier: {1}\n" +
                "Celková frekvencia: {2} GHz\nTechnológia HT: {3}",
                this.procesor.Názov, this.procesor.ViacJadier,
                this.procesor.CelkováFrekvencia,
                this.procesor.TechnológiaHT);
        }
    }
```

Vzhľadom na to, že v tele triedy **Počítač** je umiestnená deklarácia vnorenej triedy **Procesor**, môžeme v dátovej sekcii triedy **Počítač** definovať súkromnú odkazovú premennú s identifikátorom **procesor**, ktorej dátovým typom je vnorená trieda. V tele konštruktora triedy **Počítač** dochádza k založeniu inštancie vnorenej triedy. Týmto spôsobom zabezpečíme, že každá korektne vytvorená inštancia triedy **Počítač** bude disponovať svojím procesorom. Pravdaže, z hľadiska lepšej používateľskej prívetivosti by sme mohli konštruktor triedy **Počítač** preťažiť a ponúknuť tak širšie možnosti pre konfiguráciu procesora vytváraného počítača.

Keďže oblasť platnosti odkazovej premennej **procesor** je vymedzená telom nadradenej triedy, je zrejmé, že životnosť objektu triedy **Procesor** sa bude kryť so životným cyklom objektu triedy **Počítač**.

Verejne prístupná inštančná metóda **InformácieOProcesorePočítača** poskytuje klientom objektu triedy **Počítač** základné údaje o technickej charakteristike procesora, ktorým je počítač osadený.

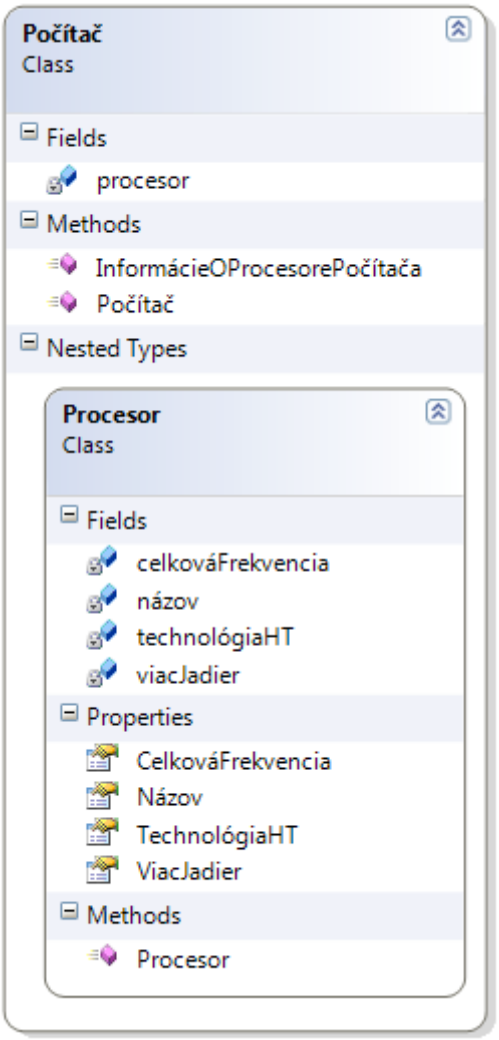
Vizuálny model reprezentujúci diagram triedy **Počítač** je znázornený na obr. 26.

Inštanciácia triedy **Počítač** prebieha podľa štandardného postupu:

```
class Program
{
    static void Main(string[] args)
    {
        // Inštanciácia triedy Počítač.
        Počítač mojePC = new Počítač();
        mojePC.InformácieOProcesorePočítača();
        Console.Read();
    }
}
```

V procese inštanciácie bude najskôr konštruovaný objekt triedy **Počítač**. Aktivácia operátora **new** spôsobí dynamické založenie objektu na riadenej halde. Operátor **new** implicitne vyvolá konštruktor alokovanej inštancie triedy **Počítač**, ktorý zabezpečí

vytvorenie vnoreného objektu triedy **Procesor**. Podotknime, že vnorený objekt nie je pre klientsky programový kód priamo prístupný, čo je v poriadku, pretože naším cieľom je dodržať základné princípy objektovo orientovaného programovania.



Obr. 26: Vizuálne stvárnenie kompozičného vzťahu medzi triedami **Počítač** a **Procesor**

2.4 Dedičnosť

Programovací jazyk C# 3.0 podporuje verejnú jednoduchú dedičnosť. Na rozdiel od jazyka C++ teda absentuje implementácia viacnásobnej dedičnosti, avšak na druhú stranu, túto skutočnosť nemôžeme pokladať za slabú stránku jazyka C# 3.0. Skôr naopak, najmä keď uvážime, že s implementáciou viacnásobnej dedičnosti sa v jazyku C++ viazali nemalé problémy, ktoré bolo často nutné riešiť virtuálnym dedením. Abstrahovanie od viacnásobnej dedičnosti však nie je vlastné len programovaciemu jazyku C# 3.0, ale tiež Jave, ďalšiemu veľmi populárnemu programovaciemu jazyku súčasnosti.

Jednoduchá dedičnosť, s ktorou sa v jazyku C# 3.0 stretávame, je implicitne verejná (nie sú vyžadované a nakoniec ani potrebné iné varianty jednoduchej dedičnosti, ako je súkromná alebo chránená dedičnosť). V procese jednoduchej dedičnosti vytvárame z bázovej triedy odvodenú triedu (podtriedu). Odvodená trieda dedí od svojej bázovej triedy všetky atribúty a metódy. Hoci zdedené budú všetky členy bázovej triedy, nie všetky musia byť z tela odvodenej triedy priamo viditeľné. O tom, ktoré zdedené členy budú viditeľné, rozhodujú ich prístupové modifikátory, ktoré sú uvedené v bázovej triede. Vzťahy vybudované na základe verejnej jednoduchej dedičnosti majú vždy generalizačno-špecializačný charakter. Bázová trieda je všeobecnou šablónou, zatiaľ čo odvodená trieda je jej špecifickejšou implementáciou. Ak sa vyskytuje v reťazci verejnej jednoduchej dedičnosti viacero tried, platí pravidlo, podľa ktorého je každá nasledujúca podtrieda vždy konkrétnejším prípadom svojej nadradenej triedy.

Okrem opätovnej použiteľnosti schopností bázovej triedy môže odvodená trieda dodať novú, doplnkovú funkcionalitu. To sa deje definovaním nových členov, ktoré sa vyskytujú len v odvodenej triede. Podtrieda sa môže taktiež rozhodnúť predefinovať správanie zdedených členov bázovej triedy. V tomto kontexte je možné prekryť zdedenú implementáciu člena bázovej triedy v triede odvodenej, čo je podstatou polymorfizmu implementovaného pomocou verejnej jednoduchej dedičnosti. Na druhú stranu, odvodená trieda je schopná ukryť zdedenú implementáciu člena bázovej triedy tak, že ho nahradí novou implementáciou. Táto technika je známa ako ukrývanie zdedených členov bázovej triedy v podtriede.

Generický syntaktický príkaz pre deklarovanie odvodenej triedy **B**, ktorá vznikla v procese verejnej jednoduchej dedičnosti z bázovej triedy **A**, má nasledujúcu podobu:

```
class B : A
{
  ...
}
```

Zapísaná deklarácia odvodenej triedy predpokladá, že existuje trieda s identifikátorom **A**, ktorá nie je ani zapečatená, ani statická.

Keď vytvoríme inštanciu odvodenej triedy, táto bude obsahovať zdedený podobjekt bázovej triedy. Generický syntaktický príkaz zakladajúci inštanciu odvodenej triedy vyzerá takto:

```
B obj = new B();
```

Keďže inštancia odvodenej triedy v sebe zapuzdruje zdedený podobjekt bázovej triedy, je nutné korektne skonštruovať najskôr ten, a až potom môže byť riadne skonštruovaná inštancia odvodenej triedy. Tomuto princípu zodpovedá aj reťazec volaní inštančných konštruktorov: najskôr je aktivovaný inštančný konštruktor odvodenej triedy, ktorý

však vzápätí (ešte pred vstupom do svojho tela) volá inštančný konštruktor bázovej triedy. Keď je zdedený podobjekt bázovej triedy skonštruovaný a korektne inicializovaný, dokončí sa konštrukcia objektu odvodenej triedy (spracujú sa programové príkazy v tele inštančného konštruktora odvodenej triedy).

Uvažujme deklaráciu bázovej triedy **GrafickýObjekt**:

```
class GrafickýObjekt
{
    public void Vykresliť()
    {
        // Zdrojový kód pre vykreslenie grafického objektu.
    }
}
```

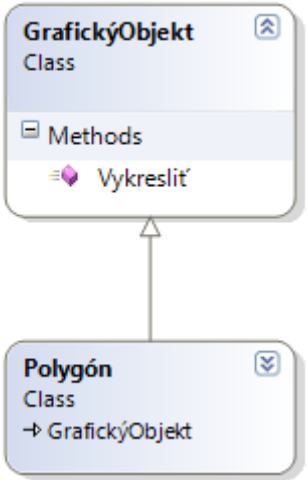
V ďalšom kroku odvodíme od triedy **GrafickýObjekt** novú podtriedu **Polygón**:

```
class  Polygón : GrafickýObjekt
{
    // Telo odvodenej triedy.
}
```

Hoci sme do tela odvodenej triedy nevložili žiadne členy, ktoré by zavádzali istú funkcionalitu, môžeme odvodenú triedu ihneď inštanciovať a zavolať zdedenú verejne prístupnú inštančnú metódu **Vykresliť**:

```
static void Main(string[] args)
{
    Polygón šesťuholník = new Polygón();
    šesťuholník.Vykresliť();
    Console.ReadLine();
}
```

Vizualizácia vzťahu medzi dvomi triedami, ktorý bol vybudovaný na báze verejnej jednoduchej dedičnosti, je zobrazená na obr. 27.



Obr. 27: Vizualizácia vzťahu verejnej jednoduchej dedičnosti medzi triedami

Odvodená trieda sa nemusí uspokojiť len so zdedenou funkcionalitou, ale smie ďalej vylepšovať svoje schopnosti:

```
class Polygón : GrafickýObjekt
{
    public void Posunúť(Vektor v)
    {
        // Zdrojový kód uskutočňujúci transláciu polygónu.
    }
}
```

Deklaráciu odvodenej triedy sme rozšírili o definíciu verejne prístupnej inštančnej parametrickej metódy **Posunúť**, ktorá dokáže uskutočniť transláciu polygónu[22]. Použitie modifikovanej triedy **Polygón** približuje nasledujúci fragment zdrojového kódu:

```
static void Main(string[] args)
{
    Polygón šesťuholník = new Polygón();
    šesťuholník.Posunúť(new Vektor(10, 2, 3));
    Console.ReadLine();
}
```

2.5 Abstraktné triedy

Abstraktná trieda je v programovacom jazyku C# 3.0 trieda, ktorú nie je možné použiť v procese inštanciácie. Nie je teda možné zakladať inštancie abstraktnej triedy. Abstraktná trieda vystupuje v pozícii bázovej triedy, od ktorej môžu byť v procese verejnej jednoduchej dedičnosti odvodzované nové podtriedy. Získané podtriedy môžu byť buď znova abstraktné, alebo inštančné. Ak je podtrieda bázovej abstraktnej triedy inštančného charakteru, dokáže produkovať svoje inštancie.

V deklarácii abstraktnej triedy sa vyskytuje modifikátor **abstract**:

```
[M] abstract class X
{
    // Telo abstraktnej triedy.
}
```

kde:

- **[M]** je prístupový modifikátor abstraktnej triedy.
- **X** je identifikátor abstraktnej triedy.

[22] Formálny parameter metódy prijíma inštanciu hodnotovej štruktúry **Vektor**, ktorá uchováva súradnice vektora, pomocou ktorého bude grafický objekt posunutý.

Pre abstraktné triedy platia tieto zásady:

1. Členy abstraktnej triedy (metódy a vlastnosti) môžu byť abstraktné alebo inštančné.
2. Abstraktný člen abstraktnej triedy je implicitne virtuálny.
3. Abstraktný člen je v tele abstraktnej triedy iba deklarovaný, jeho definícia sa objavuje až v tele inštančnej podtriedy, ktorá je od bázovej abstraktnej triedy odvodená.

Pozrime sa na deklaráciu abstraktnej triedy **BankovýÚčet**:

```
// Deklarácia abstraktnej triedy.
public abstract class BankovýÚčet
{
    // Deklarácia 1. abstraktnej metódy.
    public abstract void Vložiť(uint suma);

    // Deklarácia 2. abstraktnej metódy.
    public abstract void Uhradiť(uint suma);

    // Deklarácia abstraktnej skalárnej vlastnosti, ktorá je určená
    // len na čítanie.
    public abstract uint Stav
    {
        get;
    }
}
```

V tele našej abstraktnej triedy sa nachádzajú výhradne abstraktné členy: dve abstraktné metódy a jedna abstraktná skalárna vlastnosť. Je dôležité poukázať na skutočnosť, že abstraktné členy abstraktnej triedy sú zavádzané len prostredníctvom svojich deklaračných príkazov (prototypov). Keby sme chceli do tela deklarovanej abstraktnej triedy umiestniť definície abstraktných členov, kompilátor by nás zastavil s hlásením udávajúcim nemožnosť uskutočnenia takejto operácie. Na druhú stranu, v deklarácii abstraktnej triedy sa môžu nachádzať aj neabstraktné členy. Pritom platí, že akýkoľvek neabstraktný člen vyskytujúci sa v tele abstraktnej triedy musí byť v tejto triede riadne definovaný.

Od deklarovanej abstraktnej triedy odvodíme inštančnú triedu **BežnýBankovýÚčet**:

```
// Deklarácia podtriedy abstraktnej triedy.
public class BežnýBankovýÚčet : BankovýÚčet
{
    // Definície súkromných dátových položiek triedy.
    private uint stav, minimálnyZostatok;
    private string číslo, majiteľ;

    // Definícia parametrického inštančného konštruktora.
    public BežnýBankovýÚčet(string číslo, string majiteľ,
        uint minimálnyZostatok, uint počiatočnýVklad)
    {
```

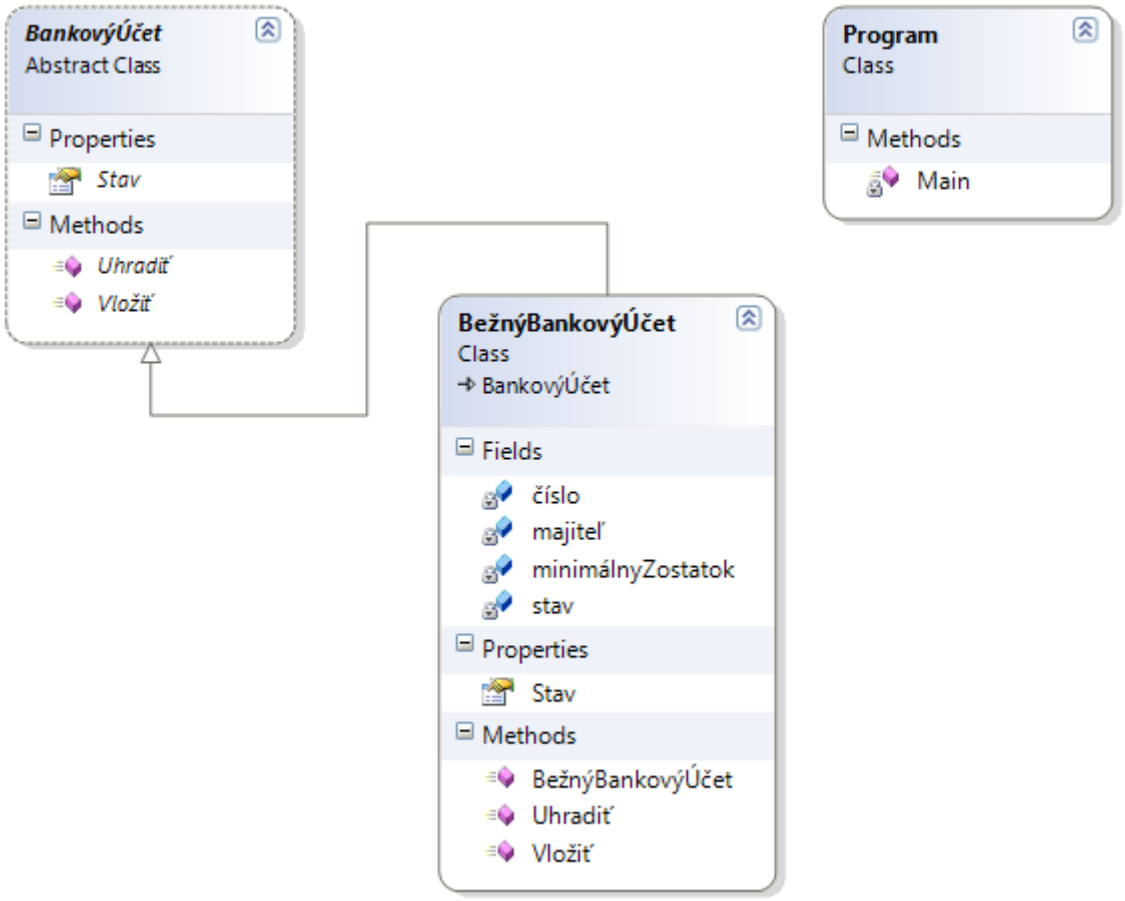
```
        this.číslo = číslo;
        this.majiteľ = majiteľ;
        this.minimálnyZostatok = minimálnyZostatok;
        stav += počiatočnýVklad;
    }

    // Definícia 1. prekrývajúcej inštančnej metódy.
    public override void Vložiť(uint suma)
    {
        stav += suma;
    }

    // Definícia 2. prekrývajúcej inštančnej metódy.
    public override void Uhradiť(uint suma)
    {
        if (!((int)stav - (int)suma < minimálnyZostatok))
            stav -= suma;
    }

    // Definícia prekrývajúcej inštančnej skalárnej vlastnosti.
    public override uint Stav
    {
        get { return this.stav; }
    }
}
```

Všetky zdedené abstraktné členy abstraktnej bázovej triedy sú v tele inštančnej odvodenej triedy explicitne prekryté. To je nevyhnutný predpoklad na to, aby sme jednotlivé implementované členy mohli obdariť požadovanou aplikačnou logikou. Keďže deklarovaná podtrieda je inštančného charakteru, môžeme vytvoriť jej inštanciu (bankový účet) a vykonať niekoľko bankových transakcií:

```
class Program
{
    static void Main(string[] args)
    {
        // Inštanciácia podtriedy.
        BežnýBankovýÚčet účet = new BežnýBankovýÚčet("1234567",
            "Jonatán Veselý", 500, 5000);
        // Prvý vklad peňazí na bankový účet.
        účet.Vložiť(10000);
        // Prvá úhrada peňazí z bankového účtu.
        účet.Uhradiť(4000);
        // Druhá úhrada peňazí z bankového účtu.
        účet.Uhradiť(12000);
        Console.WriteLine("Stav na účte: {0}  USD.", účet.Stav);
        Console.Read();
    }
}
```

Po spustení programu zistíme, že aktuálny stav na našom bežnom bankovom účte je 11000 USD. To je spôsobené odmietnutím poslednej transakcie z dôvodu prekročenia minimálneho zostatku na bankovom účte.

Vizualizácia vzťahu medzi bázovou abstraktnou triedou a inštančnou odvodenou triedou je znázornená na obr. 28.



Obr. 28: Vzťah medzi bázovou abstraktnou triedou a odvodenou inštančnou triedou

2.6 Zapečatené triedy

Zapečatená trieda je v programovacom jazyku C# 3.0 trieda, ktorú nie je možné použiť ako bázovú triedu v procese verejnej jednoduchej dedičnosti. Nie je teda možné vytvárať podtriedy zapečatenej triedy.

Hoci od zapečatenej triedy nemožno dediť, jej schopnosť generovať inštancie nie je nijakým spôsobom obmedzená. Ako zapečatené sú zvyčajne deklarované triedy, ktoré disponujú úplnou funkcionalitou.

V deklarácii zapečatenej triedy sa vyskytuje modifikátor **sealed**:

```
[M] sealed class X
{
    // Telo zapečatenej triedy.
}
```

kde:

- **[M]** je prístupový modifikátor zapečatenej triedy.

- **X** je identifikátor zapečatenej triedy.

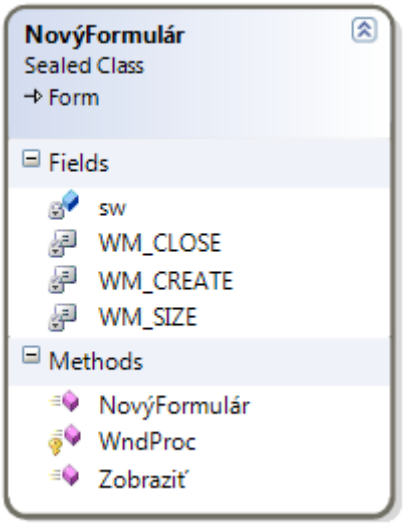
Nasleduje deklarácia zapečatenej triedy **NovýFormulár**:

```
using System;
using System.Windows.Forms;
using System.IO;

namespace diz
{
    // Deklarácia zapečatenej triedy.
    public sealed class NovýFormulár : Form
    {
        // Definície súkromných dátových položiek triedy.
        private const int WM_CREATE = 0x0001;
        private const int WM_CLOSE = 0x0010;
        private const int WM_SIZE = 0x0005;
        private StreamWriter sw;

        // Definícia inštančného konštruktora triedy.
        public NovýFormulár()
        {
            sw = new System.IO.StreamWriter
               (Application.StartupPath + "\\Súbor.txt");
            sw.WriteLine("***** ZÁZNAMNÍK SPRÁV OPERAČNÉHO " +
               "SYSTÉMU *****");
            sw.WriteLine("------- Začiatok záznamu -------");
        }

        // Definícia prekrývajúcej metódy WndProc.
        protected override void WndProc(ref Message m)
        {
            switch (m.Msg)
            {
                // Zachytenie správy WM_CREATE.
                case WM_CREATE:
                    sw.WriteLine("Okno aplikácie bolo vytvorené " +
                       " (čas [" + DateTime.Now.Hour + ":" +
                       DateTime.Now.Minute + ":" +
                       DateTime.Now.Second + "], správa " +
                       "WM_CREATE).");
                    break;
                // Zachytenie správy WM_SIZE.
                case WM_SIZE:
                    // Ak bolo okno aplikácie minimalizované...
                    if ((int)(m.WParam) == 1)
                        sw.WriteLine("Okno aplikácie bolo " +
                           "minimalizované (čas [" +
                           DateTime.Now.Hour + ":" +
                           DateTime.Now.Minute + ":" +
                           DateTime.Now.Second + "], správa " +
                           "WM_SIZE; WParam == 1).");
                    // Ak prišlo k maximalizácii okna aplikácie...
                    else if ((int)(m.WParam) == 2)
                        sw.WriteLine("Okno aplikácie bolo " +
                           "maximalizované (čas [" +
                           DateTime.Now.Hour + ":" +
                           DateTime.Now.Minute + ":" +
                           DateTime.Now.Second + "], správa " +
                           "WM_SIZE; WParam == 2).");
```

```
                    break;
                // Zachytenie správy WM_CLOSE.
                case WM_CLOSE:
                    sw.WriteLine("Okno aplikácie bolo uzatvorené " +
                       " (čas [" + DateTime.Now.Hour + ":" +
                       DateTime.Now.Minute + ":" +
                       DateTime.Now.Second + "], správa " +
                       "WM_CLOSE).");
                    sw.WriteLine("------- Koniec záznamu -------");
                    sw.Close();
                    break;
            }

            // Volanie metódy WndProc bázovej triedy.
            base.WndProc(ref m);
        }

        // Definícia metódy Zobraziť, ktorá zabezpečuje
        // zviditeľnenie okna aplikácie na obrazovke počítača.
        public void Zobraziť()
        {
            this.Show();
        }
    }
}
```

Inštancie deklarovanej zapečatenej triedy budú pôsobiť ako formuláre, ktoré dokážu monitorovať priebehy vybratých akcií svojich životných cyklov. Predmetom záznamu budú tieto akcie: vytvorenie formulára, minimalizovanie formulára, maximalizovanie formulára a uzatvorenie formulára. Keď nastane ktorákoľvek z vymenovaných akcií, jej časový výskyt sa uloží do externého textového súboru. Komunikácia medzi operačným systémom a formulárom je uskutočňovaná na báze mechanizmu správ. Ak dôjde k určitej udalosti, operačný systém podá formuláru informáciu o vzniku udalosti zaslaním správy. Túto správu následne filtruje procedúra okna formulára, ktorá rozhodne, aká bude reakcia na príjem príslušnej správy.

Diagram zapečatenej triedy je zobrazený na obr. 29.

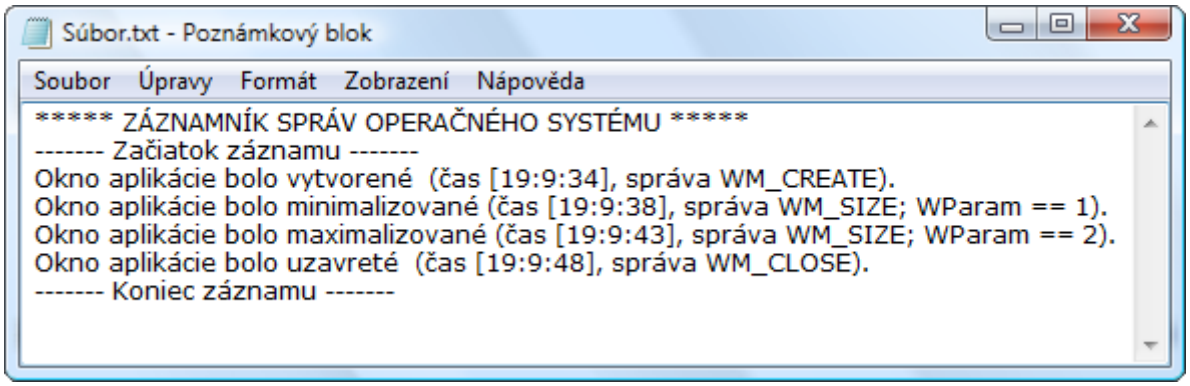


Obr. 29: Diagram zapečatenej triedy

Inštanciáciu zapečatenej triedy môžeme realizovať napríklad v tele spracovateľa udalosti **Click** tlačidla, ktoré sa nachádza na hlavnom aplikačnom formulári:

```
private void btnZobraziťFormulár_Click(object sender, EventArgs e)
{
    // Generovanie novej inštancie zapečatenej triedy (nového formulára).
    NovýFormulár nf = new NovýFormulár();
    nf.Zobraziť();
}
```

Obsah textového súboru s informáciami o priebehu životného cyklu formulára môžeme vidieť na obr. 30.



Obr. 30: Monitoring životného cyklu formulára ako inštancie zapečatenej triedy

2.7 Parciálne triedy

V programovacom jazyku C# 3.0 smieme deklaráciu triedy rozdeliť do viacerých fragmentov, pričom tieto fragmenty môžu byť uložené v rôznych zdrojových súboroch zostavenia aplikácie .NET. Ak je trieda deklarovaná takýmto spôsobom, nazývame ju parciálnou.

Pri preklade musia byť všetky zdrojové súbory s jednotlivými deklaračnými fragmentmi parciálnej triedy dostupné a viditeľné, pretože kompilátor z nich zostaví finálnu deklaráciu parciálnej triedy.

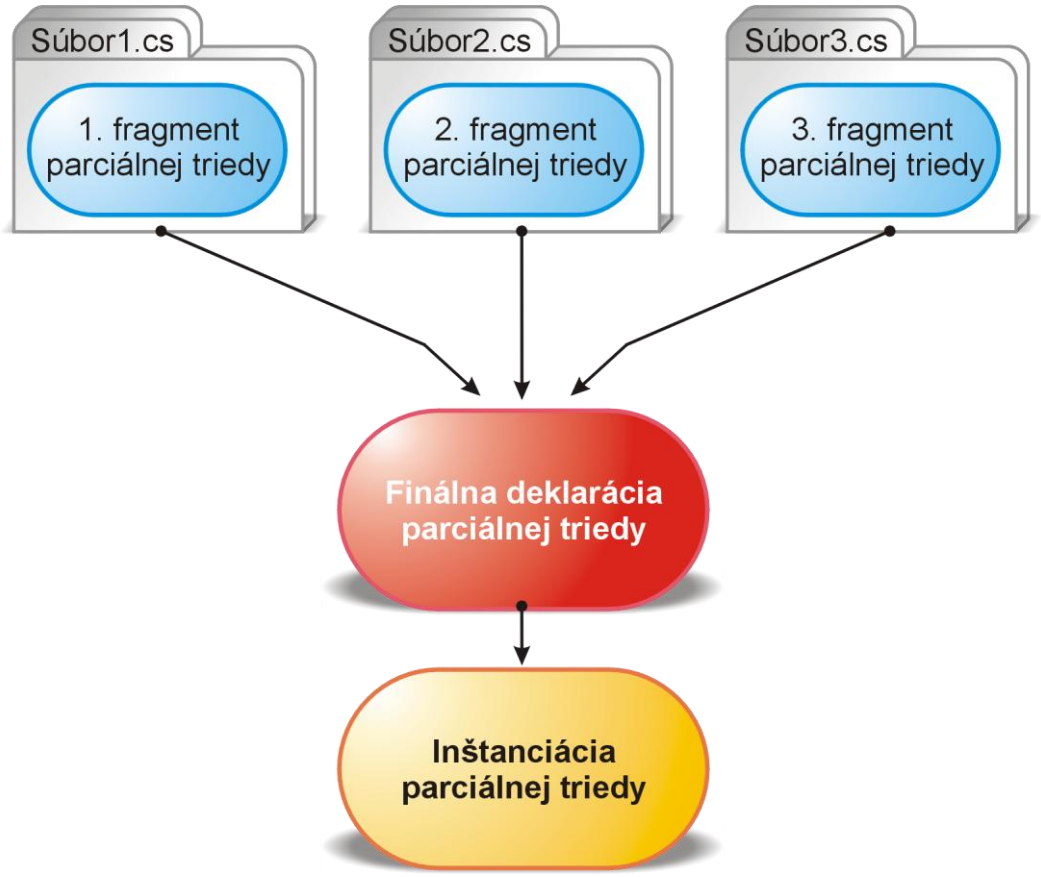
V deklaračných fragmentoch parciálnej triedy sa vyskytuje modifikátor **partial**:

```
[M] partial class X
{
    // 1. fragment parciálnej triedy.
}
[M] partial class X
{
    // 2. fragment parciálnej triedy.
}
```

kde:

- **[M]** je prístupový modifikátor parciálnej triedy.
- **X** je identifikátor parciálnej triedy.

Zostavenie finálnej deklarácie parciálnej triedy z deklaračných fragmentov je schematicky znázornené na obr. 31.



Obr. 31: Generovanie finálnej deklarácie parciálnej triedy a jej inštanciácia

Nasleduje praktická ukážka deklarácie parciálnej triedy **Doktorand**. Zdrojový kód parciálnej triedy je rozdelený na dve časti, z ktorých každá je uložená v samostatnom zdrojovom súbore.

Deklaračný fragment parciálnej triedy uložený v súbore Program.cs:

```
namespace ParciálneTriedy
{
    // 1. deklaračný fragment parciálnej triedy.
    internal partial class Doktorand
    {
        private string meno, priezvisko;
        private byte kredityZaVedeckúČinnosť;
        public Doktorand(string meno, string priezvisko,
            byte kredityZaVedu)
        {
            this.meno = meno;
```

```
            this.priezvisko = priezvisko;
            this.kredityZaVedeckúČinnosť = kredityZaVedu;
        }
    }
}
```

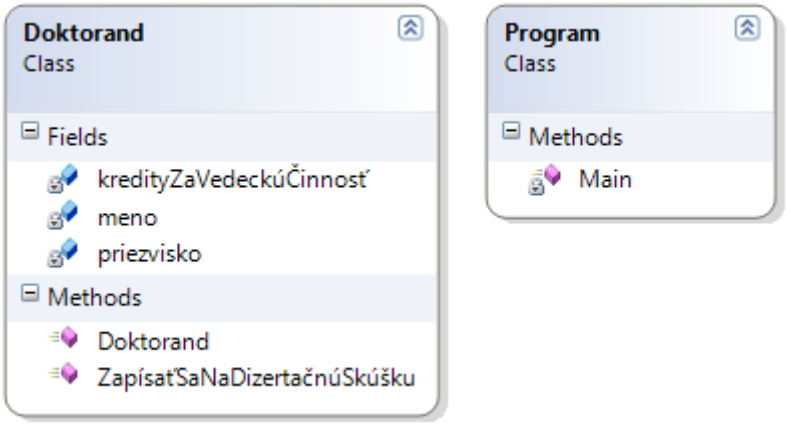
Deklaračný fragment parciálnej triedy uložený v súbore PT2.cs:

```
using System;
namespace ParciálneTriedy
{
    // 2. deklaračný fragment parciálnej triedy.
    internal partial class Doktorand
    {
        public void ZapísaťSaNaDizertačnúSkúšku()
        {
            if (this.kredityZaVedeckúČinnosť < 180)
                Console.WriteLine("Nemôžete sa zapísať na " +
                    "dizertačnú skúšku.");
            else
                Console.WriteLine("Boli ste zapísaný na " +
                    "dizertačnú skúšku.");
        }
    }
}
```

Inštanciácia parciálnej triedy prebieha podľa štandardných pravidiel v tele hlavnej metódy **Main**:

```
class Program
{
    static void Main(string[] args)
    {
        // Inštanciácia parciálnej triedy.
        Doktorand d1 = new Doktorand("Ján", "Hornák", 190);
        d1.ZapísaťSaNaDizertačnúSkúšku();
        Console.Read();
    }
}
```

Pri zobrazovaní grafických diagramov parciálnych tried pracuje Návrhár tried tak, že najskôr vyhotoví finálnu deklaráciu parciálnej triedy a až potom zobrazí jej vizuálny obraz.



Obr. 32: Vizualizácia parciálnej triedy vo vizuálnom Návrhárovi tried

2.8 Statické triedy

Počnúc verziou 2.0 programovacieho jazyka C# je možné v zdrojovom kóde vytvárať statické triedy. V deklaračnom príkaze statickej triedy sa nachádza modifikátor **static**:

```
internal static class X
{
    // Telo statickej triedy.
}
```

Statická trieda môže obsahovať iba definície statických členov, ako sú statické dátové položky, statický konštruktor, statické metódy či statické vlastnosti. Pre každú statickú triedu platia nasledujúce pravidlá:

1. Statická trieda nemôže byť inštanciovaná.
2. Statická trieda nemôže pôsobiť ako bázová trieda.
3. Statická trieda je implicitne zapečatená.
4. Statická trieda nesmie obsahovať definície inštančného konštruktora a finalizéra.

Pokračujeme uvedením zdrojového kódu statickej triedy, ktorá implementuje funkcionalitu pre vykresľovanie rastrových bitových máp[23]:

```
// Deklarácia statickej triedy.
static class Obrázok
{
    // Definícia súkromnej statickej dátovej položky.
    private static Image bitováMapa;
    // Definícia statického konštruktora.
    static Obrázok()
    {
```

[23] Predstavený zdrojový kód statickej triedy predpokladá, že je direktívou **using** zavedený menný priestor **System.Drawing**.

```
            bitováMapa = null;
        }
        // Definícia statickej metódy.
        public static void Vykresliť(string cesta, Graphics grafickýObjekt)
        {
            bitováMapa = Image.FromFile(cesta);
            grafickýObjekt.DrawImage(bitováMapa, new Point(10, 10));
            bitováMapa.Dispose();
        }
}
```

V tele statickej triedy nie sú umiestnené žiadne iné členy, len statické. Parametrická statická metóda prijíma absolútnu cestu k fyzickému súboru s rastrovými dátami, rovnako ako aj odkaz na inštanciu triedy **Graphics** (táto inštancia pôsobí ako primárny grafický objekt, prostredníctvom ktorého sú realizované grafické operácie súvisiace s plochou aplikačného formulára). Dáta rastrovej bitovej mapy načítame do inštancie triedy **Image** z menného priestoru **System.Drawing**. Všimnime si, že s načítaním dát nám pomáha parametrická metóda **FromFile** triedy **Image**, ktorá je sama statická. Metóda **FromFile** vytvorí inštanciu triedy **Image** a vráti odkaz, ktorý ukladáme do definovanej odkazovej premennej. Za vykreslenie bitovej mapy na plochu aplikačného formulára je zodpovedná inštančná metóda **DrawImage** primárneho grafického objektu. Metóde **DrawImage** musíme odovzdať dva artefakty:

1. Odkaz na inštanciu triedy **Image**, v ktorej je uchovaná bitová mapa.

2. Inštanciu hodnotovej štruktúry **Point**, ktorá determinuje súradnice bodu, na ktorom začne vykresľovanie bitovej mapy. Súradnice bodu, ktorý je reprezentovaný inštanciou štruktúry **Point**, budú totožné so súradnicami ľavého horného rohu obdĺžnikového regiónu, do ktorého bude bitová mapa vykreslená.

Ak všetky operácie prebehnú úspešne, na aplikačnom formulári sa objaví požadovaná bitová mapa. Podotknime, že statická metóda nezavádza štruktúrovanú správu chýb, a teda nepriamo očakáva, že jej budú dodané korektné argumenty.

Keďže po vykreslení bitovej mapy s ňou už neplánujeme ďalej manipulovať, môžeme systémové zdroje asociované s inštanciou triedy **Image** uvoľniť volaním metódy **Dispose**.

Praktické použitie statickej triedy je veľmi jednoduché, pretože nemusíme pracovať so žiadnymi objektmi. Jednoducho vyjadríme náš zámer spustiť statickú metódu:

```
private void btnVykresliťBitovúMapu_Click(object sender, EventArgs e)
{
    // Aktivácia statickej metódy statickej triedy.
    Obrázok.Vykresliť(@"c:\Lesné kvety.jpg",
        this.CreateGraphics());
}
```

Statickú metódu voláme v spojení so statickou triedou. Prvým odovzdávaným argumentom je doslovný textový reťazec (doslovnosť je indikovaná symbolom @ zapísaným pred reťazcom) charakterizujúci absolútnu cestu k súboru s bitovou mapou. Druhým argumentom je odkaz na primárny grafický objekt, ktorý je spojený s aktuálnym aplikačným formulárom (aktuálnou inštanciou triedy **Form**).

2.9 Anonymné triedy a inicializátory inštancií anonymných a pomenovaných tried

Pri objektovo orientovanom programovaní v jazyku C# 3.0 môžu vývojári pracovať s dvoma základnými typmi tried. Ide o pomenované a anonymné triedy. Pomenovaná trieda je každá trieda, ktorá je správne deklarovaná a disponuje svojím identifikátorom. Všetky triedy, ktoré deklarujú programátori, patria k pomenovaným triedam. Na druhú stranu, počínajúc verziou 3.0 jazyka C# 3.0 môžeme vytvárať aj anonymné triedy. Pre anonymné triedy platia nasledujúce pravidlá:

- Anonymnú triedu nedeklaruje programátor, ale kompilátor. Na základe syntaktického predpisu zostrojí kompilátor kompletnú deklaráciu anonymnej triedy.

- Anonymná trieda nemá identifikátor, ktorý by bol priamo prístupný pre programátora. Keďže vývojár nepozná identifikátor implicitne vygenerovanej triedy, táto trieda sa mu javí ako anonymná. Hoci z pohľadu programátora je kompilátorom zostavená trieda naozaj anonymná, považujeme za dôležité podotknúť, že aj anonymná trieda má svoj identifikátor. Ten je však vytváraný automaticky kompilátorom, takže prekladač je ten, ktorý samočinne prisudzuje názov anonymnej triede. Spravidla neexistuje žiaden zmysluplný dôvod, aby sa vývojár pokúšal získať identifikátor anonymnej triedy[24]. Navyše, identifikátor anonymnej triedy sa môže líšiť v rôznych prekladových reláciách.

- Inštanciácia anonymnej triedy sa realizuje operátorom **new**. Po úspešnej inštanciácii vracia operátor **new** odkaz na alokovanú a inicializovanú inštanciu anonymnej triedy. Vrátený odkaz bude uložený do odkazovej premennej, ktorej dátový typ bude musieť byť kompilátorom implicitne inferovaný. Nevyhnutnosť aktivácie mechanizmu typovej inferencie je podmienená neznalosťou identifikátora anonymnej triedy.

[24] Napriek tomu, že nejde o štandardnú programovú operáciu, je možné identifikátor anonymnej triedy zistiť pomocou mechanizmu reflexie. Reflexia je proces dynamickej analýzy dátových typov uložených v zostavení aplikácie .NET.

- Inštancia anonymnej triedy je inicializovaná pomocou množiny tzv. inicializátorov inštancií anonymných tried. Inicializátory sú hodnoty, ktorými budú inicializované automaticky generované súkromné dátové položky inštancie anonymnej triedy. Okrem prípravy súpravy dátových položiek zostrojuje kompilátor aj kolekciu verejne prístupných skalárnych inštančných vlastností, ktoré slúžia na prístup k súkromným dátovým položkám. Medzi počtom automaticky vygenerovaných súkromných dátových položiek a počtom verejne prístupných skalárnych inštančných vlastností existuje relácia typu 1:1. Je však nutné si uvedomiť, že verejne prístupné skalárne inštančné vlastnosti, ktoré sú do tela anonymnej triedy umiestnené, slúžia len na čítanie hodnôt súkromných dátových položiek. V telách týchto vlastností absentuje vetva **set**, a preto nemôžu byť použité na modifikáciu hodnôt dátových položiek.

Všeobecný vzor, opisujúci inštanciáciu anonymnej triedy, je takýto:

```
var obj = new {SIV1 = IN1, SIV2 = IN2, ..., SIVN = INN};
```

kde:

- **$SIV_{1...N}$** sú identifikátory verejne prístupných skalárnych inštančných vlastností anonymnej triedy.
- **$IN_{1...N}$** sú inicializátory.

Zoznámme sa s fragmentom zdrojového kódu jazyka C# 3.0, v ktorom dochádza k inštanciácii anonymnej triedy:

```
// Inštanciácia anonymnej triedy.
var obj = new
{
    Kniha = "C# - praktické příklady",
    Autor = "Ján Hanák",
    Vydavateľ = "Grada Publishing",
    RokVydania = 2006,
    PočetStrán = 288,
    ISBN = "80-247-0988-0"
};
```

Vzhľadom na to, že operátor **new** nie je nasledovaný žiadnym identifikátorom triedy, kompilátor dokáže usúdiť, že bude musieť zostaviť deklaráciu anonymnej triedy. Postupuje teda tak, že vytvorí hlavičku a telo triedy. Triedu samozrejme pomenuje, aby sa mohol na ňu priamo odkazovať. Kompilátor ďalej pokračuje definíciou šestice súkromných dátových položiek, ktoré budú uchovávať dáta inštancie anonymnej triedy. Po zhotovení dátových položiek vytvorí kompilátor rovnaký počet verejne prístupných skalárnych inštančných vlastností určených len na čítanie. Okrem už spomenutých členov vloží kompilátor do tela anonymnej triedy aj parametrický inštančný konštruktor. Je to práve tento konštruktor, ktorý je zodpovedný za korektnú inicializáciu založenej inštancie anonymnej triedy podľa pokynov programátora. Po úspešnej

inštanciácii bude inštancia anonymnej triedy alokovaná na riadenej halde a náležite inicializovaná. Operátor **new** nám poskytne odkaz na vytvorenú inštanciu, ktorý ukladáme do odkazovej premennej s identifikátorom **obj**. Dodajme, že dátový typ premennej **obj** je za prispenia mechanizmu typovej inferencie automaticky určený.

Ak budeme chcieť vypísať na obrazovku počítača informácie o knihe, použijeme automaticky vygenerované verejne prístupné skalárne inštančné vlastnosti:

```
Console.WriteLine("Informácie o knihe:\n" +
    "Názov: " + obj.Kniha +
    "\nAutor: " + obj.Autor +
    "\nVydavateľ: " + obj.Vydavateľ +
    "\nRok vydania: " + obj.RokVydania +
    "\nPočet strán: " + obj.PočetStrán +
    "\nISBN: " + obj.ISBN);
```

Inicializátory inštancií tried smieme využiť aj pri zakladaní inštancií pomenovaných tried. Najskôr uvádzame deklaráciu pomenovanej triedy, ktorú budeme vzápätí potrebovať pri praktickej ukážke inicializátorov:

```
class Kniha
{
    private string názov, autor, vydavateľ, isbn;
    private ushort rokVydania, početStrán;

    public string Názov
    {
        get { return názov; }
        set { názov = value; }
    }
    public string Autor
    {
        get { return autor; }
        set { autor = value; }
    }
    public string Vydavateľ
    {
        get { return vydavateľ; }
        set { vydavateľ = value; }
    }
    public ushort RokVydania
    {
        get { return rokVydania; }
        set { rokVydania = value; }
    }
    public ushort PočetStrán
    {
        get { return početStrán; }
        set { početStrán = value; }
    }
    public string ISBN
    {
        get { return isbn; }
        set { isbn = value; }
    }
}
```

Syntaktická skladba triedy je veľmi jednoduchá. Inicializátory nám umožnia efektívne skonštruovať použiteľnú inštanciu triedy:

```
// Inicializácia inštancie triedy pomocou inicializátorov.
Kniha kniha = new Kniha()
{
    Názov = "C# - praktické příklady",
    Autor = "Ján Hanák",
    Vydavateľ = "Grada Publishing",
    RokVydania = 2006,
    PočetStrán = 288,
    ISBN = "80-247-0988-0"
};
```

Inštanciácia triedy sa riadi týmto predpisom:

1. Najskôr je na riadenej halde alokovaný pamäťový priestor pre inštanciu triedy.

2. Dochádza k spusteniu konštruktora. O tom, aký konštruktor to bude, rozhoduje syntaktické znenie inštanciačného príkazu. Ak sa pozrieme na použitie operátora **new**, tak vidíme, že pri inštanciácii sme použili výraz **new Kniha()**. Alokovaná inštancia triedy bude preto inicializovaná bezparametrickým implicitným inštančným konštruktorom. Konštruktor je implicitný preto, že ho vytvára automaticky kompilátor. (Pripomeňme, že túto akciu kompilátor uskutočňuje vždy, keď nie je v tele triedy explicitne definovaný žiaden inštančný konštruktor.)

3. Po spracovaní konštruktora je zahájená inicializácia predpísaná inicializátormi. Pomocou verejne prístupných skalárnych inštančných vlastností získame prístup k súkromným dátovým položkám inštancie triedy a uložíme do nich hodnoty inicializátorov.

4. Odkaz na alokovanú a inicializovanú inštanciu triedy je uložený do odkazovej premennej.

Z preskúmaného pracovného modelu inicializátorov inštancií tried jasne vidíme, že tieto programové entity nepreberajú funkciu konštruktora, len ju dopĺňajú. Ak by pomenovaná trieda, ktorej inštanciu chceme získať, definovala adekvátny parametrický konštruktor, tak by sme mohli primárne využiť ten a nemuseli by sme sa spoliehať na inicializátory. Naopak, s výhodou môžeme inicializátory upotrebiť pri tých triedach, ktoré žiaden konštruktor nedefinujú (čo je prípad aj našej vyššie uvedenej triedy). Aj keď kompilátor automaticky zostaví bezparametrický inštančný konštruktor, ten ponúka len štandardnú[25], v žiadnom prípade nie používateľsky konfigurovateľnú, inicializáciu dátovej sekcie objektu.

[25] Štandardnou inicializáciou máme na mysli uvedenie dátových položiek inštancie triedy do východiskového stavu. Dátové položky primitívnych integrálnych hodnotových dátových typov budú inicializované nulami. Dátové položky primitívnych reálnych hodnotových dátových typov budú inicializované rovnako nulami, len

2.10 Polymorfizmus implementovaný pomocou verejnej jednoduchej dedičnosti

Polymorfizmus determinuje schopnosť objektov reagovať inak na príjem identickej správy. V programovacom jazyku C# 3.0 môže byť polymorfizmus implementovaný dvomi spôsobmi:

1. Polymorfizmus implementovaný pomocou verejnej jednoduchej dedičnosti.
2. Polymorfizmus implementovaný pomocou rozhrania.

V tejto kapitole sa budeme venovať výkladu prvého variantu, a teda polymorfizmu implementovaného pomocou verejnej jednoduchej dedičnosti. Podstata polymorfizmu v tomto vyjadrení spočíva v prekrytí zdedeného člena bázovej triedy v triede odvodenej. Zdedený člen bázovej triedy môžeme prekryť rovnakým druhom člena, ktorý bude disponovať odlišnou funkcionalitou.

Aby sme mohli v odvodenej triede prekryť zdedenú implementáciu člena bázovej triedy, musí byť takýto člen v bázovej triede definovaný ako virtuálny. Pre naše ďalšie potreby budeme uvažovať o metóde ako o člene triedy, s ktorým budeme pracovať. Keď v tele bázovej triedy definujeme virtuálnu metódu, vysielame kompilátoru signál, že táto metóda bude môcť byť v odvodenej triede prekrytá. Ak má byť istá metóda bázovej triedy virtuálna, musí sa v jej hlavičke nachádzať modifikátor **virtual**. Generická syntaktická ukážka deklarácie bázovej triedy s jednou virtuálnou metódou:

```
class A
{
    public virtual void m()
    {
        // Implementácia virtuálnej metódy bázovej triedy.
    }
}
```

Metóda **m** bázovej triedy **A** je explicitne virtuálna, čo znamená, že odvodená trieda bude môcť v prípade záujmu prekryť zdedenú implementáciu tejto virtuálnej metódy. V programovacom jazyku C# 3.0 nie sú inštančné metódy tried implicitne virtuálne. Implicitná virtualita inštančných metód je prirodzená napríklad pre jazyk Java, v ktorom keď zavedieme do tela triedy definíciu metódy, je automaticky (implicitne) považovaná za virtuálnu. V jazyku C# 3.0 však nie je implicitnou technikou prekrývanie, ale ukrývanie zdedených členov bázovej triedy. Preto ak chceme uplatniť prekrytie zdedenej inštančnej metódy bázovej triedy, musí byť takáto metóda virtuálna.

v reálnej reprezentácii s jednoduchou alebo dvojnásobnou presnosťou. Dátové položky logického hodnotového dátového typu **bool** budú inicializované logickou nepravdou (**false**). Dátové položky znakového hodnotového dátového typu **char** budú inicializované nulovými znakmi. Dátové položky primitívnych odkazových dátových typov (**string** a **object**) budú inicializované nulovými odkazmi (hodnotami **null**).

V tele odvodenej triedy definujeme rovnomennú metódu s modifikátorom **override**:

```
class B : A
{
    public override void m()
    {
        // Implementácia inštančnej metódy odvodenej triedy,
        // ktorá prekrýva rovnomennú zdedenú metódu bázovej triedy.
    }
}
```

Prekrývajúca metóda odvodenej triedy s modifikátorom **override** musí byť s prekrývanou (virtuálnou) metódou bázovej triedy zhodná v týchto aspektoch:

- signatúra (identifikátor a zoznam formálnych parametrov),
- dátový typ návratovej hodnoty,
- prístupový modifikátor.

Implementácia prekrývajúcej metódy bude samozrejme odlišná.

O tom, ktorá metóda bude v súvislosti so založenou inštanciou aktivovaná, rozhoduje pri polymorfizme implementovanom pomocou verejnej jednoduchej dedičnosti typ zrodenej inštancie. Preskúmajme nasledujúci fragment zdrojového kódu jazyka C# 3.0:

```
A obj = new B();
obj.m();
```

Rozoberme prvý inštanciačný príkaz. V tomto príkaze definujeme odkazovú premennú s identifikátorom **obj**. Do tejto odkazovej premennej je možné uložiť objektovú referenciu na inštanciu bázovej triedy **A**. My však aplikujeme operátor **new** na odvodenú triedu, čo znamená, že vytvárame jej inštanciu. Len čo je dokončený proces inštanciácie odvodenej triedy, operátor **new** vracia v podobe svojej návratovej hodnoty objektovú referenciu na vytvorenú inštanciu odvodenej triedy. Túto objektovú referenciu je však možné uložiť aj do odkazovej premennej, ktorej typom je bázová trieda. Je to preto, že medzi triedami **A** a **B** existuje puto vybudované na základe verejnej jednoduchej dedičnosti, pričom **B** je odvodenou triedou a **A** je bázovou triedou. Inštanciačný príkaz verifikuje korektnosť tvrdenia, podľa ktorého sa potomok môže vyskytnúť všade tam, kde je očakávaný jeho rodič. Z tohto titulu je možné do odkazovej premennej s typom bázovej triedy uložiť objektovú referenciu na inštanciu odvodenej triedy.

Druhý príkaz predstavuje volanie verejne prístupnej inštančnej metódy **m**. Keďže sme povedali, že o tom, ktorá metóda bude aktivovaná, rozhoduje typ zrodenej inštancie, tak musí byť zrejmé, že tento príkaz volá prekrývajúcu metódu inštancie odvodenej triedy. A to aj napriek tomu, že na vyvolanie metódy sme použili odkazovú premennú, ktorej typom je bázová trieda.

2.11 Polymorfizmus implementovaný pomocou rozhrania

Druhý spôsob, podľa ktorého je možné navrhnúť polymorfné správanie inštancií tried v jazyku C# 3.0, reprezentuje odlišná implementácia identického rozhrania viacerými triedami. Aby sme však mohli kvalifikovane hovoriť o implementácii polymorfizmu prostredníctvom rozhrania, musíme sa najskôr zoznámiť s pojmom rozhranie a jeho chápaním v jazyku C# 3.0.

Rozhranie je predpis vymedzujúci štýl správania sa entít, ktoré sa rozhodnú príslušné rozhranie implementovať. V jazyku C# 3.0 je rozhranie deskriptorom, ktorý združuje množinu deklaračných príkazov metód, vlastností, udalostí a delegátov. Rozhranie síce charakterizuje členy prostredníctvom deklaračných príkazov, no už nedefinuje ich implementáciu[26]. Všetky členy deklarované v rozhraní budú definované v triede, ktorá sa rozhodne dané rozhranie implementovať. Povedané inak, rozhranie vraví „čo sa má urobiť“, avšak už nezmieňuje konkrétny spôsob „ako sa to má urobiť“.

Rozhranie deklarujeme podľa nasledujúceho generického modelu:

```
[M] interface IRozhranie
{
    // Deklarácie členov rozhrania.
}
```

kde:

- **[M]** je prístupový modifikátor, ktorý určuje prístupové práva jednak samotného rozhrania, a taktiež aj členov, ktoré rozhranie deklaruje[27].
- **IRozhranie** je identifikátor rozhrania[28].

Uvažujme deklaráciu rozhrania **IPočítačováGrafika**:

```
public interface IPočítačováGrafika
{
    void VykonaťGrafickúTransformáciu(string zdrojováBitmapa,
        string cieľováBitmapa);
}
```

[26] Z pohľadu jazyka C++ by sme mohli povedať, že rozhranie je súborom prototypov členov bez vopred naprogramovanej funkcionality. Deklaračné príkazy, resp. prototypy, charakterizujú signatúry členov rozhrania, no abstrahujú od konkrétnej funkcionality týchto členov.

[27] Pokiaľ nie je explicitne určený žiaden prístupový modifikátor rozhrania, tak je rozhranie deklarované s implicitne internými prístupovými právami (ako keby bol aplikovaný modifikátor **internal**).

[28] Podľa odporúčania vhodných nomenklatúrnych smerníc sa rozhrania v jazyku C# 3.0 (a podobne aj vo všetkých ostatných .NET-kompatibilných programovacích jazykoch) pomenúvajú so začiatočným písmenom „I“ (z angl. Interface). Hoci nie je obligatórna, budeme túto pomenovaciu zásadu uplatňovať v celej publikácii. Je to koniec koncov programátorsky prívetivé, pretože už na základe identifikátora entity dokážeme zistiť, či ide o triedu, alebo rozhranie.

Toto rozhranie obsahuje deklaráciu jednej verejne prístupnej parametrickej metódy. Ako môžeme dedukovať z názvu deklarovanej metódy, jej úlohou bude spracovanie grafickej transformácie. Predmetom transformácie sa stane bitová mapa, ktorej rastrové dáta budú uložené vo vstupnom grafickom súbore. Absolútna cesta k vstupnému grafickému súboru bude vo forme reťazcovej konštanty uložená do prvého formálneho parametra transformačnej metódy. Po spracovaní grafickej transformácie bude modifikovaná bitová mapa uložená do nového (výstupného) fyzického grafického súboru, ktorý špecifikuje druhý formálny parameter transformačnej metódy.

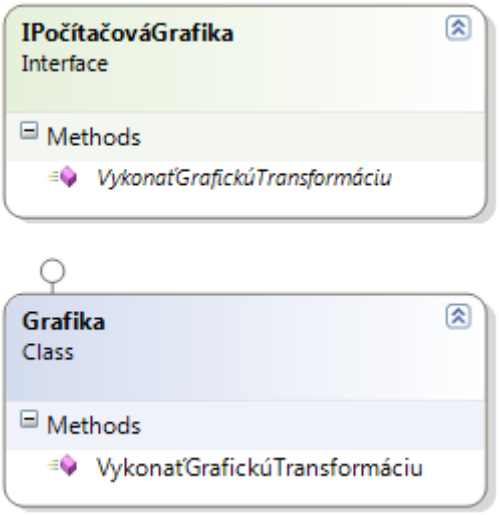
Rozhranie samo osebe predstavuje len predpis, podľa ktorého sa budú musieť správať triedy, ktoré sa rozhodnú dané rozhranie implementovať. Implementácia rozhrania **IPočítačováGrafika** triedou **Grafika** vyzerá v jazyku C# 3.0 takto:

```
// Deklarácia triedy, ktorá implementuje rozhranie.
class Grafika : IPočítačováGrafika
{
    // Definícia metódy, ktorá je deklarovaná v rozhraní.
    public void VykonaťGrafickúTransformáciu(string zdrojováBitmapa,
    string cieľováBitmapa)
    {
        int i, j;

        // Konštrukcia objektu triedy Bitmap, ktorý bude zapuzdrovať
        // rastrové dáta načítané zo vstupného grafického súboru.
        Bitmap bitováMapa = new Bitmap(zdrojováBitmapa);
        Color farbaObrazovéhoBodu, inverznáFarba;

        // Dvojica cyklov uskutočňuje inverziu bitovej mapy.
        for (i = 0; i < bitováMapa.Width; i++)
        {
            for (j = 0; j < bitováMapa.Height; j++)
            {
                farbaObrazovéhoBodu = bitováMapa.GetPixel(i, j);
                inverznáFarba = Color.FromArgb(
                    255 - farbaObrazovéhoBodu.R,
                    255 - farbaObrazovéhoBodu.G,
                    255 - farbaObrazovéhoBodu.B);
                bitováMapa.SetPixel(i, j, inverznáFarba);
            }
        }
        // Po vykonaní grafickej transformácie je upravená bitová mapa
        // uložená do nového grafického súboru.
        bitováMapa.Save(cieľováBitmapa);
        // Keďže s objektom triedy Bitmap už nebudeme pracovať,
        // môžeme s ním asociované zdroje explicitne uvoľniť.
        bitováMapa.Dispose();
    }
}
```

Zo syntaktického pohľadu sa technika implementácie rozhrania triedou ponáša na aplikáciu verejnej jednoduchej dedičnosti. Hoci je stvárnenie zdrojového kódu identické s dedičnosťou, v tomto prípade ide o implementáciu rozhrania. Ako sme už spomenuli, ak sa trieda rozhodne isté rozhranie implementovať, musí prijať záväzok definovať

všetky členy, ktoré príslušné rozhranie deklaruje. V našej praktickej ukážke obsahuje rozhranie len jednu metódu, ktorej definíciu deklarovaná trieda zavádza. Zdrojový kód umiestnený v tele metódy **VykonaťGrafickúTransformáciu** realizuje inverziu vstupnej bitovej mapy. Po vykonaní inverzie je transformovaná bitová mapa uložená do výstupného grafického súboru.

Grafické modely rozhrania a implementujúcej triedy zobrazené v Návrhárovi tried sú znázornené na obr. 33.



Obr. 33: Vizuálny model rozhrania a implementujúcej triedy

Algoritmus vykonávajúci inverziu bitovej mapy pracuje nasledujúcim spôsobom:

1. Pre každý obrazový bod, ktorý je súčasťou bitovej mapy, je nájdený jeho farebný vektor. Farebný vektor je vyjadrený v priestore RGB, pričom je tvorený usporiadanou trojicou hodnôt (zložiek). Zložky farebného vektora podávajú informácie o intenzite zastúpenia troch farebných zložiek: R – červená, G – zelená a B – modrá. Intenzita zastúpenia červenej, zelenej a modrej zložky v konečnom dôsledku určuje intenzitu výslednej farby, ktorou bude obrazový bod vykreslený na obrazovke počítača. Intenzita zastúpenia zložiek farebných vektorov je bežne kvantifikovaná pomocou celočíselnej alebo reálnej stupnice hodnôt. Pri celočíselnej stupnici môže intenzita zložky farebného vektora nadobúdať hodnoty z intervalu <0, 255>. Pri reálnej stupnici sa zasa používa interval <0, 1>. Napríklad, obrazový bod s maximálnou intenzitou svojich zložiek disponuje farebným vektorom v tvare [255, 255, 255] (v celočíselnom vyjadrení), resp. [1, 1, 1] (v reálnom vyjadrení). Zo všetkých obrazových bodov bitovej mapy vyberieme jeden, s ktorým budeme ďalej pracovať. Nech sa vybratý obrazový bod volá P_1 a nech jeho farebný vektor má nasledujúce zloženie: $P_1[R_1, G_1, B_1]$. Prijmime dohovor, že zložky farebného vektora analyzovaného obrazového bodu budeme kvantifikovať pomocou celočíselnej stupnice hodnôt.

2. Pri inverzii sú zložky farebného vektora obrazového bodu P_1 transformované tak, že vznikne nový obrazový bod s transformovaným (invertovaným) farebným vektorom. Ak invertovanému obrazovému bodu prisúdime identifikátor P_2, tak potom výpočet jeho farebného vektora stanovuje nasledujúci matematický vzťah:

$$P_2\,[R_2, G_2, B_2] \;=\; [255 - R_1, 255 - G_1, 255 - B_1]$$

   Ukážme si príklad: Nech je vstupný obrazový bod P_1 vykreslený odtieňom červenej farby s farebným vektorom [237, 28, 36]. Po inverzii získame nový farebný vektor [18, 227, 219], ktorý predstavuje tyrkysovú farbu.

3. Algoritmus teda pracuje tak, že najskôr určí farebné vektory obrazových bodov a potom ich prevedie na ich inverzné ekvivalenty. Grafická transformácia sa uskutočňuje na báze obrazového bodu, čo znamená, že finálny invertovaný obraz nezískame skôr, než budú grafickej transformácii podrobené všetky obrazové body, z ktorých je editovaná bitová mapa zložená. Ak počet obrazových bodov bitovej mapy (ako objem vstupných dát) označíme premennou n, tak operácie „zistenie aktuálneho farebného vektora, výpočet inverzného farebného vektora a aplikácia transformovaného farebného vektora“ budú musieť byť spracované n-krát. Pre bežnú bitovú mapu s rozmermi 1024x768 obrazových bodov to znamená realizáciu viac ako 786-tisíc grafických transformácií.

Vzhľadom na to, že načítaná bitová mapa je v objekte triedy **Bitmap** uložená vo forme dvojrozmernej dátovej mriežky, ako algoritmicky najjednoduchšie riešenie sa javí použitie dvojice cyklov, pomocou ktorých manipulujeme s farebnými vektormi jednotlivých obrazových bodov. Na zistenie pôvodných farebných vektorov používame inštančnú metódu **GetPixel** objektu triedy **Bitmap**, kým s aplikáciou transformovaných farebných vektorov nám pomôže metóda **SetPixel** rovnakého objektu. Dodajme, že metódy **GetPixel** a **SetPixel** patria k tzv. párovým metódam, ktoré sa vždy používajú spoločne.

Vo chvíli, keď je inverzia farebných vektorov obrazových bodov bitovej mapy hotová, voláme inštančnú metódu **Save** objektu triedy **Bitmap** a modifikovanú bitovú mapu ukladáme do nového fyzického grafického súboru. Keďže objekt triedy **Bitmap** už nemienime naďalej používať, môžeme požiadať o explicitnú dealokáciu systémových prostriedkov, ktoré sú s týmto objektom asociované. Na to nám slúži inštančná metóda **Dispose**[29].

[29] Volanie metódy **Dispose** nad objektom triedy **Bitmap** však neznamená, že dochádza k likvidácii tohto objektu. Metóda **Dispose** zabezpečuje uvoľnenie alokovaných prostriedkov, ktoré slúžili najmä na uskladnenie rastrových dát načítanej bitovej mapy. Objekt triedy **Bitmap** je preto aj po aktivácii metódy **Dispose** stále dosiahnuteľný z programového kódu, a teda „živý“. Programátori by však takýto objekt už nemali naďalej používať, pretože nie je zaručený jeho dátovo konzistentný stav. Objekt triedy **Bitmap** (a objekt akejkoľvek triedy, v súvislosti s ktorým smie byť zavolaná metóda **Dispose**) bude zlikvidovaný vo chvíli, keď dôjde

Triedu **Grafika** s implementovaným rozhraním **IPočítačováGrafika** by sme mohli prakticky použiť napríklad takto:

```
using System.Diagnostics;
class Program
{
    static void Main(string[] args)
    {
        Grafika g = new Grafika();
        Stopwatch sw = new Stopwatch();
        sw.Start();
        Console.WriteLine("Vykonávam grafickú transformáciu...");
        g.VykonaťGrafickúTransformáciu(@"c:\obr_01.jpg", @"c:\obr_02.png");
        sw.Stop();
        Console.WriteLine("Grafická transformácia je hotová.");
        Console.WriteLine("Operácia trvala {0} s.",
            sw.Elapsed.TotalSeconds);
        Console.Read();
    }
}
```

Spracovanie grafickej transformácie je za prispenia našej triedy veľmi jednoduché. Vo chvíli, keď máme alokovanú a inicializovanú inštanciu triedy **Grafika**, zavoláme jej verejne prístupnú inštančnú metódu **VykonaťGrafickúTransformáciu**, ktorej odovzdáme zmysluplnú kolekciu vstupných argumentov. Program informuje používateľa nielen o uskutočnení grafickej transformácie, ale rovnako aj o nameranom exekučnom čase, ktorý bol na jej spracovanie potrebný. Exekučný čas meriame pomocou objektu triedy **Stopwatch** z menného priestoru **System.Diagnostics**.

O polymorfizme implementovanom rozhraním vravíme vtedy, ak dve triedy zavedú odlišnú implementáciu identického rozhrania. To sa udeje vtedy, ak navrhneme triedu **Grafika2**, do tela ktorej zavedieme definíciu metódy **VykonaťGrafickúTransformáciu** s odlišnou funkcionalitou. Definovaná metóda triedy **Grafika2** bude uskutočňovať iný typ grafickej transformácie – my sme zvolili prevod bitovej mapy do sivotónu. Zdrojový kód jazyka C# 3.0, ktorý zobrazuje deklaráciu triedy, uvádzame ďalej:

```
// Deklarovaná trieda zavádza odlišnú implementáciu rozhrania.
class Grafika2 : IPočítačováGrafika
{
    public void VykonaťGrafickúTransformáciu(string zdrojováBitmapa,
    string cieľováBitmapa)
    {
        int i, j, I;
        Bitmap bitováMapa = new Bitmap(zdrojováBitmapa);
        Color farbaObrazovéhoBodu;
        for (i = 0; i < bitováMapa.Width; i++)
        {
            for (j = 0; j < bitováMapa.Height; j++)
            {
                farbaObrazovéhoBodu = bitováMapa.GetPixel(i, j);
```

k zrušeniu aj posledného odkazu, ktorý bol na tento objekt nasmerovaný. Potom automatický správca pamäte finalizuje objekt a odstráni ho z riadenej haldy.

```
                I = (int)(0.299 * farbaObrazovéhoBodu.R +
                    0.587 * farbaObrazovéhoBodu.G +
                    0.114 * farbaObrazovéhoBodu.B);
                bitováMapa.SetPixel(i, j, Color.FromArgb(I, I, I));
            }
        }
        bitováMapa.Save(cieľováBitmapa);
        bitováMapa.Dispose();
    }
}
```

Algoritmus grafickej transformácie je pri prevode obrazu do sivotónu založený na výpočte intenzity jasu pre farby vykreslené v odtieňoch sivej. Vychádzame pritom z matematického vzorca, ktorý reflektuje mieru citlivosti ľudského oka na jednotlivé zložky farebných vektorov:

$$I = 0{,}299 \times R + 0{,}587 \times G + 0{,}114 \times B$$

kde:

- I je výsledná intenzita jasu zodpovedajúca sivotónovému obrazu.
- R, G a B sú zložky farebného vektora spracúvaného obrazového bodu.

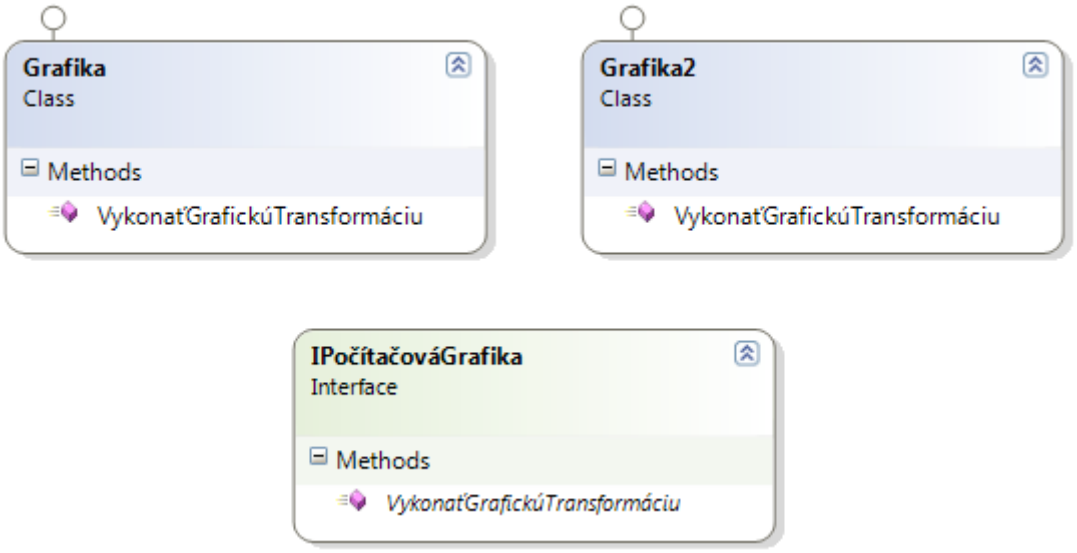
Predostretá matematická rovnica vyjadruje, s akou intenzitou vníma ľudské oko jednotlivé farebné zložky modelu RGB. Z použitých multiplikatívnych koeficientov je zrejmé, že najcitlivejšie je vnímaná zelená farba (koeficient 0,587), potom červená farba (koeficient 0,299) a nakoniec modrá farba (koeficient 0,114). Povedané priamočiarejšie, vo farebnom obraze si všímame najviac intenzitu odtieňov zelenej farby, o niečo menej sú pre nás signifikantné odtiene červenej farby a vôbec najnižšiu citlivosť preukazujeme voči odtieňom modrej farby. Číselné hodnoty modelu RGB, o ktorých sme sa zmienili, nám podávajú informácie o intenzite jasu určitej farby, resp. farebného odtieňa. Napríklad, najjasnejšia je bezpochyby biela farba, ktorej farebný vektor má hodnotu [255, 255, 255]. Vyššie uvedený vzťah dokáže podľa vstupných informácií o zložkách farebného vektora obrazového bodu (alebo presnejšie o intenzite zložiek farebného vektora) vypočítať celkový jas, ktorý je typický pre obrazový bod vykreslený v sivotóne.

Celý matematický algoritmus si najlepšie objasníme na praktickom príklade. Nadviažeme pritom na predchádzajúcu rozpravu o bitovej mape, z ktorej sme si vybrali jeden obrazový bod s názvom P_1. Intenzita jasu tohto obrazového bodu je determinovaná trojicou číselných hodnôt zodpovedajúcich zložkám jeho farebného vektora vyjadrených v modeli RGB. Ak sú $[R_1, G_1, B_1]$ zložky farebného vektora obrazového bodu P_1, tak obrazový bod P_2 prevedený do odtieňov sivej bude mať takúto číselnú interpretáciu:

$$P_2 = \left[0{,}299 \times R_1 + 0{,}587 \times G_1 + 0{,}114 \times B_1, \right.$$

$$0{,}299 \times R_1 + 0{,}587 \times G_1 + 0{,}114 \times B_1,$$
$$0{,}299 \times R_1 + 0{,}587 \times G_1 + 0{,}114 \times B_1,$$
]

Po vykonaní grafickej transformácie už obrazový bod P_2 nebude „farebný“, pretože jeho intenzita jasu bude prislúchať niektorému z odtieňov sivej farby.



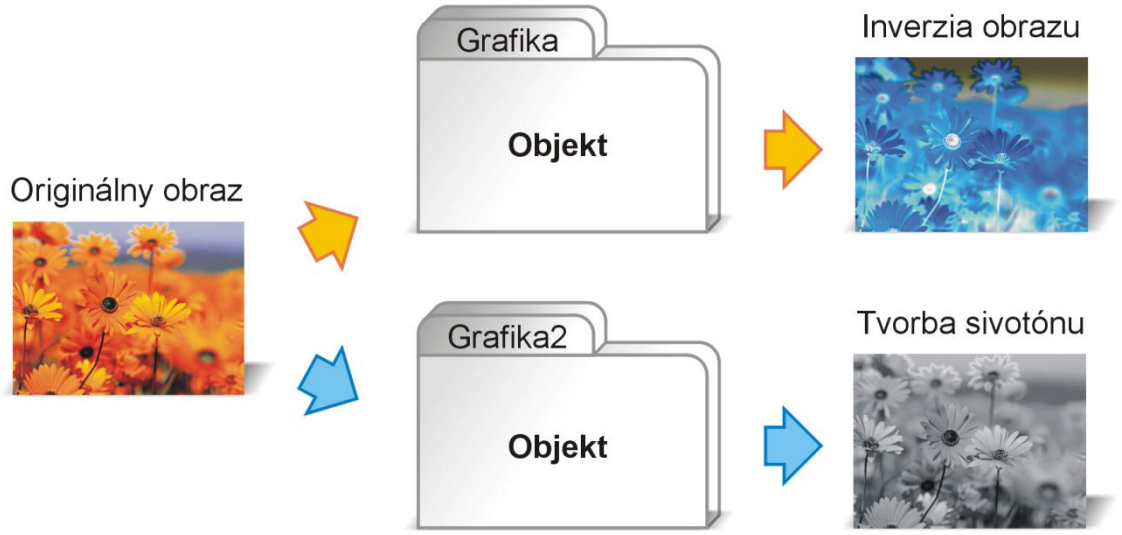
Obr. 34: Polymorfizmus implementovaný pomocou rozhrania

Inštancie tried **Grafika** a **Grafika2** sa správajú polymorfne, pretože na rovnaký podnet (žiadosť o vykonanie grafickej transformácie) reagujú odlišne (inštancia triedy **Grafika** vyhotoví inverzný obraz, zatiaľ čo inštancia triedy **Grafika2** vytvorí sivotónový obraz).

Zdrojový kód jazyka C# 3.0, ktorý inštanciuje triedu **Grafika2** a vykonáva tvorbu sivotónového obrazu, vyzerá takto:

```
class Program
{
    static void Main(string[] args)
    {
        Grafika2 g = new Grafika2();
        Stopwatch sw = new Stopwatch();
        sw.Start();
        Console.WriteLine("Vykonávam grafickú transformáciu...");
        g.VykonaťGrafickúTransformáciu(@"c:\obr_01.jpg", @"c:\obr_03.png");
        sw.Stop();
        Console.WriteLine("Grafická transformácia je hotová.");
        Console.WriteLine("Operácia trvala {0} s.",
            sw.Elapsed.TotalSeconds);
        Console.Read();
    }
}
```

Vizualizáciu polymorfného správania inštancií tried, ktoré zavádzajú odlišnú implementáciu totožného rozhrania uvádzame na obr. 35.



Obr. 35: Schematická demonštrácia polymorfného správania inštancií tried, ktoré zavádzajú odlišnú implementáciu rovnakého rozhrania

2.12 Delegáti

Delegát je v jazyku C# 3.0 používateľsky deklarovaný odkazový dátový typ, ktorého inštancia dokáže v sebe zapuzdrovať odkaz na cieľovú inštančnú alebo statickú metódu. Len čo je inštancia delegáta prepojená s cieľovou metódou, možno danú metódu vyvolať nepriamo, prostredníctvom inštancie delegáta. Podľa svojej praktickej aplikácie sa inštancia delegáta podobá na použitie smerníka na funkciu, ktorý je známy z prostredia programovacích jazykov C a C++.

Pre nájdenie vhodných analógií medzi smerníkmi na funkcie a inštanciami delegátov najskôr ozrejmíme použitie smerníkov na funkcie v jazykoch C/C++, a potom sa budeme sústrediť na výklad delegátov v jazyku C# 3.0.

V jazykoch C a C++ môžeme definovať špeciálnu smerníkovú premennú, do ktorej smie byť uložený smerník na vstupný bod tela istej cieľovej funkcie[30]. Každá funkcia je v rámci fyzického procesu aplikácie umiestnená na určitej pamäťovej adrese. Adresa, ktorá identifikuje prvý príkaz tela funkcie je známa ako adresa vstupného bodu tela funkcie. Práca so smerníkmi na funkcie sa uskutočňuje podľa tejto triády:

[30] Smerníky na funkcie sú nepomerne zložitejším mechanizmom ako štandardné smerníky. V našom ponímaní budeme za „štandardné smerníky“ považovať smerníky predstavujúce pamäťové adresy, na ktorých sa nachádzajú objekty neobjektových alebo objektových dátových typov.

1. Definícia smerníkovej premennej, do ktorej môže byť uložený smerník na funkciu.
2. Inicializácia smerníkovej premennej smerníkom ukazujúcim na vstupný bod tela cieľovej funkcie[31].
3. Nepriama aktivácia cieľovej funkcie pomocou smerníka na funkciu, ktorý je uložený v definovanej smerníkovej premennej.

Nasledujúci definičný príkaz jazyka C zakladá novú smerníkovú premennú, ktorá môže byť inicializovaná smerníkom na vstupný bod tela funkcie:

```
int (*p_f)(char *x);
```

Definovaná smerníková premenná sa volá **p_f** a môže byť inicializovaná smerníkom na vstupný bod tela akejkoľvek cieľovej funkcie, ktorá vyhovuje týmto kritériám:

1. Funkcia vracia celočíselnú návratovú hodnotu typu **int**.
2. Funkcia je parametrická, pričom definuje jeden formálny parameter: týmto formálnym parametrom je smerníková premenná typu **char***[32].

Skôr, ako môžeme definovanú smerníkovú premennú inicializovať, musíme pripraviť deklaráciu (funkčný prototyp) a definíciu cieľovej funkcie[33]. Pre naše potreby sme zvolili funkciu **ZistitDlzkuRetazca**, ktorá dokáže analyzovať počet znakov v ľubovoľnom textovom reťazci.

```
#include <stdio.h>

// Deklarácia cieľovej funkcie (funkčný prototyp).
int ZistitDlzkuRetazca(char *p);

// Definícia hlavnej funkcie.
int main()
{
    // Definícia smerníkovej premennej, do ktorej bude môcť byť
    // uložený smerník na funkciu.
    int (*p_f)(char *x);
    return 0;
}
```

[31] Prvé dva kroky triády (definícia a inicializácia smerníkovej premennej) možno spojiť do jedného agregovaného kroku, tzv. definičnej inicializácie smerníkovej premennej. Definičnú inicializáciu sa odporúča použiť vtedy, ak už pri vytváraní smerníkovej premennej poznáme jej konkrétnu inicializačnú hodnotu.

[32] Podotknime, že identifikátor formálneho parametra môže v definičnom príkaze absentovať. Z pohľadu kompilátora je dôležité, aby bol korektne zadaný dátový typ formálneho parametra, identifikátor už nemusí byť nevyhnutne prítomný. Napriek tomu budeme identifikátory vždy uvádzať, pretože ich prispením je zapísaný zdrojový kód zrozumiteľnejší a lepšie čitateľný.

[33] Hoci jazyk C nevyžaduje, aby bola každá definovaná funkcia opatrená svojím prototypom, riadime sa v tomto smere modelom jazyka C++ a všetky definované funkcie vždy správne deklarujeme. Pripomeňme, že ak sa definícia funkcie v jazyku C nachádza pred volaním tejto funkcie, vystupuje zároveň ako prototyp funkcie.

```
// Definícia cieľovej funkcie.
int ZistitDlzkuRetazca(char *p)
{
    int znaky = 0;
    while(*p != '\0')
    {
        znaky++;
        p++;
    }
    return znaky;
}
```

Ďalší fragment zdrojového kódu jazyka C predvádza, ako prebieha inicializácia smerníkovej premennej a nepriama aktivácia cieľovej funkcie:

```
int main()
{
    int (*p_f)(char *x);
    int znaky;
    // Inicializácia smerníkovej premennej smerníkom na cieľovú funkciu.
    p_f = ZistitDlzkuRetazca;
    // Nepriame vyvolanie cieľovej funkcie pomocou smerníka na funkciu.
    znaky = p_f("Paralelne programovanie");
    printf("Dlzka retazca (znaky): %d.", znaky);
    getchar();
    return 0;
}
```

Inicializácia smerníkovej premennej sa uskutočňuje v priraďovacom príkaze. Všimnime si, že na pravej strane od operátora priradenia sa nachádza identifikátor cieľovej funkcie. Ak kompilátor analyzuje prítomnosť identifikátora cieľovej funkcie v tomto programovom kontexte, implicitne ho konvertuje na smerník na vstupný bod tela cieľovej funkcie. Týmto smerníkom je potom inicializovaná smerníková premenná.

Po svojej inicializácii môžeme smerníkovú premennú použiť tak, ako keby sme pracovali so samotnou cieľovou funkciou. Vzhľadom na to, že naša funkcia je parametrická, pričom očakáva smerník na dočasné pole s textovými znakmi, odovzdávame jej reťazcovú konštantu. Nasleduje spustenie kódu funkcie, analýza predmetného textového reťazca a vrátenie celočíselnej návratovej hodnoty, ktorá predstavuje počet znakov reťazca[34].

Hoci sme cieľovú funkciu nepriamo aktivovali prostredníctvom identifikátora smerníkovej premennej vo forme, v ktorej sme identifikátor smerníkovej premennej stotožnili s názvom cieľovej funkcie, môžeme použiť aj iný variant. V ňom dáme explicitne najavo, že v skutočnosti dochádza k dereferencii smerníka na funkciu, ktorý je uložený v smerníkovej premennej.

[34] Naša používateľská funkcia **ZistitDlzkuRetazca** sa správa rovnako ako vstavaná knižničná funkcia jazyka C s názvom **strlen**, ktorá je deklarovaná v hlavičkovom súbore string.h. Návratovou hodnotou oboch funkcií je skutočný počet znakov v skúmanom textovom reťazci bez nulového (ukončovacieho) znaku.

```
int main()
{
    int (*p_f)(char * x);
    int znaky;
    p_f = ZistitDlzkuRetazca;

    // Explicitná dereferencia smerníka na funkciu, ktorý sa nachádza
    // v smerníkovej premennej.
    znaky = (*p_f)("Paralelne programovanie");
    printf("Dlzka retazca (znaky): %d.", znaky);
    getchar();
    return 0;
}
```

S delegátmi v jazyku C# 3.0 pracujeme podobne ako so smerníkovými premennými obsahujúcimi smerníky na funkcie v jazykoch C a C++. Keďže delegát je používateľsky deklarovaným odkazovým dátovým typom, prvým krokom je jeho deklarácia. Nasledujúci deklaračný príkaz zavádza deklaráciu delegáta s identifikátorom **Delegát**:

```
// Deklarácia delegáta ako nového používateľsky deklarovaného
// odkazového dátového typu.
internal delegate int Delegát(string reťazec);
```

Deklarovaný delegát disponuje internými prístupovými právami a jeho inštancia bude môcť byť asociovaná s cieľovou inštančnou alebo statickou metódou, ktorá vyhovuje týmto požiadavkám:

1. Metóda vracia celočíselnú návratovú hodnotu primitívneho dátového typu **int**.

2. Metóda je parametrická, pracuje s jedným formálnym parametrom, ktorého typom je primitívny odkazový dátový typ **string**. Formálny parameter metódy bude inicializovaný odkazom na inštanciu triedy **System.String**, v ktorej je zapuzdrený textový reťazec.

Deklarácia delegáta sa syntakticky podobá na definíciu smerníkovej premennej jazyka C/C++, do ktorej môže byť uložený smerník na funkciu. Sémantický rozdiel však spočíva v tom, že delegát je novým dátovým typom: jeho kompletnú deklaráciu vykoná kompilátor jazyka C# 3..0 vo svojej vlastnej réžii. Ak by sme sa pozreli na ekvivalentný nízkoúrovňový MSIL kód, uvideli by sme, že kompilátor automaticky vytvoril triedu delegáta, ktorá je podtriedou systémovej triedy **System.MulticastDelegate**.

V tele triedy delegáta sú situované definície pre inštančný konštruktor a trojicu parametrických inštančných metód s názvami **BeginInvoke**, **EndInvoke** a **Invoke**.

Druhým krokom pri práci s delegátom je definovanie cieľovej metódy, ktorá bude s jeho inštanciou spojená[35].

Syntaktický obraz definície metódy **ZistiťDĺžkuReťazca** vyzerá takto:

```
class Program
{
    internal delegate int Delegát(string reťazec);

    static void Main(string[] args)
    {
        // Telo hlavnej metódy je zatiaľ prázdne.
    }

    // Definícia metódy, ktorá bude spojená s inštanciou delegáta.
    static int ZistiťDĺžkuReťazca(string reťazec)
    {
        return reťazec.Length;
    }
}
```

Pre programátorov, ktorí prichádzajú z jazykov C/C++, je dôležité oznámenie, že v jazyku C# 3.0 pracujeme bežne len s definíciami, nie s deklaráciami metód[36].

Definovaná metóda je statická, čo je dané technickými požiadavkami na prehľadnejší zdrojový kód. Keby sme totiž chceli inštanciu delegáta spojiť s inštančnou (a teda nestatickou) cieľovou metódou, museli by sme deklarovať triedu a v nej definovať požadovanú inštančnú metódu.

Napokon založíme inštanciu delegáta a uložíme do nej odkaz na cieľovú statickú metódu:

```
static void Main(string[] args)
{
    // Inštanciácia delegáta a asociovanie založenej inštancie
    // s cieľovou statickou metódou.
    Delegát del = new Delegát(ZistiťDĺžkuReťazca);

    // Indirektná aktivácia cieľovej statickej metódy pomocou
    // zostrojenej inštancie delegáta.
    int znaky = del("Paralelné programovanie");
```

[35] Naším cieľom je ukázať základné použitie delegátov, a preto budeme náležite definovať cieľovú metódu, ktorá bude s inštanciou delegáta asociovaná. V praktickom nasadení však nemusia vývojári vždy cieľové metódy definovať, pretože môžu použiť metódy tried, resp. ich inštancií, ktoré už existujú a nachádzajú sa v spoločnej bázovej knižnici (FCL) vývojovo-exekučnej platformy Microsoft .NET Framework 3.5.

[36] Jazyk C# 3.0 nevyžaduje použitie funkčných prototypov. Za istých okolností však môžeme s prototypmi funkcií pracovať. Ide predovšetkým o situácie, kedy sme nútení spolupracovať s natívnymi funkciami aplikačného programového rozhrania Win32 API. Funkcie tvoriace jadro Win32 API majú natívnu povahu (nejde o riadený kód) a ak chceme s nimi kooperovať, musíme do zdrojového kódu jazyka C# 3.0 zaviesť deklarácie ich riadených prototypov. Tak sa na syntaktickej úrovni vytvorí prepojenie medzi vrstvami riadeného (.NET) a natívneho kódu (Win32 API). Interoperabilita je realizovaná pomocou nízkoúrovňových softvérových služieb Platform Invoke (P/Invoke).

```
    Console.WriteLine("Dĺžka reťazca (znaky): {0}.", znaky);
    Console.Read();
}
```

Keďže je deklarovaný delegát odkazový dátový typ, jeho inštanciu vytvoríme pomocou operátora **new**. Vo chvíli, keď je na riadenej halde alokovaná inštancia delegáta, dochádza k jej inicializácii. V procese inicializácie je do inštancie zapuzdrený odkaz na cieľovú statickú metódu s uvedeným identifikátorom. Napokon je odkaz na inštanciu delegáta uložený do odkazovej premennej, ktorá je vytvorená v rámci inštanciačného príkazu. Keď máme inicializovanú inštanciu delegáta, môžeme prostredníctvom nej nepriamo zavolať cieľovú statickú metódu. Všimnime si, že odkazovú premennú (cez ktorú je inštancia delegáta dosiahnuteľná) používame podobne, ako by to bola cieľová metóda, ktorú chceme spustiť. Po aktivácii metódy budú vykonané všetky príkazy, ktoré sa v jej tele nachádzajú (v našom prípade pôjde len o jeden príkaz). Napokon nám metóda poskytne svoju návratovú hodnotu, ktorú ukladáme do vopred pripravenej premennej.

Predchádzajúci výpis zdrojového kódu jazyka C# 3.0 sa dá prepísať nasledujúcim spôsobom:

```
static void Main(string[] args)
{
    // Syntakticky zjednodušená inštanciácia delegáta.
    Delegát del = ZistiťDĺžkuReťazca;

    // Cieľovú metódu voláme explicitne pomocou inštančnej
    // metódy Invoke delegáta.
    int znaky = del.Invoke("Paralelné programovanie");
    Console.WriteLine("Dĺžka reťazca (znaky): {0}.", znaky);
    Console.Read();
}
```

Líši sa len syntaktická stránka zdrojového kódu, finálny efekt je rovnaký. Charakterizujme modifikácie, ktoré sme v kóde uskutočnili:

1. Po prvé, zdrojový kód využíva zjednodušenú formu inštanciácie a inicializácie delegáta. Kód explicitne nepoužíva operátor **new**, a teda ani nie je vidieť priamu aktiváciu inštančného konštruktora. Z pohľadu jazykov C/C++ sa celý inštanciačný príkaz ponáša na definičnú inicializáciu odkazovej premennej **del**. Hoci kód navodzuje dojem, že do uvedenej premennej je priradený odkaz na vstupný bod tela cieľovej metódy, nie je to tak. Do odkazovej premennej je totiž uložená objektová referencia, ktorá je nasmerovaná na inštanciu zrodeného delegáta. Táto inštancia leží na riadenej halde a obsahuje pamäťovú adresu vstupného bodu tela cieľovej metódy.

2. Po druhé, v zdrojovom kóde je explicitne aktivovaná metóda **Invoke** inštancie delegáta. V skutočnosti je metóda **Invoke** volaná vždy, ak nie je stanovené inak.

V súvislosti s aktiváciou cieľovej metódy je dôležité uviesť, že zdrojový kód tejto metódy je spracúvaný synchrónne. To znamená, že vykonávanie zdrojového kódu hlavnej metódy **Main** bude pozastavené dovtedy, pokiaľ nebudú spracované všetky príkazy synchrónne aktivovanej cieľovej metódy. Exekúcia kódu v tele hlavnej metódy bude pokračovať v okamihu, keď synchrónne spustená cieľová metóda vykoná celú svoju činnosť.

Ďalej si ukážeme, ako inštanciu delegáta prepojiť s cieľovou inštančnou (nestatickou) metódou:

```
// Deklarácia triedy.
internal class HraciaKocka
{
    // Definície súkromných dátových položiek triedy.
    private uint číslo, šestka;
    private Random generátor;

    // Definícia verejne prístupného bezparametrického konštruktora.
    public HraciaKocka()
    {
        generátor = new Random();
    }

    // Definícia inštančnej metódy triedy.
    public void Hodiť(ushort početHodov)
    {
        for (int i = 1; i <= početHodov; i++)
        {
            číslo = (uint)generátor.Next(1, 7);
            Console.WriteLine("Bolo hodené číslo {0}.", číslo);
            if (číslo == 6)
                šestka++;
        }
        if (šestka == 0)
            Console.WriteLine("Číslo 6 nepadlo ani jedenkrát.");
        else
        {
            Console.WriteLine("Číslo 6 padlo {0}krát.", šestka);
            Console.WriteLine("Vaša úspešnosť bola " +
                ((float)šestka / početHodov * 100).ToString("0") + " %.");
        }
     }
}
// Deklarácia delegáta.
internal delegate void Delegát(ushort x);

class Program
{
    static void Main(string[] args)
    {
        HraciaKocka kocka = new HraciaKocka();

        // Vytvorenie inštancie delegáta a spojenie
        // inštancie s cieľovou inštančnou metódou.
        Delegát prvýDelegát = new Delegát(kocka.Hodiť);

        // Aktivovanie cieľovej inštančnej metódy pomocou
```

```
        // inštancie delegáta.
        prvýDelegát.Invoke(10);

        Console.ReadLine();
    }
}
```

Asociovanie inštancie delegáta s inštančnou cieľovou metódou je jednoduché, pretože všetko, čo musíme urobiť, je odovzdať inštančnému konštruktoru delegáta odkaz na požadovanú metódu. Keďže cieľová metóda je inštančná a nie statická, vždy musíme špecifikovať odkazovú premennú, prostredníctvom ktorej je príslušná inštancia dosiahnuteľná. Bodkovým operátorom (.) následne určíme metódu, s ktorou chceme pracovať.

2.12.1 Spojenie inštancie delegáta s anonymnou cieľovou metódou

Počnúc verziou 2.0 programovacieho jazyka C# je možné prepojiť inštanciu delegáta nielen s pomenovanou inštančnou alebo statickou cieľovou metódou, ale taktiež s anonymnou metódou. Anonymná metóda je metóda, ktorá je automaticky generovaná kompilátorom jazyka C# podľa pokynov programátora, pričom kompilátor túto metódu implicitne pomenuje. Identifikátor metódy však nie je známy vývojárovi, takže z jeho pohľadu je zostavená metóda anonymná.[37]

```
class Program
{
    // Deklarácia delegáta.
    delegate void Delegát(int min, int max);

    static void Main(string[] args)
    {
        // Inštanciácia delegáta a spojenie inštancie
        // s anonymnou metódou.
        Delegát druhýDelegát = delegate(int min, int max)
        {
            Random generátor = new Random();
            for (int i = 0; i < 10; ++i)
            {
                Console.WriteLine("{0}. číslo je {1}.",
                    i + 1, generátor.Next(min, max + 1));
            }
        };

        druhýDelegát(1, 100);
        Console.Read();
    }
}
```

[37] Hoci pomocou mechanizmu reflexie je možné diagnostikovať identifikátor anonymnej metódy, v bežných podmienkach nie je potrebné (a spravidla ani užitočné) tento identifikátor poznať. Identifikátor anonymnej metódy sa môže odlišovať v rôznych prekladových reláciách zdrojového kódu.

Pozrime sa bližšie na príkaz, ktorý zakladá novú inštanciu deklarovaného delegáta. Na pravej strane od operátora priradenia sa nachádza kľúčové slovo **delegate** s určením signatúry anonymnej metódy. Ako si môžeme všimnúť, v našej praktickej ukážke bude anonymná metóda parametrická, pričom jej formálne parametre budú môcť prijať dve celočíselné hodnoty. Po špecifikácii signatúry anonymnej metódy dochádza k definícii tela tejto metódy. Telo anonymnej metódy je (podobne ako telo pomenovanej metódy) tvorené programovými príkazmi, ktoré sa nachádzajú v bloku ohraničenom zloženými zátvorkami ({}). Za uzatváracou zloženou zátvorkou nakoniec zadávame symbol bodkočiarky, pretože musíme ukončiť príkaz, ktorý inštanciuje a inicializuje delegáta.

Použitie anonymných metód v súvislosti s inštanciami delegátov je veľmi užitočná technika, ktorá nás zbavuje povinnosti za každých okolností definovať samostatné metódy, ktoré budú previazané s inštanciami delegátov. Namiesto toho kompilátoru odovzdáme požadované znenie zoznamu formálnych parametrov a programových príkazov, na čo nám kompilátor automaticky vygeneruje korektnú anonymnú metódu.

2.12.2 Kovariancia a kontravariancia delegátov

Programovací jazyk C# vo verziách 1.0 a 1.1 vyžadoval, aby bola signatúra deklarovaného delegáta úplne zhodná so signatúrou cieľovej metódy, s ktorou bude inštancia tohto delegáta spojená. Vo verzii 2.0 prišlo k uvoľneniu spomenutého formalizmu. Jazyk C# 2.0 (a rovnako aj jazyk C# 3.0) definuje dve vlastnosti delegátov, ktoré vymedzujú pravidlá pre prípustné odlišnosti signatúr. Tieto vlastnosti sú známe ako kovariancia a kontravariancia delegátov.

Podľa kovariancie je možné, aby bol dátový typ návratovej hodnoty cieľovej metódy špecifickejší ako dátový typ návratovej hodnoty deklarovaného delegáta. Praktickú aplikáciu kovariancie demonštruje nasledujúci fragment zdrojového kódu jazyka C# 3.0:

```
// Deklarácia  bázovej triedy.
class A
{
    // Definícia virtuálnej metódy bázovej triedy.
    public virtual A m()
    {
        return this;
    }
}
```

```
// Deklarácia odvodenej triedy.
class B : A
{
    // Definícia inštančnej metódy odvodenej triedy.[38]
    public B w()
    {
        return this;
    }
}

class Program
{
    // Deklarácia delegáta.
    delegate A Delegát();

    static void Main(string[] args)
    {
        // Inštanciácia a inicializácia delegáta.
        Delegát del = new Delegát(metóda);
        // Indirektná aktivácia cieľovej metódy.
        Console.WriteLine(del().ToString());
        Console.Read();
    }

    // Definícia cieľovej metódy.
    static B metóda()
    {
        B b = new B();
        return b.w();
    }
}
```

Z deklarácie delegáta vyplýva, že dátovým typom jeho návratovej hodnoty je bázová trieda **A**. Z tohto titulu môžeme vyhlásiť, že deklarovaný delegát bude môcť byť po svojej inštanciácii prepojený s takou cieľovou metódou, ktorá v podobe svojej návratovej hodnoty vracia odkaz na inštanciu triedy **A** alebo odkaz na inštanciu akejkoľvek podtriedy triedy **A**.

Keď sa pozrieme na definíciu cieľovej statickej metódy, s ktorou je inštancia delegáta spojená, vidíme, že dátovým typom návratovej hodnoty tejto metódy je trieda **B**. Pretože trieda **B** je podtriedou triedy **A**, je podľa pravidiel kovariancie spojenie inštancie delegáta a cieľovej metódy úplne korektné. Po tom, ako bude prostredníctvom inštancie delegáta vyvolaná cieľová statická metóda, odohrajú sa tieto činnosti:

- V riadenej halde sa alokuje inštancia podtriedy **B**.
- Alokovaná inštancia podtriedy **B** je inicializovaná bezparametrickým inštančným konštruktorom.
- Aktivuje sa metóda **w** inštancie podtriedy **B**.
- Metóda **w** vráti odkaz na aktuálnu inštanciu podtriedy **B**.

[38] Metóda odvodenej triedy neprekrýva zdedenú metódu bázovej triedy.

- Statická metóda prijme poskytnutý odkaz na inštanciu podtriedy **B** a odošle ho späť klientskemu programovému kódu (hlavnej metóde **Main**).

- V tele hlavnej metódy je navrátený odkaz na inštanciu podtriedy **B** konvertovaný do textovej podoby. Výsledkom konverzného procesu je zobrazenie kvalifikovaného mena triedy, z ktorej inštancia vznikla.

Kontravariancia delegátov sa sústreďuje na rozdielnosť medzi dátovými typmi formálnych parametrov cieľovej metódy a deklarovaného delegáta. Podľa kontravariancie je možné, aby cieľová metóda disponovala generickejšími dátovými typmi svojich formálnych parametrov v porovnaní s dátovými typmi formálnych parametrov, ktoré sú uvedené v deklarácii delegáta.

Nasleduje praktická ukážka kontravariancie delegátov v jazyku C# 3.0:

```
// Deklarácia bázovej triedy.
class A
{
    // Definícia prekrývajúcej metódy bázovej triedy.
    public override string ToString()
    {
        return "A";
    }
}

// Deklarácia odvodenej triedy.
class B : A
{
    // Definícia prekrývajúcej metódy odvodenej triedy.
    public override string ToString()
    {
        return "B";
    }
}

class Program
{
    // Deklarácia delegáta.
    delegate string Delegát(B b);

    static void Main(string[] args)
    {
        // Inštanciácia a inicializácia delegáta.
        Delegát del = metóda;
        B obj = new B();

        // Nepriama aktivácia cieľovej metódy.
        string s = del(obj);

        Console.WriteLine(s);
        Console.Read();
    }
```

```
    // Definícia cieľovej metódy.
    static string metóda(A a)
    {
        return a.ToString();
    }
}
```

Komentár k zdrojovému kódu: V kóde deklarujeme dve triedy, **A** je bázová trieda a **B** je odvodená trieda. Do tela bázovej triedy **A** je zavedená definícia prekrývajúcej metódy s identifikátorom **ToString**. Táto metóda bázovej triedy **A** prekrýva rovnomennú zdedenú metódu primárnej bázovej triedy **System.Object**. Ak teda založíme inštanciu bázovej triedy **A** a zavoláme jej metódu **ToString**, získame textový reťazec „A“. Odvodená trieda **B** poskytuje komparatívnu funkcionalitu, s tým rozdielom, že definovaná metóda **ToString** prekrýva rovnomennú zdedenú metódu bázovej triedy **A**. Analogicky, v spojení s inštanciou podtriedy **B** bude volaná spomenutá prekrývajúca metóda. Na výstupe teda získame textový reťazec „B“.

Interesantný je príkaz, ktorý deklaruje delegáta. Deklaračný príkaz definuje jeden formálny parameter, ktorého dátovým typom je podtrieda **B**. Môžeme teda povedať, že inštanciu delegáta budeme môcť asociovať s akoukoľvek cieľovou metódou, ktorej formálny parameter dokáže prijať odkaz na inštanciu podtriedy **B**.

Vzťah medzi deklarovaným delegátom a definovanou cieľovou statickou metódou je vybudovaný na báze kontravariancie. To znamená, že je dovolené, aby cieľová statická metóda definovala formálny parameter, ktorého dátovým typom je bázová trieda **A**.

Keď bude pomocou inštancie delegáta nepriamo spustená cieľová statická metóda, jej formálny parameter získa odkaz na inštanciu podtriedy **B**. Tento odkaz bude implicitne konvertovaný na odkaz na inštanciu bázovej triedy **A**, pričom týmto skonvertovaným odkazom bude inicializovaný formálny parameter cieľovej statickej metódy. Predostretý mechanizmus je realizovateľný a vždy bude fungovať. Je to preto, že do formálneho parametra, ktorého dátovým typom je bázová trieda **A** môže byť kedykoľvek uložený odkaz na inštanciu odvodenej triedy **B**. Funkčnosť charakterizovaného mechanizmu je daná jedným zo základných princípov objektovo orientovaného programovania, podľa ktorého sa môže potomok vyskytnúť všade tam, kde je očakávaný jeho rodič.

2.13 λ-výrazy a λ-kalkul

V kolekcii syntakticko-sémantických inovácií jazyka C# 3.0 majú svoje pevné miesto λ-výrazy (lambda-výrazy). Lambda-výrazy sú súčasťou širšej matematicko-informatickej vedeckej teórie, ktorá sa nazýva λ-kalkul (lambda-kalkul). λ-kalkul patrí k jednej z paradigiem tvorby počítačového softvéru, ktorej sa vraví funkcionálne programovanie. Funkcionálne programovanie sa na stavbu softvéru pozerá inak ako imperatívne programovanie. Pripomeňme, že v ponímaní imperatívnej paradigmy je program

považovaný za konečnú a neprázdnu množinu inštrukcií, ktoré sú spracúvané jedna za druhou a ktoré formujú jednotlivé kroky implementovaných imperatívnych algoritmov.

Imperatívne programovanie je vývojárom blízke, pretože sa s ním stretávajú už od nepamäti: jazyky BASIC, C, C++, C#, Visual Basic a mnohé ďalšie môžeme zaradiť do kategórie imperatívnych jazykov. Na druhú stranu, funkcionálne programovanie je postavené na aparáte λ-kalkulu a počítačový program definuje z matematického hľadiska. Z pohľadu funkcionálneho programovania je program indikovaný ako konečná a neprázdna množina funkcií. Na funkcie, ktoré zapuzdrujú požadované algoritmy, sú aplikované rôzne transformácie (najmä redukcie a konverzie), vďaka ktorým sa program dopracuje k požadovanému výsledku. λ-kalkul intenzívne pracuje s λ-výrazmi, prostredníctvom ktorých sú funkcie anonymne definované.

λ-kalkul vo svojom jadre pracuje s unárnymi funkciami, teda s funkciami, ktoré sú schopné pracovať len s jedným argumentom. Argumentom unárnej funkcie môže byť ďalšia unárna funkcia a unárna funkcia môže inú unárnu funkciu vracať tiež ako svoju návratovú hodnotu. Unárna funkcia je definovaná anonymne, čo znamená, že ide o anonymnú funkciu, ktorá nemá vlastný identifikátor. Na definíciu unárnych anonymných funkcií sa používajú λ-výrazy. λ-výraz vytvára funkciu a podáva rovnako informácie o jej formálnom parametre.

Napríklad matematický zápis funkcie $f(x) = x + 7$ vyjadríme nasledujúcim λ-výrazom: $(\lambda\, x \mid x + 7)$. λ-výrazy zapisujeme vždy do zátvoriek, presne tak, ako sme uviedli. Každý λ-výraz formujú tieto štyri časti:

1. lambda-symbol (λ),
2. formálny parameter funkcie (x),
3. vertikálny oddeľovač (|),
4. telo funkcie, resp. telo λ-výrazu ($x + 7$).

V matematike by sme na výpočet hodnoty funkcie so vstupným argumentom 2 použili zápis $f(2)$. V λ-kalkule zapíšeme výraz $(\lambda\, x \mid x + 7)\, 2$, kde hodnota 2 predstavuje argument, ktorý bude dosadený do formálneho parametra (x) a použije sa v tele funkcie na určenie jej návratovej hodnoty. Návratová hodnota funkcie je v skutočnosti hodnota λ-výrazu. Je zrejmé, že hodnotou λ-výrazu bude číslo 9.

λ-výraz je z formálnej stránky definovaný nasledujúcim spôsobom:

1. **X** je premenná, ktorá je definovaná svojím identifikátorom.
2. $(\lambda\, \mathrm{X} \mid V)$ je abstrakcia, v rámci ktorej je **X** premenná a **V** je telo λ-výrazu. Termín „abstrakcia“ môžeme substituovať termínom „definícia anonymnej funkcie“.
3. $(\lambda\, \mathrm{X} \mid V)\ A$ je aplikácia, teda volanie anonymnej funkcie s argumentom **A**.

Identifikátorom premennej **X** môže byť aj symbol zložený z viacerých znakov. Premenné, ktoré sa nachádzajú v λ-výrazoch, rozdeľujeme na viazané a voľné. Klasifikačné kritérium je celkom jednoduché. Pokiaľ je premenná asociovaná s lambda-symbolom, je ponímaná ako viazaná. V opačnom prípade ide o voľnú premennú. V λ-výraze $(\lambda\, x \mid 2x + 1)$ je **x** viazanou premennou. Keď predchádzajúci λ-výraz upravíme na $(\lambda\, \mathrm{x} \mid 2x + y)$, tak **x** zostáva i naďalej viazanou premennou, no premenná **y** je voľná (neexistuje žiaden λ-symbol, ktorý by bol s ňou spojený).

Keďže λ-kalkul dokáže pracovať len s unárnymi funkciami, musíme reálnu funkciu s viacerými premennými (formálnymi parametrami) v tomto prostredí modelovať ako kompozíciu jednej unárnej funkcie, ktorá ako argument prijíma ďalšiu unárnu funkciu.

Uvažujme tento príklad: Nech je počet tranzistorov umiestnených v jednom viacjadrovom procesore daný funkciou $f_1(n) = 10^8 \times n$, kde n je počet exekučných jadier uložených v procesorovom balení. Povedzme, že budeme chcieť sledovať celkový počet inštalovaných tranzistorov v kolekcii viacjadrových procesorov, ktorá je vyrobená počas jedného výrobného cyklu. Tejto relácii nech zodpovedá funkcia $f_2(p, n) = p \times 10^8 \times n$, kde p je celkový počet vyprodukovaných procesorov. Definíciu funkcie f_2 by sme mohli zapísať aj takto: $f_2(p, f_1(n)) = p \times f_1(n)$. Použitím nástrojov λ-kalkulu zapíšeme vzťahy medzi funkciami nasledujúcim spôsobom:

1. abstrakcia: $(\lambda\, n \mid 10^8 \times n)$
2. abstrakcia: $(\lambda\, p \mid (\lambda\, n \mid p \times (10^8 \times n)))$

Aplikácia pre 100 štvorjadrových procesorov, ktoré sú vyrobené v jednom cykle, bude potom vyzerať takto:

$$(\lambda\, p \mid (\lambda\, n \mid p \times (10^8 \times n)))\ 100\ 4$$

V zloženom λ-výraze možno rozpoznať vnútornú (vnorenú) abstrakciu:

$$(\lambda\, n \mid p \times (10^8 \times n))$$

Vyhodnotenie zloženého λ-výrazu prebieha v týchto krokoch:

1. Najskôr nahradíme viazané premenné (p a n) konkrétnymi hodnotami:

$$(\lambda\, p \mid (\lambda\, n \mid 100 \times (10^8 \times 4)))$$

2. Vyhodnotíme vnútornú a vonkajšiu abstrakciu:

$$(\lambda\, p \mid (\lambda\, n \mid 10^{10} \times 4))$$

3. Výsledná hodnota analyzovaného λ-výrazu je 4×10^{10}. Na základe vypočítanej hodnoty môžeme povedať, že 100 procesorov osadených štyrmi jadrami obsahuje 40 miliárd tranzistorov.

Vyhodnocovanie λ-výrazov sa uskutočňuje pomocou vyhodnocovacích pravidiel, ktoré sa nazývajú konverzie. λ-kalkul pozná tri základné typy konverzných operácií: α-konverzia, β-redukcia a η-redukcia.

2.14 λ-výrazy v jazyku C# 3.0

Implementácia λ-výrazov do jazyka C# 3.0 je ďalším evolučným stupňom použitia anonymných metód. λ-výraz sa v jazyku C# 3.0 definuje podľa nasledujúceho vzoru:

(formálne parametre) => telo λ-výrazu

Ak je množina formálnych parametrov jednoprvková, zátvorky v zápise nie sú povinné. V opačnom prípade musíme uvádzať zátvorky, v ktorých sú jednotlivé formálne parametre oddelené čiarkami. Formálne parametre sú reprezentované identifikátormi, ktoré môžu byť doplnené explicitnými špecifikáciami dátových typov. Pokiaľ parameter nedisponuje priamo zadaným dátovým typom, kompilátor sa pokúsi jeho typ odvodiť implicitne na základe kontextu, v akom je parameter použitý. Túto akciu vykonáva mechanizmus typovej inferencie. Podstatou mechanizmu typovej inferencie je strojová dedukcia dátového typu analyzovanej entity. Bez ohľadu na to, či je dátový typ parametra zadaný explicitne, alebo je implicitne inferovaný, typ formálneho parametra je určený vždy v čase prekladu zdrojového kódu.

Symbol => predstavuje λ-operátor. Tento operátor sa používa len v λ-výrazoch, pričom pôsobí ako oddeľovač formálnych parametrov a tiel λ-výrazov. λ-operátor => má rovnakú prioritu ako priraďovací operátor (=) a je asociatívny v smere sprava doľava.

Telo λ-výrazu je tvorené syntakticky a sémanticky správne zapísanou postupnosťou operandov a operátorov, ktoré možno spracovať a vyhodnotiť. Keďže λ-výrazy sú veľmi úzko prepojené s anonymnými metódami, tak telo λ-výrazu bude zaliate do automaticky vygenerovanej anonymnej metódy. Samozrejme, aby mohla byť vygenerovaná anonymná metóda použiteľná, musíme deklarovať a inštanciovať delegáta, ktorý bude s anonymnou metódou prostredníctvom syntaxe λ-výrazu asociovaný.

Nasledujúci výpis zdrojového kódu jazyka C# 3.0 demonštruje elementárne použitie λ-výrazu:

```
class Program
{
    // Deklarácia delegáta.
    delegate int Del(int a);
    static void Main(string[] args)
    {
        // Inštanciácia delegáta a inicializácia zrodenej
        // inštancie delegáta λ-výrazom.
        Del d = new Del(x => x + 10 * 2);
        // Aktivácia cieľovej anonymnej metódy.
        Console.WriteLine(d(10));
        Console.Read();
    }
}
```

Deklarácia delegáta nám vraví, že jeho inštancia bude môcť byť asociovaná s metódou, ktorá prijíma jeden 32-bitový celočíselný argument a vracia rovnako dlhú návratovú hodnotu. V tele hlavnej metódy **Main** je významný hneď prvý príkaz, ktorý predstavuje definičnú inicializáciu odkazovej premennej. Výraz, nachádzajúci sa napravo od priraďovacieho operátora, vytvára inštanciu delegáta, ktorú automaticky spája s implicitne generovanou anonymnou metódou. V tele samočinne zhotovenej anonymnej metódy je uložené telo definovaného λ-výrazu. λ-výraz, ktorý používame, má v λ-kalkule takúto podobu:

$$(\lambda\, x \mid x + 10 \times 2)$$

Ekvivalentný λ-výraz zapísaný podľa syntaktických pravidiel jazyka C# 3.0 vyzerá takto:

```
x => x + 10 * 2
```

Formálnym parametrom je premenná **x**, ktorá nemá explicitne determinovaný dátový typ. Ak absentuje priama špecifikácia dátového typu, vieme, že kompilátor využije mechanizmus typovej inferencie a dátový typ určí implicitne. Telo λ-výrazu tvorí jednoduchý aritmetický výraz, ktorého typ bude opäť implicitne inferovaný.

Z inštanciačného príkazu je zrejmé, že λ-výraz je v podobe vstupného argumentu odovzdaný formálnemu parametru inštančného konštruktora delegáta. Hoci zápis zdrojového kódu naznačuje túto operáciu, v skutočnosti je situácia iná. λ-výraz bude konvertovaný na parametrickú anonymnú metódu, telo ktorej bude totožné s telom λ-výrazu. Po inštanciácii delegáta bude zrodená inštancia spojená s anonymnou metódou. Kompilátor vie, že je nutné zostaviť parametrickú anonymnú metódu, pretože deklarovaný delegát je rovnako parametrický. Konečne, konštruktor inštancie delegáta získa odkaz na implicitne definovanú anonymnú metódu.

Po spracovaní inštanciačného príkazu je vytvorená väzba medzi inštanciou delegáta a anonymnou metódou. Táto metóda však nie je spustená okamžite – k aktivácii metódy

prostredníctvom inštancie delegáta dochádza až vtedy, keď je stanovený vstupný argument, ktorý bude metóde ponúknutý. To sa deje v nasledujúcom príkaze:

```
Console.WriteLine(d(10));
```

Výraz **d(10)** zahajuje synchrónnu aktiváciu cieľovej anonymnej metódy s tým, že metóde je odovzdaná celočíselná konštanta. Vo vzťahu k λ-kalkulu je výraz **d(10)** ekvivalentom aplikácie funkcie. Pri ďalšej analýze zistíme, že prebehne tento reťazec operácií:

1. Inštancia delegáta nepriamo vyvolá cieľovú parametrickú anonymnú metódu.

2. Formálny parameter cieľovej anonymnej metódy sa inicializuje kópiou argumentu (implicitné sú dáta odovzdávané hodnotou), teda kópiou celočíselnej konštanty.

3. Vyhodnotí sa λ-výraz uložený v tele cieľovej anonymnej metódy. Hodnota λ-výrazu je 30 $(10 + 10 \times 2)$. Túto hodnotu vráti anonymná metóda vo forme svojej návratovej hodnoty.

4. 32-bitová návratová hodnota anonymnej metódy sa konvertuje na postupnosť textových znakov, odošle sa do výstupného dátového prúdu a zobrazí sa na obrazovke počítača.

Pre signatúru deklarovaného delegáta a formu definovaného λ-výrazu platia tieto pravidlá:

- λ-výraz musí obsahovať toľko formálnych parametrov, koľko formálnych parametrov je definovaných v deklarácii delegáta.

- Dátový typ každého formálneho parametra λ-výrazu musí byť implicitne konvertovateľný na dátový typ zodpovedajúceho formálneho parametra definovaného v deklarácii delegáta.

- Dátový typ návratovej hodnoty λ-výrazu musí byť implicitne konvertovateľný na dátový typ návratovej hodnoty, ktorý je uvedený v deklarácii delegáta.

Vďaka flexibilnej jazykovej špecifikácii C# 3.0 možno inštanciačný príkaz uvedený v predchádzajúcom fragmente zdrojového kódu prepísať aj takýmto spôsobom:

```
// Syntakticky jednoduchší zápis inštanciačného príkazu.
Del d = x => x + 10 * 2;
```

Napriek tomu, že sme modifikovali syntaktický obraz príkazu, jeho pôvodná sémantika zostala zachovaná. Pri štúdiu podobne navrhnutých inštanciačných príkazov delegátov kooperujúcich s λ-výrazmi však musíme presne poznať praktické implikácie, ktoré zo syntakticky jednoduchšie zapísaných príkazov plynú.

V prípade potreby môžeme formálny parameter uviesť s explicitne určeným dátovým typom:

```
// Explicitné stanovenie dátového typu formálneho parametra λ-výrazu.
Del d = (int x) => x + 10 * 2;
```

Upozorňujeme, že ak zapíšeme dátový typ formálneho parametra, tak typ i identifikátor parametra musia byť vložené do zátvoriek (inak kompilátor generuje chybové hlásenie).

Použitie λ-výrazu nás zbavuje povinnosti definovať samostatnú cieľovú metódu, ktorá bude spojená s inštanciou deklarovaného delegáta. Nemusíme dokonca definovať ani anonymnú metódu, pretože tú kompilátor vytvára automaticky podľa znenia λ-výrazu.

λ-výraz smieme použiť napríklad pri projektovaní programu, ktorý na základe vstupných dát klasifikuje nárast výkonnosti pôvodne sekvenčného algoritmu podľa Amdahlovho zákona.

2.15 Praktická aplikácia λ-výrazov v jazyku C# 3.0

Americký vedec Gene Amdahl vynašiel v roku 1967 matematický vzorec na určenie nárastu výkonnosti medzi sekvenčnou a paralelnou verziou počítačového programu. Amdahlov zákon je daný nasledujúcim matematickým vzťahom:

$$N_v = \frac{s + p}{s + \frac{p}{n}}$$

kde:

- N_v je nárast výkonnosti po paralelizácii pôvodne úplne sekvenčného programu.
- s je relatívna početnosť sekvenčných algoritmov počítačového programu.
- p je relatívna početnosť paralelných algoritmov počítačového programu.
- n je počet procesorov počítača, resp. počet exekučných jadier viacjadrového procesora.

Keďže súčet relatívnych početností sekvenčných a paralelných algoritmov počítačového programu bude vždy rovný 1, môžeme vzorec reflektujúci Amdahlov zákon modifikovať takto:

$$N_v = \frac{1}{s + \frac{p}{n}}$$

Uveďme príklad praktického nasadenia Amdahlovho zákona:

$$N_v = \frac{1}{0{,}5 + \frac{0{,}5}{2}} = \frac{1}{0{,}5 + 0{,}25} = \frac{1}{0{,}75} = 1{,}33$$

Príklad predstavuje program, ktorého algoritmická skladba je z 50 % tvorená sekvenčnými algoritmami a z ďalších 50 % paralelnými algoritmami. Takýto program teda presne polovicu úloh rieši pomocou sekvenčných algoritmov a druhú polovicu pomocou paralelných algoritmov. Samozrejme, rozdiely existujú v povahách algoritmov. Zatiaľ čo sekvenčné algoritmy sú spracúvané synchrónne, pre paralelné algoritmy je príznačná asynchrónna exekúcia. Paralelné algoritmy dokážu prostredníctvom dátového alebo úlohového paralelizmu rozdeliť výpočtovo náročný problém na menšie časti (podproblémy), ktoré môžu byť realizované v rovnakom čase na viacerých procesoroch viacprocesorových strojov alebo na viacerých jadrách viacjadrových procesorov. (V tomto ponímaní abstrahujeme od pseudoparalelizmu a namiesto neho pracujeme so skutočným paralelizmom.)

Praktický príklad približuje situáciu, kedy je program v spomenutej algoritmickej skladbe spustený na počítači osadenom dvojjadrovým procesorom. Z výpočtu vyplýva, že maximálny teoretický nárast výkonnosti bude 33 %, čo znamená, že po paralelnej optimalizácii bude program pracovať o tretinu rýchlejšie ako predtým.

```
class Program
{
    delegate float Del(float s, float p, int n);
    static void Main(string[] args)
    {
        Del d = (s, p, n) => 1 / (s + p / n);
        float nárastRýchlosti = d(0.5f, 0.5f, 2);
        Console.WriteLine("Nárast rýchlosti paralelizovanej " +
            "verzie programu je {0} %.",
            ((nárastRýchlosti - 1) * 100).ToString("0.00"));
        Console.Read();
    }
}
```

2.16 λ-príkazy v jazyku C# 3.0

V programovacom jazyku C# 3.0 je možné pracovať aj s tzv. λ-príkazmi, o ktorých hovoríme vtedy, keď telo λ-výrazu tvorí konečná a neprázdna množina programových príkazov. Generický vzťah pre definíciu λ-príkazu má nasledujúcu podobu:

(formálne parametre) =>

{
  príkaz$_1$;
  príkaz$_2$;
  ...
  príkaz$_n$;
}

λ-príkaz nachádza uplatnenie napríklad pri paralelnej inicializácii štvorcovej matice celočíselnými hodnotami:

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;


namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            int[,] pole = new int[10000, 10000];
            Stopwatch sw = new Stopwatch();
            sw.Start();
            Console.WriteLine("Inicializácia matice bola zahájená.");
            // Konštrukcia Parallel.For využíva λ-príkaz.
            Parallel.For(0, 10000, i =>
            {
                for (int j = 0; j < 10000; j++)
                {
                    pole[i,j] = 2 * i + j;
                }
            });
            sw.Stop();
            Console.WriteLine("Matica je inicializovaná.");
            Console.WriteLine("Exekučný čas: {0} s.",
                sw.Elapsed.TotalSeconds);
           Console.Read();
        }
    }
}
```

Tento program využíva riadenú knižnicu Parallel Extensions (známu tiež ako Parallel FX) spoločnosti Microsoft na podporu paralelného spracovania programových inštrukcií. Aby sme mohli využívať triedy pre podporu paralelizácie, musíme vykonať dve akcie:

1. Do aktuálne otvoreného projektu jazyka C# 3.0 vložíme odkaz na zostavenie knižnice Parallel Extensions. To vykonáme takto:

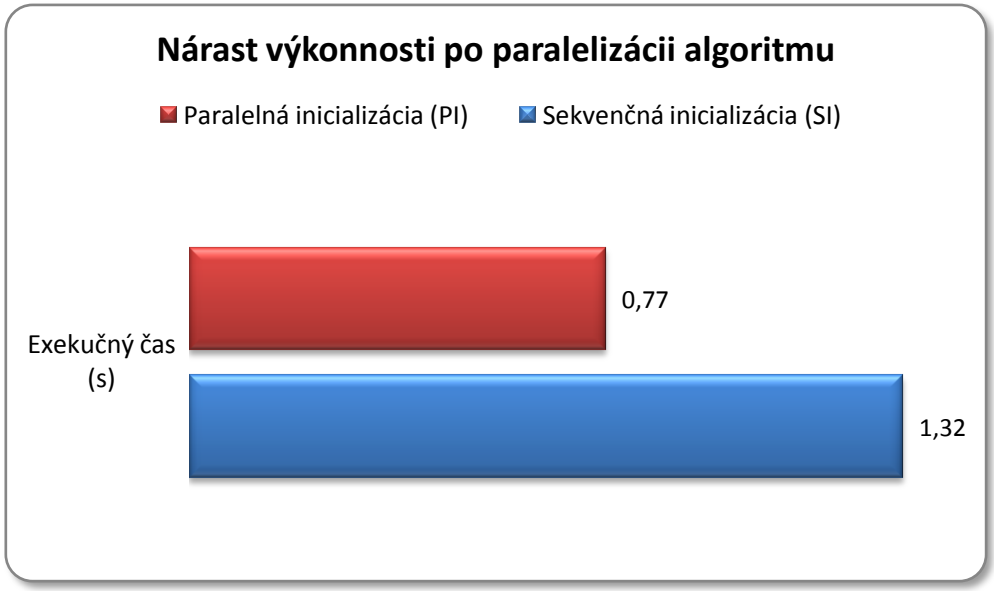
- Z ponuky **Project** vyberieme príkaz **Add Reference**.
- Keď sa objaví dialógové okno **Add Reference**, uistíme sa, že je vybratá záložka **.NET**.
- V zozname vyhľadáme zostavenie **System.Threading**, ktoré zodpovedá knižnici Parallel Extensions.
- Po kliknutí na tlačidlo **OK** sa odkaz na knižnicu Parallel Extensions vloží do projektu jazyka C# 3.0.

2. Do zdrojového súboru vložíme príkaz **using** a zabezpečíme import menného priestoru **System.Threading**.

Keďže máme za úlohu inicializovať štvorcovú maticu typu 10000 x 10000, budeme potrebovať 2 programové cykly. Pre potreby tohto praktického experimentu sme sa rozhodli paralelizovať nadradený cyklus. Namiesto toho, aby sme použili štandardný cyklus **for**, aplikujeme jednu z množiny preťažených definícií statickej metódy **For** triedy **Parallel**. Tá predstavuje cyklus **for** s možnosťou paralelizácie jeho iterácií.

λ-príkaz je umiestnený v signatúre statickej metódy **Parallel.For**, ktorá implementuje techniku imperatívneho dátového paralelizmu.

Praktické experimenty na počítači osadenom 2-jadrovým procesorom AMD Turion 64 X2 a 2 GB operačnej pamäte ukázali, že paralelizácia znižuje exekučný čas potrebný na spracovanie inicializačnej časti algoritmu o približne 42 % (obr. 36).



Obr. 36: Vizualizácia nárastu výkonnosti po paralelizácii pôvodne sekvenčného inicializačného algoritmu

Záver

Vážení vývojári, programátori a softvéroví experti,

ďakujeme vám za váš záujem o túto knihu. Pevne veríme, že sa jej poradilo splniť vytýčený cieľ a že ste s ňou boli spokojní. Keďže ako vývojári moderného počítačového softvéru sa vzdelávame celý život, dovoľujeme si pripojiť kolekciu odkazov na ďalšie hodnotné informačné zdroje:

1. Visual C# Developer Center → http://msdn.microsoft.com/en-us/vcsharp/default.aspx.
2. C# 3.0 Language Specification → http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc.
3. Blogs from the C# Team → http://msdn.microsoft.com/en-us/vcsharp/aa336719.aspx.
4. C# Samples for Visual Studio 2008 → http://code.msdn.microsoft.com/csharpsamples.
5. Visual C# Technical Articles → http://msdn.microsoft.com/en-us/vcsharp/bb466181.aspx.
6. How Do I Videos – Visual C# → http://msdn.microsoft.com/en-us/vcsharp/bb798022.aspx.
7. Visual Studio 2008 and .NET Framework 3.5 Training Kit → http://www.microsoft.com/DOWNLOADS/details.aspx?familyid=8BDAA836-0BBA-4393-94DB-6C3C4A0C98A1&displaylang=en.
8. MSDN Forums – Visual C# → http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=9&SiteID=1.
9. Webcasty v českom a slovenskom jazyku → http://www.microsoft.com/cze/msdn/webcasts/default.mspx.
10. Praktické cvičenia v slovenskom jazyku → http://www.microsoft.com/slovakia/msdn/hols/default.mspx.
11. MSDN Magazine → http://msdn.microsoft.com/en-us/magazine/default.aspx.
12. C# Corner → http://www.c-sharpcorner.com/.

Prajeme vám veľa úspechov pri vytváraní tých najlepších počítačových aplikácií v jazyku C# 3.0!

Autor

O autorovi

Ing. Ján Hanák, Microsoft MVP, vyštudoval Ekonomickú univerzitu v Bratislave. Tu, na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky (KAI FHI), pracuje ako vysokoškolský pedagóg.

Prednáša a vedie semináre týkajúce sa programovania a vývoja počítačového softvéru v programovacích jazykoch C, C++ a C#. Okrem spomenutej trojice jazykov patrí k jeho obľúbeným programovacím prostriedkom tiež Visual Basic a C++/CLI. Aktívne vynachádza nové postupy tvorby softvéru, ktorý bude pomáhať nielen študentom, ale aj širokej verejnosti.

Je nadšeným autorom odbornej počítačovej literatúry. V jeho portfóliu môžete nájsť nasledujúce knižné tituly:

1. **Inovácie v jazyku Visual Basic 2008**. Praha: Microsoft, 2008.
2. **Visual Basic 2008: Grafické transformácie a ich optimalizácie**. Bratislava: Microsoft Slovakia, 2008.
3. **Programovanie B – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C++)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
4. **Programovanie A – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
5. **Expanzívne šablóny: Príručka pre tvorbu "code snippets" pre Visual Studio**. Bratislava: Microsoft Slovakia, 2008.
6. **Kryptografia: Príručka pre praktické odskúšanie symetrického šifrovania v .NET Framework-u**. Bratislava: Microsoft Slovakia, 2007.
7. **Príručka pre praktické odskúšanie vývoja nad Windows Mobile 6.0**. Bratislava: Microsoft Slovakia, 2007.
8. **Príručka pre praktické odskúšanie vývoja nad DirectX**. Bratislava: Microsoft Slovakia, 2007.
9. **Príručka pre praktické odskúšanie automatizácie aplikácií Microsoft Office 2007**. Bratislava: Microsoft Slovakia, 2007.
10. **Visual Basic 2005 pro pokročilé**. Brno: Zoner Press, 2006.
11. **C# - praktické příklady**. Praha: Grada Publishing, 2006 (kniha získala ocenenie „Najúspešnejšia novinka vydavateľstva Grada v oblasti programovania za rok 2006“).
12. **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Praha: Microsoft, 2006.
13. **Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005**. Praha: Microsoft, 2005.

14. **Visual Basic .NET 2003 – Začíname programovat**. Praha: Grada Publishing, 2004.

V rokoch 2006, 2007 a 2008 bol jeho prínos vývojárskym komunitám ocenený celosvetovými vývojárskymi titulmi **Microsoft Most Valuable Professional (MVP)** s kompetenciou **Visual Developer – Visual C++**.

Kontakt s vývojármi a programátormi udržiava najmä prostredníctvom technických seminárov a odborných konferencií, na ktorých vystupuje. Za všetky vyberáme tieto:

- Konferencia **Software Developer 2007**, príspevok na tému „Představení produktu Visual C++ 2005 a jazyka C++/CLI“. Praha 19. 6. 2007.

- Technický seminár **Novinky vo Visual C++ 2005**. Microsoft Slovakia. Bratislava 3. 10. 2006.

- Technický seminár **Visual Basic 2005 a jeho cesta k Windows Vista**. Microsoft Slovakia. Bratislava 27. 4. 2006.

V súčasnosti pôsobí ako stály spolupracovník počítačových časopisov PC World a Connect!. V minulosti zastával pozíciu odborného redaktora rovnako v magazínoch Software Developer, ComputerWorld, Infoware, PC Revue a Chip.

Dovedna publikoval viac ako 250 odborných a populárnych prác venovaných vývoju počítačového softvéru.

Ak sa chcete s autorom spojiť, môžete využiť jeho adresu elektronickej pošty: hanja@stonline.sk.

Akademický blog autora môžete sledovať na adrese: http://blog.aspnet.sk/hanja/.

Použitá literatúra

1. Akhter, S., Roberts, J.: Multi-core Programming. Hillsboro: Intel Press, 2006.
2. Hanák, J.: Inovácie v jazyku Visual Basic 2008. Praha: Microsoft, 2008.
3. Hanák, J.: Visual Basic 2005 pro pokročilé. Brno: Zoner Press, 2006.
4. Hanák, J.: Programujeme v jazycích C++ s Managed Extensions a C++/CLI. Praha: Microsoft, 2006.
5. Hanák, J.: C# - praktické příklady. Praha: Grada Publishing, 2006.
6. Hanák, J.: VB 6.0 → VB 2005: Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005. Praha: Microsoft, 2005.
7. Hanák, J.: Visual Basic .NET 2003 – Začínáme programovat. Praha: Grada Publishing, 2004.
8. Kraval, I.: Základy objektově orientovaného programování za pomoci jazyka Microsoft Visual Basic 5.0. Brno: Computer Press, 1998.
9. Kraval, I., Ivachiv, P.: Základy komponentní technologie COM. Brno: Computer Press, 1998.
10. Vaníček, J., Papík, M., Pergl, R., Vaníček, T.: Teoretické základy informatiky. Praha: Kernberg Publishing, 2007.

Ján Hanák

Objektovo orientované programovanie v jazyku C# 3.0
(Príručka pre vývojárov, programátorov a softvérových expertov)

Recenzenti:	Ing. Jiří Burian, Mgr. Miroslav Kubovčík
Vydanie:	prvé
Rok prvého vydania:	2008
Náklad:	150 ks
Jazyková korektúra:	Ing. Peter Kubica
Vydal:	Artax a.s., Žabovřeská 16, 616 00 Brno pre Microsoft s.r.o., Vyskočilova 1461/2a, 140 00 Praha 4
Tlač:	Artax a.s., Žabovřeská 16, 616 00 Brno

ISBN: 978-80-87017-02-9

Ing. Ján Hanák je Microsoft MVP (Most Valuable Professional – najcennejší odborník) s kompetenciou Visual Developer – Visual C++. Je autorom 15 odborných kníh, príručiek a praktických cvičení o programovaní a vývoji počítačového softvéru. Pracuje ako vysokoškolský pedagóg na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky Ekonomickej univerzity v Bratislave. Prednáša a vedie semináre z programovania v jazykoch C, C++ a C#. V rámci svojej vedeckej činnosti sa zaoberá problematikou štruktúrovaného, objektovo orientovaného, komponentového, funkcionálneho a paralelného programovania.

ISBN: 978-80-87017-02-9