Vzory, vzory a zase vzory

Miroslav Virius

V tomto článku navážeme na článek *Neobjevovat Ameriku, nevynalézat kolo* z tohoto čísla tištěného Chipu (str. 108), ve kterém jsme se seznámili s *návrhovými vzory* (design patterns) a s jejich významem pro dnešní programování. Zde se na ně podíváme podrobněji a ukážeme si i programové realizace některých z nich.

Malé opakování

Návrhové vzory se dostaly do popředí zájmu programátorů a vůbec všech, do se zabývají vývojem softwaru, v posledních několika letech. Jak víme, jde o ustálená řešení nejběžnějších problémů, s nimiž se při návrhu softwaru můžeme setkat. (Martin Fowler je popsal jako řešení, která se osvědčila v jednom případě a o kterých lze předpokládat, že se osvědčí i v mnoha jiných případech.)
Návrhové vzory vznikly přirozeným vývojem a teprve pak si jich skupina softwarových odborníků povšimla a publikovala 23 z nich – nejprve na konferencích OOPSLA o objektových technologiích a pak v monografii [1]. To ale samozřejmě neznamená, že tím jsou návrhové vzory vyčerpány. Tak, jako se vyvíjejí techniky programování a zároveň s nimi požadavky uživatelů, možnosti a dostupnost různých technologií, budou se vyvíjet i návrhové vzory. Lze předpokládat, že velké softwarové firmy budou časem mít (nebo nejspíš už mají) přísně utajované katalogy svých vlastních návrhových vzorů.
Podívejme se ale na ty, o nichž se hovoří v [1]. Připomeňme si, že návrhové vzory se zpravidla dělí na vzory tvořivé (kreacionální), vzory strukturální a vzory chování (behaviorální).

Vzory tvořivé

Jak víme, zabývají se tvořivé vzory způsoby vytváření instancí objektových typů. Výsledkem jsou řešení, která jsou nezávislá na způsobu, jakým instance různých objektových typů vznikají.

Příklad: jedináček neboli singleton

O tomto vzoru jsme se letmo zmínili už v úvodním článku v tištěném Chipu. Podívejme se na něj podrobněji. Jde o to, že někdy potřebujeme, aby jistá třída – řekněme třída X – měla pouze jedinou instanci. Když se nad tím na chvilku zamyslíme, napadnou nás nejméně tři možnosti, jak se s tímto požadavkem vypořádat:

1. Můžeme tento požadavek ignorovat, navrhnout třídu X zcela normálním způsobem, a vytvoření jedné instance přenechat na disciplíně programátora, který naši třídu bude používat.
2. Můžeme do třídy X vložit statickou datovou složku („datovou složku třídy“), která bude obsahovat informaci o počtu vytvořených instancí. Konstruktory třídy X budou kontrolovat počet existujících instancí a v případě pokusu o vytvoření druhé instance vyvolají výjimku.
3. Můžeme vytvořit jednu globální instanci třídy X, nějakým způsobem ji ukrýt a definovat globální funkci, která bude vracet ukazatel na tuto instanci.

Podívejme se na výhody a nevýhody jednotlivých možností:

První možnost je nepochybně nejjednodušší, neboť neznamená žádné změny v návrhu třídy X. Také ovšem neposkytuje žádnou záruku, že omezení kladené na počet instancí bude dodrženo – a to ani v případě, že třídu X navrhujeme pouze pro svou vlastní potřebu. Při pozdějších úpravách programu na toto omezení snadno zapomeneme a důsledky si lze lépe představit než popsat.
Druhá možnost je o něco lepší, nelze ji ale označit za dobrou. Na jedné straně poskytuje záruku, že v programu nevznikne více než jedna instance třídy X, což je dobré. Na druhé straně neošetřená výjimka znamená ve většině programovacích jazyků okamžité ukončení programu, a to si většinou nemůžeme dovolit. Ale i když výjimku zachytíme a ošetříme, je situace problematická. Je totiž jasné, že nejméně celý úsek programu, který počítal s vytvořením nové instance třídy X, nemůže proběhnout – a to není to, co jsme vlastně měli v úmyslu, neboť my jsme chtěli použít onu jedinou instanci třídy X.
Úvahy nad prvními dvěma možnostmi nás vedou k závěru, že nejlepším řešením bude, když programátor používající třídu X vůbec nebude mít možnost vytvářet její instance – jinými slovy, se sami postaráme o vytvoření jediné instance této třídy, ukryjeme ji a poskytneme k ní jeden globální přístupový bod. To je v podstatě třetí z uvedených možností a je základem návrhového vzoru *jedináček*.
Programovací jazyky, jako je C++, Java, C# nebo Object Pascal, opravdu poskytují nástroje, s jejichž pomocí lze zabezpečit, že programátor, který používá naši třídu X, nebude moci vytvářet její instance. Třída sama se může postarat o vytvoření jediné své instance a může nabídnout veřejně přístupnou metodu, která bude vracet ukazatel nebo referenci na vytvořenou instanci. V C++ můžeme postupovat takto:

- Ve třídě X deklarujeme soukromou statickou datovou složku `_x` typu `*X`.
- Dále deklarujeme veřejně přístupnou statickou metodu `getX()`, která při prvním volání vytvoří instanci třídy `X` a ukazatel na ni uloží do `_x`. Tato metoda představuje „přístupový bod“ k jediné existující instanci.
- Všechny konstruktory třídy `X` deklarujeme jako chráněné. Podívejme se na schématický příklad:

```
class X
{
public:
     static X* getX() {
          if(_x == 0)
               return _x = new X(1);
          else
               return _x;
     }
     // Nějaké užitečné metody
     int getI(){ return i; }
protected:
     X(int i): i(i){}
private:
     static X* _x;
     int i;         // Nějaká užitečná data
};

// Inicializace statické složky
X* X::_x = 0;
```

Možná namítnete, že by bylo jednodušší vytvořit onu jedinou instanci třídy `X` už při inicializaci statické datové složky `_x` a zjednodušit deklaraci metody `getX()`,

```
// Tohle nemusí fungovat
X* X::_x = new X(5);

class X {
public:
     static X* getX()
     {
          return _x;
     }
// Ostatní složky jako předtím
};
```

To by ale nemuselo fungovat. Jazyk C++ totiž nestanoví přesně dobu, kdy inicializace statických datových složek proběhnou, a pokud bychom např. chtěli použít instanci jedináčka v jiném samostatně překládaném souboru k inicializaci globální proměnné, mohlo by se stát, že by ještě neexistoval.

Poznámky

1. Skutečnost, že jsme konstruktor (příp. všechny konstruktory) deklarovali jako chráněný, nám umožní v případě potřeby odvozovat od třídy `X` potomky. Kdyby byly všechny konstruktory soukromé, znemožnili bychom tím dědění.
2. Takto navrženou třídu lze snadno upravit, aby umožňovala vytvořit zadaný počet instancí (více než jednu, ale omezený počet).
3. Inicializační část konstruktoru je deklarována správně: Pravidla pro vyhledávání jmen v C++ zaručují, že se hodnota parametru `i` uloží do datové složky `i`. V zápisu `i(i)` totiž musí být „vnější“ `i` buď jménem složky nebo předka třídy `X`, takže parametr `i` při vyhledávání vůbec nepřipadá v úvahu. Na druhé straně v závorce je očekáván výraz, a proto zde parametr `i` zastíní složku `i`.
4. Někdy bývají jako příklad jedináčka uváděny třídy, které nemají vůbec žádné instance, které byly navrženy pouze jako „obalové“ třídy pro skupinu statických metod a konstant. (Vzpomeňme si např. na třídu `java.lang.Math` z jazyka Java. Takovéto třídy slouží v čistě objektových jazycích pouze jako objektová kamufláž pro skupinu „obyčejných“ funkcí.) Třídy bez instancí jsou ovšem jednodušší než jedináčci, nepotřebují – a také neobsahují – žádný přístupový bod, stačí, když mají soukromý konstruktor.

Další tvořivé vzory

Podívejme se alespoň ve stručnosti na některé další tvořivé vzory.
Abstraktní továrna (abstract factory) poskytuje obecné rozhraní, které umožňuje centralizovaně vytvářet objekty několika příbuzných datových typů, aniž by bylo nezbytné přesně zadat typ. Jinými slovy, abstraktní továrna zapouzdřuje rozhodování o typu a dalších podrobnostech instance a její následné vytvoření. Abstraktní továrna zpravidla vrací vytvořenou instanci prostřednictvím odkazu na společného předka typů všech takto vytvářených instancí nebo prostřednictvím odkazu na společné rozhraní.
Lze si např. představit, že v grafickém editoru reprezentujeme grafické objekty pomocí objektů, které jsou instancemi tříd odvozených od abstraktní třídy `GrafickýObjekt`.

Použijeme-li k jejich vytváření abstraktní továrnu, bude vracet instance odvozených tříd – úseček, kružnic, atd. – např. pomocí ukazatelů na typ `GrafickýObjekt`.
S použitím abstraktních továren se setkáme např. u kódu komponent COM, generovaného automaticky pomocí různých průvodců na základě popisu tříd v IDL.
Podobný účel jako abstraktní továrna má i *tovární metoda* (factory method), občas označovaná také jako *virtuální konstruktor*. Lze ho charakterizovat tak, že umožňuje třídě, aby přenechala rozhodnutí o vytvoření instance svým podtřídám (a přitom může nabídnout implicitní implementaci). Podobně jako abstraktní továrna vrací vytvořenou instanci prostřednictvím ukazatele na společného předka.
S použitím továrních metod se lze často setkat v knihovnách jazyka Java. Jedním takovým příkladem může být statická metoda `newInstance()` třídy `java.lang.Class`, která umožňuje vytvořit instanci třídy, máme-li k dispozici instanci třídy `Class` popisující typ požadované instance.
Návrhový vzor *prototyp* umožňuje vytvářet nové instance kopírováním (klonováním) existujícího prototypu. Tento vzor se hodně uplatňuje v jazycích se statickým typovým systémem, jako je C++, kde nejsou za běhu k dispozici téměř žádné informace o typu. (Třída `std::type_info` je pro vytvoření nové instance za běhu programu v současné verzi C++ prakticky nepoužitelná.)

Strukturální vzory

Tyto vzory, jak víme, popisují strukturu objektů nebo jejich seskupení. Také v oddílu o strukturálních vzorech se podíváme na jeden z nich podrobněji a některých dalších se jen zmíníme.

Příklad: most

Tento vzor odděluje abstrakci od implementace, což umožňuje nezávisle měnit jedno i druhé. *Most* bývá také označován jako vzor *držák/tělo* (handle/body). V poslední době se lze v anglicky psané literatuře setkat také s popisným označením *the pimpl idiom*, přičemž *pimpl* je zkratka slov *pointer to implementation*, tedy ukazatel na implementaci.
Podívejme se na příklad: Ve své aplikaci potřebujeme binární vyhledávací strom. Zatím není jasné, zda nám vystačí obyčejný binární strom nebo zda budeme požadovat některou z variant vyváženého stromu, např. červeno-černý strom nebo třeba AVL-strom. Co s tím?
Můžeme samozřejmě použít implementaci stromu, o níž se domníváme, že je pro naše účely vhodná, a doufat, že se domníváme správně. Ovšem v případě, že budeme později chtít použít jinou implementaci, se můžeme dostat do problémů, neboť toto řešení není právě nejpružnější.
Poněkud lépe dopadneme, když nejprve definujeme abstraktní třídu `Strom` (nebo třeba `Tree`) a od ní odvodíme potomka – implementaci, kterou v programu použijeme. Budeme-li později chtít použít jinou implementaci, stačí, když ji odvodíme od stejného předka.
Návrhový vzor *most* jde ještě o něco dál. Jeho základní podoba v našem případě vypadá takto:

1. Definujeme třídu `Strom`, která bude obsahovat pouze veřejně přístupné metody požadované ve třídě reprezentující strom v našem programu, a soukromou složku představující ukazatel na instanci třídy, která poskytuje skutečnou implementaci stromu. Instance této třídy budou představovat rozhraní pro přístup ke stromu.
2. Dále definujeme třídu `StromImpl`, která poskytne implementaci stromu.
3. Metody třídy `Strom` se budou odvolávat na metody implementační třídy `StromImpl`.

Podívejme se na konkrétní implementaci v nejjednodušším případě. Nejprve zjednodušená třídu `StromImpl`:

```
class StromImpl
{
public:
     StromImpl;
     void Vloz(int n);
     void Odstran(int n);
     bool Obsahuje(int n);
     // A další metody
protected:
     class Prvek // Třída prvků stromu
     {
          // ...
     };
     // Zde jsou také metody, které zpřístupníme
     // potomkům (implementační detaily)
private:
     Prvek * koren; // Ukazatel na kořen stromu
};
```

Implementace jednotlivých metod zde neuvádíme, stejně jako neuvádíme řadu dalších veřejných i chráněných metod, neboť pro nás nejsou důležité.
Třída `Strom`, využívající vzor *most*, bude jednoduchá, neboť nebude obsahovat žádné chráněné ani soukromé metody a kromě ukazatele na implementační instanci ani žádné datové složky:

```
class Strom
{
public:
     Strom(StromImpl *p);
     void Vloz(int n);
     void Odstran(int n);
     bool Obsahuje(int n);
     // ... a další veřejné metody
protected:
     StromImpl* GetImpl(){return _pImpl;}
     void SetImpl(StromImpl *_strom){_pImpl = _strom;}
private:
     StromImpl* _pImpl;
};
```

Metody třídy `Strom` prostě využijí služeb třídy `StromImpl`:

```
Strom::Strom(StromImpl *p)
: _pImpl(p) { }

void Strom::Vloz(int n)
{
     _pImpl->Vloz(n);
}
```

```
void Strom::Odstran(int n)
{
     _pImpl->Odstran(n);
}

bool Strom::Obsahuje(int n)
{
     return _pImpl->Obsahuje(n);
}
```

Poznamenejme, že v typickém případě bude `StromImpl` abstraktní třída a program bude využívat její konkrétní potomky.

Poznámky

Co tím získáme, když použijeme *most*?

1. Tím, že jsme oddělili rozhraní od implementace, jsme získali možnost měnit implementaci bez výraznějších zásahů do programu. V případech, kdy to má smysl, lze (po jednoduchých úpravách dokonce měnit implementaci za běhu programu.
2. Vzor *most* umožňuje odvozovat potomky od třídy, představující rozhraní, stejně jako od třídy představující implementaci. To zvyšuje pružnost návrhu.
3. V některých situacích může být výhodné, že několik rozhraní bude pracovat se stejnou implementací. Např. několik instancí třídy `Strom` může čerpat data z jedné instance třídy `StromImpl`.
4. Získáváme tím také možnost oddálit rozhodování o použité implementaci.
5. V C++ nám vzor *most* poskytuje možnost utajit rozhraní implementační třídy. Hlavičkový soubor, který obsahuje deklaraci třídy `StromImpl`, není třeba zveřejňovat, neboť je potřebný pouze při překladu souboru s implementací metod. Hlavičkový soubor s definicí třídy `Strom` se na něj nemusí odvolávat.
6. Do implementačních tříd lze ukrýt kód závislý na platformě. Například při tvorbě programu s grafickým uživatelským rozhraním, který chceme spouštět pod Windows i pod Linuxem, bude výhodné mít pro okna, tlačítka apod. třídy nezávislé na platformě, které se pomocí mostu odvolají na konkrétní implementaci pro použité prostředí.

S použitím tohoto návrhového vzoru se setkáme mimo jiné v automaticky generovaném kódu při vytváření distribuovaných programů podle standardu CORBA, se standardní knihovně jazyka C++ atd.

Další strukturální vzory

Podívejme se opět ve stručnosti na další strukturální vzory.
Jedním z nejpoužívanějších je *adaptér*, označovaný také jako *obal* (wrapper) který přizpůsobuje rozhraní existující třídy požadavkům nebo očekáváním uživatele. Jestliže např. zakoupíme implementaci AVL-stromu a budeme ji chtít použít v předchozím příkladu, budeme požadovat, aby se metody této třídy jmenovaly `Vloz()` apod. Pokud to nesplňují, „zabalíme“ zakoupenou třídu do pomocné třídy, která převede volání metody `Vloz()` na volání metod zakoupené třídy.
Máme-li k dispozici implementaci dvoustranné fronty, můžeme ji využít k implementaci zásobníku tak, že vytvoříme adaptér, který dvoustrannou frontu „zabalí“ a zveřejní jen ty z jejích metod, které patří do rozhraní zásobníku.
Návrhový vzor *skladba* (composite) umožňuje skládat objekty do stromových struktur a tak efektivně vyjadřovat vztahy mezi částí a celkem. Nabízí cestu, jak jednotným způsobem zacházet s celkem i s jeho částmi. Typickým příkladem může být vyjádření obrázku

v grafickém editoru: Obrázek se skládá z úseček, obdélníků, kružnic atd.; často ale potřebujeme, aby obrázek mohl v sobě obsahovat také jiný obrázek a aby uživatel mohl s tímto vloženým obrázkem manipulovat jako s celkem a přitom podobně jako s úsečkami či kružnicemi.
Vzor *dekorátor* použijeme, jestliže potřebujeme k objektu dynamicky připojovat další funkčnost a nepokládáme za rozumné užít dědičnost. (Ta vede ke statické změně funkčnosti.) Dekorátor např. umožní za běhu připojit k tlačítku v grafickém uživatelském rozhraní zvláštní ohraničení (tedy změnit způsob zobrazení). Pro klientskou část programu ovšem zůstane třída „ozdobená“ dekorátorem stejná jako předtím. Setkáme se s ním často mj. v knihovnách pro tvorbu GUI.
Vzor *fasáda* (facade) poskytuje jednotné rozhraní pro přístup k subsystému, typicky tvořenému velkým množstvím dalších objektů. Usnadňuje tak použití tohoto subsystému.
Vzor *muší váha* (flyweight) se uplatní, jestliže v systému máme velké množství „jemnozrnných“ objektů. Typickým příkladem mohou být reprezentace znaků v textovém editoru. Objekty muší váhy mají jednak vnitřní stav, který je reprezentován daty objektu, a za druhé mají vnější stav, který je dán kontextem použití. (Vnitřní stav může být např. hodnota znaku, vnější stav – velikost písma, zda jde o normální písmo, kurzivu nebo tučné písmo – je dán kontextem použití.
Vzor *zástupce* (proxy) poskytuje náhradníka za jiný objekt a tak umožňuje řídit přístup k zastupovanému objektu. Tento vzor se používá v různých variantách; podívejme se na některé z nich:

- V distribuovaných aplikacích (COM, CORBA) se setkáme se *vzdáleným zástupcem*. To je „místní zástupce“ vzdáleného objektu – serveru – v adresovém prostoru klienta. Klient používá zástupce, jako kdyby to byl server, a zástupce zprostředkovává komunikaci se skutečným serverem.
- Ochranný zástupce umožňuje např. kontrolu správnosti přístupu. Do této kategorie patří např. „automatické ukazatele“, používané v C++ (implementované mj. prostřednictvím šablony `auto_ptr<>`).

Vzory chování

Tyto vzory se zabývají algoritmy, rozdělováním povinností mezi objekty, způsoby jejich spolupráce atd. Můžeme se na ně dívat jako na nástroj, který posunuje pozornost programátora od procesu řízení ke způsobu propojení mezi objekty. Popisují ovšem nejen způsoby interakcí mezi objekty, ale i pomocné objekty, které takováto propojení realizují.

Příklad: iterátor

S *iterátory* (označovanými také *enumerátory* nebo *kurzory*) se dnes setkáme nejen ve standardní šablonové knihovně jazyka C++, ale i v knihovnách mnoha jiných jazyků, jako je Java nebo C#. Na iterátor se můžeme dívat jako na abstrakci zobecňující ukazatele na prvky pole, seznamu nebo jiné posloupnosti (tedy kontejneru, ve kterém jsou prvky seřazeny nějakým způsobem jeden za druhým – např. podle pořadí, v jakém do něj byly vloženy).
O kontejneru, se kterým iterátor pracuje, hovoříme jako o *podkladovém* kontejneru.
Typický iterátor:

- Poskytuje přístup k uloženému prvku – ve standardní knihovně jazyka C++ pomocí přetížených operátorů `*` nebo `->`, v C# pomocí vlastnosti `Current`, v Javě můžeme použít odkaz vrácený metodou `next()`.
- Obsahuje operaci, která umožňuje přejít na následující prvek posloupnosti. V C++ je to přetížený operátor ++, v Javě k tomu slouží metoda `next()`, v C# metoda `Next()`.

- Umožňuje nastavit iterátor na počátek kontejneru. Např. ve standardní knihovně jazyka C++ obsahuje řada kontejnerů metodu `begin()`, která vrací iterátor ukazující na první prvek. V Javě ukazuje nově vytvořený iterátor na první prvek kontejneru, v C# ukazuje před první prvek.
- Poskytuje metodu, která z podkladového kontejneru odstraní prvek, na který tento iterátor právě ukazuje. (Tím se ovšem může narušit platnost ostatních iterátorů, které pracují na podkladovém kontejneru.)

Podle druhu posloupnosti, na níž pracuje, může iterátor poskytovat také operace pro přechod na předchozí prvek, pro náhodný přístup k prvkům kontejneru atd.
Iterátory umožňují implementovat mnohé algoritmy nezávisle na druhu kontejneru. Ve standardní knihovně C++ jsou iterátory implementovány pomocí šablon. To nám umožňuje napsat univerzální algoritmus (tedy funkci), která provede určitou operaci (téměř) nezávisle na druhu kontejneru. Jestliže např. máme frontu a seznam,

```
#include <list>
#include <deque>
#include <iostream>
using namespace std;

int main()
{
     list<int> li;
     deque<double> qi;
     // a další kód
}
```

můžeme napsat univerzální funkci `Vypis()`, která vypíše obsah zadaného úseku posloupnosti do zadaného datového proudu:

```
template<typename T>
void Vypis(T beg, T end, ostream& Proud)
{
     for(T i = beg; i != end; i++)
          Proud << *i << endl;
}
```

Obsah celého obou kontejnerů, `li` a `qi`, vypíšeme příkazy

```
Vypis(li.begin(), li.end(), cout);
Vypis(qi.begin(), qi.end(), cout);
```

Poznamenejme, že `end()` je metoda, která vrací iterátor ukazující za poslední prvek kontejneru.
Podobným způsobem je v STL implementována řada funkcí – připomeňme si např. `sort()`, `remove_if()`, `copy()` atd.
Dodejme, že v jazycích jako je Java nebo C#, lze k napsání podobné funkce využít skutečnosti, že všechny iterátory implementují jisté rozhraní.

Poznámka

Iterátory bývají občas přirovnávány k indexu pole – zejména při výkladu v programovacích jazycích, které nezavádějí ukazatele. Je tu ale jeden významný rozdíl: Index je anonymní v tom smyslu, že neurčuje kontejner, o jehož prvek jde. Například číslo 1 určuje první prvek v jakémkoli poli nebo jiném indexovatelném kontejneru. Naproti tomu iterátor je vázán na kontejner, určuje konkrétní prvek v konkrétním kontejneru.

Další vzory chování

Návrhový vzor *řetěz odpovědnosti* (chain of responsibility) umožňuje vyhnout se přímému spojení objektu, který odesílá nějakou zprávu, s příjemcem této zprávy tím, že zprávu předává posloupnosti („řetězu“) objektů. Tyto objekty si zprávu předávají, až ji jeden z nich zpracuje. (Trochu to připomíná způsob, jakým se v operačním systému DOS ošetřovala přerušení: Programátor definoval novou proceduru pro obsluhu přerušení, v níž toto přerušení „odchytil“ a podíval se, zda jde o případ, který ho zajímá. Pokud ano, zpracoval ho, pokud ne, zavolal původní funkci pro obsluhu přerušení.)

Vzor *příkaz* (command), označovaný také *transakce* nebo *akce*, zapouzdří zprávu vyjadřující žádost nebo příkaz do objektu. To umožňuje parametrizovat klienty pomocí různých typů příkazů, ukládat příkazy do fronty nebo zásobníku (a tak např. usnadní implementaci příkazu *Zpět* v různých editorech) atd.

Vzor *interpret* se používá, jestliže potřebujeme interpretovat data představující vyjádření jisté formální gramatiky. To možná zní nestravitelně, jde ale o poměrně častý problém, na který narazíme např. při vyhledávání podřetězců ve znakových řetězcích pomocí regulárních výrazů, v překladačích programovacích jazyků atd.

Jestliže potřebujeme zachytit vnitřní stav objektu, tedy přenést ho ven („externalizovat“ ho), a nechceme přitom porušit zapouzdření, použijeme návrhový vzor *obnovitel* (memento). Uplatní se mj. při implementaci funkce *Zpět* v editorech a podobných programech.

Velmi rozšířený je návrhový vzor *pozorovatel* (observer), známý také pod jmény *vydavatel-předplatitel* (publisher/subscriber). Setkáme se s ním mj. v Javě nebo v C#, kde se standardně používá při zpracování událostí. Je také základem architektury dokument-pohled (document-view), známé nejen z knihovny MFC. Jde o to, že instance, která může způsobit nějakou událost (označovaná jako zdroj události, např. tlačítko v GUI) si vede seznam instancí, které mohou na tuto událost reagovat (pozorovatelů). Pokud událost nastane, zdroj vyrozumí všechny pozorovatele – pošle jim zprávu, tj. zavolá odpovídající metodu.

Návrhový vzor *stav* (state) umožňuje objektu změnit chování, když se změní jeho vnitřní stav.

Jestliže potřebujeme v určitých situacích změnit algoritmus, používaný pro řešení jistého problému, vezmeme vzor *strategie* (strategy), označovaný také jako *politika* nebo *zásady* (policy). Dosti podobný je i vzor *šablonová metoda* (template method), který definuje kostru algoritmu a odkládá některá rozhodnutí na podtřídy.

Při práci s kontejnery se často uplatní vzor *návštěvník* (visitor). Někdy totiž předem nevíme, co budeme s daty, uloženými v kontejneru, dělat. Například syntaktický strom, vzniklý analýzou aritmetického výrazu, můžeme interpretovat, můžeme ho použít ke generování strojového kódu atd. Abychom nemuseli implementovat všechny možnosti spolu se stromem, a abychom také později mohli snadno doplnit možnost, o níž v době prvního návrhu nemáme ani tušení, použijeme vzor návštěvník. Je založen na metodě, kterou nazveme třeba `accept()` a kterou definujeme v prvku datové struktury. Tato metoda „přijme“ návštěvníka, tedy objekt, který s prvkem provede vše potřebné.

Na závěr

Pokud jste dočetli až sem, začínáte mít představu, co to návrhové vzory vlastně jsou. A pokud se mi podařilo vzbudit vaši zvědavost, budu rád, neboť znalost a používání těchto vzorů může ušetřit mnoho práce.

Odkaz:

[1] Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides: *Design Patterns*. Addison-Wesley 1995. K dispozici je i český překlad (bohužel nepříliš kvalitní).