ASPEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Marek Pícka

Katedra informačního inženýrství, PEF ČZU, Kamýcká 129, 165 21 Praha 6 - Suchdol, ČR
picka@pef.czu.cz

Abstrakt
Aspektově orientované programování se snaží řešit problém, kdy některé funkce systému, typickým příkladem je logování, nelze modularizovat pomocí stávajících technik. Tyto funkce jsou rozptýleny v mnoha modulech, což snižuje kvalitu kódu, zhoršuje znovupoužití kódu, prodražuje vývoj. Aspektově orientované programování tyto problémy se snaží řešit pomocí modularizace těchto záležitostí do aspektů. Nejpoužívanějším aspektově orientovaným jazykem dnešní doby je AspectJ.

1. Úvod

Původně se programy pro počítače vyvíjeli ve strojovém kódu, to sice vedlo k efektivním programům, ale za cenu pomalého vývoje – vývojář musel přemýšlet na úrovni instrukční sady procesoru. Následně se začali používat vyšší programovací jazyky, které už umožňovaly lepší abstrakci. Potom se objevily strukturované jazyky, které rozkládaly systém na procedury, řešící jednotlivé problémy. Každé nové programovací paradigma umožňovalo přemýšlet na abstraktnější úrovni a tím bylo možno řešit složitější úlohy. V současné době je nejpoužívanější objektově orientované paradigma (OOP), které systém rozděluje na třídy, které ukrývají svojí implementaci. Třídy jsou díky polymorfismu komunikovat s okolím, aniž by se třídy musely ně příliš starat. OOP ukazuje svou sílu zejména při modelování obvyklého chování objektů. Má však potíže s chováním, které je rozprostřeno mezi mnoho modulů (tříd). Aspektově orientované programování (AOP) se snaží řešit tyto problémy. AOP je žhavým kandidátem na nové, obecně přijímané paradigma.

2. Principy aspektově orientovaného programování

Na složitý softwarový systém se můžeme dívat jako na složenou implementaci mnoha záležitostí[1]. Typický systém se skládá z několika druhů záležitostí jako jsou například obchodní logika, výkonnost, perzistence dat, logování, bezpečnost, debugování, ošetření chyb atd. Lze se setkat i s záležitostmi týkajícími se samotného vývojového procesu jako jsou srozumitelnost, snadnost použití, udržovatelnost, rozšiřitelnost atd.

Některé z těchto záležitostí (jsou to typicky pomocné problémy), jako jsou například logování, bezpečnost atd., jdou špatně zachytit pomocí klasických způsobů návrhu. Při klasických způsobech návrhu (objektově, strukturovaně) jsou tyto záležitosti rozptýlené v mnoha modulech, jsou to tzv. crosscutting concerns (česky asi nejlépe protínající, nebo rozptýlené záležitosti). Řešit problém těchto protínajících záležitostí se snaží, jejich modularizací, řešit aspektově orientované programování (AOP – viz. [1] a [6]).

[1] Záležitost (concern) je jednotlivý cíl, koncept nebo oblast zájmu.

2.1 Problémy protínajících záležitostí (crosscutting concerns)

Vývojář vytváří systém na základě požadavků, které lze rozdělit na základní (týkající se typicky hlavních funkcí systému – někdy se jim říká module-level) a pomocných (někdy se jim říká system-level). Tyto pomocné požadavky jsou typicky nezávislé na ostatních základních a pomocných požadavcích. Tyto pomocné požadavky (a z nich odvozrné záležitosti) mají tendenci být rozptýlené v mnoha modulech (crosscutting concerns).

Ačkoliv protínající záležitosti (crosscutting concerns) se vyskytují v mnoha modulech, dnešní implementační techniky implementují tyto požadavky pomocí metodologií, které se snaží mapovat požadavky do jedné dimenze. Tato dimenze je typicky vytvořena pomocí základních funkcí systému. Ostatní záležitosti v systému jsou roztroušeny v této základní dimenzi. Jinými slovy jsou požadavky na systém vícedimenzionální, zatímco metodologie nám poskytují pouze jednu hlavní dimenzi. To vede k těmto dvěma hlavním problémům:

- *K zmotání požadavků dohromady* – modul v systému může vyplňovat více požadavků a tak musí vývojář najednou myslet na obchodní logiku, výkon, synchronizaci, bezpečnost, logování atd. a tím se znepřehledňuje návrh.
- *K rozptýlení požadavků* – protože jsou požadavky rozptýleny v mnoha modulech, tak je jejich implementace také rozptýlena v mnoha modulech. Například pokud systém používá databázi, tak jsou výkonové záležitosti, rozptýleny v mnoha modulech a tak ja lze obtížně řešit.

Kombinace těchto dvou problémů se projevuje v těchto problémech:

- *Horší produktivita* – soustředění se na více požadavků naráz odvádí návrháře od hlavních problémů k vedlejším.
- *Špatná kontrola splnění požadavků* – protože je více požadavků implementováno v jednom modulu, je těžší kontrola jejich splnění.
- *Menší znovupoužití kódu* – protože je více záležitostí implementováno v jednom modulu, tak je tento modul méně znovupoužitelný (např. jedna záležitost implementovaná modulem nám nevyhovuje).
- *Špatná kvalita kódu* – všechny předchozí problémy zapříčiňují horší kvalitu výsledného systému.
- *Obtížnější evoluce* – přidání dalšího požadavku na systém způsobuje obtížné přepisování kódu.

2.2 Základy AOP

Z předchozích částí článku vyplývá, že je užitečné modularizovat protínající záležitosti. Jedním z řešení je aspektově orientované programování.

AOP zahrnuje tyto tři kroky:

1. *Dekompozice aspektů* – dekompozice požadavků kvůli identifikaci obecných a protínajících záležitostí (crosscutting concerns). Potom se musí oddělit obecné záležitosti (ty jdou typicky rozdělit do modulů pomocí klasických metod) od protínajících (vyskytují se přes mnoho modulů). Na příkladu kreditní karty lze nalézt tyto záležitosti: vlastní zpracování kreditní karty, logování a autentizaci.
2. *Implementace záležitostí (concern)* – implementace každé nalezené záležitosti zvlášť. Pro náš případ to bude modul vlastního zpracování kreditní karty, logovací modul a modul provádějící autentizaci.

3. *Rekompozice aspektů* – v tomto posledním kroku musíme spojit dohromady vytvořené moduly pomocí aspektů. Tomuto kroku, který vytváří z modulů výsledný systém, se říká integrace nebo splétání (weaving). Pro příklad kreditní karty musíme specifikovat, že každý začátek a konec operace s ní bude logován, a každá operace musí být povolena.

AOP se nejvíce liší od OOP způsobem adresování protínajících záležitostí (crosscutting concerns). Při použití AOP žádná záležitost neví že ostatní na ní berou ohled. V našem příkladě kreditní karta neví, že před každou operací je ověřována a že všechny operace jsou logovány.

Implementace AOP může použít jinou technologii (tj třeba strukturované nebo objektové programování) jako svůj základ, který je použitý pro rozdělení obecných záležitostí do modulů (tj. v našem případě použít pro zpracování kreditní karty OOP) a pomocí aspektů modularizovat pouze protínající záležitosti.

2.3 Výhody AOP

AOP pomáhá řešit problémy způsobené zmotáním a rozptýlením záležitostí v kódu. Zde jsou uvedeny přínosy AOP:

- *Modularizace implementace protínajících záležitostí* – AOP tím dokáže separovat jednotlivé protínající záležitosti a to vede k jednoduššímu návrhu.
- *Snadnější vývoj systému* – oddělením protínajících záležitostí vede k jednoduššímu kódu s oddělenými požadavky.
- *Možnost učinit rozhodnutí později* – AOP umožňuje odložit některá rozhodnutí, která by případně mohla vést v velkým změnám v designu, na pozdější dobu. Mnoho záležitostí lze přidat odděleným aspektem.
- *Lepší znovupoužitelnost kódu* – oddělením protínajících záležitostí bude návrh jednodušší a mnohem generičtější a lze ho pak lépe znovupoužít.

3. Aspektově orientovaný jazyk AspectJ

Jazyk AspectJ (více [2], [5]) se vyvíjí v laboratořích PARC od poloviny 90 let (První dostupná verze v roce 1998, dnes aktuální verze1.1.1). Tento jazyk představuje jednoduché a mocné rozšíření Javy o možnost zachytit a modularizovat protínající záležitosti (crosscuting concerns) . Jednou z největších výhod AspectJ je binární kompatibilita s klasickou Javou – tj. lze používat neupravené virtuální stroje a lze rozšiřovat javovské programy, které máme jen přeložené do bajtkódu.

Pro podporu AOP AspectJ přidává do Javy pouze těchto několik konceptů:

- *Joinpoint* – (pokus o český překlad – bod vstupu) je dobře definovaný bod v průběhu provádění programu. Například joinpoint může být definován voláním určité metody.
- *Pointcut* – () sdružuje několik joinpointů a shromažďuje informace o jejich kontextu.
- *Advice* – (česky pokyn, doporučení) kód, který je proveden, když program projde joinpointem, například advice může logovat zprávy před (nebo po) projití joinpointem.
- *Inter-type declaration* – (česku mezi-typové deklarace) dovoluje programátorovi modifikovat statickou strukturu programu, jako jsou atributy třídy a vztahy (například dědičnost) mezi třídami.

- *Aspect* – (česky aspekt) je jednotkou modularity v AOP. Aspekt se chová podobně jako javovská třída, ale navíc může obsahovat konstrukce aspektově orientovaného programování (tj. pointcuty, advice a vnitro-typové deklarace).

Zatím co joinpointy, pointcuty a doporučení jsou dynamické konstrukce (tj. provádějí a vyhodnocují se až za běhu programu), vnitro-typové deklarace jsou statické – vyhodnocují a provedou se v době překladu.

3.1 Hello World příklad

Pro prvním seznámení s jazykem je zvykem uvést příklad HelloWorld. Aby se zvykům učinilo za dost, příklad následuje.

Nejdříve si nadefinujeme jednoduchou třídu, která obsahuje metody pro zobrazování zpráv.

```
// HelloWorld.java
public class HelloWorld {
    public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```

Dále je uveden příklad aspektu, který zajistí dobré chování třídy HelloWorld. Před vytištěním každé zprávy program vytiskne pozdrav (`"Good day!"`) a po ní se slušně poděkuje (`"Thank you!"`).

```
// GoodMannersAspect.java
public aspect GoodMannersAspect {
    pointcut callSayMessage() : call(public static void
HelloWorld.say*(..));

    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
```

Soubor `GoodMannersAspect.java` má podobnou strukturu jako třída v Javě. Je v něm deklarován aspekt `GoodMannersAspect`. Aspekt definuje pointcut `callSayMessage()`, který zachycuje všechna volání public static void metod třídy `HelloWorld`, začínající na say s jakýmikoliv argumenty. V našem případě to jsou metody `say()` a `sayToPerson()`. Dále jsou definovány dva pokyny (advice), co má aspekt dělat. Před vyvoláním metod definovaných v pointcutu (příkaz `before()`) je vytiskne `"Good day!"` a po vyvolání metod (příkaz `after()`) se vytiskne `"Thank you!"`.

3.2 JoinPointy

JoinPointy, které jsou hlavní koncept AspectJ, jsou dobře definovaná místa v programu – tj. kandidáty jsou volání metod, přístupy k proměnným, testy podmínek, začátky cyklů, přiřazení atd. Každý joinPoint má svůj kontext, například joinpoint týkající se volání metod zná svůj objekt a vstupující argumenty.

AspectJ rozeznává tyto joinpointy:

- volání a provádění metod,
- volání a provádění konstuktorů,
- přístupy (jak čtení i zápis) k proměnným,
- vyvolání výjimek,
- inicializace tříd a objektů.

AspectJ nepodporuje jemněji definované joinpointy nacházející se uvnitř metod, jako jsou například testy, nebo začátky cyklů.

3.3 Pointcuty

Pointcuty jsou programové konstrukce, které definují joinpointy a specifikují jejich kolekce. Dále také mohou získat kontext joinpointu a předat ho dál ke zpracování.

V předchozím případě jsme použili definici pointcutu:

```
pointcut callSayMessage() :
   call(public static void HelloWorld.say*(..));
```

Ta obsahuje jméno pointcutu (`callSayMessage()`). Prázdné závorky () vyjadřují, že pointcut nepotřebuje žádný kontext – jinak řečeno, že pokyny (advice) reagující na tento pointcut namají žádné vstupní parametry. Dále část `call(public static void HelloWorld.say*(..))` se skládá z určení typu jointpointu – `call` znamená, že join point je definován při volání metody po vyhodnocení jejích vstupních argumentů, a z `public static void HelloWorld.say*(..)` což je signatura metody – v tomto případě všechny public static void metody třídy HelloWorld zažínající na say s jakýmikoliv parametry. Je možno použít i zástupné znaky (hvězdičku) a dvě tečky u parametrů. Pokud bychom potřebovali definovat více pointcutů, udělalo by se to pomocí spojky or (||), složitější podmínky se dají definovat i pomocí and (&&) a not (!). Následující příklad ukazuje, jak spojit více joinpointů dohromady a plenou definici signatury metody.

```
pointcut callSayMessage() :
   call(public static void HelloWorld.say(String)) ||
   call(public static void HelloWorld.sayToPerson(String,String));
```

Zástupné znaky se mohou používat i k určení tříd i návratových hodnot. Například pokud bychom potřebovali do pointutu zařadit všechny metody začínající na say* všech tříd, použili bychom `call(*.say*(..))`.

Velmi podobné direktivě `call()` u pointcutu je `execution()`. Rozdíl proti call je, že je joinpoint definován již před vyhodnocením argumentů Například

```
execution( * Foo *(..) throws IOException)
```

definuje joinpoint, který zahrnuje všechny metody třídy `Foo`, které vyhazují výjimku `IOException`.

Dalšími důležitými joinpointy jsou ty, které se týkají přístupu k proměnným (třídním nebo instančním). Například joinpoint

```
get(int Foo.x)
```

se týká všech čtení proměnné `x` třídy `Foo`. Pro hlídání zápisu lze použít pointcut `set()`. Například

```
set( !private * Foo.*)
```

se týká všech zápisu do neprivátních proměnných třídy `Foo`.

Dalším důležitým pointcutem je typ `handler()`, týkající se vyvolání výjimky s damou signaturou. Například

```
handler(IOException+)
```

se týká všech výjimek IOException a z ní odvozených podtypů.

Joinpoint může být definován i pomocí spuštěného řídícího toku instrukcí (control-flow). Například pokud metoda a() volá metodu b(), tak do toku instrukcí patří i metoda b() – tj joinpoint můžeme definovat pro volání metody a() a všech z nich volaných metod (tj. a() i b()) pomoci konstrukce `cflow()` nebo jen metod z ní volaných (tj jenom b()) pomocí konstrukce `cflowbelow()`. Pomocí těchto konstrukcí můžeme definovat joinpoint i v pokynu (advice). Tj.

```
cflow(callSayMessage())
```

bude určen, když programový tok vstoupí do pokynu (advice) `callSayMessage()`.

Další možností určení joinpointu je podmínka. Joinpoint

```
if( booleanExpression)
```

bude splněn, pokud bude výraz `booleanExpession` splněn.

Poslední skupinou jsou joinpointy určené pomocí konstrukcí `this()`, `target()` a `args()`. Konstrukce

```
this(MyClass+)
```

určuje joinpointy, v kterých platí, že `this` je instancí `MyClass` a jejich potomků. Konstrukce

```
target(MyClass)
```

určuje joinpointy u kterých je objekt, kterého metoda je volána, třídy `MyClass`. A konstrukce

```
args(String,..,int)
```

určuje joinpointy, kde první argument metody je typu `String` a poslední je typu `int`.

Pomocí těchto konstrukcí lze předávat kontext (tj cíl a argumenty) do pokynů (advice). Pokud se vrátíme k našemu HalloWorld příkladu, potom konstukce

```
pointcut callSayMessageWithParams(String name, String message) :
   call(public static void HelloWorld.sayToPerson(String,String)) &&
        args(String message, String name);
```

předává parametry `message` a `name` do pokynu (advice) `callSayMessageWithParams` s parametry `name` a `message` třídy `String`. Podobně lze předávat kontext pomocí `this` a `tagret`.

Podrobnější popis možností vytváření pointcutů lze najít v The AspectJ Programming Guide [4].

3.4 Pokyny (advices)

Zatímco pointcut definuje místa v průběhu vykonávání programu, pokyn (advice) definuje co, kdy a místo čeho se má stát. AspectJ má tři možnosti – `before`, `after` a `around`.

Při použití označení before se pokyn (advice) vykoná před místem označeným joinpointem (např. při po joincutu `call`, se advice provede před voláním metody).

Při označení after se pokyn (advice) provede po skončení metody. Protože z Javě může metoda skončit buďto po příkaze `return` nebo výjimce, máme také modifikace `after() returning` a `after() throwing`.

Při použití `around` se vykoná pokyn (advice) místo volané metody. Například pokud budeme chtít modifikovat náš `HelloWorld` příklad, aby byl slušný i v Japonsku (přidáme za každé jméno příponu -san) přidáme k již vytvořeným částem (tj. `HelloWorld.java` a `GoodMannersAspect.java`) nový aspekt:

```
// JapaneseGoodMannersAspect.java
public aspect JapaneseGoodMannersAspect {
    pointcut callSayMessageToPerson(String person)
        : call(* HelloWorld.sayToPerson(String, String))
              && args(*, person);

    void around(String person)
        : callSayMessageToPerson(person) {
        proceed(person + "-san");
    }
}
```

V tomto aspektu se místo metody `sayToPerson` vykoná náš pokym (advice), která přidá příponu –san za jméno. Pomocí příkazu `proceed(String person)` znovu spustíme metodu `sayToPerson`, tentokrát už s japanizovaným jménem.

3.5 Mezi-typové deklarace

Mezi-typové deklarace (inter-type declaration) jsou deklarace, které modifikují strukturu napříč tříd a jejich hierarchiemi. Zatímco předchozí konstrukty byly dynamické, jsou mezi-typové deklarace statické – jsou vyhodnoceny již při překladu. Například můžeme přidávat do tříd (nebo interfaců) nové metody a atributy. Zde přidáváme metodu `foo` a statickou proměnnou `instanceCount` do třídy `MyClass`.

```
aspect IntroduceMethodExample {
        private void MyClass.foo() {
                System.out.println("This is foo");
        }

        private static int MyClass.instanceCount = 0;
}
```

Dále můžeme modifikovat strukturu třídní hierarchie změnou předka nebo přidáním interfacu do třídy. Na tomto malém příkladě přidáváme třídě `MyClass` rozhraní `Serializible`.

```
aspect MakeMyClassSerializable {
        declare parents : MyClass implements Serializable;
}
```

Dále můžeme měnit klasické výjimky vyvolané v aspektu na výjimky třídy `org.aspect.lang.SoftException` – ty jsou definovány jako podtřída `RuntimeException`, které nemusí být deklarovány. Dále můžeme zajistit, aby daný join point nebyl nikdy použit pomocí deklarace chyby nebo varování. Dále můžeme deklarovat preference mezi aspekty (tj. jaký aspekt se bude vykonávat dřív a jaký později). Více o mezitypových deklaracích v [4].

3.6 Aspekty

Aspekty hrají roli jednotek modularizace, podobnou jako třídy v Javě. Aspekty dávají dohromady pointcuty a pokyny (advice). Aspekty se podobají třídám i v jiných ohledech. Aspekty mohou obsahovat proměnné i metody, mohou dědit z ostatních tříd a aspektů a implementovat rozhraní. Nicméně se aspekty liší od tříd tím, že nemůžeme vytvářet instance pomocí konstruktoru `new`.

V AspectJ mohou třídy definovat pointcuty, no pouze jako statické prvky. Třídy nemohou obsahovat pokyny (advice). Aspekt a pointcut v něm obsažený může být označen jako abstraktní – tj. nechává implementaci na svých potomcích.

Aspekty mají několik modifikátorů:

- pokud je aspekt privilegovaný (`privileged aspect A { ... }`), potom může aspekt A přistupovat k privátním prvkům

- aspekt může dědit z třídy i implementovat interfacy – `aspect A extend B implements I,J`.
- aspekt může vytvářet pokaždé novou instanci po projití joinpointem – `aspect A percflow ( call(void Foo.m()))`.

4. Závěr

Aspektově orientované programování řeší problémy s rozptýlenými záležitostmi, které jsou pomocí klasických metod špatně modularizovatelné.Tím umožňují jednodušší, snadněji udržovatelný, znovupoužitelný a čistější návrh.V současné době se principy AOP nejlépe používají spolu s objektově orientovaným návrhem, kdy základní funkčnost a základní modularizace je řešena objektově a rozptýlené záležitosti jsou řešeny pomocí aspektů.

V současné době je nejrozšířenějším aspektově orientovaným jazykem AspectJ, který je mocným rozšířením objektově orientovaného jazyku Java. AspectJ přidává do Javy pouze jeden základní koncept (joinpoint) a čtyři nové jazykové konstrukce (pointcut, advice, inter-type declaration a aspect). Jednou z hlavních výhod jazyku AspectJ je jeho kompatibilita s klasickou Javou (virtuální stroj je kompatibilní, Javovský program je programem v AspectJ), která umožňuje postupný přechod k aspektům.

Literatura:

1. Kiczales, G., at al.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Findland (1997)
2. Kiczales, G., at al.: An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2001
3. Laddad, Ramnivas. I want my AOP! . JawaWorld, 2002, č. 1, 3, 4. JavaWorld.com, an IDG company
4. The AspectJ Team: AspectJ Programming Guide. http://aspectj.org
5. The AspectJ Website. http://aspectj.org
6. The aspect-oriented software development Website. http://aosd.net