Aspect Oriented Programming (AOP)

Úvod

- AOP doplňuje OOP o nový způsob rozšiřování struktury programu. Zatímco v OOP je klíčovým prvkem modularity třída, v AOP je to **aspekt**.
- V typické aplikaci se vyskytují části kódu, které prolínají do mnoha vrstev aplikace, ale do žádné vrstvy je nelze vhodně začlenit (logování, auditování, profilování, autorizace, synchronizace, použití transakcí apd).
- Tyto části kódu se separují do tzv. aspektů. AOP zajišťuje zařazení aspektů do stávajícího kódu, aniž by se kód musel modifikovat (pomocí mechanismu tvorby proxy objektů spravovaných v kontejneru).

Implementace AOP ve Springu

- Využití AOP modulu prolíná celým Springem, stejně jako využití IoC kontejneru. Pomocí AOP je implementována např. podpora transakcí.
- AOP není výhradně technologií Springu, Spring nabízí vlastní jednoduše použitelnou implementaci AOP (**Spring AOP**), která neimplementuje celou specifikaci, ale vyhovuje zhruba 80 % možných požadavků enterprise aplikací,
  - při specifických náročnějších požadavcích lze použít integrovaný framework **AspectJ** (aspekty pro atributy, drobné doménové objekty ...).
- IoC kontejner Springu není na AOP závislý, AOP je pouze jeho doplňkem. Pokud nechceme AOP využívat, nemusíme. Podpora AOP je dostupná i mimo aplikační server.

AspectJ

- Od verze 2.0 nabízí Spring kromě vlastní implementace Spring AOP také integraci s AOP frameworkem AspectJ, díky kterému lze využívat:
  - Pointcut EL (Expression Language) – jazyk pro popis přiřazení aspektů do toku kódu,
  - AspectJ definice – označení třídy implementující aspekt pomocí anotace @Aspect (je třeba zapnout podporu autoproxyingu aspektů v konfiguraci: `<aop:aspectj-autoproxy/>`).
- Spring AOP a AspectJ lze volně kombinovat (aspekty ze Springu lze využívat v AspectJ a naopak).

Obecná terminologie AOP (I)

- **Aspekt** – část kódu zařazovaná do mnoha dalších tříd. Ve Springu je aspekt implementován jako beana s konfigurací AOP v XML (tzv. schema-based aproach), nebo jako beana označená v Java kódu anotací @Aspect (styl převzatý z frameworku AspectJ). Aspekty jako třídy obsahují deklarace specifických členů (v podobě metod): pointcutů, advice.
- **Join point** – bod při vykonávání kódu programu, ke kterému je přiřazován kód aspektu – např. spuštění metody (jiné body pro zařazení aspektu Spring AOP zatím nepodporuje).
- **Pointcut** – vzor (predikát, zástupný výraz v Pointcut EL), který obecně popisuje join points, na kterých se bude provádět advice (vykonání aspektu).
- **Advice** – akce vykonávaná při aplikaci aspektu na daný joint point, je asociovaná s pointcutem – spouští se na každém joint point, který vyhovuje danému pointcutu.

Logovací aspekt I.

```
import java.util.Arrays; import org.aspectj.lang.*;
import org.aspectj.lang.annotation.*; import org.slf4j.*;

@Aspect @Component
public class LoggingAspect {
   private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

   @AfterThrowing(value = "within(@org.springframework.stereotype.Service *)", throwing = "ex")
   public void logServiceException(JoinPoint joinPoint, RuntimeException ex) {
      Signature signature = joinPoint.getSignature();
      String methodName = signature.getName();
      String stuff = signature.toString();
      String arguments = Arrays.toString(joinPoint.getArgs());
      logger.info("Exception in method: " + methodName + " with arguments " + arguments
            + "\nand the full toString: " + stuff + "\nthe exception is: " + ex.getMessage(),
ex);
   }
}
```

Aspekt bude zapojen na metodách servisní vrstvy.

Tato metoda se vyvolá, když se vyhodí výjimka typu RuntimeException

Logovací aspekt II.

Spring konfigurace:

Nesmíme zapomenout na oscanování anotací aspektu!

```
<context:component-scan base-package="com.test.aspect" />

<aop:aspectj-autoproxy />
```

Zapojení aspektu

NEBO POMOCÍ JAVACONFIG: `@EnableAspectJAutoProxy`

pom.xml:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.6.6</version>
</dependency>
```

Plus dependency na slf4j-log4j
a konf. soubor log4j.properties :-)

Obecná terminologie AOP (II)

- **Introduction** (inter-type declaration) – dodatečná deklarace a implementace rozhraní v určitém (již existujícím) datovém typu.
- **Target object** – objekt, na kterém je vykonávána advice. Jelikož Spring AOP je implementováno pomocí proxy objektů, je target object vždy proxy objektem.
- **AOP proxy** – proxy objekt vytvořený AOP frameworkem pro zařazení aspektů do kódu (spouštění advice). Proxy objekty jsou ve Spring frameworku implementovány jako standardní JDK dynamic proxies (J2SE dynamic proxies), což umožňuje zavést proxy objekty pro libovolné rozhraní, nebo jako CGLIB proxies, které lze zavést i nad třídami bez rozhraní.
- **Weaving** – spojování aspektů s ostatním kódem aplikace. Toto se provádí buď při kompilaci aplikace (například modifikací byte kódu speciálním AspectJ kompilátorem), nebo až za běhu aplikace (ve Spring AOP vytvořením proxy objektu, který bude obsahovat původní kód včetně kódu dodaného aspektem).

Různé typy advice

- **before advice** – advice spouštěná před joint pointem (metodou),
- **after returning advice** – advice spouštěná za joint pointem v případě, že kód joint pointu (metody) skončil normálně (bez vyvolání výjimky),
- **after throwing advice** – advice spouštěná za joint pointem v případě, že kód joint pointu (metody) vyvolal výjimku,
- **after (finally) advice** – advice spouštěná vždy za joint pointem (metodou), tj. v případě, že kód joint pointu (metody) skončil normálně, i v případě, že vyvolal výjimku,
- **around advice** – advice spouštěná „kolem“ joint pointu (metody). Joint point je ve skutečnosti spouštěn v libovolné fázi uvnitř kódu, který je definován v advice, přičemž joint point lze vyvolat i vícekrát, nebo vůbec. Může být vrácena i jiná/modifikovaná hodnota, nebo vyvolána určitá výjimka. Jedná se o nejobecnější advice s největší kontrolou.

Pointcut Expression Language (EL)

- Pointcut je vzor (pattern), pomocí kterého se vyhledávají joint points (metody bean), na které budou aplikovány aspekty spuštěním určité advice. Vyhledávat lze pouze veřejné metody.
- Ve výrazech pointcutu lze používat následující značky (**PCD – pointcut designators**):
  - execution – nejpoužívanější, vyhledání metod podle názvu,
  - within – vyhledání metod v určitém specifikovaném typu,
  - this – vyhledání metod, které se nachází v proxy daného typu,
  - target – vyhledání metod, které jsou v cílových business objektech daného typu (klasických neproxy objektech),
  - args – vyhledání metod, které obsahují argumenty daných typů,
  - bean – vyhledání metod, které se nachází v beaně s daným identifikátorem,
  - ...
- a následující zástupné symboly: *, logické operátory: and, or, not. Logické operátory mohou být použity pro zřetězení více PCD.

Formát pointcut výrazů

- `execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`
- Nepovinné části jsou zakončeny otazníkem.
- **ret-type-pattern** – návratový typ, nejčastěji se používá * (libovolný i void), jinak lze použít plně kvalifikovaný název typu,
- **name-pattern** – jméno metody (* pro zastoupení libovolné části),
- **param-pattern** – specifikace parametrů, jedna z těchto možností:
  - () – metody bez parametrů,
  - (..) – metody s libovolným (i nulovým) počtem parametrů,
  - (*) – metody s jedním parametrem libovolného typu,
  - (*, String) – metody se dvěma parametry, kde první je libovolného typu a druhý typu String atd.

Příklady pointcut výrazů

- `execution(public * *(..))` – libovolná veřejná metoda
- `execution(* set*(..))` – libovolná metoda začínající na „set“
- `execution(* com.xyz.service.AccountService.*(..))` – libovolná metoda definovaná v rozhraní/třídě AccountService
- `execution(* com.xyz.service.*.*(..))` – libovolná metoda definovaná v balíčku service (předposlední * zastupuje libovolné rozhraní/třídu v balíčku, poslední * zastupuje libovolné jméno metody)
- `within(com.xyz.service.*)` – libovolná metoda v balíčku service
- `within(com.xyz.service..*)` – libovolná metoda v balíčku service nebo v jeho podbalíčcích
- `target(com.xyz.service.AccountService)` – libovolná metoda, jejíž cílový objekt implementuje rozhraní AccountService
- `args(java.io.Serializable)` – libovolná metoda, která má pouze jeden parametr typu Serializable, který je vyhodnocován za běhu aplikace (nemusí se jednat o typ deklarovaný v hlavičce metody, jako je tomu v případě pointcutu execution(* *(java.io.Serializable))

Tvorba aspektů pomocí XML

- V konfiguraci kontejneru musí být uveden jmenný prostor aop, následně může být veškerá konfigurace AOP umístěna uvnitř elementu <aop:config> (konfigurace může být rozdělena i do více těchto elementů).
- Element <aop:config> může obsahovat elementy pro deklaraci pointcutů (pointcut), advices (before, after-returning, after-throwing, after, around) a aspektů (aspect).
- Alternativně může být konfigurace prováděna pomocí anotací (ve stylu `@AspectJ`).

Tvorba aspektů pomocí XML

- Deklarace aspektu (třída aspektu je obyčejný objekt Javy)

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="..."/>
```

- Deklarace pointcutu
  - Pointcut deklarovaný vně aspektu může být sdílen více aspekty:

```
<aop:config>
 <aop:pointcut id="businessService"
 expression="execution(* com.xyz.myapp.service.*.*(..))"/>
</aop:config>
```

Tvorba aspektů pomocí XML

- Deklarace pointcutu
  - Pointcut deklarovaný uvnitř aspektu:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    <aop:pointcut id="businessService"
 expression="execution(* com.xyz.myapp.service.*.*(..))"/>
    ...
  </aop:aspect>
</aop:config>
```

- Deklarace before advice

```
<aop:aspect id="beforeExample" ref="aBean">
    <aop:before
      pointcut-ref="businessService"
      method="doAccessCheck"/>
    ...
</aop:aspect>
```

  - Namísto pointcut-ref lze použít přímo in-line pointcut expression:
    `pointcut="execution(* com.xyz.myapp.service.*.*(..))"`

Tvorba aspektů pomocí XML

- Deklarace before advice
  - Ve třídě aspektu musí být definována metoda odkazovaná pomocí atributu “method“ (implementace advice). Tato metoda implementuje vlastní kód (chování) aspektu:
  - `public void doAccessCheck() {...}`
- Deklarace after returning advice

```
<aop:aspect id="afterReturningExample" ref="aBean">
    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      returning="retVal"
      method="doAccessCheck"/>
    ...
</aop:aspect>

// metoda pro advice umístěná v kódu aspektu,
// s přístupem k návratové hodnotě joint pointu (metody)
public void doAccessCheck(Object retVal) {...
```

Tvorba aspektů pomocí XML

- Deklarace after throwing advice

```
<aop:aspect id="afterThrowingExample" ref="aBean">
    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      throwing="dataAccessEx"
      method="doRecoveryActions"/>
    ...
</aop:aspect>

// metoda pro advice umístěná v kódu aspektu,
// s přístupem k objektu vyvolané výjimky
public void doRecoveryActions(DataAccessException
  dataAccessEx) {...
```

- Deklarace after (finally) advice

  Obdobně použitím elementu <aop:after>.

Tvorba aspektů pomocí XML

- Deklarace around advice
  - Prvním parametrem advice metody musí být argument typu ProceedingJoinPoint, který umožňuje zavolat metodu proceed() ke spuštění joint pointu (metody). Metoda proceed může být volána také s argumentem typu Object[], který představuje argumenty pro joint point (metodu):

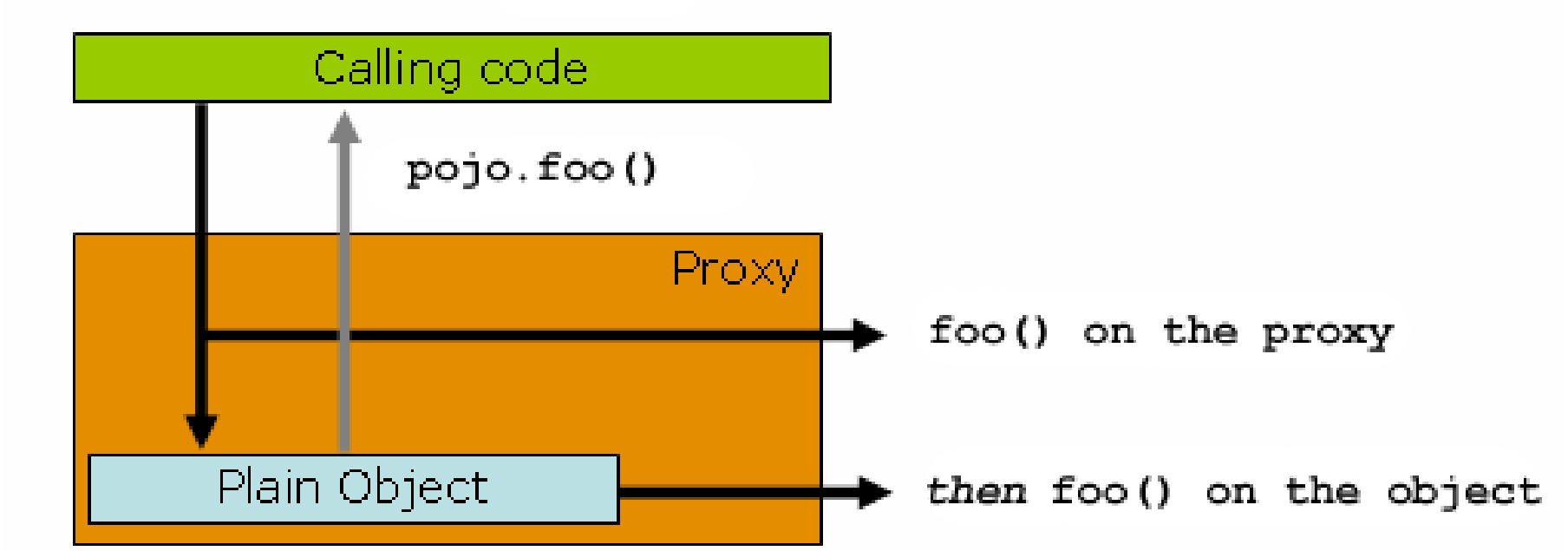
```
<aop:aspect id="aroundExample" ref="aBean">
    <aop:around
      pointcut-ref="businessService"
      method="doBasicProfiling"/>
    ...
</aop:aspect>
```

```
// implementace advice metody ve třídě aspektu:
public Object doBasicProfiling(ProceedingJoinPoint pjp)
  throws Throwable {
    // začátek měření času
    Object retVal = pjp.proceed();
    // konec měření času
    return retVal;
}
```

AOP proxy

- Spring používá interně dva druhy proxy:
  - Standardní JDK dynamic proxies (J2SE dynamic proxies) – umožňují zavést proxy objekty pro libovolné rozhraní,
  - CGLIB proxies – umožňují zavést proxy i nad třídami bez rozhraní.

- Princip zapouzdření volání metod a advice do proxy objektu:



Self invocation

- Aby se vyvolala logika aspektu, je nutné projít přes proxy objekt. Nejjednodušší je dát logiku „jinam“, do další beany. Co když to ale nedává smysl? Pak nám nezbývá nic jiného než něco takového:

```
@Autowired
private ApplicationContext applicationContext;

public String doStuff() {
  return applicationContext.getBean(Main.class).doStuffInternal();
}
```

Nebo:

```
@Service
class DummyService {
    @Lazy
    @Autowired
    private DummyService self;
}
```