Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 26.02. 2001
Url: http://www.builder.cz/art/cpp/cpp_polymorf.html

Polymorfismus - dokončení

Slovo polymorfismus znamená něco jako jako "vícetvarost", nebo "mnohotvarost". Polymorfismus je v programování velmi obecný pojem. V souvislosti s OOP se jedná o to, že instance různých tříd na stejný podnět (na vyvolání stejné metody) reagují různě. Instance více tříd poskytují svému okolí stejnou službu, ale každá instance na vyžádání této služby provede něco jiného. Troufám si říci, že právě pro spojení polymorfismu s dědičností se vlastně používá OOP. Pro třídy poskytující stejné služby se nabízí nadtřída, která má společné vlastnosti všech svých podtříd. Velká část návrhu programů v OOP je vlastně hledání těchto společných služeb (vlastností) pro různé třídy, a vytváření nadtříd podobných tříd. Mějme jako příklad třídu kruh, a třídu obdélník. U obou chceme mít nějakou metodu, vracející obvod tohoto geometrického útvaru. V OOP se podobný problém řeší většinou tak, že se vytvoří nějaká nadtřída - už jsem ji vlastně pojmenoval na "geometrický útvar" a tato třída bude mít metodu pro vypočítání obvodu. Z této třídy budou dědit třídy kruh a obdélník, které si metodu pro výpočet obvodu implementují po svém. Konkrétně v C++ bude metoda `float dejObvod();` deklarována jako virtuální a třídy kruh a obdélník si předefinují podle potřeby její tělo.

Abstraktní třídy, čiré metody

Zmiňované zavedení nadtřídy je vlastně abstrakcí tříd kruh a obdélník. Taková abstrakce se může na první pohled zdát zbytečná, proto se podívejme na výhody. V místech, kde budu používat instance tříd kruh, nebo obdélník, budu s těmito instancemi co nejvíce pracovat jako s instancemi jejich nadtřídy. Nebudu tím vlastně znát se kterou podtřídou zrovna pracuji a program bude algoritmus pracující s obecným geometrickým útvarem. Výhodu si uvědomíme, až když fungující program budu chtít rozšířit o další grafický útvar - třeba trojúhelník. V OOP se předpokládá, že při takové změně (rozšíření) se bude muset minimálně (nejlépe vůbec) zasahovat do napsaného zdrojového textu. Ve skutečně objektově orientovaném jazyce (Tím bohužel C++ není.) a při opravdu kvalitním návrhu je možné skutečně pří rozšiřování znovu použít hotové třídy beze změn. V C++ to s takovou ideální znovupoužitelností zdrojového textu není zase tak žhavé. Ale správně navržený program lze rozšiřovat s minimálními úpravami.

Abstrakce sebou nesou také určité problémy. Například v mém příkladě je otázka jaké tělo má mít metoda `virtual float GrafickyUtvar::dejObvod();` . Nabízí se odpověď žádné, to by jsme ale měli nějak překladači dát najevo. K tomuto účelu existují tak zvané "pure" metody. Pure je v těchto souvislostech vhodné přeložil asi jako čirý. Nehodí se zde překlad prázdný, protože prázdná metoda je v C++ něco jiného. Čirou metodou může být jen metoda virtuální. Čirá metoda nemá tělo a nelze jí vyvolat. Čirá metoda se označí v deklaraci symboly = 0. Tedy například: `virtual float dejObvod() = 0;` . Aby se zabránilo vyvolání takové metody, nelze vytvořit instanci třídy, která má alespoň jednu čirou metodu. Třídu, která obsahuje alespoň jednu čirou metodu, nazýváme abstraktní třídou. Abstraktní třída nemůže mít své instance. (Třídu nelze instanciovat - podivné slovo, které raději moc nepoužívám.) Vytvořím-li potomka, ve kterém nedefinuji tělo čiré metody, je metoda čirá i v potomkovi a potomek je také abstraktní třída.

Příklad:

```
#include <iostream.h>

class GrafickyUtvar
{
 public:
   virtual float dejObvod() = 0;
};

class Kruh : public GrafickyUtvar
{
 private:
   float r;
 public:
   Kruh():r(0){};
   Kruh(float polomer):r(polomer){};
   virtual float dejObvod();
};

class Obdelnik : public GrafickyUtvar
{
 private:
   float a,b;
 public:
   Obdelnik():a(0),b(0){};
   Obdelnik(float s1, float s2):a(s1),b(s2){};
   virtual float dejObvod();
};

class Plocha
{
 private:
   int Pocet;
   GrafickyUtvar *utvary[10];
 public:
   Plocha():Pocet(0){};
   void pridej(GrafickyUtvar *u)
        { utvary[Pocet++] = u;};
   float dejObvodyVsech();
};

float Plocha::dejObvodyVsech()
{
   float navrat = 0.0;
   for(int p = 0; p<Pocet; p++)
      {navrat += utvary[p]->dejObvod();}
```

```
   return navrat;
}

float Kruh::dejObvod()
{
   return 2*3.14*r;
}

float Obdelnik::dejObvod()
{
   return 2*(a+b);
}

int main(void)
{
   Plocha p;
   GrafickyUtvar *a,*b,*c,*d,*e = NULL;
   p.pridej(a = new Kruh(2));
   p.pridej(b = new Kruh(10.3));
   p.pridej(c = new Obdelnik(12,45));
   p.pridej(d = new Obdelnik(1.4,100));
   cout << p.dejObvodyVsech() << endl;
   delete a;
   delete b;
   delete c;
   delete d;
   delete e;
/* Zkuste pro zkoušku odstranit // na následujících dvou řádcích. Nastane chyba. */
   //GrafickyUtvar g;
   //GrafickyUtvar *gg = new GrafickyUtvar;
   return 0;
}
```

Třída plocha je napsána jen pro tento příklad, jinak není moc dobře implementována.

Nyní si představte situaci, že tento program chci rozšířit o třídu rovnostranný trojúhelník. Třída bude vypadat asi následovně:

```
class RovnoStrannyTrojuhelnik : public GrafickyUtvar
{
 private:
   float a;
 public:
   RovnoStrannyTrojuhelnik() : a(0){};
   RovnoStrannyTrojuhelnik(float strana):a(strana) {};
   virtual float dejObvod();
```

```
};

float RovnoStrannyTrojuhelnik::dejObvod()
{
   return 3*a;
}
```

Nyní ve funkci main před řádek `cout << p.dejObvodyVsech() << endl;` vepište řádek `p.pridej(e = new RovnoStrannyTrojuhelnik(10));` . Přece jenom jsme museli již napsaný zdrojový text měnit - funkci main. Ale všimněte si, že jsem NIJAK nemusel měnit třídy Plocha, Grafický útvar, kruh, obdélník, nebo jejich metody! To není vše, ja jsem pro tohle rozšíření ani NEPOTŘEBOVAL ZDROJOVÉ TEXTY TĚCHTO TŘÍD!!!! Stačila mi pouze deklarace (V nějakém hlavičkovém souboru, který bych vložil pomocí include). Zdrojové texty těl metod tříd jsem nepotřeboval. Stačilo kdybych je měl zkompilovány v binárním tvaru (*.obj, *.dll, *.o, atd...), které bych na závěr po kompilaci "přilinkoval".

Závěrem

Rozhodně Vám doporučuji používat polymorfismus v kombinaci s dědičností k vytváření abstraktních tříd. Je-li třída opravdu abstraktní, pro jistotu nějakou její metodu označte jako čirou. Nezapomeňte, že ne každá nadtřída musí být nutně abstraktní. Pro tento způsob abstrakcí musíte používat metody volané pozdní vazbou (virtuální). Má-li třída alespoň jednu virtuální metodu, měla by mít i virtuální destruktor - o tom jsem psal v minulém článku. Používání metod volaných pozdní vazbou je trochu pomalejší, než používání metod volaných časnou vazbou. Musí se totiž adresa podprogramu vypočítávat z TVM - viz předchozí článek. Přesto Vám doporučuji metody volané pozdní vazbou používat. Metody volané pozdní vazbou do OOP prostě patří. Existují dokonce jazyky (Java, atd...), které nemají časnou vazbu, vše je voláno pozdní vazbou. Časnou vazbu používejte, jen jste-li si jistí, ale opravdu jistí na 100%, že metodu v nějakém potomkovi nebudete měnit.

Ještě si Vás dovolím upozornit na poměrně častou chybu v přístupu k podobným problémům. Bylo by chybou dávat do třídy grafický útvar nějakou proměnnou, pomocí níž by jsem poznal, zda se jedná o obdélník kruh atd... Dědičnost bych nepoužil. Metoda na výpočet obvodu by byla vlastně jeden veliký switch a každý case by byl pro konkrétní útvar (kruh, atd...). Tím bych přišel o všechny výhody abstrakcí při rozšiřování programu. Není ani dobré dědičnost použít a v každé instanci nějaké podtřídy nastavit zmiňovanou proměnnou stejně. Cíl by byl identifikovat, o jakou podtřídu jde. Abstrakce spočívá právě v tom, že pracuji s nadtřídou a nevím jaká podtřída to vlastně je. Ale uznávám, že poměrně často je potřeba zjistit o jakou podtřídu se jedná. Kdybych náhodou já v uvedeném programu musel zjistit o jakou podtřídu jde, použil bych dynamickou identifikaci typu (O té si povíme později.), ale snažil bych se o to co nejméně, protože bych tím přišel o obecnost.

Tolik k polymorfismu. Příštích několik kapitol se budu věnovat zase dědičnosti, tentokrát vícenásobné a všem potížím, které může způsobit.