Komentované zásady návrhu objektově orientovaného programového systému

Jan M Honzík

1 Úvod

Po řadě článků a publikací popisujících a propagujících objektově orientované vlastnosti různých programovacích jazyků se začínají objevovat také informace, podrobně a kriticky hodnotící proces použití těchto nástrojů při výstavbě rozsáhlejšího programového systému. Jádrem příspěvku jsou zásady uvedené v [1] , komentované stručně na základě zkušenosti ze studia nejnovějších literárních zdrojů v oblasti objektově orientovaného návrhu.

Technologie OOP slibovala, že problémy softwarového vývoje se stanou minulostí. Výsledkem OOP by měly být automaticky zvýšená produktivita programátorů a znovupoužitelné části programu. S takovými představami začal tým Billa Hunta vyvíjet rozsáhlý programový systém HP VEE (Visual Engineering Environment). Následující poznámky, glosy, rady a návody jsou výsledkem kritického zpětného hodnocení procesu vývoje celého systému. Diskuse s tvůrci programového systému Control Panel, InView aj. (Moravské přístroje-Alcor Zlín), jejichž systémy dosahují rozměrů mnoha set tisíců zdrojových řádků, vedla k plnému souhlasu s uvedenými závěry. V souhrnu lze vyjádřit závěr, že OOP představuje velmi užitečný nástroj v prostředí nového technologického přístupu k řešení problému. Některé sliby a očekávané vlastnosti je však třeba přijímat s jistou rezervovaností.

Nejdůležitějším poznatkem je, že výsledky OOP nepřicházejí automaticky. Jsou výsledkem pečlivého návrhu a implementačních zkušeností, které dovedou maximálně využít objektově-orientovaný přístup. Výsledný produkt se pak snadno udržuje a zlepšuje.

2 Slibované příspěvky

Tvůrci nové metodiky slibují mnoho nových vlastností OOP a mnohé z nich lze opravdu využít. Na rozdíl od předpokladů, však tyto nové vlastnosti nejsou dosaženy vždy pouhým použitím jazyka pro OOP.

Jde především o tyto vlastnosti:

- OOP slibuje znovupoužitelné programy, protože s použitím OOJ je téměř nemožné, nebo jen velmi těžké napsat kód, který by nebyl znovupoužitelný. Každý řádek, který se jednou vytvoří bude použitelný v příštích projektech. Z toho vyplývá, že zakrátko nebudeme psát žádný software. Budeme jen dávat dříve napsané moduly dohromady.

- OOP slibuje redukci údržby hotového programu. Seskupením operací a dat ve třídě a omezením přístupu k implementačním detailům se vytváří program nezávislý na implementačních detailech jiného programu. Takové zapouzdření programu umožňuje rozsáhlé změny v jedné třídě bez vlivu na třídu jinou.

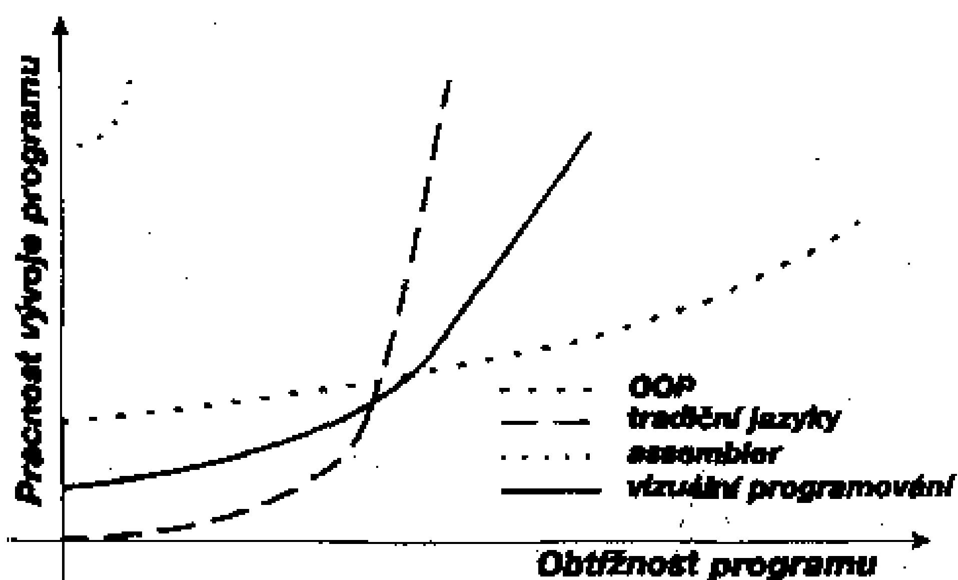
• OOP slibuje zlepšení produktivity programátora. Jednou z příčin je skutečnost, že třídy jsou vzájemně nezávislé. Produktivita roste také proto, že v důsledku dědičnosti programátor nemusí psát mnoho úseků znovu.

Jedním z nejdůležitějších poznatků, ke kterému dojde nejeden řešitel je skutečnost, že **znovunepoužitelný** program lze napsat v kterémkoli jazyce. Vyspělost většiny jazyků není zárukou, že ho nevyspělý programátor bude využívat vhodným a efektivním způsobem. Převzít úsek programu, který je funkční a rychlý, ale je napsán chaotickým způsobem, který může ukrývat řadu chyb, může vést k náročnější údržbě, než v případě jeho nového vytvoření. Náročnost údržby, jejíž snížení slibuje OOP, je dáno především použitím dobrých metod návrhu. Nepříznivý dopad špatného návrhu s použitím OOP je v praxi velmi častým jevem. Mnohem významnější než znovupoužití je však proces znovuvytvoření programu, které je s použitím OOP mnohem snadnější než s tradičními jazyky.

Potvrzuje se skutečnost, že knihovny tříd většinou fungují, jak měly. Rozšiřování a vylepšování tříd s použitím dědičnosti je snadná a užitečná metoda, která pomáhá při dělbě práce a nepůsobí problémy v jiných částech projektu. Je však skutečností, že je snadné napsat pomalu běžící objektově orientovaný program. Problém rychlosti se vždy vplíží do systému, který se snaží sdílet příliš hodně programových úseků s příliš mnoha různými operacemi. OOP nejen dovoluje, ale přímo podporuje takové praktiky prostřednictvím vlastností jako je dědičnost a předávání zpráv.

3 Úspěšné OOP

Tvůrci programů mohou učinit několik kroků, kterými se software učiní znovupoužitelný, udržovatelný a přenositelný. Znamená to ale nějakou námahu navíc. Největší potíže jsou s rozsáhlým a špatně navrženým programem. Bez ohledu na použitý programovací jazyk existuje určitá velikost programu, do jejíhož dosažení lze program rozumně udržovat. Tím se nechce říci, že nelze vytvořit větší programy; stojí to ale mnohem více úsilí.



Zdá se, že pro jazyk C je tato mez někde mezi 500 a 5000 procedurami. Není stanoven limit pro OOJ, ale je někde mezi 1000 a 10000 tříd. Jedním z faktorů, který se na tom podílí je i konflikt identifikátorů (pro globální proměnné, třídy a konstanty), který u rozsáhlého systému může vést až k zmatení mezi jednotlivými členy řešitelského týmu.

Srovnání programovacích prostředků ukazuje přibližný graf na obr.1. Pro rozsáhlé programy je nejvhodnější OOP. Ačkoliv začátek je velmi pracný a obtížný, při dosažení určité velikosti již není třeba vyvíjet tak velké úsilí k dalšímu rozšiřování.

Na obr.1. jsou hranice vyznačeny v souvislosti s různými vývojovými prostředky. Osa x představuje hranici obtížnosti nebo velikosti programu, osa y představuje potřebné programátorské úsilí. Pro ideální situaci by byl vztah, který je representován úhlopříčkou pravoúhelníku vymezeného Tato situace se na každém grafu najde jen v určitém úseku. Vizuální programování je výhodné pro nerozsáhlé problémy, ale použití tohoto prostředku pro výstavbu rozsáhlejšího programu vede k dramatickému vzrůstu potřebné práce. Použití tradičních jazyků pro malé problémy nemusí být účinné, ale tyto jazyky jsou velmi vhodné pro středně velké problémy. Zvládnutí naučení se a použití OOP je obtížnější, než u tradičního programování. Vyplatí se však, jakmile program překročí určitou velikost. Pro srovnání, asembler se také zvládá obtížně, ale inkrementální úsilí, které je třeba vynaložit, když program vzroste, je obrovské. Asembler je opravdu vhodný jen pro doladění určité vlastnosti, nebo pro vytvoření procedury či funkce, kterou nelze dobře udělat jinak.

Následující zásady jsou shrnutím zkušenosti po dokončení velkého projektu. Mohou pomoci nejen při využití OOP, ale i při programování s jinými prostředky.

4 Redukce složitosti

Složitost je pravděpodobně největším nepřítelem tvorby dobrých programů. Jestliže si i členové tvůrčího kolektivu musí pamatovat příliš mnoho informací, stane se výsledný systém nestabilním, protože jedna osoba neovládne mnoho detailů při vytváření dobré práce.

Největší předností OOP je možnost snižování složitosti s využitím "zapouzdření" (uzavírání, obalení - encapsulation). 4.1 Rozdělení velkých systémů na malé části (Divide et impera) když nic víc, přináší to snížení velikosti systému, kterému lze porozumět vcelku, než se do něj zasahuje při změně. Podle [2] je základním posláním OOP snižování složitosti

4.1 Snižování počtu globálních jmen a funkcí

Globální proměnné jsou vždy nebezpečné, protože zapomenout na zodpovědnost za ně a zapomenout na jejich význam je tak snadné. Globální funkce nejčastěji představují nežádané závislosti.

4.2 Ukrývání implementačních detailů

Vytvořený systém, podobně jako mnoho OO systémů, představuje síť křížově provázaných objektů. Ukázalo se, že požadavek na objekt následující za následujícím objektem je nebezpečný, protože takový požadavek předpokládá existenci takového objektu. To už

vyžaduje vědět více o celé struktuře, než je absolutně nezbytné a k přepisování a dodělávání dalších funkcí, když se struktura změní.

4.3 Odstranění závislosti na pořadí

Představte si systém, v němž vzájemná výměna dvou řádků zdánlivě nezávislých volání zpráv způsobí chybu. Tento případ může nastat mnohokrát. Jako příklad lze uvést třídu textového okna, která přijímají zprávy pro nastavení fontu a velikosti okna. Zpráva o nastavení fontu však implicitně nastavují rozměry okna, aby se tam text s novými fonty vešel. Záměna pořadí zaslání zprávy o fontech a o velikosti okna vedou k různým velikostem okna.

4.4 Omezování všech závislostí

Programové moduly systémů s mnoha závislostmi lze jen ztěží znovuvyužít. I v objektově orientovaném přístupu se zdůrazňují staré známé pojmy technologií se strukturovaným přístupem jako je potřebná vysoká míra soudržnosti (cohesion) a nízká míra spřaženost (coupling).

4.5 Snižování organizačních závislostí

Organizační závislosti se ovládají nejsnadněji. Týkají se pouze struktury vytvářeného systému. Jejich redukcí se výrazně usnadní extrakce částí velkého systému pro nové použití

4.6 Omezování externích odkazů

Všechny externí odkazy ve dané třídě způsobí obtíže při pokusu vyjmout třídu z okolního prostředí. Jestliže např. daná třída používá tři další třídy, z nichž každá používá tři jiné třídy, pak bude obtížné použít jen malý úsek z této části programu znova.

4.7 Rozvážné používání dědičnosti

Dědičnost dovoluje vytvářet třídy prostřednictvím "programování na základě rozdílu". Novou třídu lze vytvořit přidáním nebo změnou chování existující třídy. Je-li zapotřebí např. třída zásobník a existuje již třída fronta, je možné implementovat zásobník s využitím dědičnosti od třídy. Zásobník je fronta s jedním koncem. Pak se tedy popíše, čím se liší zásobník od fronty. Zásobník potřebuje např. operace Push a Pop místo AddFirst, RemoveFirst, AddLast a RemoveLast. S použitím dědičnosti mohou tyto dvě třídy automaticky sdílet některé administrativní úkoly, jako je např. dynamické přidělování paměti.

Dědičnost je přínosem i obtíží. Při odpovídajícím využití přináší mimořádnou výrazovou sílu. Dědičnost může zlepšit programátorovu produktivitu a zvýšit spolehlivost programu, protože třídy snadněji sdílejí úseky programu. Protože však dědičnost kombinuje všechny operace definované ve třídě s operacemi definovanými u tříd předků, může to vnášet nenápadné a nežádoucí závislosti.

Jestliže třída zásobník zdědí všechny operace nad frontou, může uživatel záměrně nebo omylem použít operace pro frontu nad zásobníkem. Na první pohled se to nezdá být velký problém. Může to však způsobit dodatečnou práci, když se má přepsat třída zásobník tak

aby se zamezilo dědičnosti od fronty. V tom případě se buď musí dodat frontě operace, které se v programu používají pro práci s frontou, nebo přepsat program, který zásobník používá tak, aby používal pouze zásobníkové operace.

Alternativním řešením pro třídu zásobník může být, že zásobník je raději instancí fronty než jejím dědicem. Zásobník pak může implementovat operace push a pop zasláním zprávy frontě. Tento přístup vede k lepší organizaci, protože uživatel fronty používá frontu přímo a uživatel zásobníku je omezen na zásobníkové operace. Tato organizace je jednoduchá a vyžaduje jen malé programovací úsilí.

Je třeba se rozhodnout se rozhodnout zda "zdědit nebo koupit". Je to otázka, zda třída B má být dědicem nebo klientem jiné třídy A. Jednoduchou pomůckou může být úvaha, že dědictví znamená "bytí", zatímco klientství znamená "vlastnictví". Dědický vztah je správný, jestliže na každou instanci třídy B můžeme nahlížet jako by to byla také instance třídy A. Klientský vztah je správný, když instance B vlastní jeden nebo více atributů třídy A. Klasickou ukázkou chyby je deklarovat dům, jako předka okna a podlahy. Ani na okno, ani na podlahu nelze nahlížet jako na instanci domu, ale jen jako na jeho komponenty.

Dědičnost má za výsledek jednodušší a účinnější přístup k vlastnostem třídy předka. Díky redefinici lze použít operací rodičů bez nutnosti zachovat jejich implementaci. I typy zděděných entit lze předefinovat. Takové vlastnosti nemají přístup ke službě prostřednictvím rozhraní.

Na druhé straně je dědičnost mnohem více zavazující než "nákup služby". Použije-li se postävení klienta, vidíme třídu A jen prostřednictvím jeho rozhraní a jsme ochráněni před eventuálními změnami v jeho implementaci. Dědictví zpřístupňuje implementaci; dává tím více moci a méně ochrany.

4.8 Ostražité použití vícenásobné dědičnosti

Vícenásobná dědičnost dovoluje implementovat novou třídu s použitím více než jedné třídy předka. Bez ohledu na obhájce vícenásobné dědičnosti lze říci, že vícenásobná dědičnost může být nebezpečná a to z následujícího pohledu: Zmnohanásobuje nevýhody dědičnosti bez skutečného zjednodušení architektury programu. Bill Hunt uvádí, že v systému HP VEE, který řešili, bylo několik situací, které lákaly k využití mnohanásobné dědičnosti. Když se však provedla detailní analýza, vždy se dospělo k alternativnímu řešení, které bylo jednodušší z pohledu implementace a které bylo méně závislé.

Vždy, když obhájci dědičnosti demonstrují význam vícenásobné dědičnosti, používají příklad z každodenního života, který zdánlivě dokazují její správnost. Jako příklad lze vzít řiditelnou kolovou sekačku trávy, která je jak motorovým vozidlem, tak nástrojem na sekání trávy. Protože ne všechny nástroje na sekání trávy jsou motorizované a pohyblivé, nemůže třída "nástrojů na sekání trávy" zdědit vlastnosti třídy "motorová vozidla".

Protože ale řiditelná kolová motorová sekačka je jak motorové vozidlo, tak nástroj na sekání trávy, musí dědit vlastnosti obou tříd. Tento argument má svou váhu, může ale vést k následným komplikacím. Nejdůležitější je uvědomit si, že vícenásobná dědičnost není nikdy nutná; zdá se být jen pohodlná. Podíváte-li se na motorová vozidla a na sekací nástroje

jako na atributy řiditelné kolové sekačky trávy, pak jednoduchou a vhodnější alternativou implementace řiditelné kolové motorové sekačky je použití instance třídy "motorové vozidlo" a třídy "sekací nástroje" než vytvoření nové třídy, dědící vlastnosti obou tříd.

Jednou z největších předností OOP je snadnost přepsání nebo náhrady modulu bez ovlivnění okolních modulů systému. Dědičnost, a zejména vícenásobná dědičnost, má tendenci působit proti této přednosti, protože dědic dědí i řadu zděděných vlastností, které nejsou nezbytně potřebné. Tyto vlastnosti zvyšují množství práce potřebné pro přepsání modulu buď z toho důvodu, že je třeba přepsat více operací v novém modulu, nebo je třeba upravit program uživatele modulu, aby nemohl využívat některé nežádoucí operace.

4.9 Nepoužívání přímého přístupu k proměnné

Seskupením dat (proměnných) a procedur (metod) do objektu se ukrývají implementační detaily před zbytkem systému. Jedním z důsledků je skutečnost, že třídu lze zcela přepsat a změnit, aniž by byly nutné změny ostatních částí systému. V některých situacích je však toto zapouzdření (encapsulation) poněkud nešikovné, nebo vede k snížení rychlosti programu. Protože většina jazyků umožňuje přímý přístup k proměnným, je programátor v častém pokušení obejít zapouzdření. Tomuto pokušení je vhodné se vyhnout, pokud opravdu není jiné řešení. Přepis třídy, která obsahuje proměnnou, při čemž se změní její význam této proměnné, zavádí do systému možnost závažné chyby.

4.10 Vhodná volba jmen a identifikátorů

Tuto prostou zásadu nelze podcenit a stojí za to o ní hovořit velmi vážně. Správná volba jména proměnné, zprávy nebo funkce může plně popisovat smysl pojmenované entity.

4.11 Potřeba včasné tvorby komentářů

Není-li autor schopen úsek programu okomentovat jednoduchou větou, pak nikdo jiný nebude schopen pochopit co dělá, aniž by důkladně studoval manuál popisující program (pokud něco podobného vůbec existuje).

4.12 Omezení vedlejších jevů

Prověřte důkladně, že každá metoda provádí správně právě jednu funkci. Vedlejší jevy mohou být při znovupoužití velmi zrádné.

5 Závislosti na dané úrovni

Závislosti počítačových systémů rozhodují o jejich portabilitě. Skutečnost, že program využívá určitých vlastností operačního systému jako je multitasking nebo použití specializovaných knihoven, může způsobit potíže při pokusu o převedení programu do jiného systému.

5.1 Oddělení závislosti

Všechny užitečné programy mají nějaké závislosti na počítači. Tyto závislosti lze minimalizovat efektivním použitím modulů pro rozhraní (interface module). Jestliže určitý program závisí na určitém systému oken, lze portabilitu zlepšit omezením přístupu k systému na přístup přes třídu ovladače systému oken. Když se použije jiný systém oken, stačí přepsat jen třídu ovladače.

5.2 Tvorba programu s dobrou časovou složitostí

Vytváříte-li program na vysoce výkonném počítači, aniž sledujete jeho časovou složitost, můžete narazit na potíže s neadekvátním chováním programu na pomalejším a lacinějším počítači uživatele. Špatné časové vlastnosti programu vznikají nejčastěji na základě zásadních architektonických rozhodnutí, které neberou v úvahu výkonnostní možnosti a požadavky. V takovém případě nepomůže dodatečné "ladění" programu za účelem získání větší rychlosti, ale většinou je nutná pouze zásadní změna a přepsání velké části hotového projektu. Je-li časová výkonnost důležitým požadavkem projektu, je třeba architektonická rozhodnutí dělat v její prospěch. Výsledkem bude nejen program, který je uspokojivě rychlý na levném počítači, ale který je vynikající při chodu na výkonném počítači.

5.3 Neplýtvejme pamětí

Podobně jako čas i paměťový prostor je zdroj, který musí být dostupný v přiměřené velikosti. Tam kde se nešetří využitím paměti, tam může dojít k situaci, kdy budou potíže s programem na počítači s omezenou kapacitou paměti. Využití virtuální paměti způsobuje potíže s rychlostí nebo i s nemožností provedení programu na počítačích s omezenou pamětí.

5.4 Program nemá využívat znalosti konkrétní velikosti a reprezentace proměnných nebo pořadí ukládání parametrů do zásobníku

Je snadné na to zapomenout, ale počítače systémů 80x86 ukládají datové elementy v jiném pořadí, než jiné systémy. Za normálních okolností nejsou detaily souvisejícím s reprezentací dat v paměti něčím, čím by se měl programátor zabývat, pokud ovšem program nepřistupuje k datům, jako k sekvenci bytů. Je vhodné si uvědomit, že řada systémů pracujících se zásobníkem implementovaným sekvencí po sobě jdoucích paměťových míst, může postupovat při vkládání odlišným směrem, než se očekává.

5.5 Je vhodné přenést aplikaci do dvou jiných prostředí

O programu nelze hovořit, že je portabilní, neběží-li alespoň na třech různých počítačových systémech. To, že program běží jen na dvou počítačích, není postačující, protože snadno může dojít k přehlédnutí a k neuplatnění porušení některé podmínky portability.

6 Závislosti na vývojovém prostředí

Závislost na vývojovém prostředí může mít závažné důsledky, předpokládáme-li přesun software na jiný počítačový systém a tam pokračovat ve vývoji programu. Chceme-li přenést program na jiný systém ale nepokračovat v jeho vývoji, pak prostředky automatického překladu mohou být pro svůj úkol vhodné.

6.1 Volba dostupného jazyka

Ačkoliv se někteří lidé domnívají, že volba jazyka je izolovaný a technický problém, má řídit také dostupností jazyka a jeho popularitou. Nedoporučuje se volba systému který má mocné jazykové prostředky a velmi zajímavé vývojové prostředí, ale je dostupný na malém počtu systémů.

6.2 Využívání hlavních vlastností jazyka

Velmi často nabízejí okrajové a speciální vlastnosti programovacího jazyka přesně to, co potřebujeme jen ve speciální situaci. Navíc, málo užívané vlastnosti jazyka jsou nejméně otestované. Jestliže Vás baví objevovat vady kompilátorů zkoušením různých obskurních ale legálních konstrukcí, pak je vše v pořádku. Ostatním se doporučuje držet se vlastností, o nichž lze s úspěchem předpokládat, že budou za všech okolností fungovat.

6.3 Omezování použití direktiv kompilátorů a makroinstrukcí

Toto pravidlo je zvlášť důležité, když máme v úmyslu importovat program z jiného prostředí ve zdrojové podobě. Stalo se, že se v projektu velmi rozsáhle využívalo definovaného typu nazvaného BOOL, což byla zkratka pro typ se dvěma stavy. Pak se pomocí include vložil soubor s poněkud odlišnou definicí téhož BOOL, a bylo hodně práce s tím, jak přesvědčit kompilátor, že v případě těchto dvou konfliktních definicí jde o totéž.

6.4 Nepoužívání speciálních nástrojů

Je vhodné si vybírat jen ty nejnutnější nástroje. Jestliže se použije vyspělý nástroj, pak se může stát, že při přenosu do systému, který podobné nástroje nemá může být řada potíží. Jako vždy při tvorbě programového vybavení platí: klíč k úspěchu je v jednoduchosti!

7 Alespoň jedna špičková vlastnost programu

Jestliže píšete programové dílo, které umí dělat všechno, pak to nejspíše skončí tak, že program dělá sice vše, ale chabě. Je dobré zaměřit pozornost na to, aby váš program dělal alespoň jednu věc opravdu dobře. To pak může vytvořit programu jeho jméno na trhu.

7.1 Hlavní pozornost hlavním požadavkům

Často se stává, že programátor, který tvoří úseky programu, o nichž předpokládá znovuvyužití, se snaží přehnaně předvídat jejich budoucí použití a přitom zanedbá jejich hlavní funkci.

7.2 Vyřeš současný problém úplně

Jestliže nevyřešíte aktuální problém úplně ihned jak vznikne, bude ho muset asi jednou modifikovat uživatel.

7.3 Návrh programu z pohledu uživatele

Je třeba řešit problémy na základě požadavku uživatele, je ale dobré skrýt před ním implementační detaily.

7.4 Použití nejjednoduššího přijatelného algoritmu

Pro vyhledávání v sekvenci je nejednodušší algoritmus lineárního vyhledávání. Složitější algoritmus se vyplatí jen pro dlouhý seznam.

8 Hledání rychlosti

Zákazník má rád systémy s krátkou odezvou a programátoři jim při tvorbě programů dávají přednost. Dobrá rychlost je ceněna a může kompenzovat některé neobratnosti nebo jiné nedoceněné vlastnosti vytvořeného systému.

8.1 Rychlost je podstatná

Vysoká rychlost programu bude vždy podstatný požadavkem každého programu. Bez ohledu na vlastnosti počítačového hardware, uživatel vždy požaduje účinný program. Bude to ještě dlouho trvat, než MIPSy ztratí svůj smysl. (MIPS = Mega Instructions Per Second).

8.2 Netvořme pomalý program

Programátoři nebudou znovupoužívat programy a jejich části, které jsou známy svou pomalostí. Pomalé programy, které běží normálně na rychlých počítačích, omezují typy počítačů, které jsou s danými programy použitelné.

8.3 Dobrý návrh neznamená pomalý program

Mnozí programátoři tvrdí, že požadavek dostatečné rychlosti programu se vylučuje s použitím metodiky OOP. Rychlost a dobrý návrh programu se vzájemně nevylučují. Ve skutečnosti je dobrá rychlost jedním z požadovaných výsledků dobrého návrhu. Vezměme v úvahu např. množství algoritmů, které se nabízejí pro řazení. Ve většině případů platí, že dobrý algoritmus je rychlejší, než sebelépe optimalizovaný slabý algoritmus.

8.4 Každá operace právě jednou

Toto pravidlo se obtížně dodržuje při práci na velkém systému. Stává se např., že jeden modul má mnoho různých použití a lze obtížně určit, zda jedno použití bude obsahovat

duplicity operací, zatím co v jiném bude něco chybět. Je např. velmi snadné zapomenout na aktualizaci obrazovky v určité situaci, zatím co jindy ji aktualizujeme zbytečně několikrát.

8.5 Zachování co největší jednoduchosti

Zlepšení rychlosti a chování jednoduchého systému je snazší. Nejčastější cestou ke zlepšení rychlosti je přepsání kritické a špatně se chovající části programu. Předělání zdánlivě jednoduché funkce pro ekvivalenci, tak aby byla kompaktní a neobsahovala žádná další volání, může v programu, který ji používá v nejzanořenějším cyklu, vést k výrazně rychlejšímu chodu.

8.6 Použití výkonných algoritmů

Je zde vždy rozdíl mezi účinností a jednoduchostí. Velmi jednoduché algoritmy se snadněji implementují a udržují. Výkonné algoritmy jsou většinou také složitější. Při výběru algoritmu je vhodné odhadnout rozměr a charakter celé vytvářené aplikace. Je mnoho programátorů, kteří pro vyhledávací tabulky bez velkého přemýšlení použijí mechanismu tabulek s rozptýlenými položkami (hashing-tables). Jestliže však tabulka nebude mít nikdy více než 20 položek, nebo přístup do tabulky je časově velmi řídký, je mnohem vhodnější prostý lineární algoritmus vyhledávání. Často se vyplatí začínat s nejprostšími algoritmy, jejichž výkon je pod průměrem. Za výkonnější se vymění až když se to ukáže nutným.

8.7 "Co můžeš odložit na zítřek, nedělej dnes" a nepočítej neaktuální záležitosti předčasně

Není-li něco třeba udělat právě teď, neztrácejme čas předčasným zpracováváním toho, co se možná použije později. Může se stát, že to nebude potřebné vůbec a čas, který byl výpočtu věnován, je ztracený. Čas ztrávený předčasným zpracováním nějakého úkonu, se koncovému uživateli bude jevit jako zbytečné zpoždění. Odloží-li se výpočet až do okamžiku, kdy jeho výsledek je opravdu potřebný, bude se uživateli jevit celkové zpoždění menší v důsledku jeho rozdělení do více úseků a celý program bude budit zdání větší rychlosti.

8.8 Nezasílejme zprávy a nevolejme funkce, nejsou-li absolutně nezbytné!

Důvodem není, jak by se zdálo, že zaslání zprávy či volání funkce je pomalé. Zprávy a volání funkce ale zkrývají či maskují kroky, které jsou výsledkem jejich provedení. Jakmile jednou objekt vyšle zprávu, neexistuje jednoduchý způsob předvídat, kolik dalších objektů vyšle zprávy a kolik kroků se provede, jako důsledek prvního zaslání zprávy. Také některé změny volaných metod, mohou dramaticky zpomalit volající metodu.

Optimalizovat nekompaktní úsek programu je velmi svízelné. Je-li nutné radikálně optimalizovat programový úsek, pak je nejlepší přepsat (možná i do asembleru) koncové metody, které samy nikoho nevolají.

8.9 Pozornost globální organizaci

Nejlepší chování programu dosáhneš na základě promyšlené komplexní organizace. Často výkon aplikace trpí v důsledku stovek zpráv, které se zasílají pro provedení jednoduchého úkonu.

8.10 Duplicita maličkostí nemusí být na škodu

Není třeba se obávat dovolit nějaké metodě, aby udělala sama něco jednoduchého, místo toho, aby volala kvůli této jednoduché, byť samostatné činnosti další metodu. Mnozí objektově-orientovaní tvůrci působí dojmem přesvědčení, že žádná metoda nesmí mít více než jeden řádek. Pak se snaží znovupoužívat kdejakou bezvýznamnou a malou metodou zasíláním zpráv sem a tam, místo, aby tuto maličkost zduplikovali na potřebném místě.

8.11 Pozor na nepoměr mezi časovou složitostí metod a jejich volání

Vždy je dobré se přesvědčit, že doba práce metody je větší, než čas spotřebovaný na její volání. V jiném případě se čas spotřebuje na skákání sem a tam, aniž se něco pořádného děje. Zdá se být užitečným vytvářet metody, jejichž volání spotřebuje celkově méně než 25% jejich celkového času zpracování.

9 Hodnocení výsledného produktu

V tomto odstavci jsou stručně shrnuty hodnotící postřehy vedoucího projektu HP VEE a autora [1].

"Hlavní přínos OOP lze využít pouze tehdy, jestliže se úzkostlivě dodržují zásady pečlivého návrhu a pečlivé implementace. V naší tvůrčí skupině jsme se o tom mnohokrát přesvědčili. Výsledný produkt HP VEE měl 350 tříd a více než 5000 metod. Tyto počty nezahrnují několik hlavních modulů, které jsme převzali z jiných systémů. Tradiční přístup by měl za výsledek možná dvojnásobek uvedených hodnot.

Nemohli jsme k vývoji HP VEE přistoupit s použití tradičních postupů, protože by to vyžadovalo příliš mnoho aktivních členů tvůrčího kolektivu. Výsledek by byl tak složitý, že by nebyl zvládnutelný. Protože jsme však navrhovali systém s využitím objektově-orientovaného přístupu s dobrou a disciplinovanou technologií návrhu, produkt se ukázal být spolehlivý a snadno rozšiřitelný bez projevů nestability, která je v podobných případech běžná."

10 Závěr

Uvedený příspěvek byl původně zpracován pro výuku studentů oboru Informatika a výpočetní technika na FEI VUT v Brně v kursu "Modulární a objektově orientované programování". Text je studentům dostupný na síťovém serveru, jako jeden z mnoha textových zdrojů informací, jež nejsou předmětem konkrétní přednášky, ale vytvářejí knihovnu statí vybraných k šířeji pojaté tematice kursu.

Většina školních projektů, ale i projektů výzkumné činnosti na vysoké škole nemá rozměry, ze kterých lze věrohodně vyvodit podobné závěry. O to cennější je zdroj pragmatických, ale systematicky uspořádaných zkušeností z projektu, jehož rozsah působí jako silná lupa zesilující účinek každé nevhodně zvolené Snad žádné z uvedených doporučení zkušeného programátora nepřekvapilo. Nechť si však každý sám sáhne do svědomí, kolik z nich vědomě nedodržuje.

11 Literatura

[1] Hunt,B.: Practical Object orieneted Programming:
38 Guidelines for making OOP work, EDN - Technology future, July 6, 1992 ,

[2] Booch,G.: Object Oriented Design With Applications,
The Benjamin/Cummings Publ.Comp.1991, ISBN 0-8053-0091-0

Doc.Ing.Jan M Honzík,CSc., UIVT FEI VUT v Brně
Božetěchova 2, 612 66 Brno 12
tel: 05-7275 242, fax: 05-4121 1141
email: honzik@dcse.fee.vutbr.cz