ASPEKTOVÉ PROGRAMOVÁNÍ V JAZYKU PYTHON

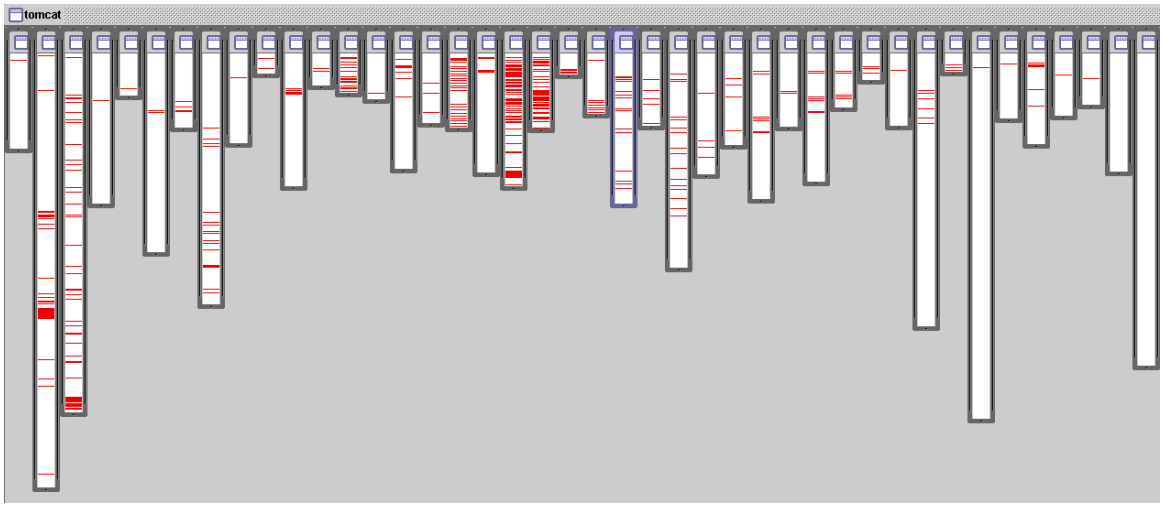
Marek Pícka
Česká zemědělská univerzita v Praze, Provozně-ekonomická fakulta, katedra informačního inženýrství, picka@pef.czu.cz

ABSTRAKT:
Tento článek pojednává o aspektově orientovaném programování, o problémech, které k němu vedly a o jeho základních principech. Dále tento článek ukazuje různé možnosti implementace aspektů a zaměřuje se na nejčastější způsob implementace pomocí kompozice filtrů. V druhé části je článek zaměřuje na implementaci aspektově-orientovaného programování v jazyce Python a uvádí jednoduché příklady implementace aspektů.

KLÍČOVÁ SLOVA:
Aspekt, Aspektově orientované programování (AOP), Python

ÚVOD

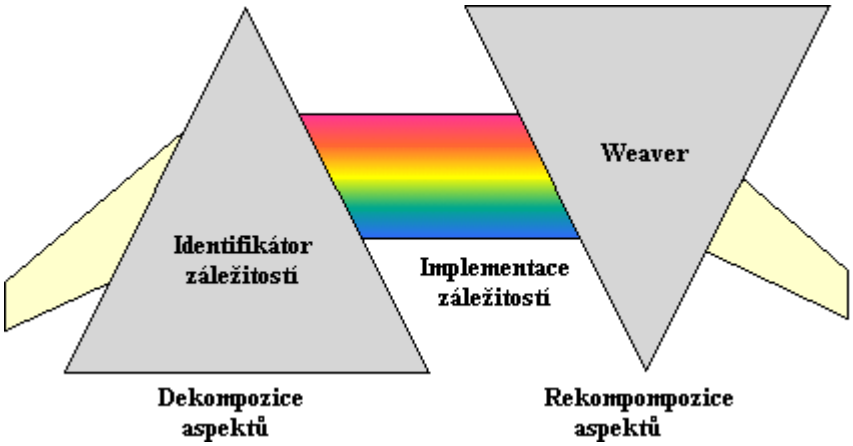
Myšlenky aspektově-orientovaného programování (AOP) vznikly cca v polovině 90. let jako reakce na problémy s modularizací některých funkcí programu při použití standardních postupů objektově-orientovaného programování. Asi nejtypičtějším příkladem je logování. Potřebujeme logovat spousty věcí na spoustě místech programu. Na obrázku 1 je zachyceno, kde všude je v aplikačním serveru Tomcat potřeba logovat. Je vidět, že logování je *rozptýleno* po celém programu mezi různé třídy. O takovýchto funkcích programu, které nejde rozumně modularizovat a jsou rozházené po celém programu se říká, že jsou tzv. „crosscutting concerns“ (česky asi nejlépe rozptýlená, roztroušená záležitost, vlastnost, další možná anglická synonyma jsou feature, behaviour). Aspektově-orientované je takový pokus, jak provést modularizaci takovýchto rozptýlených vlastností.



Obrázek 1. Logování v aplikačním serveru Tomcat (převzato z www.apache.org)

PRINCIPY ASPEKTOVĚ ORIENTOVANÉHO PROGRAMOVÁNÍ

Jako analogie aspektového programování se uvádí hranol (viz obrázek 2). Nejdříve se problém rozdělí na jednotlivé problémy, které jsou mezi sebou co nejvíce nezávislé (anglický



Obrázek 2. Analogie AOP jako hranolu

termín zní – separace záležitostí (separation of concerns), tento výraz zavedl E. W. Dijkstra ve svém článku O roli vědeckého myšlení - viz [1]) . AOP nám umožňuje provést tuto separaci lépe než pomocí jiných programovacích paradigmat – postihneme i roztroušené vlastnosti. V „separovaném stavu“ vytvoříme zdrojový kód programu. Druhý hranol naopak skládá tyto záležitosti dohromady. Tento hranol je v AOP realizován speciální částí virtuálního stroje (anglicky weaver) aspektově-orientovaného jazyka, která za běhu splétá (nebo spřádá, anglický termín je weaving) tyto oddělené části zase dohromady.

Možnosti realizace AOP

AOP se realizuje několika základními přístupy (viz [21]). Vyjmenujme si ty nejdůležitější:

- Kompozicí filtrů – aspekty jsou implementovány jako wrappery (obaly) a adaptéry metod. Toto je asi nejběžnější způsob implementace AOP. Viz například AspectJ [9].
- Subjektově orientovaným programováním – základním konceptem je transformace problému do „strukturálních aspektů“, které mohou být spojeny s ostatními aspekty (jako jsou logika zpracovávané domény, uživatelské rozhraní, ukládání dat atd.). Dohromady tyto aspekty tvoří aplikaci. S tímto přístupem přišla firma IBM (viz [7]). Jako ilustrativní příklad použití lze uvést vytváření nových tříd v aspektově orientovaném prostředí TransWarp (viz [16]): `MyClass = (domainLogicAspect + structuralAspect + guiAspect + …)().`
- Adaptivní programování (Demeter) – základním přístupem je rozdělení programu na část definující objekty a část definující operace s nimi. Cílem je udržovat co nejmenší soudržnost mezi nimi (změnou jednoho se nezmění druhé). Adaptivní programování je speciální příklad AOP, kde se pro zachycení souvislostí mezi objekty a operacemi používají grafy. Více o projektu Demeter se lze dočíst v [8].

Ideální stav (z hlediska AOP) by byl, kdyby se používalo tzv. čisté AOP. Hlavní záležitosti (core concern) programu (tj. business logika) a rozptýlené záležitosti (cross-cutting concerns) jsou aspekty. Dnes se ale používá AOP v kombinacemi s jinými styly programování, tj. core concerns jsou vyjádřeny typicky objekty (ale také i pomocí procedur – například AspectC – viz [10]).

Dále se budu věnovat zejména kompozici filtrů, protože je to nejpoužívanější (a také implementačně nejjednodušší) přístup k AOP.

AOP založené na kompozici filtrů

Tento způsob implementace aspektů je nejběžnější. Používá ho první (a také nejrozšířenější) implementace aspektů – AspectJ (viz [3]). Tento typ implementace aspektů je založen na tom,

že vykonávání programu v nějakém určitém bodě může být modifikováno pomocí filtru. Tomuto definovanému bodu v programu se říká *joinpoint* a může to být okamžik volání metody, přístup k atributu, provedení výjimky atd. A tomuto filtru, který je proveden před, během, nebo po dosažení *joinpointu*, se říká *advice*. Další používaný termín je *pointcut* a to je sloučení několika *joinpointů* dohromady. V tomto pojetí AOP je *aspekt* struktura, která jednotlivým *pointcutům* přiřazuje daný filtr (advice). Podrobnější vysvětlení těchto termínů je třeba v [6].

Praktický postup je takovýto:

1. Vytvořím aspekt pro každou rozptýlenou záležitost.
2. Určím si místa v programu (tyto místa jsou ty, kde se rozptýlené záležitosti vyskytují) pomocí joinpointů – typicky to jsou metody, atributy atd. Tyto místa typicky určuji pomocí jejich jmen (u některých jazyků můžu použít i hvězdičkovou notaci například `A.me*(..)` by v AspectJ určilo všechny metody třídy `A`, které začínají na `me` a mají libovolné parametry.
3. *Joinpointy* sloučím do *pointcutu* a přiřadím aspektu.
4. Vytvořím filtry (kusy kódu, něco jako metody - *advice*), který se bude provádět těsně před dosažením *joinpointu* (direktiva *before*), nebo místo příkazu definovaného *joinpointem* (*around*), nebo po provedení příkazu definovaného *joinpointem* (*after*).
5. Spojím patřičné filtry a *pointcuty*.

Doporučené použití aspektů

V dnešní době je AOP na tom podobně jako objektově orientované programování před 15 – 20 lety. V odborné veřejnosti se o AOP mluví (ale málokdo ví o co jde), některé nástroje se dostávají do „produkční kvality“ (AspectJ, JBoss atd.), použití v praxi je však poměrně vzácné, neexistují metodiky založené na aspektech, dokonce neexistuje ani standardní modelovací jazyk. Další typickou věcí je to, že existující aplikace jsou spíše programy (typicky objektově orientované) s aspekty než aspektově orientované programy. Z tohoto vychází i doporučený postup použití AOP:

1. Vytvořím si seznam vlastností aplikace.
2. Rozdělím vlastnosti na hlavní (core concerns) a vedlejší (ty jsou typicky rozprostřené – croscutting concerns)
3. Vytvořím si objektový model aplikace, kterým budu implementovat její kostru. Těmto objektům se také někdy říká datové (viz [5]).
4. V tomto okamžiku již lze naprogramovat kostru aplikace. V tomto stavu aplikace sice moc užitečná nebude (bude jí chybět mnoho věcí – zajištění persistence, bezpečnostní záležitosti, logování, pravděpodobně i GUI atd.), ale půjdou na ní spouštět testy.
5. K naprogramované kostře aplikace budu přidávat „vedlejší záležitosti“. Pokud to bude z hlediska modularizace výhodné, tak pro jejich implementaci použiji aspekty. Program po přidání nové vlastnosti by měl být nadále funkční, to ověřuji pomocí testů, které průběžně dopisuji.

JAZYK PYTHON

Python (viz [11]) je jazyk podporující několik paradigmat programování – objektové programování, strukturované programování, funkcionální programování a také pro design by contract. Python má automatickou správu paměti, dynamické typy a dynamické vyhodnocování jmen (jména proměnných jsou za běhu přiřazovány svým hodnotám). Python převzal mnoho vlastností od SmallTalku, ale používá mnohem obvyklejší syntaxi.

Python se snaží být co nejjednodušší. Snaží se jednu věc dělat pouze jedním způsobem (na rozdíl od Perlu). Jeho duch se dá shrnout do vlastností – Python je „krásný“, „explicitní“ a

„jednoduchý“ (více například v [12]). Z těchto důvodů také častokrát je vyučován jako první programovací jazyk a také se používá jako jazyk pro neprogramátory.
O Pythonu se někdy uvádí, že to je skriptovací jazyk. Ve skutečnosti je to jazyk, který je vhodný jak pro rozsáhlé projekty, tak pro malé skripty. Mezi nejznámější projekty napsané v Pythonu patří aplikační server s objektovou databází Zope, klient decentralizované P2P sítě BitTorrent, z her například Civilizace IV. Firma Google (a z českých Seznam) používá Python ve svých projektech.

Aspekty v Pythonu
Nejdřív musím proč chci ukázat principy AOP právě na Pythonu:

1. Je to jednoduchý a elegantní jazyk.
2. Python se častokrát používá jako „spustitelný pseudokód“ pro zápis algoritmů – výpisy jsou srozumitelné i pro ty, kdo ho neovládají.
3. AOP je implementováno jednoduchým a přímočarým způsobem – pomáhá to při pochopení.
4. Výklad AOP na Pythonu jsem použil při výuce (v předmětu, kde také ostatní příklady byly použity v Pythonu).

Implementace aspektů v Pythonu
Aspektově orientované programování je v jazyce Python realizováno pomocí následujících implementací:

1. Aspektový modul laboratoří Logilab – tato implementace (více v [13]) je založena na kompozici filtrů. Aktuální verze je 0.1.4 ze září 2006.
2. Lightweight Python AOP – pokus o co nejjednodušší implementaci AOP do jazyka Python. Poslední verze je 0.4 z března 2007. Více v ([14]).
3. Pythius – tato implementace AOP založená na kompozici filtrů je součástí programového balíku, který slouží k ověřování kvality kódu napsaného v Pythonu počítáním různých metrik). Viz [15].
4. TransWarp – je to programový balík (toolkit) pro snadné programování podnikových aplikací v Pythonu. Vychází z principů blízkým subjektově-orientovanému programování. Více o TransWarpu je na [16].
5. PEAK (Python Enterprise Application Kit) – nástupe Transwarpu. Umožňuje vytvoření podnikové aplikace z komponent. Více viz [17].
6. PyPy AOP – PyPy je speciální překladač, který překládá Python do Pythonu (nebo do jiného vyššího programovacího jakyka – C, C#, Javascript, plánuje se Java atd.) . Výsledkem je, že mohu snadno (i za běhu) modifikovat modifikovat gramatiku jazyka. To mi umožňuje snadno přidávat k Pythonu nové syntaktické konstrukce (například aspekty). Více o implementaci aspektů v PyPy je v [20].

Pro demonstraci AOP v Pythonu požiji aspektový modul od Logilabu, protože je jednoduchou implementací, která ovšem názorně ukazuje základy aspektového programování.

PŘÍKLADY POUŽITÍ ASPEKTOVÉHO MODULU
Použití této knihovny bude nejlepší vysvětlit na příkladech.

Příklad č.1 – Logování
Tento příklad zachycuje snad nejklasičtější případ použití aspektů – logování. Zde zaznamenávám všechna volání sledované metody a vypisuji jméno metody, jaké třídě metoda patří, parametry metody a návratovou hodnotu.

```
class MyClass:
    #privátní atribut
    __jmeno = None
    #konstruktor
    def __init__(self, jmeno = 'Ja'):
        self.__jmeno = jmeno
    def dobryDen(self):
        print 'Dobry den preji,', self.__jmeno
        return 'poprano'
    def dobrouNoc(self, jmeno):
        print 'Dobrou noc preji,', self.__jmeno
        return 'poprano pred spanim'

from logilab.aspects.core import AbstractAspect

class LogAspect(AbstractAspect):
    #advice before
    def before(self, wobj, context, *args, **kwargs):
        metoda = context['method_name']
        trida = context['__class__'].__name__
        print trida,".",metoda,args
    #advice after
    def after(self, wobj, context, *args, **kwargs):
        metoda = context['method_name']
        trida = context['__class__'].__name__
        navrat_hodnota = context['ret_v']
        print trida,".",metoda, "-> ", navrat_hodnota

#vlastní program
pozdrav = MyClass('Marku')
#spojení všech metod instance pozdrav s aspektem LogAspect
from logilab.aspects.weaver import weaver
weaver.weave_methods(pozdrav, LogAspect)

pozdrav.dobryDen()
#odpojení všech metod instance pozdrav od aspektu LogAspect
weaver.unweave_methods(pozdrav, LogAspect)
```

Výstup programu je:

```
MyClass . dobryDen ()
Dobry den preji, Marku
MyClass . dobryDen ->  poprano
```

Aspekt je děděn ze třídy `AbstractAspect`. Abychom AbstractAspect mohli používat, tak ho musíme nejdříve importovat:

```
from logilab.aspects.core import AbstractAspect
```

V aspektu definujeme, kdy a co máme spustit – metody `before()`, `after()` nebo `around()`. Tyto metody mají takovouto definici:

```
def after(self, wobj, context, *args, **kwargs)
```

kde `self` znamená odkaz na samotný aspekt, `wobj` odkaz na objekt na který je aspekt aplikován, `context` je ditionary s kontextem objektu (obsahuje jméno metody, třídu objektu, návratovou hodnotu atd.) a proměnné `*args` a `**kwargs` v kterých jsou uloženy parametry volané metody.
Pomocí metody

```
weaver.weave_methods(object,aspect)
```

spojíme všechny metody objektu nebo třídy `object` s aspektem `aspect`. Případné odpojení metod provedeme pomocí metody `unweave_methods()`.

Příklad č.2 – Profiler
Druhý příklad ukazuje implementaci profileru. V tomto výpisu programu nejsou uvedeny nepodstatné definice a privátní metody jsou zkráceny.

```
class ProfilerAspect(AbstractAspect):
    def __init__(self, pointcut):
        self.__profile_dict = {}
        AbstractAspect.__init__(self, pointcut)

    def around(self, wobj, context, *args, **kwargs):
        met_name = context['method_name']
        wclass = context['__class__']
        start_time = time.time()
        try:
            return self._proceed(wobj, wclass, met_name, *args,
**kwargs)
        finally:
            end_time = time.time()
            self.__add_to_dict(wobj,met_name,start_time, end_time -
start_time)

    def __add_to_dict(self, wobj, met_name, start_time, call_time):
        #tato privátní metoda přidává do dictionary, v kterém se
        #skladují výsledky, nový záznam o vykonání metody

    def print_info(self):
        for cls in self.__profile_dict:
            print 'Ve tride', cls,":"
            for met_name in self.__profile_dict[cls]:
                entries = self.__profile_dict[cls][met_name]
                time = reduce(lambda x,y: x+y[1],entries, 0)
                print "Metoda", met_name,"probehla",
len(entries),"krat za ",time,"s"

class MyClass:
    def write_thousands_join(self, char):
        str_list = []
        for index in range(100000):
            str_list.append(char)
        return ''.join(str_list)

tmp = MyClass()
```

```
#inicializace AOP
pointcut = PointCut()
pointcut.add_method(MyClass, 'write_thousands_join')
weaver.weave_pointcut(pointcut , ProfilerAspect)
#Měřený úsek
for i in range(100):
    tmp.write_thousands_join('a')
###

aspect_instance = weaver.get_aspect(ProfilerAspect)
aspect_instance.print_info()
weaver.unweave_method(MyClass, ProfilerAspect)
```

Výstup programu je:

```
Ve tride __main__.MyClass :
Metoda write_thousands_join probehla 100 krat za  4.04699993134 s
```

Oproti předcházejícímu příkladu se zde používám u aspektu metodu `around()`, která zabezpečuje spuštění svého kódu místo metody. Metoda má strukturu, že si na začátku poznamenáme startovní čas, následně spustíme původní metodu pomocí `self._proceed()` a po jejím vykonání si poznamenáme konečný čas. Nakonec se v této metodě zabýváme uložením získaných výsledků do dictionary. Zajímavý programátorský obrat je v metodě `print_info()`, kde pro sečtení jednotlivých časů spuštění používáme funkcionální prostředky (`reduce` a `lambda` funkci). Další odlišnost proti předcházejícímu programu je vytváření pointcutu. Zde jsou to metody:

```
pointcut = PointCut()
pointcut.add_method(MyClass, 'write_thousands_join')
weaver.weave_pointcut(pointcut , ProfilerAspect)
```

Zde vytváříme pointcut ne pro všechny metody třídy, ale pouze pro vybranou metodu. Zde `write_thousands_join`().

Příklad č.3 – Návrh podle kontraktu

Aspektová knihovna od Logilabu podporuje také kontrolu vstupních a výstupních podmínek metod a jejich invariantů. Na těchto základech je založena metodika Návrh podle kontaktu (Design by contract) od B. Meyera. Více o této metodice je např. v [18] nebo [19].

Tato knihovna umožňuje definovat (a také kontrolovat) vstupní, výstupní podmínky a invarianty metod. Na následujícím výpisu je uveden příklad metody `push()` vkládající prvek do zásobníku. Podmínky, jejichž splnění je nutné před vložením prvku do zásobníku (uvozené direktivou `pre:`), jsou vkládaný objekt musí existovat (`obj is not None`) a zásobník nesmí být plný `( not self.is_full() )`. Výstupní podmínky (uvozené direktivou `post:`) jsou – zásobník (po vložení prvku) není prázdný a na jeho vrcholu je právě vložený objekt. Případně lze definovat invariant metody (je uvozen klíčovým slovem `inv:`).

```
def push(self, obj):
    """
    pre:
        obj is not None
        not self.is_full()
    post:
```

```
            not self.is_empty()
            self.top() == obj
        """
        raise NotImplementedError
```

ZÁVĚR

Aspektově orientované programování řeší problémy s rozptýlenými záležitostmi, které jsou pomocí klasických metod špatně modularizovatelné. Tím umožňují jednodušší, snadněji udržovatelný, znovupoužitelný a čistší návrh. V současné době se AOP používá ve své „hybridní“ podobě v kombinaci zejména s objektově-orientovaným přístupem. Nejpoužívanějším přístupem k implementaci AOP je implementace pomocí skládání filtrů.
AOP v jazyce Python je jednoduše a přehledně implementováno. Některé jednodušší implementace založené na kompozici filtrů (Logilab, Lightweight, Pythius) se nehodí pro nasazení ve velkých aplikacích, ale pro seznámení se základy AOP jsou dostačující. Další implementace tj. TransWarp a zejména PEAK jsou nejenom implementacemi AOP založenými na subjektově orientovaném programování, ale také aplikačními balíky pro podnikové systémy. Novinkou je implementace AOP pomocí překladače Pythonu do Pythonu PyPy, která slibuje velkou pružnost při vytváření a modifikování základních konceptů.

LITERATURA

[1] DIJKSTRA, E.W. *On the Role of Scientific Thought.* http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html
[2] KICZALES, G., at al.: *Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag, Findland (1997)
[3] KICZALES, G., at al.: An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2001
[4] LADDAD, Ramnivas. I want my AOP! . *JawaWorld,* 2002, č. 1, 3, 4. JavaWorld.com, an IDG company.
[5] MERUNKA, V: Datové objekty. *In Objekty 2005*.
[6] PÍCKA, M.: Aspektově orientované programování. *In Tvorba softwaru 2004*.
[7] Subject-oriented programming http://www.research.ibm.com/sop/
[8] Demeter/Adaptive programming. http://www.ccs.neu.edu/research/demeter/
[9] The AspectJ Website. http://aspectj.org
[10] The AspectC Homepage. http://www.aspectC.org
[11] Python Homepage. http://www.python.org
[12] Python Philosophy. http://c2.com/cgi/wiki?PythonPhilosophy
[13] Logilab’s Aspect module. http://www.logilab.org/projects/aspects
[14] A Light-weight Approach to Aspect Oriented Programming in Python. http://www.cs.tut.fi/~ask/aspects/aspects.html
[15] Pythius Homepage. http://pythius.sourceforge.net/
[16] TransWarp Wiki. http://zope.org/Members/pje/Wikis/TransWarp/.
[17] PEAK – Python Enterprise Application Kit. http://peak.telecommunity.com/
[18] MEYER, B: *Building bug-free O-O software: An introduction to Design by Contract.* http://archive.eiffel.com/doc/manuals/technology/contract/
[19] PITNER, T.: Návrh podle kontraktu – klasická metodika a moderní nástroje. *In Tvorba softwaru 2005*.
[20] PyPy[aspect_oriented_programming]. http://codespeak.net/pypy/dist/pypy/doc/aspect_oriented_programming.html
[21] LAU, Sean: Aspect Oriented Programming, University of Waterloo, Canada