



Desatero, aneb nejčastější chyby v programování

Aleš Kepřt
Univerzita Palackého

listopad 2008, březen 2009

1. Nepoužívejte pole na místě veřejného rozhraní

- Používejte třídy obsahující pole uvnitř
- Pole na místě veřejného rozhraní je, hlavně v C++, velmi nebezpečná věc
- V OOP a C++ zvlášť byste nikdy neměli používat pole přímo, ale zabalit jej do nějaké třídy
 - U čistě datových polí můžete použít kolekci typu pole (vector v C++, List<> v C#) - toto je však jen fiktivní „řešení“ problému.
 - Je lepší celé pole zabalit do třídy a zveřejnit jen několik přístupových metod, jejichž jména a typy vstupních parametrů vytvoříte tak, aby přesně vystihovaly, k čemu je to pole určeno.
- Nesnažte se splnit tento bod tím, že prostě zabalíte pole do třídy a dáte mu přístupové metody pro čtení a zápis dat v poli.
 - Každé pole má nějaký sémantický význam a v 90% případů k němu lze vytvořit vhodnější rozhraní, než jen tupé put/get. (Typicky to platí o šachovnici - tam si můžete s metodami vyloženě vyhrát.)
- Smyslem tohoto opatření je, že objekt nezveřejňuje svůj stav v podobě pole, ale výhradně po jednotlivých datových položkách
- Potřebujete-li naopak předat někomu víc dat (jako parametr jeho metody), můžete mu poslat tyto parametry v poli – není to problém
 - Nikdy ale nesmíte poslat cizímu objektu své vnitřní pole jako parametr jeho metody!!!
 - Pokud vás i sémantika vede k tomu, abyste někam posílali své vnitřní pole jako parametr, musíte udělat kopii toho pole a poslat jen kopii. Tak totiž zamezíte tomu, aby onen cizí objekt změnil ono pole.
- V jazycích umožňujících definovat operátor přístupu do pole můžete tento konstrukt využít. A je to dokonce velmi výhodné - syntakticky se to tváří jako pole, snadno se to používá, přitom je to bezpečné.

3. Nepoužívejte makra místo konstant

- Použijte typované konstanty
- Použijte deklaraci konstantní proměnné (const)
 - Konstantní proměnná má typ, to je výhodné
- Pojmenované konstanty nevytvářejte pomocí #define (v C++)
 - Makro se navíc nahrazuje jako text, tedy i tam, kde to ani nechcete.

4. Nepoužívejte čísla, když nic nepočítáte

- Použijte výčtový typ
- Příklad:
 - Rozhodnete se, že vytvoříte 4 konstanty pro směry pohybu vlevo, vpravo, nahoru a dolů
 - Budou to čísla 1 až 4 (pochopitelně), 0 může znamenat „bez pohybu“
 - Tyto konstanty nevytvářejte jako 4 samostatné konstanty
 - Použijte výčtový typ, čili deklaraci enum
 - V C# je navíc výhoda, že pojmenované hodnoty z výčtového typu nelze použít v jiném kontextu
 - Když se za půl roku na kód podíváte, nemusíte bádát „A ta dvojka je nahoru, nebo doleva..?“

5. Nepoužívejte maďarskou notaci

- Maďarská notace je, když... přidáváte na začátek názvů entit (nejčastěji u proměnné) zkratku typu entity
- Takové přívěšky nepoužívejte – je to nějaký přežitek z beztypových a slabě typovaných jazyků (jako např. ANSI C)
- Bohužel Microsoft to stále hodně používá (ve Win32 API, čili opět zejména v kódu dotýkajícím se slabě typovaných jazyků), takže hodně začátečníků se tím nechá zblbnout
- Typ proměnné správně stojí mimo název a je sémanticky bezvýznamné, abyste typ strkali ještě do názvu
- Další příklad chybných názvů je dávání písmene C na začátek názvu třídy. C je zde dokonce odvozeno od názvu metaentity (class), což je vyloženě nepotřebný přívěšek
- Maďarská notace se hodí pro hodnoty výčtových typů v C/C++, protože ty jsou bezkontextové
 - V ostatních jazycích to ale u výčtových typů nepoužívejte

6. Data a kód patří k sobě

- V OOP je jasně dáno, že kód pracující s nějakými daty patří do stejné třídy jako tato data
 - Tato věta je samozřejmě značně zjednodušená(!!!), ale zkuste se někdy zamyslet, jestli náhodou v programu nemáte třídu XY, která úplně zbytečně stojí mimo datovou třídu (třeba se to může stát u šachovnice a nějakých algoritmů).
- Pozor ale na jednu věc:
 - Vyšší prioritu než tato poučka má princip odpovědnosti, tj. že třída dělá přesně to, co je v její kompetenci, za co je odpovědná
 - Nic navíc třída dělat nemá.
 - Proto někdy stojí kód mimo dat – když prostě není odpovědností té datové třídy, aby dělala to, co je v tom algoritmu
 - Je to vždy otázka návrhu programu

7. Šetřete s dědičností

- Často je lepší použít běžnou asociaci
- Místo zdědění třídy vytvoříte proměnnou té původní třídy uvnitř nové třídy

8. Proměnné nejsou public

- Výjimkou jsou čistě datové třídy, které z logiky věci neobsahují žádné metody, jen samé public proměnné
 - Toto je možné udělat, ale buďte velmi opatrní, kde to použijete
- Příklad:
 - Pozice na šachovnici bude zcela evidentně datovým párem dvou číselných proměnných int x,y;
 - Je to třída, ale zda položky x a y jsou public, nebo private, závisí na tom, zda tuto třídu budete chápat jako doslova datovou, nebo k ní dáte další metody
- Místo public proměnné je často vhodnější používat property

9. Nepoužívejte návratové hodnoty pro signalizaci chyby

- Použijte výjimky
- Výjimky (exceptions) jsou přirozeným nástrojem pro „oznamování“ chyb
- Na návratové hodnoty ve stylu „0 je OK, <0 je chyba“ jednoduše zapomeňte
- Literatura obvykle uvádí, že asi 50% kódu je v C++ jen kvůli tomu, aby ošetřovalo chybové stavy
 - Čili samé příkazy if...
 - A takový chaos v kódu přece nikdo nechce 😊
- Poznámka: Toto je velmi obtížné dodržet v C++, takže v C++ toto pravidlo můžete ignorovat

10. Optimalizujte algoritmy, ne implementaci

- Cílem optimalizací je obvykle zrychlení programu při běhu
- Optimalizaci musíte dělat tak, že zvolíte lepší algoritmy
- Snaha optimalizovat kód po implementační stránce tím, že nebudete dodržovat zde uvedené poučky, je cestou do pekla
 - Řada zde uvedených pouček vede k pomalejšímu kódu
 - Ale jde o konstantní nárůstek výpočetních časů, zatímco změnou algoritmu dosáhnete mnohem více – např. třídění bubble sort nahradíte quick sortem...
- Navíc platí: Napřed to celé naprogramujte, pak teprve optimalizujte
 - Bude-li to vůbec třeba

11. Rozděľujte zdrojáky do více souborů

- Bývá dobrým zvykem, že každá třída má svůj vlastní soubor
- Některé jazyky umožňují i jiné členění, avšak pro lepší přehlednost je velmi doporučeno dodržovat standardní styl, tj. pro každou třídu mít samostatný soubor téhož jména

12. Nepoužívejte globální proměnné

(od Michala Krupky)

- Nevýhoda globálních proměnných je, že je to jakési fluidum prostupující celým programem – odporuje to principu lokality, který je základním kamenem kvality a bezchybnosti kódu
- Jakmile se vám objevuje jedna entita (globální proměnná) na různých místech kódu, nejste schopni uhlídat, co se kde děje, a odladit případné chyby s touto entitou související

13. Rozlišujte specifikaci a implementaci

- Základní požadavek OOP: zapouzdření
- Specifikace rozhraní, kterým třída nabízí svou funkcionalitu okolnímu světu, musí být zásadně zcela nezávislá na vnitřní implementaci
- Rozhraní třídy navrhujte vždy s ohledem na sémantiku („co má ta třída dělat“) a bez ohledu na implementaci („jak to ta třída dělá“)
- Výhody správného řešení poznáte velmi zřetelně sami

14. Kontrolujte, že vstupy patří do domény

(od Milana Řezníčka)

- Vysvětlení příkladem:
 - Program šachových koncovek v Lispu rozumí příkazu (umisti-figurku '(cerny pesak) 1 1)
 - Otázkou je, co program udělá, když zadáme (umisti-figurku '(zeleny kominik) 110 120)
 - Statické jazyky odfiltrují zeleného kominíka při překladu, neboť pěšák celkem jistě bude členem výčtového typu (čili enum Figurka)
 - Platnost souřadnic pak musíme zkontrolovat na začátku funkce umisti-figurku.
- Poznámka: V OOP se tyto testy vstupů mohou vynechat, pokud se jedná o privátní metodu třídy, která z logiky kódu nemůže být zavolána s neplatným vstupem
 - Nicméně i tam v zájmu bezpečnosti test být může

15. Rozlišujte specifikaci a implementaci i na straně klienta

- Implementaci a specifikaci musí rozlišovat i klient, ne jen dodavatel funkcionality
- V bodě 13 byla řeč o tom, že třída nesmí mít rozhraní závislé na tom, jak organizuje svá data uvnitř
- Obráceně ale ani klient nesmí lpět na „nedokumentované“ funkcionalitě
- Ta nedokumentovaná funkcionalita může být právě způsobena tím, co je v bodě 13: Z třídy se dostává ven něco z vnitřní implementace
- Příklad:
 - ID čísla jsou de facto pointery do nějaké tabulky, takže jsou to vlastně všechna čísla v násobcích 4 a mají dolní dva bity vždycky nulové
 - Klient udělá strašnou chybu, že do těch dvou bitů obratně začne ukládat další svoje pomocná data
 - Chyba na sebe v takovém případě nechá třeba hodně dlouho čekat, než se něco opravdu pokazí v praxi, ale následky jsou o to horší...

16. GUI musí být především standardní, až potom pěkné

- Změna barev u jednotlivých tlačítek a podobných součástí okna vašeho programu jistě každého láká a máte z ní dobrý pocit, že výsledný program je hezčí než standard
- Mějte ale na paměti, že uživatele každá nestandardní, tedy hlavně nečekaná, věc v první řadě vyděsí!
 - A nejspíše se polovině z nich ani nebude líbit totéž, co se líbí vám
- Proto nevymýšlejte žádné specialitky a vytvářejte GUI hlavně standardní
- Pro celou řadu platforem existují podrobné specifikace, jak mají programy vypadat
 - Slovo „platforma“ zde používám jako obecné označení pro počítače, operační systémy a další prostředí, kde programy běží.



© Mgr. Aleš Kepřt, Ph.D., 2006-2009

Vytvořeno pro potřeby přednášky na UP Olomouc. Tento text není určen pro samostudium, ale jen jako vodítko pro přednášku, takže jeho obsah se může čtenáři zdát stručný, nekompletní či možná i chybný. Použití je povoleno dle vlastní libosti, ale jen na vlastní nebezpečí. ☺ V případě dalšího šíření je NUTNO uvádět původního autora a odkaz na původní dokument. Komentáře můžete posílat e-mailem autorovi (adresu najdete přes Google).