Ing. Jiří V O L Á K

INCOMA Gottwaldov

Úsporné programování

Cílem příspěvku je kromě kritiky některých živelných programovacích metod naznačení cesty k vytváření skutečně efektivních aplikačních programů s vynaložením minimální námahy. Zásady úsporného programování, byť jsou uvedeny poněkud striktní formou, je třeba chápat pouze jako doporučení, ověřená praxí a autor si nepřeje být za ně upálen na hranici. Snahou je spíše vzbudit diskusi než podat ucelený přehled nebo metodiku.

Širší souvislosti programování směrem k analýze či uživateli jsou pro omezený rozsah příspěvku pouze naznačeny, ač si jistě zaslouží pozorného rozboru.

Programátorské neřvary.

Prvním dosti rozšířeným neřvarem je tzv. bezhlavé programování. Termín tlačí a programátor se pouští do díla ve chvíli, kdy jeho informovanost o problému jemalá. Nedostatek informací nahrazuje fantazií a programuje možnosti, které s hlediska zralé úvahy nemohou nastat. V klokotu činnosti mu postupně unikají souvislosti a programátor se stále více zaplétá. Příznačná je obrovská spotřeba papíru všeho druhu, času i energie. Ladění vzniklého produktu je hotovým horrorem, o údržbě

nemluvě. Veškeré opravy je nutno provádět "flikováním", čímž se dostáváme k dalšímu oblíbeném hříchu.

Princip flikování spočívá v záměně příčiny a následku. Poněkud vulgárním příkladem je přičtení konstanty K k výsledku, který z neznámých důvodů vyšel právě o K menší. Flikování úzce souvisí s bezhlavým programováním a je do jisté míry jeho produktem. Je charakteristické pro poslední fázi ladění složitých nebo nepřehledných programů, kde najít skutečnou příčinu chyby se nepodařilo a trpělivosti i času valem ubývá. Programy jsou brzy záplatovány v několika vrstvách a každá další oprava přináší zajímavé překvapení.

Za zmínku stojí okolnost, že oba uvedené neřvary mají lavinovitý charakter. Prvních několik bezhlavých instrukcí spustí řetězovou reakce, která se prakticky nedá zastavit.

Výsledkem zmíněných hříchů je metoda "pokus-omyl", používaná při údržbě aplikačního software. Následky lze obvykle těžko dohlédnout a metodu "pokus-omyl" je možno doporučit nanejvýš pro badatelské práce v hlubinách operačního systému.

Instrukce pro strýčka Příhodu. Spočívá v programování pasáží, které "by mohly být někdy užitečné". Tento nešvar je dosti zakořeněn i u mnoha zkušených programátorů. Faktem je, že pravděpodobnost využití instrukcí pro strýčka Příhodu je nesrovnatelně malá a tyto jsou dobré pouze jako líheň chyb.

Za pozornost stojí ještě tzv. nekonečné testování. Program zhotovený kombinací předešlých metod je podroben řadě testovacích běhů, při nichž se programátor pracně prodírá od záhadných chyb k jejich příčinám. Ježto variant bývá obvykle několik, jsou postupně vyčerpány všechny bez valného efektu. Na stole se kupí informace z ladících běhů, jenže jakési stále chybějí ty pravé.

Výsledkem je záplava bezduchých instrukcí a programátor bezradně zírá na své dílo, které spíše než program připomíná následky uragánu. V této chvíli se dotyčný obrací na svého kolegu a rozvine se charakteristický dialog:

Programátor: "Proč mi ten program nechodí?"
Kolega : "Program nemá hlavu ani patu, je flikován a obsahuje instrukce pro strýčka Příhodu. Algoritmus je zbytečně složitý, program hýří skokovými instrukcemi a jako takový je zavrženíhodný."
Programátor: "Teď to přece nebudu celé předělávat, když mi to dalo tolik práce. Ostatně jsem se ptal na něco jiného." (Opakuje otázku, atd.)

Poučné je, že nejlepší rada - začít docela znova - není nikdy akceptována. Je mnoho programátorů, kteří nejsou ani po letech praxe ochotni připustit, že je třeba napřed přemýšlet a potom psát a ne naopak! Mnohem častější je dotaz "proč mi to nechodí" než "jak to mám napsat".

Omlouvám se za tuto poněkud přehnanou kapitolu, která se ostatně čtenáře netýká, neboť shora citovaný programátor nemá pro přemíru práce čas na četbu sborníků.

Úsporný program.

Za úsporný program považujme takový program, který neobsahuje ani jednu zbytečnou instrukci a nezabírá v paměti počítače ani jeden zbytečný byte. Zbytečná instrukce je pak ta, které není bezpodmínečně třeba k provedení požadované funkce programu. Jinými slovy, úsporný program provádí pouze životně nutné funkce a to nejjednodušším algoritmem. Praxe ukazuje, že tyto přirozené požadavky jsou často hrubě ignorovány nejen na úrovni programování, ale i v analýzách programátorských i systémových. Obdobně jako o úsporném programu je možno hovořit o úsporné analýze či úsporné systémové analýze, což se sice vymyká rámci tohoto příspěvku, nicméně možnosti úsporného programování se od nich přímo odvíjejí.

Myšlenka úsporného programování vychází z členění instrukcí programu na:

- vstupní/výstupní
- procedurální (aritmetika, přesuny, formátování ap.)
- pomocné (skoky, rozhodování, větvení)

Logika programu stojí a padá s poslední skupinou, t.j. s pomocnými instrukcemi. Tyto instrukce samy o sobě nic neprovádějí, zabírají místo a jsou příčinou téměř všech programových chyb. Chyby způsobené těmito instrukcemi se navíc mnohem hůře hledají, než u procedurálních a vstupních / výstupních operací. Maximálním omezením pomocných instrukcí lze dosáhnout :
- výrazného zkrácení programu
- prudkého poklesu četnosti chyb
- snadného ladění i údržby programu.

Vytvoření úsporného programu předpokládá v prvé řadě co nejpřesnější ujasnění požadovaných funkcí a vyloučení všech nepodstatností (přípravná fáze). Tuto fázi se nevyplácí podceňit, jak je zřejmé z kapitoly o programátorských nešvarech. Je nutno se k ní vracet i po odladění programu a předání do provozu. V době tvorby programu programátor nemá nikdy zcela přesné informace o frekvenci využívání jednotlivých větví programu - ty se dozví až z praktického provozu.

Součástí přípravné fáze je nalezení nejjednoduššího algoritmu, který vede k vytčenému cíli. Je to ten algoritmus, který vede k nejkratšímu přeloženému programu. Obvykle je též nejpřehlednější, nejrychlejší a přináší nejméně chyb. Tento algoritmus se mj. nezdržuje ošetřováním chyb v datech, které nemohou nastat, v praxi se nevyskytují nebo jim lze zabránit jinak (numerický děrovač pro numerické údaje, omezený počet znaků, aby chyba ani nemohla vzniknout, zásada jediného vstupu údaje do počítače, atd.). Nejúčinnější prevencí je zařídit věc tak, aby každá chyba komplikovala život především tomu, kdo ji způsobil.

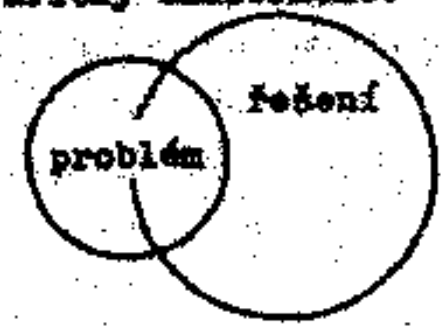
Ladění je provázeno snahou programátora program dále co nejvíce zkrátit. Sleduje, zda skutečně program využívá všech možností, které byly naprogramovány. Málo "navštěvované" nebo zcela "mrtvé" kouty programů jsou totiž nespolehlivějším zdrojem záhadných chyb, neboť nejsou nikdy zcela odladěny. (Není na ně čas ani trpělivost). Proto bude lepší, budou-li raději včas amputovány.

Teprve v okamžiku, kdy je programátor po několikaměsíčním bezchybném rutinním provozu programu do hloubi duše přesvědčen, že další zkracování programu je vyloučeno, lze hovořit o dobrém programu. Tento program je zcela stručný, přehledný, přímočarý a pochopitelný pro všechny, kteří s ním přijdou do styku. Autor sám se pak chlubí nikoliv tím, že jeho dílu nikdo nerozumí, nýbrž průzračností a primitivností svých triků (a někdy je jimi sám překvapen). Zjednodušování a "narovnávání" programu je ostatně z programátorského hlediska velmi zajímavé a není zrhánim času! Kompilační výtisk je poslední, nejpodrobnější a mnohdy také jedinou dokumentací, která skutečně"sedí". Je možno též snadno se poučit u souseda; po prolistování jeho programu, který je prost veškerého balastu a odskoků do neznámých houštin je vzápětí zřejmé o co v něm jde.

Vlastní ladění takto vytvořeného programu vyžaduje v průměru 4 kompilace:

1. kompilace - formální chyby, škrty
2.a3.kompilace - logické chyby, škrty, formální úpravy
4. kompilace - seškrtaná verze po několika měsících = konečná podoba

Graficky znázorněno:



1. verze
- nepřesná znalost problému
- nedokonalý algoritmus
- nevhodná obecnost
- chyby

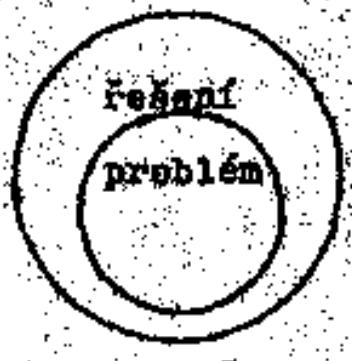
2.a 3.verze je lepší

Konečná verze
- přesná znalost problému
- nejjednodušší algoritmus
- řešení "na míru"

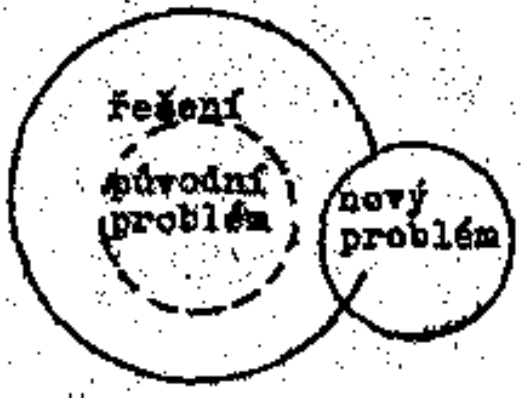
Časový odstup několika měsíců před vytvořením konečné verze je nutný nejen pro ověření souhlasu problému a modelu, ale umožní též objektivnější pohled na věc v době, kdy programátor není již daným problémem emocionálně zatížen, což je v době řešení nesporně žádoucí ("tvůrčí muka"). Později si snáze připustí, že to či ono je zbytečné.

Zde je nasnadě námitka, že řešením "na míru" se ztrácí obecnost. Je tomu přesně naopak. Když se Newton zabýval zákony gravitace, nenechal se unést výpočtem doby pádu pro jednotlivá jablka a hrušky. Nikoliv, dal si tu práci, že proniknul k podstatě problému, že totiž všechny předměty padají stejně.

Otázku neoprávněě pochopená obecnosti je možno znázornit graficky:



Při této metodě jsou do řešení zahrnuty všechny funkce, které v budoucnosti přicházejí v úvahu. Základním motivem bývá snaha vyřešit problém "jednou provždy", což se stejně nepodaří. Odhad budoucího vývoje reálného systému a skutečných požadavků na programy je příliš obtížný. Nakonec se ukáže, že skutečné požadavky "trochu přečnívají":

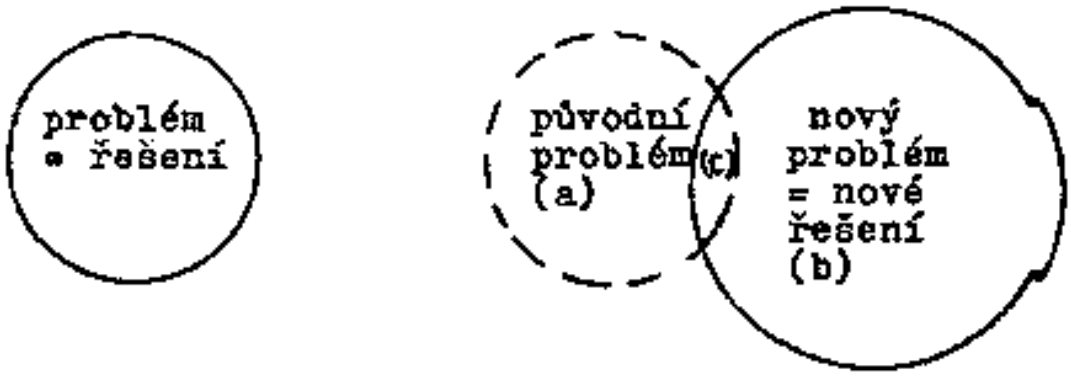


Důsledky jsou následující:

- řešení je nutno předělat
- značná část původního řešení nebyla nikdy využita, představuje tedy zbytečnou práci.

Z obrázku se zdá, že stačí původní řešení trochu vylepšit. Všichni ale víme, jak to vypadá s "nepatrnou" změnou v pracně odladěném složitém programu (flikování).

Postup "řešením na míru" je daleko méně pracný. Toto řešení zahrnuje pouze ty funkce, které jsou v daném okamžiku nezbytné:



časový bod t_1 časový bod t_2

Je třeba zdůraznit, že opravené "minimální" řešení zahrnuje :

1. Odstranění nepoužívaných funkcí (a)
2. Upřesnění a zdokonalení společných funkcí (c)
3. Přidání nových nezbytných funkcí (b)

Zvláště na bod 1, tj. odstranění nepoužívaných funkcí se snadno zapomíná, přestože je životně důležitý - zabrání rozrůstání modelu, které nutně vede k jeho celkové degeneraci. Analogii je možno najít v přírodě: nepoužívaný orgán zakrní, trénovaný roste, a to je existenční záležitost! Vybytnutí druhohorních obludi je sice otřepaný, nicméně dosti instruktivní příklad neschopnosti rychlé reakce na dynamiku okolí.

Zásady disciplinovaného programování

"Život" programu probíhá zhruba v těchto fázích:
Na začátku je impuls, kterým se rozhodne o vytvoření příslušného programu (předložení analýzy, požadavek uživatele, výpočetního střediska, vlastní rozhodnutí či nápad programátora apod.). Je to pouze časový bod.

Přípravná fáze :

Trvá zhruba 10 - 20 dnů, někdy i déle podle charakteru problému a schopností programátora. V období přípravné fáze programátor

- shání informace o problému a snaží se poznat a definovat jeho podstatu. Drží se zásady jeden problém = jeden program = jeden programátor
- provede rozčlenění na vstup, proceduru a výstup. Drží se zásady jediného vstupu, jediné procedury a jediného výstupu, pokud to lze. Pokud ne, upozorňuje na nedostatky v analýze či ve firemním software
- hledá nejjednodušší algoritmus, který vede k řešení nejkratší cestou, tj. nejmenším počtem strojových instrukcí
- volí programovací jazyk, v němž lze tento algoritmus nejsnáze zakódovat.

Tato fáze probíhá v myslí programátora a použitím velmi stručných poznámek. Navenek na programátorovi není vidět žádná intenzivní činnost. Dotyčný se věnuje v této době též jiným úkolům. Řešení zraje v jeho hlavě a vyznačuje se tím, že není provázeno spotřebou papíru, strojového času ani času nikoho jiného.
Přípravná fáze představuje největší tvůrčí přínos z celého období tvorby programu. Jak uvidíme dále, ostatní fáze jsou víceméně mechanickou záležitostí.
Je třeba upozornit, že <u>přípravnou fázi nelze urychlit</u> tlakem zvenčí ani zevnitř bez rizika katastrofálních následků. (viz bezhlavé programování a jeho důsledky)

Kódování

Je poměrně hbitou záležitostí. Ježto má programátor nyní zcela jasnou ideu, vytvoří koncept programu a krasopisně jej přepíše do formulářů, to vše v průběhu 2 - 4 hodin. Formuláře překontroluje a pošle do děrovny a k 1.kompilaci. Program není delší než 5 - 10 listů programovacího formuláře.
Potřebné pomůcky:
- definice vstupních informací (1 list)
- definice výstupních informací (1 list)
- poznámky o algoritmu procedury, tj. osnova s body co a jak naprogramovat, naznačené některé triky. Žádný vývojový diagram
- konceptní papír, tužka a blok programovacích formulářů.

Nejprve programátor provede programovou definici vstupních a výstupních informací. Drží se standardní symboliky a pokud nic podobného neexistuje, používá co nejjasnějších identifikátorů. Jejich délku volí mezi 3 - 7 znaky, mj.s ohledem na úroveň a frekvenci jejich výskytu. Tam, kde jde do větší podrobnosti, volí identifikátory kratší. Nejčastěji používané identifikátory zkrátí na 1 znak, aby mu déle vydržela tužka.

Při psaní procedurní části programu se programátor maximálně snaží o lineární strukturu programu. Stejnou strukturu má totiž kompilační výtisk i operační paměť počítače (číslování adres a postup programu). Podprogramům se vyhýbá a snaží se spíše o to, aby každý obrat byl zakódován pouze jednou a zařazen na správné místo v programu.
Nepodmíněný skok ve směru programu nepoužívá vůbec (jde to) a proti směru jen v krajním případě. Za každý nepodmíněný skok se potrestá poznámkou, proč a kam se skáče. Stejně označí každé větvení v programu. Každý logicky uzavřený úsek programu zřetelně oddělí a opatří stručnou poznámkou, nadpisem nebo zastránkováním. Logiku programu se snaží koncipovat tak, aby daná skupina instrukcí se buď provedla nebo přeskočila ve směru programu. Nepoužívá referencí,

po kterých se nikdo neptá. Každá reference tak označuje místo, na které se skutečně skáče, což později při opravách každý ocení. Snaží se, aby bylo možno bez postranních účinků kamkoliv vložit další instrukci a kteroukoliv vyřadit. Hojně používí do sebe vložených smyček, např. ve struktuře soubor-věta-pole-znak.

Ladění

Probíhá rovněž rychle a vcelku bez problémů. Jelikož je program krátký a přehledný, neobsahuje ani mnoho chyb. Přiléhavější je tedy spíše termín "doladování". Při první kompilaci bývá 3-7 formálních chyb, z toho obvykle polovina z děrovny. Pozorným pročtením kompilačního výtisku lze odhalit i některé chyby v logice (1-2). Kromě toho provede programátor některé škrty a zjednodušení v programu, zruší nepoužívané reference na základě seznamu cross-referencí a přidá několik vysvětlujících poznámek. Při dobré organisaci je první kompilace hotova do 24 hodin po napsání programu.

Druhá kompilace s testovacím během odhalí většinu logických chyb (3-5). Programátor se při tomto testovacím běhu snaží otestovat všechny instrukce programu, což při jeho lineární struktuře není problém. Dále provede některá zjednodušení a přeskupení instrukcí, které vedou ke zkrácení přeloženého programu. Druhá kompilace se provádí do 24 hodin po první. Základní chyby v logice se nevyskytují, pokud přípravná fáze proběhla normálně.

Třetí kompilace se provádí do 24 hodin po předchozí. V této 3.verzi jsou již vychytány veškeré logické chyby. Kromě toho provedl programátor poslední formální úpravy pro oko čtenáře. Je třeba důrazně připomenout, že žádné sestručnění programu nesmí jít na úkor poznámek či jakékoliv přehlednosti!

Ověřování

Po třetí kompilaci je program obvykle schopen samostatného života ve společnosti. Ověřují se jeho funkce v širších souvislostech a sleduje se frekvence jeho využití i funkční spolehlivost z hlediska celku i jednotlivých částí. Programátor sleduje připomínky uživatele (zadavatele) a zatímco se již aktivně věnuje jiné práci, v duchu i formou poznámek (tužkou do kompilačního výtisku) připravuje konečnou verzi. Ověřování trvá několik měsíců, přičemž je program rutinně využíván.

Rutinní verze

Jelikož všechny funkce programu jsou již konfrontovány s praktickou potřebou i statisticky ověřeny, je možno přistoupit k vytvoření rutinní verze programu, do níž jsou promítnuty zkušenosti z ověřovací fáze. Jelikož od zadání uplynulo již dosti času, projeví se patrně i dynamika okolí, do nějž je program zasazen. Překvapením i odměnou pro programátora je fakt

- jak snadno se v programu provádějí opravy
- jak málo oprav je ve skutečnosti požadováno.

Sešlost věkem

Nastává v okamžiku, kdy nároky na změnu funkcí programu představují radikální zásah do jeho logiky, což není nic jiného než impuls k vytvoření zcela nového programu. Nutno říci, že programátor se s programem loučí bez valné lítosti, neboť aktivní podíl jeho práce za psacím stolem nepřesáhl 10 hodin (přemýšlení se nepočítá).

Závěr.

Důsledné uplatňování zásad úsporného programování přináší dobré výsledky. Základní přístup spočívá v tom, postihnout pouze to, co je nutné a ne vše, co je možné. Od programátora vyžaduje především schopnost odlišit podstatné věci od podružných či bezvýznamných a trpělivost v přípravné fázi. Samozřejmým požadavkem je dokonalá znalost příslušného programovacího jazyka od syntaxe až po obraz zdrojového jazyka ve strojovém kódu. S tím souvisí i znalost operačního systému i hardware počítače, zejména v oblasti komunikace s periferiemi (přístupové časy apod.).

Hlavní přínos je třeba vidět ve výrazném zlepšení kvality programátorské práce. Zvýšení produktivity práce se dostaví jako vedlejší produkt.

Vytvoření úsporného programu je charakterizováno podstatně pečlivější přípravnou fází. Toto zvýšené úsilí se vrací formou

- úspory času při kódování
- úspory strojového času při ladění
- nenáročností při údržbě a všeobecné pochopitelnosti
- flexibility programu, přestože je "ušit na míru"
- klidnější práce programátora

Podstatné snížení podílu mechanické práce (na 8-10 hodin pro běžný aplikační program) dává programátorovi více času na hodnocení problému, vlastní práce i na studium.