Podpora aplikační logiky v J2EE aplikačních rámcích

Petr Matulík, Tomáš Pitner

Masarykova univerzita v Brně, Fakulta informatiky

Abstrakt. Prostředí J2EE (Java2 Enterprise Edition) je dobrou volbou všude tam, kde jsou požadována robustní, platformově nezávislá řešení podnikových aplikací různého rozsahu. V příspěvku se zaměříme na velmi podstatnou vrstvu těchto aplikací – vrstvu aplikační logiky. Hlavní pozornost budeme logicky věnovat aplikačním rámcům (frameworks), neboť rámce pomáhají řešit základní úkoly návrhu architektury rozsáhlejších aplikací a jsou rovněž místem praktického uplatnění moderních programovacích technik a principů podporujících aplikační logiku – programování orientovaného na aspekty (či Aspect-Oriented Programming) a obrácení řízení (Inversion of Control). Vše bude ilustrováno na příkladech jednoduchého rámce VRaptor a komplexního rámce Spring.

1 Úvod

Prostředí J2EE (Java2 Enterprise Edition) je dobrou volbou všude tam, kde jsou požadována robustní, platformově nezávislá řešení podnikových aplikací různého rozsahu.

V příspěvku se zaměříme na velmi podstatnou vrstvu těchto aplikací – vrstvu aplikační logiky – a její podporu v prostředích J2EE. Hlavní pozornost budeme logicky věnovat aplikačním rámcům (frameworks), jichž je pro J2EE k dispozici celá řada a patří již k základní výbavě soudobého vývojáře webových, ale i jiných javových aplikací.

Rámce pomáhají řešit základní úkoly návrhu architektury rozsáhlejších aplikací a jsou rovněž místem praktického uplatnění moderních programovacích technik a principů podporujících aplikační logiku – programování orientovaného na aspekty (či Aspect-Oriented Programming) a obrácení řízení (Inversion of Control).

Tyto techniky budeme demonstrovat na příkladech rámce jednoduchého (VRaptor) i komplexního (Spring).

1.1 Požadavky na aplikační logiku

Jelikož Java je objektový jazyk, je aplikační logika reprezentována metodami objektů. Snahou vývojáře je, aby aplikační logika byla kromě funkční správnosti – také *přehledná*, snadno *testovatelná, udržovatelná* a *znovupoužitelná* – a to i mimo prostředí, pro nějž primárně vznikla. Zajištění těchto mimofunkčních požadavků je mnohdy obtížnější než vyřešení samotné funkcionality. V poslední době se však objevilo (či znovuobjevilo) několik technik a přístupů, které to usnadňují. Nejprve se podívejme na to, jakými technikami je aplikační logika řízena.

1.2 Řízení aplikační logiky

Valná část moderních aplikací odděluje jednotlivé vrstvy a je vybudována na modelu MVC (model – view/pohled – controller/řadič). Aplikační vrstva je skryta v metodách *modelu* a aktivace těchto metod je posláním *řadiče*. O prezentaci výstupů uživateli se stará *pohled*. Velké procento aplikací – zejména webových – je realizovaných s pomocí tzv. aplikačních rámců.
Rámce jsou z hlediska řízení charakteristické zejména použitím tzv. *Hollywood Principle*: „Do not call us, we will call you!“, což vyjadřuje, že nastává *obrácení řízení* (Inversion of Control, IoC). Namísto samotné výkonné komponenty obsahující vlastní aplikační logiku se o tok výpočtu stará rámec na základě pravidel specifikovaných vývojářem aplikace – rámec tedy volá metody komponent. Rámec tedy není prostá knihovna (i když knihovny může rovněž nabízet); jeho hlavní role spočívá v řízení toku aplikace a v jednotném zajištění nezbytných podpůrných služeb aplikacím. Jak ale rámec aplikaci řídí?

1.3 Deklarativní řízení

Valná většina rámců používá *deklarativní řízení toku* výpočtu. Tok výpočtu je řízen pravidly zachycenými v popisných souborech (deskriptorech), které dnes mají převážně strukturovanou XML podobu. Vezměme příklad webové aplikace. Uživatel zastupovaný webovým prohlížečem vyšle HTTP požadavek směrem k aplikaci vytvořené pomocí některého z webových aplikačních rámců. Deskriptor dané aplikace obvykle podle cílového URL nasměruje dotaz na příslušný řetěz zpracování:

1. nejprve je získán patřičný *model*,
2. na něm jsou volány metody *aplikační logiky* a
3. následně je na výsledek aplikován vhodný *pohled*, který se dostane zpět klientovi.

Toto vše řídí aplikační rámec, nikoli samotná komponenta s aplikační logikou. To, že *řízení je obráceno*, že je uplatňováno zvenčí, dovoluje aplikační logiku znovupoužít i mimo její původní místo.

1.4 Injektáž závislostí

Obrácení řízení je již dlouho používaný návrhový vzor, jenž v původním významu představoval techniku, s jejíž pomocí dochází k přenesení řízení běhu programu z kódu, který navrhuje programátor, na podpůrný aplikační rámec. Lze tedy říci, že jakýkoliv moderní aplikační rámec včetně dále diskutovaného rámce Spring či EJB je aplikací návrhového vzoru IoC.

S nástupem aplikačních rámců, jakými jsou Spring či PicoContainer, se však význam IoC začal posunovat a byl spíše používán pro popis způsobu, jakým je v těchto rámcích zajištěno *provázání spravovaných komponent* a řešení závislostí mezi nimi.

Na toto zmatení pojmů reagovali přední teoretici i praktici v této oblasti (včetně Roda Johnsona) a rozhodli se pro zavedení nového pojmu *injektáž závislostí*

(Dependency Injection, DI). Zjednodušeně řečeno lze postup při použití DI popsat takto. Mějme závislost komponenty A na komponentě B; jinými slovy komponenta A obsahuje odkaz na komponentu B. Při použití DI budou při startu kontejneru, který je spravuje, obě komponenty vytvořeny a v komponentě A bude vytvořen odkaz na komponentu B. Existuje několik způsobů, jak toho lze dosáhnout, my však zmíníme jen dva zdaleka nejrozšířenější.

První způsob umožňuje uspokojení závislostí prostřednictvím nastavovacích („set“) metod komponenty a proto bývá označován jako *Setter Injection*. Druhou možností je pak varianta s názvem *Constructor Injection*, která využívá standardních konstruktorů jazyka Java. A právě tyto dvě varianty jsou podporovány rámcem Spring[2].

V minulosti se provázání komponent dosahovalo buď přímou instanciací v kódu třídy, případně různými vyhledávacími („lookup“) mechanismy (např. JNDI) či aplikací návrhového vzoru *Service Locator*. Přestože v tomto pořadí šlo vždy o krok kupředu, všechna tato řešení jsou horší než IoC, a to zejména z hlediska příliš úzkého vzájemného svázání komponent a zhoršené čitelnosti a udržovatelnosti kódu.

Nyní se podívejme na vybrané aplikační rámce a způsob, jakým uvedené techniky využívají. Za příklady zvolíme jeden komplexní rámec – Spring – a jako protiváhu jednoduchý rámec VRaptor. Začneme přitom tím jednodušším, kde jsou principy na první pohled vidět.

2 Jednodušší řešení – rámec VRaptor

V poslední době se jako odpověď na složitost „plných“ webových aplikačních rámců jako je Struts, WebWork nebo Spring zmíněný dále objevují jednoduché rámce neřešící „vše“. Příkladem takového rámce je VRaptor [2].

2.1 Řízení toku

Nejdůležitějším posláním aplikačního rámce je řízení toku – rámec tedy realizuje obrácení řízení. Tok výpočtu vyřízení klientského požadavku je deklarativně specifikován popisným souborem `vraptor.xml`, který může vypadat např. takto (podrobněji viz dokumentace rámce VRaptor):

```
<package name="chain">
   <chain name="productlist">
      <logic class="org.vraptor.example.chain.ProductLogic"
             method="listAll"/>
      <view>chain/productList.vm</view>
   </chain>
   <chain name="newproduct">
      <view>chain/productForm.vm</view>
```

[2] Kromě toho se hovoří i o tzv. *interface injection*, která ale vyžaduje, aby komponenty, do nichž se závislost bude injektovat, implementovaly určité předepsané rozhraní, což je poměrně úzce svazuje s daným prostředím.

```
   </chain>
   <!-- our first real chain -->
   <chain name="saveproduct">
      <logic class="org.vraptor.example.chain.ProductLogic"
             method="save"/>
      <logic class="org.vraptor.example.chain.ProductLogic"
             method="listAll"/>
      <view>chain/productList.vm</view>
   </chain>
   <chain name="editproduct">
      <logic class="org.vraptor.example.chain.ProductLogic"
             method="edit"/>
      <view>chain/productForm.vm</view>
   </chain>
</package>
```

Popisovač rámci sděluje, jak reagovat na jednotlivé klientské požadavky – akce. Každá akce je realizovaná jako řetěz (element `chain`) operací (elementy `logic`) zakončený obvykle zobrazením výsledku operací (element `view`).

Jak vidět, samotná aplikační logika proces neřídí a má být znovupoužitelná mezi různými akcemi – viz např. metoda `listAll` třídy `ProductLogic`. Aby byla logika opravdu znovupoužitelná, musí však být vhodně napsaná.

2.2 Nezávislost aplikační logiky

Podívejme se, jak VRaptor umožňuje vyčlenění aplikační logiky mimo prostředí úzce svázané se servlety a dalšími prvky JavaServlet API.

Příklad převzatý z dokumentace VRaptor ukazuje na triviálním úloze obrácení zadaného řetězce, jak zbavit aplikační logiku závislosti na konkrétním prostředí (API):

```
public class TextInverser {
   private HttpServletRequest request;
   private RaptorContext context;

   public TextInverser(HttpServletRequest request,
                       RaptorContext context) {
      this.request = request;
      this.context = context;
   }

   public void invertName() {
      String submittedName = this.request.getParameter("name");
      String inversedName = null;
      if (submittedName != null) {
         StringBuffer buffer = new StringBuffer(submittedName);
         buffer.reverse();
         inversedName = buffer.toString();
```

```
      }
      this.context.put("name", inversedName);
   }
}
```

Výstup výše uvedené metody `invertName` je vložen do kontextu (objekt `context`) rámce. To umožní tento výsledek následně zobrazit v prezentační vrstvě realizované např. rámcem *Velocity*, populárním *FreeMarkerem* nebo klasickými JSP. Izoluje tedy aplikační logiku od prezentační. Souhra aplikační logiky a následné prezentace je zajištěna deklarací v souboru `vraptor.xml` (umístěn v adresáři /WEB-INF webové aplikace):

```
<vraptor>
  <package name="simple">
    <chain name="helloWithServletAndVRaptorStuff">
      <logic class="org.vraptor.example.simple.TextInverser"
             method="invertName"/>
      <view>simple/templateWithServletAndVRaptorStuff.vm</view>
    </chain>
  </package>
</vraptor>
```

Tento postup však stále předpokládá, že aplikační logika realizovaná třídou `TextInverser` zná JavaServlet API (např. třídu `HttpServletRequest`) a rovněž API rámce VRaptor (třída `Context`) a je na nich tedy závislá. Odstranění závislosti na obou zmíněných API lze provést takto:

```
public class TextInverser {
   private HttpServletRequest request;
   private String name;
   public TextInverser(HttpServletRequest request) {
      this.request = request;
   }
   public void invertName() {
      String submittedName = this.request.getParameter("name");
      String inversedName = null;
      if (submittedName != null) {
         StringBuffer buffer = new StringBuffer(submittedName);
         buffer.reverse();
         inversedName = buffer.toString();
      }
      this.name = inversedName;
   }
   public String getName() {
                return this.name;
   }
}
```

Místo, abychom výsledek inverze uložili do přímo kontextu VRaptoru, je zde místo toho dána k dispozici metoda `getName()`, kterou klient (např. rámec VRaptor, ale i kdokoli jiný) po provedení inverze zavolá, aby získal výsledek. Tím je odstraněna svázanost s rámcem VRaptor. Zbývá odstranit závislost na JavaServlet API:

```
public class TextInverser {
   private String name;
   public void invertName() {
      if (this.name != null) {
         StringBuffer buffer = new StringBuffer(this.name);
         buffer.reverse();
         this.name = buffer.toString();
      }
   }
   public String getName() {
                return this.name;
   }
   public void setName(String name) {
                this.name = name;
   }
}
```

Vstupní parametr `name` lze nyní nastavit metodou `setName`, o jejíž zavolání se ovšem musí postarat rámec, který to zajistí za běhu aplikace pomocí reflexe. Přijde-li ve vstupu od klienta – webového prohlížeče parametr `name`, rámec vyvolá `setName` a jméno nastaví podle hodnoty parametru. Pak zavolá metodu `invertName` a výsledek bude k dispozici metodou `getName`. Rámec tedy k nastavení používá tzv. *Setter Injection*, tj. injektáž hodnoty pomocí metod `set`*`XXX`*. VRaptor takto umí nastavovat i parametry jiných typů než String – automaticky provede konverzi.

Po poslední úpravě je aplikační logika představovaná třídou `TextInverser` nezávislá na API servletů i VRaptoru – a tedy 100% znovupoužitelná.

2.3 Interceptory

VRaptor umožňuje podobně jako u programování orientovaného na aspekty definovat speciální ortogonálně (znovu)použitelné části aplikační logiky zvané *interceptory* (Interceptors). Jsou to třídy, jejichž metody jsou volány ve význačných okamžicích vyřizování klientského požadavku:

- před vstupem do řetězce,
- po jeho realizaci (před aplikací pohledu)
- a po ní.

Takto lze např. velni snadno realizovat protokolování (logging) běhu aplikace:

```
public class LoggingInterceptor implements ChainInterceptor {
   private static Logger logger =
                  Logger.getLogger(LoggingInterceptor.class);
    public void beforeChain(RaptorData data) {
        logger.info("before chain");
    }
    public void afterChain(RaptorData data) {
        logger.info("after chain");
    }
```

```
    public void afterViewRendering(RaptorData data) {
        logger.info("after view rendering");
    }
}
```

Podobné úlohy je někdy možné zvládnout i s pouhým JavaServlet API – použitím tzv. *filtrů* [3], ale tam vzhledem k neoodělení fáze aplikační logiky od prezentační není tak snadné proniknout k jednotlivým fázím životního cyklu požadavku. Interceptory jsou také, jak vidět, daleko elegantnější. Vlastní „intercepce“ je zajištěna deklarací *domény*, v níž interceptor použijeme:

```
<domain name="logging-interceptor-domain">
<interceptor
    class="org.vraptor.example.interceptor.LoggingInterceptor"/>
</domain>
```

Vlastní aplikační logika (představovaná třídou `SomeLogic` a metodou `doSomething` musí deklarovat, že je v příslušné doméně s intercepcí:

```
<package name="interceptor"
         domains="logging-interceptor-domain">
   <chain name="log">
      <logic class="org.vraptor.example.interceptor.SomeLogic"
             method="doSomething"/>
      <view>interceptor/logging.vm</view>
   </chain>
</package>
```

3 Komplexní přístup – rámec Spring

Hitem v oblasti vývoje pokročilých J2EE aplikací se v průběhu posledních tří let stal rámec pro jejich správu s názvem Spring. V základech tohoto open-source projektu se nachází kód, který byl publikován v roce 2003 Rodem Johnsonem v jeho knize J2EE Design and Development [1] a který odráží Johnsonovu několikaletou analytickou, konzultantskou ale i programátorskou zkušenost s tvorbou rozsáhlých J2EE aplikací. Tento kód byl autorem navržen pro zefektivnění řešení některých běžných problému, se kterými se každý vývojář pokročilých Java aplikací setkává téměř denně, a celkově pro zjednodušení a snížení ceny návrhu a vývoje těchto programů. Jeho intenzivním rozšiřováním vznikl aplikační rámec, jehož celkový počet stažení se k dnešnímu dni rychle blíží k půl miliónu.

3.1 Charakteristika

Spring [4] je modulární Java/J2EE aplikační rámec, jenž také bývá někdy označován jako odlehčený (lightweight) kontejner. Využíván je stejně jako Enterprise JavaBeans zejména pro tvorbu webových aplikací, ale lze jej použít v podstatě pro jakýkoliv typ aplikace, včetně aplikací s grafickým rozhraním.

Cílem tohoto projektu je samozřejmě zejména usnadnění vývoje Java (a zejména J2EE) aplikací. Prostředků k dosažení tohoto primárního cíle poskytuje Spring několik. Prvním z nich je podpora pro *aplikační vrstvu* programů, což je vlastnost, která je na trhu unikátní a která představuje hlavní lákadlo pro vývojáře J2EE aplikací. Budeme se jí proto později věnovat podrobněji.

Dalším důležitým cílem použití rámce Spring je *snadná testovatelnost* výsledné aplikace. Spring, jak později uvidíme, umožňuje čistým a pohodlným způsobem vzájemně oddělit nejen jednotlivé vrstvy, ale dokonce i jednotlivé objekty, což je klíčovou podmínkou pro možnost využití *testování jednotek* (unit testing). Vzhledem k tomu, že agilní metodiky vývoje software, které jsou na testování ve většině případů postaveny, jsou stále populárnější a rozšířenější, je rovněž tato vlastnost rámce považována za klíčovou.

Drtivá většina existujících aplikačních rámců se soustřeďuje na podporu jen určité architekturní vrstvy aplikace. Příkladem budiž oblíbený rámec Struts, který usnadňuje vývoj webové *prezentační vrstvy* aplikací, nebo neméně populární objektově-relačně mapovací nástroj Hibernate, orientující se na *datovou (perzistenční) vrstvu*. Spring naproti tomu konzistentním způsobem podporuje všechny vrstvy aplikací, prezentační vrstvou počínaje, přes již zmíněnou aplikační vrstvu až k datové a perzistenční vrstvě.

Zásadou vývoje aplikačního rámce Spring je nevynalézat kolo**.** Integrace velkého množství rozšířených softwarových nástrojů poskytuje uživateli možnost využít specializovaných a časem prověřených řešení na danou problémovou oblast, a to konzistentním způsobem. K těmto podporovaným nástrojům patří samozřejmě již zmíněné Struts a Hibernate, jejich celkové množství však jde řádově do desítek.

3.2 Podpora aplikační vrstvy

Jak už jsme se výše zmínili, chtěli bychom se v tomto příspěvku věnovat zejména podpoře, kterou Spring poskytuje aplikační vrstvě programů.

Objekty, které tvoří aplikaci, jsou během svého životního cyklu spravovány kontejnerem rámce Spring a výjimku netvoří ani objekty aplikační vrstvy. Spring opravdu spravuje aplikaci již na úrovni objektů, nikoliv ucelených komponent, jako je tomu u EJB.

Ke službám, které může návrhář aplikace pro správu objektů využít, patří zejména jednotné procedurální i deklarativní řízení transakcí, jednotný způsob konfigurace aplikace v době nasazení, pokročilá procedurální i deklarativní správa zabezpečení, správa provázání a závislostí objektů, pooling objektů a další.

Stěžejními technologiemi, které poskytování těchto služeb umožňují jsou aspektově orientované programování a obrácení řízení. Pokud jde o AOP, tak lze použít jednak vlastního řešení rámce Spring, které je postaveno na standardních dynamických proxy jazyka Java, je ale také možno využít integrace s populárním open-source nástrojem AspectJ. Implementace obrácení řízení, respektive injektáže závislostí v rámci Spring představuje pravděpodobně nejkomplexnější řešení těchto technik vůbec.

3.3 Injektáž závislostí

Kontejner rámce Spring (nazývaný také aplikační kontext) při svém startu načte definiční soubor. Na základě definic v něm uvedených vytvoří pomocí reflexe specifikované objekty, vzájemně je prováže a tuto síť objektů spravuje po dobu běhu aplikace. Definiční soubor mívá v drtivé většině případů formát XML, lze ovšem použít i jiné formáty, například tzv. „properties“ soubor. Uveďme si nyní příklad obsahu takového XML definičního souboru:

```
<beans>
  <bean id="dao" class="cz.neco.NejakeDaoImpl" />
  <bean id="sluzba" class="cz.neco.NejakaSluzba" >
   <property name="konf">
     <value>nejakaHodnota</value>
   </property>
   <property name="dao" ><ref bean="nejakeDao" /></property>
  </bean>
</beans>
```

Předpokládejme tedy existenci následujících tříd a rozhraní:

```
interface NejakeDao {}
class NejakeDaoImpl implements NejakeDao {}
interface NejakaSluzba {}
class NejakaSluzbaImpl implements NejakaSluzba {
   private NejakeDao dao;
   private String konf;
   public void setDao(NejakeDao dao) {
      this.dao = dao;
   }
   public void setKonf(String konf) {
      this.konf = konf;
   }
}
```

V tomto případě by kontejner použil k injektáži závislostí *Setter Injection* a prostřednictvím reflexe provedl kód ekvivalentní tomuto:

```
NejakeDao nejakeDao = new NejakeDaoImpl();
NejakaSluzba nejakaSluzba = new NejakaSluzbaImpl();
nejakaSluzba.setKonf("nejakaHodnota");
nejakaSluzba.setDao(nejakeDao);
```

Nastavení atributu *konf* třídy *NejakaSluzbaImpl* ilustruje konfigurační možnosti rámce Spring. Stejným způsobem lze inicializovat hodnoty atributů ostatních primitivních typů. Nastavení atributu *dao* třídy *NejakaSluzbaImpl* naopak ukazuje, jakým způsobem lze využít Spring pro správu provázání a závislostí objektů.

Pojmenujeme-li tento definiční soubor *kontext.xml* a umístíme-li jej do `classpath`, pak kontejner nastartujeme například takto:

```
ApplicationContext context =
   new ClassPathXmlApplicationContext("/kontext.xml");
```

Vytvořenou službu s `id` *nejakaSluzba* získáme z kontejneru voláním:

```
NejakaSluzba sluzba =
   (NejakaSluzba)context.getBean("nejakaSluzba");
```

Pomocí dalších deklarací v tomtéž souboru lze také definovat, jakým způsobem budou dříve vyjmenované služby poskytnuty našim objektům. Naše třídy se tedy nemusí například transakčním či bezpečnostním managementem zabývat, vše zajistí Spring.

3.4 Programování orientované na aspekty

Na podrobný vhled do techniky zvané *programování orientované na aspekty* (Aspect-Oriented Programming, AOP) není v tomto článku prostor, dostatek informací je však možné najít např. v [5] nebo [6].

AOP v zásadě směřuje k tomu, jak elegantně dosáhnout vyčlenění opakovaných „nezáživných“, „režijních“ částí kódu, řešících obvykle nějaké (mimofunkční, napříč jdoucí) *záměry* (*cross-cutting concerns*, např. autentizaci/autorizaci klienta při vstupu do určité metody, protokolování volání metod, přístup k perzistentnímu úložišti dat …) do tzv. *pokynů* (*advices*). Tyto jsou vyčleněny do speciálních tříd a deklarativním způsobem je řečeno, kam všude se příslušný pokyn má aplikovat (ta místa se označují jako *point-cut*s). Běhové prostředí příslušného nástroje pro AOP (pro Javu je známý např. AspectJ [7]) pak zajistí aplikaci příslušného pokynu a vzniká tak tzv. *aspekt* (*aspect*).

Podívejme se teď na příklad zapojení AOP v rámci Spring:

```
<bean id="nejakyInterceptor" class="cz.neco.NejakyInterceptor"/>

<bean id="nejakyAdvisor"
      class="org.springframework...RegexpMethodPointcutAdvisor">
   <property name="advice">
      <ref bean="nejakyInterceptor"/>
   </property>
   <property name="patterns">
      <list>
         <value>.*get.*</value>
         <value>.*set.*</value>
      </list>
   </property>
</bean>

<bean id="nejakaSluzba" class="cz.neco.NejakaSluzbaImpl" />

<bean id="proxySluzba"
     class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="proxyInterfaces">
      <value>cz.neco.NejakaSluzba</value>
   </property>
```

```
   <property name="target">
      <ref local="nejakaSluzba"/>
   </property>
   <property name="interceptorNames">
      <list>
         <value>nejakyAdvisor</value>
      </list>
   </property>
</bean>
```

V tomto příkladu jsme opět využili objekt aplikační vrstvy *NejakaSluzbaImpl* z předchozího příkladu. Dále jsme definovali objekt třídy *NejakyInterceptor*, který představuj samotný interceptor, tedy kus kódu, který se bude vykonávat před nebo po provedení nějaké metody. Objekt s `id` *nejakyAdvisor* definuje, že interceptor *nejakyInterceptor* bude aplikován na „get“ a „set“ metody cílového objektu a konečně objekt s `id` *proxySluzba* představuje proxy pro náš objekt aplikační vrstvy. V konečném důsledku tedy definuje, že před a/nebo po provedení jakékoliv „get“ nebo „set“ metody cílového objektu s `id` *nejakaSluzba* bude proveden kód objektu *nejakyInterceptor*. Jelikož jde o příklad použití AOP na rozhraních, lze výsledný objekt s `id` *proxySluzba* přetypovat pouze na specifikovaná rozhraní – v našem případě rozhraní *NejakaSluzba*. Za pomoci knihovny CGLIB lze však v rámci Spring aplikovat AOP i na konkrétní třídy.

3.5 Řízení transakcí

Tímto jsme si připravili půdu pro krátkou ukázku *deklarativního řízení transakcí*, které je postaveno na právě předvedených možnostech AOP.

```
<bean id="transakcniProxySluzba"
      class="org.springframework...TransactionProxyFactoryBean">
   <property name="transactionManager">
      <ref bean="transactionManager"/>
   </property>
   <property name="target">
      <ref bean="nejakaSluzba"/>
   </property>
   <property name="transactionAttributes">
      <props>
         <prop key="insert*">
            PROPAGATION_REQUIRED,-NejakaVyjimka
         </prop>
         <prop key="update*">PROPAGATION_REQUIRED</prop>
         <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
   </property>
</bean>
```

Zde jsme definovali transakční proxy pro naši třídu aplikační vrstvy a transparentním způsobem jsme z ní v podstatě vytvořili transakční kód. Konkrétně zde je specifikováno, že na všechny metody, jejichž název začíná řetězcem „insert“,

bude uplatněno standardní transakční chování s propagací platnosti transakcí do nižších úrovní, a navíc že pokud daná metoda vyvolá výjimku třídy *NejakaVyjimka*, pak transakce bude vrácena zpět (proběhne rollback). Pokud bychom definovali příznak *+NejakaVyjimka*, pak by se při vyvolání výjimky třídy *NejakaVyjimka* provedl commit transakce. U jiných než „insert“ a „update“ metod pak definujeme, že transakce je pouze pro čtení.

Velmi podobným deklarativním a snadným způsobem lze definovat například přístupová práva k jednotlivým objektům a metodám, pooling objektů a další užitečné služby, bez nichž se aplikační vrstva J2EE systémů často neobejde.

4 Perspektivy

O výhodnosti použití technik obrácení řízení a programování orientovaného na aspekty svědčí i fakt, že podobné směry zvolili autoři návrhu standardu **Enterprise JavaBeans 3.0**. Tento aplikační rámec by také měl při správě aplikační logiky programů využívat výhod IoC a AOP a do jisté míry kopírovat přístupy, které se do povědomí širokých vývojářských vrstev dostaly do značné míry díky aplikačnímu rámci Spring. Čím dříve jako vývojáři J2EE aplikací tyto principy zvládneme, tím lépe. Časem nám stejně nic jiného nezbude…

Reference

1. Rod Johnson. *J2EE Design and Development*. Wiley Publishing, 2003, Indianapolis. ISBN:0764543857.
2. *VRaptor – Simple Web MVC Framework*. http://vraptor.dev.java.net
3. *JavaServlet API*. http://java.sun.com/products/servlet/docs.html
4. *Spring Framework.* http://www.springframework.org
5. *Aspect-Oriented Programming*. http://dmoz.org/Computers/Programming/Methodologies/Aspect-Oriented
6. Kroha, P. *Aspektově orientované programování.* Tutoriál konference DATAKON 2005, Brno, 2005.
7. *AspectJ*. http://eclipse.org/aspectj