UNIVERZITET METROPOLITAN BEOGRAD

IT355 - WEB SISTEMI 2

Aspektno - orijentisano programiranje u Springu

Lekcija 03

PRIRUČNIK ZA STUDENTE

IT355 - WEB SISTEMI 2

Lekcija 03

ASPEKTNO - ORIJENTISANO PROGRAMIRANJE U SPRINGU

UVOD

Lekcija će se baviti naprednim tehnikama razvoja softera, aspektno-orijentisanim programiranjem.

U ovoj lekciji biće prezentovana upotreba Spring okvira u *aspektno-orijentisanom programiranju* (AOP). Poseban akcenat će biti na nekim naprednim temama koje su u vezi sa ključnim aspektno-orijentisanim konceptom - *savetom*. Aspektno-orijentisano programiranje podržano je sa više različitih notacija i okvira. U ovoj lekciji će biti pokazano kako se koristi *AspectJ* okvir za Spring aplikacije.

Sve novije verzije Spring okvira dozvoljavaju da se aspekti definišu u POJO formi putem AspectJ anotacija ili XML konfiguracija u konfiguracionoj datoteci zrna. Ključna implementaciona tehnologija za Spring AOP je identična u svim verzijama Spring okvira, a to je dinamički ***proxy*** . To znači da je Spring AOP unazad kompatibilan, a to znači da se standardni AOP koncepti: ***saveti*** *(*advices*)*, *tačke prekida (*pointcuts*)* i *auto-proxy* kreatori mogu koristiti u svim ranijim verzijama Spring AOP.

AspectJ je najstariji i najpopularniji AOP okvir. Spring podržava pisanje *AspectJ* POJO aspekata pomoću ***AspectJ*** anotacija iz vlastitog *AspectJ AOP* okvira. Budući da ove anotacije podržavaju različiti ***AspectJ*** okviri, kreirane aspekte moguće je ponovo koristiti bilo kojim drugim *AOP* okvirom koji podržava ***AspectJ*** .

Ovde bi posebno trebalo napomenuti da primena AOP koncepata u Springu ima određena ograničenja. Spring dozvoljava upotrebu aspekata za zrna koja su deklarisana isključivo u Spring IoC kontejneru. U drugom slučaju, ukoliko se aspekti koriste izvan oblasti kontejnera, neophodno je koristiti *AspectJ* okvir.

Konačno, savladavanjem ove lekcije, studenti će biti u mogućnosti da kreiraju vlastite POJO aspekte primenom *Spring AOP* okvira.

Poglavlje 1

Podrška AspectJ anotacijama u Springu

OBEZBEĐIVANJE PODRŠKE ZA ASPECTJ ANOTACIJE

Neophodno je dodati prazan XML element aop:aspectj-autoproxy

PROBLEM: Spring podržava primenu POJO aspekata kreiranih pomoću AspectJ anotacija iz vlastitog AOP okvira.

REŠENJE: Za obezbeđivanje podrške *AspectJ* anotacijama, neophodno je dodati prazan XML element *<aop:aspectj-autoproxy>* u konfiguracionu datoteku zrna. U tom slučaju, Spring će automatski generisati proxy za svako zrno koje se podudara sa odgovarajućim AspectJ aspektom.

U slučaju kada interfejsi nisu dostupni ili nisu primenjeni u dizajnu aplikacije, moguće je kreirati *proxy* oslanjajući se na *CGLIB* (Byte Code Generation Library - https://github.com/cglib/cglib).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Za lakše razumevanje problema biće uveden primer aritmetičkog kalkulatora. Za početak, tu su dva interfejsa.

```
package com.metropolitan.calc;
public interface ArithmeticCalculator {
public double add(double a, double b);
public double sub(double a, double b);
public double mul(double a, double b);
public double div(double a, double b);
}
--------------------------------------------------------------------------------
package com.metropolitan.calc;
public interface UnitCalculator {
public double kilogramToPound(double kilogram);
public double kilometerToMile(double kilometer);
}
```

U nastavku će biti izvršena implementacija gore kreiranih interfejsa.

```
package com.metropolitan.calc;
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
public double add(double a, double b) {
```

```
double result = a + b;
System.out.println(a + " + " + b + " = " + result);
return result;
}
public double sub(double a, double b) {
double result = a - b;
System.out.println(a + " - " + b + " = " + result);
return result;
}
public double mul(double a, double b) {
double result = a * b;
System.out.println(a + " * " + b + " = " + result);
return result;
}
public double div(double a, double b) {
if (b == 0) {
throw new IllegalArgumentException("Division by zero");
}
double result = a / b;
System.out.println(a + " / " + b + " = " + result);
return result;
}
}
--------------------------------------------------------------------------------
--------------------
package com.metropolitan.calc;
public class UnitCalculatorImpl implements UnitCalculator {
public double kilogramToPound(double kilogram) {
double pound = kilogram * 2.2;
System.out.println(kilogram + " kilogram = " + pound + " pound");
return pound;
}
public double kilometerToMile(double kilometer) {
double mile = kilometer * 0.62;
System.out.println(kilometer + " kilometer = " + mile + " mile");
return mile;
}
}
```

PODRŠKA ZA ASPECTJ ANOTACIJAMA

Kada Spring IoC kontejner primeti <aspectjautoproxy> element, automatski će kreirati proksije za zrna koja se poklapaju sa aspektima.

Za obezbeđivanje AOP podrške u ovom primeru, neophodno je kreirati prazan XML element *<aop:aspectj-autoproxy>* u konfiguracionoj datoteci zrna. Dalje, neophodno je dodati definiciju ***aop*** šeme u korenskom *<beans>* elementu. Kada Spring IoC kontejner primeti *<aop:aspectjautoproxy>* element, automatski će kreirati *proksije* za zrna koja se poklapaju sa aspektima.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
          http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring-aop-4.3.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
          http://www.springframework.org/schema/util http://www.springframework.org/
schema/util/spring-util-4.3.xsd
">
    <aop:aspectj-autoproxy />
    <bean id="arithmeticCalculator"
          class="com.metropolitan.calc.ArithmeticCalculatorImpl" />
    <bean id="unitCalculator"
          class="com.metropolitan.calc.UnitCalculatorImpl" />
    <bean class="com.metropolitan.calc.CalculatorLoggingAspect" />
</beans>
```

NAPOMENA: Za korišćenje AspectJ anotacija u Spring aplikacijama, neophodno je uključiti odgovarujeće zavisnosti. Primenom Maven-a, u *pom.xml* datoteku, neophodno je dodati sledeće linije:

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-aop</artifactId>
<version>${spring.version}</version>
</dependency>
```

Sledi listing datoteke pom.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.metropolitan</groupId>
    <artifactId>Calc</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
```

```
            <version>4.3.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
</project>
```

PODRŠKA ZA ASPECTJ ANOTACIJAMA - JAVA KONFIGIRACIJE

Neophodno je dati način savremenijeg podešavanja Spring AOP aplikacija

Pored uključivanja podrške za *AspectJ* anotacije, iz *XML* konfiguracionog fajla, moguće je na, savremeniji način uključiti ovu podršku primenom dobro poznatih Java konfiguracionih klasa. Da bi aplikacija mogla da koristi AOP anotacije, neophodno je konfiguracionu klasu zrna obeležiti anotacijom *@EnableAspectJAutoProxy*. Definicija ove anotacije data je sledećim listingom:

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented

@Import(value=org.springframework.context.annotation.AspectJAutoProxyRegistrar.class
)
public @interface EnableAspectJAutoProxy
```

Ukoliko programer nastoji da koristi konfiguracionu Java klasu, umesto definisane XML konfiguracije (videti prethodni OU), on može da u vlastiti projekat doda klasu, na primer AppConfig.java, sa sledećim listingom:

```
@Configuration
 @EnableAspectJAutoProxy
 public class AppConfig {

     @Bean
     public ArithmeticCalculatorImpl arithmeticCalculatorImpl() {
         return new ArithmeticCalculatorImpl();
     }

     @Bean
     public UnitCalculatorImpl unitCalculatorImpl() {
         return new UnitCalculatorImpl();
     }

     @Bean
     public CalculatorLoggingAspect calculatorLoggingAspect() {
         return new CalculatorLoggingAspect();
     }
 }
```

Poglavlje 2

Deklarisanje aspekata pomoću AspectJ anotacija

ASPECTJ ANOTACIJE

AspectJ dozvoljava da aspekti budu pisani kao anotirani POJO.

PROBLEM: Nakon spajanja sa još jednim AOP pristupom ***AspectWerkz*** , *AspectJ* dozvoljava da aspekti budu pisani kao anotirani POJO pomoću skupa *AspectJ* anotacija. Ovakve aspekte podržava *Spring AOP*. Međutim, da bi mogli da budu upotrebljeni, moraju da budu registrovani u *Spring IOC kontejneru*.

REŠENJE: *AspectJ* aspekti u Springu se registruju jednostavno deklarisanjem odgovarajućih instanci zrna u *Spring IoC kontejneru*. Kada je *ApsectJ* omogućen u Spring IoC kontejneru, kreiraju se proksiji za zrna koja se podudaraju sa aspektima.

Primenom *AspectJ* anotacija, aspekti predstavljaju JAVA klasa obeležene anotacijom @Aspect. Savet je jednostavna JAVA metoda sa odgovarajućom anotacijom. Na savete je moguće primeniti sledeće anotacije: @Before, @After, @AfterReturning, @AfterThrowing i @Around.

SAVET TIPA @BEFORE

@Before savet je vezan za kod koji se primenjuje pre izvršenja izvesne funkcionalnosti programa.

Aspektno-orijentisano programiranje je zamišljeno i realizovano sa namerom da se eliminiše zamršeni kod (crosscutting) sa kojim standardi objektno-orijentisani pristup nije u potpunosti mogao da se izbori. AOP povećava stepen modularnosti softvera i na taj način olakšava njegovo održavanje. Kreiranje ***@Before*** saveta je neophodno za savladavanje problema zamršenog koda koji se primenjuje u programu nekoj tački pre izvršenja izvesne funkcionalnosti.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

/**
 *
```

```
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}
```

Ovaj izraz presečne tačke (mesto u programu gde se primenjuje aspekt) odgovara izvršenju metode metode *add()* interfejsa *ArithmeticCalculator.* Džoker znak * u izrazu odgovara bilo kojem modifikatoru (***public***, ***protected*** ili ***private***) i povratnom tipu. Za registrovanje ovog aspekta u IoC kontejneru neophodno je deklarisati instancu odgovarajućeg zrna.

```
<beans ...>
...
<bean class="com.metropolitan.calculator.CalculatorLoggingAspect" />
</beans>
```

Aspekt je moguće testirati sledećom glavnom klasom.

```
package com.metropolitan.calc;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 *
 * @author Vladimir Milicevic
 */
public class Main {

    public static void main(String[] args) {
        ApplicationContext context
                = new ClassPathXmlApplicationContext("classpath:/META-INF/
beans.xml");
        ArithmeticCalculator arithmeticCalculator
                = (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);
        UnitCalculator unitCalculator
                = (UnitCalculator) context.getBean("unitCalculator");
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}
```

Tačke izvršenja saveta, koje se podudaraju sa presečnom tačkom, u AOP žargonu se nazivaju *tačke dodira* ili *tačke unije* (join point). To znači da je presečna tačka izraz koji se poklapa sa skupom tačaka dodira u kojima se izvršava akcija - savet.

SAVET TIPA @BEFORE - DOPUNSKA RAZMATRANJA

Argument tipa JoinPoint se kreira u metodi saveta.

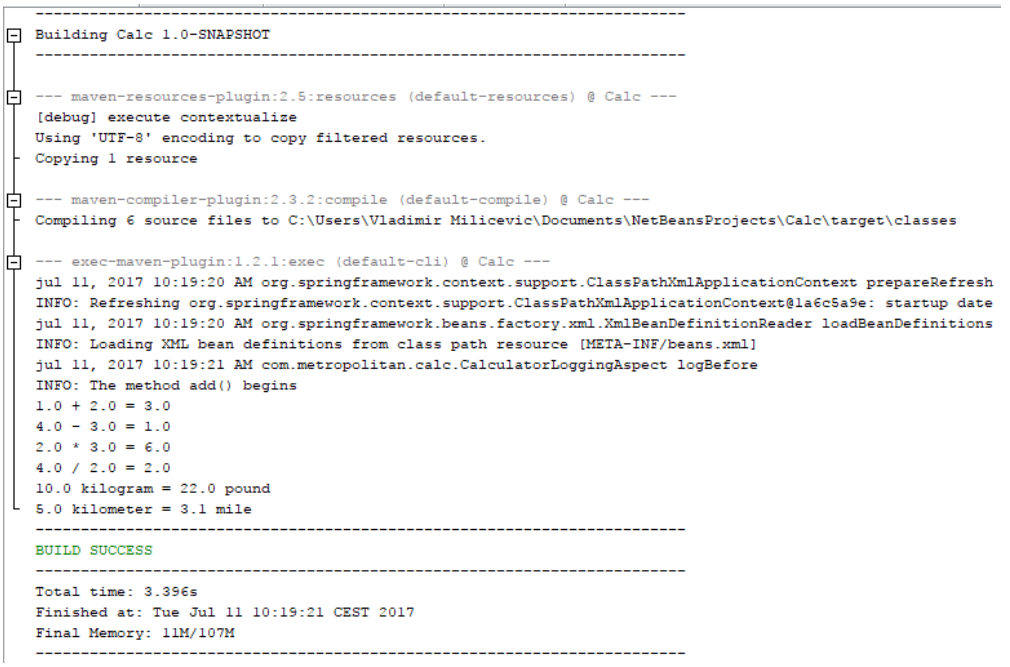
Da bi savet pristupio detaljima izvesne tačke dodira, neophodno je kreirati argument tipa JoinPoint u metodi saveta. Tada je moguće pristupiti implementacionim detaljima tačke dodira kao što su naziv metode i vrednosti argumenata. Sada je moguće proširiti definiciju presečne tačke za podudaranje sa svim metodama, primenom džokera za nazive klasa i metoda.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}
```

TESTIRANJE IZVRŠAVANJA SAVETA @BEFORE

Testira se kreirani kod

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.



Slika 2.1 Testiranje izvršavanja saveta @Before [izvor: autor]

SAVET TIPA @AFTER

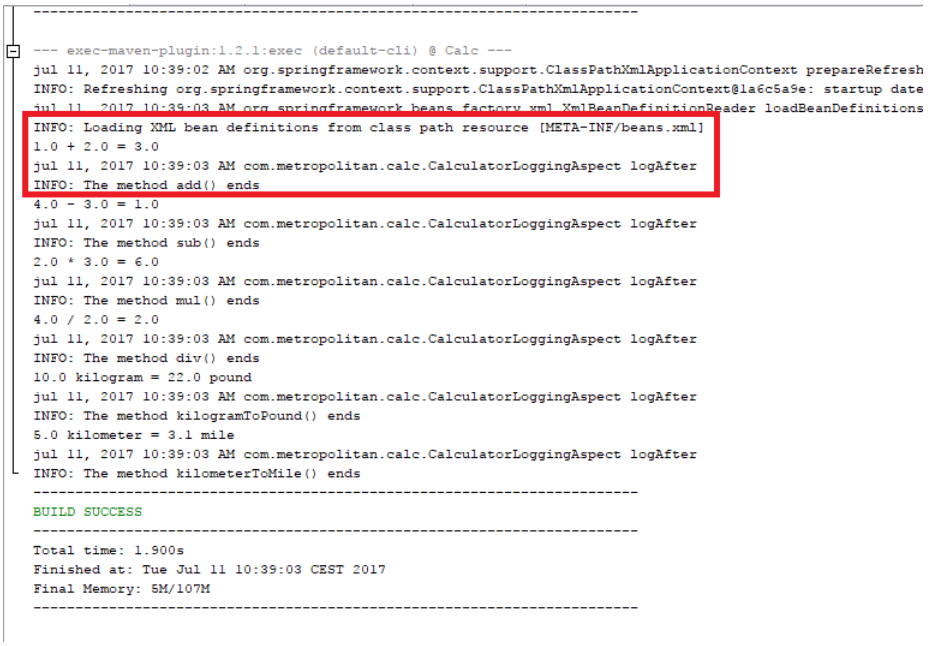
Savet @After se izvršava nakon realizovanja tačke dodira.

Savet ***@After*** se izvršava nakon realizovanja tačke dodira bilo da je vraćen rezultat ili se javio izuzetak. Sledeći savet ukazuje kraj metode kalkulatora.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends");
```

```
    }
}
```

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.



Slika 2.2 Testiranje izvršavanja saveta @After [izvor: autor]

SAVETI TIPA @AFTERRETURNING I @AFTERTHROWING

Savet After se se izvršava bez obzira da li tačka dodira vraća rezultat ili izuzetak.

Savet ***@After*** se se izvršava bez obzira da li tačka dodira vraća rezultat ili izuzetak. Ukoliko je cilj realizovanje povezivanja isključivo u slučaju vraćanja rezultata, primenjuje se anotacija ***@AfterReturning***.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

/**
 *
 * @author Vladimir Milicevic
```

```
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends");
    }
}
```

Ovim savetom moguće je pristupiti povratnoj vrednosti tačke dodira dodavanjem povratnog atributa anotaciji ***@AfterReturning***. Vrednost atributa odgovara nazivu argumenta metode saveta koja prosleđuje vraćenu vrednost.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @AfterReturning(
            pointcut = "execution(* *.*(..))",
            returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends with " + result);
    }
}
```

Ukoliko je cilj realizovanje logovanja u slučaju vraćanja izuzetka, primenjuje se anotacija ***@AfterThrowing***.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
```

```
import org.aspectj.lang.annotation.AfterThrowing;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @AfterThrowing("execution(* *.*(..))")
    public void logAfterThrowing(JoinPoint joinPoint) {
        log.error("An exception has been thrown in "
                + joinPoint.getSignature().getName() + "()");
    }
}
```

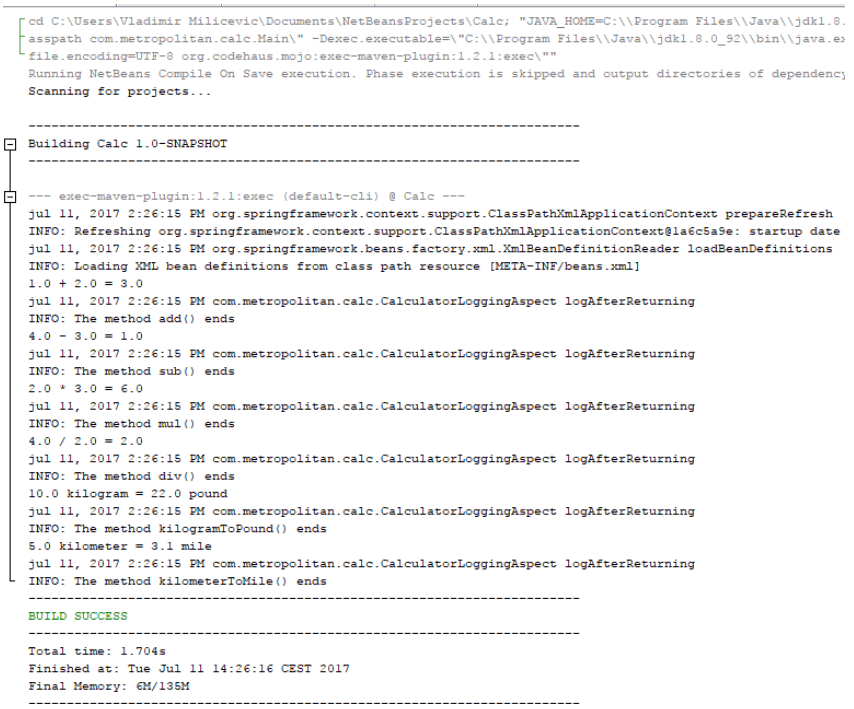
Slično se pristupa izuzetku tačke dodira, dodavanjem povratnog atributa anotaciji ***@AfterThrowing***.

```
package com.metropolitan.calc;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @AfterThrowing(
            pointcut = "execution(* *.*(..))",
            throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.error("An exception " + e + " has been thrown in "
                + joinPoint.getSignature().getName() + "()");
    }
}
```

TESTIRANJE IZVRŠAVANJA SAVETA @AFTERRETURNING

Testira se kreirani kod za nov savete AfterReturning.

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.



Slika 2.3 Testiranje izvršavanja saveta @AfterReturning [izvor: autor]

```
asspath com.metropolitan.calc.Main\" -Dexec.executable=\"C:\\Program Files\\Java\\jdk1.8.0_92\\bin\\java.ex
file.encoding=UTF-8 org.codehaus.mojo:exec-maven-plugin:1.2.1:exec\""
Running NetBeans Compile On Save execution. Phase execution is skipped and output directories of dependency
Scanning for projects...

------------------------------------------------------------------------
Building Calc 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Calc ---
jul 11, 2017 2:30:45 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1a6c5a9e: startup date
jul 11, 2017 2:30:45 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [META-INF/beans.xml]
1.0 + 2.0 = 3.0
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method add() ends with 3.0
4.0 - 3.0 = 1.0
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method sub() ends with 1.0
2.0 * 3.0 = 6.0
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method mul() ends with 6.0
4.0 / 2.0 = 2.0
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method div() ends with 2.0
10.0 kilogram = 22.0 pound
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method kilogramToPound() ends with 22.0
5.0 kilometer = 3.1 mile
jul 11, 2017 2:30:45 PM com.metropolitan.calc.CalculatorLoggingAspect logAfterReturning
INFO: The method kilometerToMile() ends with 3.1
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.629s
Finished at: Tue Jul 11 14:30:46 CEST 2017
Final Memory: 5M/107M
------------------------------------------------------------------------
```

Slika 2.4 Testiranje izvršavanja saveta @AfterReturning sa povratnim rezultatom [izvor: autor]

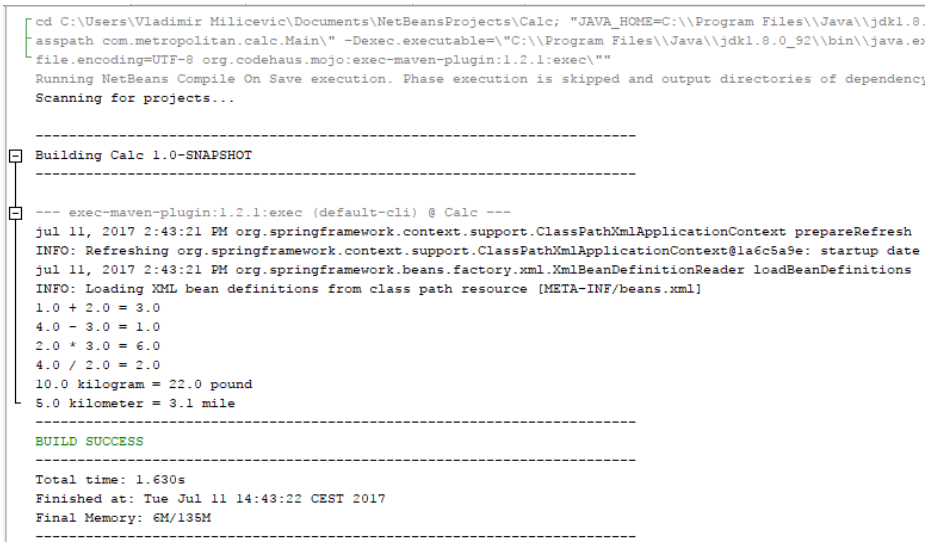
TESTIRANJE IZVRŠAVANJA SAVETA @AFTERTHROWING

Testira se kreirani kod za nov savete AfterThrowing

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.

```
------------------------------------------------------------------------
Building Calc 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Calc ---
jul 11, 2017 2:39:13 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1a6c5a9e: startup date
jul 11, 2017 2:39:13 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [META-INF/beans.xml]
1.0 + 2.0 = 3.0
4.0 - 3.0 = 1.0
2.0 * 3.0 = 6.0
4.0 / 2.0 = 2.0
10.0 kilogram = 22.0 pound
5.0 kilometer = 3.1 mile
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.531s
Finished at: Tue Jul 11 14:39:14 CEST 2017
Final Memory: 4M/107M
------------------------------------------------------------------------
```

Slika 2.5 Testiranje izvršavanja saveta @AfterThrowing [izvor: autor]



Slika 2.6 Testiranje izvršavanja saveta @AfterThrowing sa povratnim rezultatom [izvor: autor]

SAVET TIPA @AROUND

Savet tipa @Around daje punu kontrolu nad tačkom dodira.

Poslednji tip saveta je ***around*** koji je najmoćniji i daje punu kontrolu nad tačkom dodira. U ovom slučaju je moguće kombinovati sve akcije prethodnih saveta u okviru jednog saveta. Sledeći savet predstavlja kombinaciju ***before, after returning*** i ***after throwing*** saveta koji su prethodno kreirani. Neophodno je još primetiti da argument tačke dodira, za ovaj savet, mora biti ***ProceedingJoinPoint***. Radi se o podinterfejsu interfejsa ***JoinPoint*** koji omogućava kontrolisanje nastavka procesiranja sa originalnom tačkom dodira.

```
package com.metropolitan.calc;

import java.util.Arrays;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Around("execution(* *.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("The method " + joinPoint.getSignature().getName()
```

```
                + "() begins with " + Arrays.toString(joinPoint.getArgs()));
        try {
            Object result = joinPoint.proceed();
            log.info("The method " + joinPoint.getSignature().getName()
                    + "() ends with " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Illegal argument "
                    + Arrays.toString(joinPoint.getArgs()) + " in "
                    + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
}
```

Savet tipa ***@Around*** je veoma moćan i fleksibilan u smislu da je moguće menjati originalne vrednosti argumenata, a samim tim i vrednost vraćenog rezultata. Međutim, ovaj savet se mora koristiti sa velikom pažnjom budući da se lako može dogoditi da se zaboravi nastavak izvršenja aplikacije u originalnoj tački dodira.

TESTIRANJE IZVRŠAVANJA SAVETA @AROUND

Testira se kreirani kod za nov savet Around

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.

```
------------------------------------------------------------------------
Building Calc 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Calc ---
jul 11, 2017 2:50:12 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1a6c5a9e: startup date
jul 11, 2017 2:50:12 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [META-INF/beans.xml]
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method add() begins with [1.0, 2.0]
1.0 + 2.0 = 3.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method add() ends with 3.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method sub() begins with [4.0, 3.0]
4.0 - 3.0 = 1.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method sub() ends with 1.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method mul() begins with [2.0, 3.0]
2.0 * 3.0 = 6.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method mul() ends with 6.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method div() begins with [4.0, 2.0]
4.0 / 2.0 = 2.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method div() ends with 2.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilogramToPound() begins with [10.0]
10.0 kilogram = 22.0 pound
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilogramToPound() ends with 22.0
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilometerToMile() begins with [5.0]
5.0 kilometer = 3.1 mile
jul 11, 2017 2:50:12 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilometerToMile() ends with 3.1
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.549s
Finished at: Tue Jul 11 14:50:12 CEST 2017
```

Slika 2.7 Testiranje izvršavanja saveta @Around [izvor: autor]

Poglavlje 3

Informacija tačke dodira

PRISTUP INFORMACIJI TAČKE DODIRA

Aspekt može da pristupi detaljima tačke dodira deklarisanjem argumenta tipa JoinPoint.

PROBLEM: U *AOP*-u savet se primenjuje u različitim tačkama izvršavanja aplikacije koje se nazivaju tačkama dodira ili tačkama unije. Da bi aspekt primenio konkretnu akciju neophodno je, često, da dobije izvesne informacije o tački dodira.

REŠENJE: Aspekt može da pristupi detaljima tačke dodira deklarisanjem argumenta tipa ***org.aspectj.lang.JoinPoint*** u potpisu metode saveta.

Za demonstriranje i lakše razumevanje problema biće realizovan odgovarajući primer. Informacija obuhvata vrstu tačke dodira (*method-execution* u Spring AOP), potpis metode (tip i naziv metode) i vrednosti argumenata, kao i ciljni i *proxy* objekti.

```
package com.metropolitan.calc;

import java.util.Arrays;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* *.*(..))")
    public void logJoinPoint(JoinPoint joinPoint) {
        log.info("Join point kind : "
                + joinPoint.getKind());
        log.info("Signature declaring type : "
                + joinPoint.getSignature().getDeclaringTypeName());
        log.info("Signature name : "
                + joinPoint.getSignature().getName());
```

```
        log.info("Arguments : "
                + Arrays.toString(joinPoint.getArgs()));
        log.info("Target class : "
                + joinPoint.getTarget().getClass().getName());
        log.info("This class : "
                + joinPoint.getThis().getClass().getName());
    }
}
```

Originalno zrno koje je obuhvata proxy naziva se ciljno zrno ili ciljni objekat tj. target . Proxy objekat nosi naziv ***this***. Iz priloženog izlaza, generisanog kodom, moguće je videti da ovi objekti ne pripadaju istim klasama.

```
------------------------------------------------------------------------
Building Calc 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Calc ---
jul 11, 2017 2:59:55 PM org.springframework.context.support.ClassPathXmlApplication
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext
jul 11, 2017 2:59:55 PM org.springframework.beans.factory.xml.XmlBeanDefinitionRead
INFO: Loading XML bean definitions from class path resource [META-INF/beans.xml]
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Join point kind : method-execution
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Signature declaring type : com.metropolitan.calc.ArithmeticCalculator
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Signature name : add
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Arguments : [1.0, 2.0]
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Target class : com.metropolitan.calc.ArithmeticCalculatorImpl
jul 11, 2017 2:59:55 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: This class : com.sun.proxy.$Proxy5
```

Slika 3.1 Pristup informaciji tačke dodira [izvor: autor]

Poglavlje 4

Određivanje prednosti među aspektima

REDOSLED PRIMENE ASPEKATA

Prednost među aspektima se definiše implementacijom interfejsa Ordered ili anotacijom @Order.

PROBLEM: Kada postoji više aspekata koji se primenjuju na istu tačku dodira, prednost u njihovoj primeni nije definisana ukoliko se ne navede jasno.

REŠENJE: Prednost među aspektima može da se definiše na dva načina - implementacijom interfejsa Ordered ili primenom anotacije ***@Order***.

Neka je kreiran još jedan aspekt za proveru argumenata kalkulatora. U ovom aspektu se koristi samo jedan ***before*** savet.

```
package com.metropolitan.calc;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorValidationAspect {

    @Before("execution(* *.*(double, double))")
    public void validateBefore(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            validate((Double) arg);
        }
    }

    private void validate(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Positive numbers only");
        }
    }
}
```

Zatim je neophodno registrovati aspekt.

```
<beans ...>
...
<bean class="com.metropolitan.calc.CalculatorLoggingAspect" />
 <bean class="com.metropolitan.calc.CalculatorValidationAspect" />
</beans>
```

Prednost između aspekata je nedefinisana za sada. Nju je moguće definisati primenom metode *getOrder()* interfejsa ***Ordered*** ili anotacijom ***@Order***. Niža povratna vrednost metode, ili vrednost obuhvaćena anotacijom, znače veći prioritet.

```
package com.metropolitan.calc;
...
import org.springframework.core.Ordered;
@Aspect
public class CalculatorValidationAspect implements Ordered {
...
public int getOrder() {
return 0;
}
}


package com.metropolitan.calc;
...
import org.springframework.core.Ordered;
@Aspect
public class CalculatorLoggingAspect implements Ordered {
...
public int getOrder() {
return 1;
}
}
```

```
package com.metropolitan.calc;
...
import org.springframework.core.annotation.Order;
@Aspect
@Order(0)
public class CalculatorValidationAspect {
...
}


package com.metropolitan.calc;
...
import org.springframework.core.annotation.Order;
@Aspect
@Order(1)
public class CalculatorLoggingAspect {
```

```
...
}
```

ZADATAK ZA SAMOSTALNI RAD 1

Pokušajste sami da uradite

Prikazani kod iz ovog objekta učenja pokušajte da samostalno implementirate u postojeći primer koji ste preuzeli preko aktivnosti Shared Resources.

Poglavlje 5

Ponovna upotreba definicije tačke prekida

DEFINICIJA PRESEČNE TAČKE

Isti izraz presečne tačke može biti ponovljen u više saveta.

PROBLEM: Kada se kreira *AspectJ* aspekt, moguće je direktno ugraditi izraz presečne tačke u anotaciju saveta. Isti izraz presečne tačke može biti ponovljen u više saveta.

REŠENJE: *AspectJ* dozvoljava definisanje tačke prekida nezavisno tako da može biti višestruko upotrebljena.

AspectJ deklariše presečnu tačku kao prostu metodu obeleženu anotacijom @Pointcut. Telo metode je obično prazno jer je nelogično mešanje definicije presečne tačke sa programskom logikom. Modifikator pristupa metodi presečne tačke kontroliše, takođe, i vidljivost presečne tačke. Ostali saveti se obraćaju ovoj presečnoj tački pomoću naziva metode.

```
package com.metropolitan.calculator;
...
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class CalculatorLoggingAspect {
...
@Pointcut("execution(* *.*(..))")
private void loggingOperation() {}
@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
...
}
@AfterReturning(
pointcut = "loggingOperation()",
returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
...
}
@AfterThrowing(
pointcut = "loggingOperation()",
throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
...
}
@Around("loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
```

```
...
}
}
```

Obično, ako je presečna tačka deljena između više aspekata, dobar pristup je njihovo obuhvatanje zajedničkom klasom, pri čemu moraju da budu deklarisani kao javni.

```
package com.metropolitan.calculator;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class CalculatorPointcuts {
@Pointcut("execution(* *.*(..))")
public void loggingOperation() {}
}
```

Kada se vrši obraćanje ovoj presečnoj tački, neophodno je uključiti i naziv klase. Ukoliko se klasa i aspekt ne nalaze u istom paketu, neophodno je uključiti i naziv paketa.

```
package com.metropolitan.calculator;
...
@Aspect
public class CalculatorLoggingAspect {
...
@Before("CalculatorPointcuts.loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
...
}
@AfterReturning(
pointcut = "CalculatorPointcuts.loggingOperation()",
returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
...
}
@AfterThrowing(
pointcut = "CalculatorPointcuts.loggingOperation()",
throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
...
}
@Around("CalculatorPointcuts.loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
...
}
}
```

PONOVNA UPOTREBA PRESEČNE TAČKE - DEMO

Test koda sa višestrukom primenom presečne tačke

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*, izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom. Pre toga sledi kod klase *CalculatorLoggingAspect.java* koja je posebno modifikovana za potrebe primera ponovne upotrebe presečne tačke.

Ovaj primer, kao i većina pokaznih primera u ovim materijalima, priložen je kompletno urađen na kraju ove secije kao dodatna akivnost Shared Resources.

```
package com.metropolitan.calc;

import java.util.Arrays;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.core.Ordered;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorLoggingAspect implements Ordered {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("CalculatorPointcuts.loggingOperation()")
    public void logJoinPoint(JoinPoint joinPoint) {
        log.info("Join point kind : "
                + joinPoint.getKind());
        log.info("Signature declaring type : "
                + joinPoint.getSignature().getDeclaringTypeName());
        log.info("Signature name : "
                + joinPoint.getSignature().getName());
        log.info("Arguments : "
                + Arrays.toString(joinPoint.getArgs()));
        log.info("Target class : "
                + joinPoint.getTarget().getClass().getName());
        log.info("This class : "
                + joinPoint.getThis().getClass().getName());
    }

    @Around("CalculatorPointcuts.loggingOperation()")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() begins with " + Arrays.toString(joinPoint.getArgs()));
        try {
```

```
            Object result = joinPoint.proceed();
            log.info("The method " + joinPoint.getSignature().getName()
                    + "() ends with " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Illegal argument "
                    + Arrays.toString(joinPoint.getArgs()) + " in "
                    + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }

    public int getOrder() {
        return 1;
    }
}
```

```
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilogramToPound() begins with [10.0]
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Join point kind : method-execution
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Signature declaring type : com.metropolitan.calc.UnitCalculator
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Signature name : kilogramToPound
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Arguments : [10.0]
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: Target class : com.metropolitan.calc.UnitCalculatorImpl
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logJoinPoint
INFO: This class : com.sun.proxy.$Proxy8
10.0 kilogram = 22.0 pound
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilogramToPound() ends with 22.0
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilogramToPound() ends with 22.0
jul 11, 2017 6:27:15 PM com.metropolitan.calc.CalculatorLoggingAspect logAround
INFO: The method kilometerToMile() begins with [5.0]
```

Slika 5.1 Ponovna upotreba presečne tačke - demo [izvor: autor]

Poglavlje 6

Izrazi presečnih tačaka

PISANJE IZRAZA PRESEČNIH TAČAKA U ASPECTJ

Spring AOP podržava samo metodu execution() za zrna deklarisana u Spring IoC kontejneru.

PROBLEM: Problem zamršenog koda može da se javi u različitim tačkama izvršavanja programa poznatima kao tačke dodira. Zbog različitosti tačaka dodira neophodno je obezbediti moćan jezik izraza za obezbeđivanje podudaranja sa njima.

REŠENJE: AspectJ jezik presečnih tačaka (AspectJ pointcut language) je moćan alat za pisanje izraza podudaranja sa tačkama dodira. Spring AOP podržava samo metodu za izvršavanje tačaka dodira za zrna deklarisana u *Spring IoC* kontejneru. Iz ovog razloga će biti obrađeni samo izrazi presečnih tačaka podržani sa *Spring AOP*. Ukoliko se koriste Spring tačke izvan okvira kontejnera, javiće se izuzetak ***IllegalArgumentException***.

Spring AOP koristi elemente *AspectJ* jezika presečnih tačaka za definisanje presečnih tačaka. *Spring AOP* interpretira ove izraze tokom vremena izvršavanja aplikacije koristeći biblioteke koje obezbeđuje *AspectJ*.

ŠABLONI POTPISIVANJA METODA

Izrazi presečnih tačaka se koriste za povezivanje sa brojnim metodama na osnovu njihovih potpisa.

Jedan od najčešćih načina primene izraza presečnih tačaka jeste povezivanje sa brojnim metodama na osnovu njihovih potpisa. Na primer, sledećim izrazom je obezbeđeno podudaranje sa svim metodama koje su deklarisanje u ***ArithmeticCalculator*** interfejsu. Džoker znak * menja metode sa bilo kojim tipom ili modifikatorom. Dve tačke u listi argumenata zamenjuju bilo koji broj argumenata.

```
execution(* com.metropolitan.calcr.ArithmeticCalculator.*(..))
```

Moguće je isključiti naziv paketa ukoliko su ciljna klasa ili interfejs locirani u istom paketu kao i aspekt.

```
execution(* ArithmeticCalculator.*(..))
```

Sledeći izraz obezbeđuje preklapanje sa svim javnim metodama deklarisanim u ***ArithmeticCalculator*** interfejsu.

```
execution(public * ArithmeticCalculator.*(..))
```

Moguće je ograničiti povratni tip metode, na primer na *double.*

```
execution(public double ArithmeticCalculator.*(..))
```

Ograničenja je moguće uvesti i u listi argumenata. Na primer, prvi argument je tipa *double,* a ostali su proizvoljni.

```
execution(public double ArithmeticCalculator.*(double, ..))
```

Moguće je definisati i sve dozvoljene tipove argumenata metoda sa kojima se traži podudaranje.

```
execution(public double ArithmeticCalculator.*(double, double))
```

DODATNA RAZMATRANJA

Ponekad nije moguće pronaći zajedničke osobine traženih metoda.

Iako je *AspectJ* jezik izraza presečnih tačaka moćan alat za podudaranje sa različitim tačkama dodira, ponekad nije moguće pronaći zajedničke osobine poput: modifikatora, povratnih tipova, šablona naziva metoda ili argumenata za tražene metode. Tada bi trebalo razmotriti primenu izvesnih anotacija koje je moguće primeniti na nivoima metoda i tipova.

```
package com.metropolitan.calc;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target( { ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LoggingRequired {
}
```

Sada je moguće anotirati sve metode koje zahtevanju *logovanje* sa ovom notacijom.

```
package com.metropolitan.calc;
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
@LoggingRequired
public double add(double a, double b) {
...
}
@LoggingRequired
public double sub(double a, double b) {
...
}
```

```
@LoggingRequired
public double mul(double a, double b) {
...
}
```

Na kraju, moguće je napisati izraz presečne tačke za podudaranje sa svim metodama koje poseduju anotaciju ***@LoggingRequired***.

```
@annotation(com.metropolitan.calc.LoggingRequired)
```

ŠABLONI POTPISIVANJA TIPOVA

Sledeći tip izraza odnosi se na traženje svih tačaka dodira po određenom tipu.

Sledeći tip izraza odnosi se na traženje svih tačaka dodira po određenom tipu. Primenjeno na *Spring AOP*, oblast ovih tačaka prekida je sužena na traženje svih izvršavanja metoda u okviru konkretnih tipova. Na primer, sledeća presečna tačka traži sva izvršenja metoda tačaka dodira u okviru paketa: ***com.metropolitan.calc*** .

```
within(com.metropolitan.calc.*)
```

Za traženje tačaka dodira u paketu i njegovim potpaketima, neophodno je dodati još jednu tačku pre džoker znaka *.

```
within(com.metropolitan.calc..*)
```

Ukoliko se pretraga odnosi na jednu konkretnu klasu, moguće je napisati izraz na sledeći način:

```
within(com.metropolitan.calc.ArithmeticCalculatorImpl)
```

Ukoliko je ciljna klasa smeštena u istom paketu kao i aspekt, moguće je izostaviti naziv paketa.

```
within(ArithmeticCalculatorImpl)
```

Takođe, moguće je tražiti preklapanje sa izvršenjem posmatranih metoda u svim klasama koje implementiraju interfejs *ArithmeticCalculator* dodavanjem džoker znaka +

```
within(ArithmeticCalculator+)
```

Sada anotacija *@LoggingRequired* može biti korišćena na nivou klasa, umesto na nivou metoda.

```
package com.metropolitan.calc;
@LoggingRequired
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
```

```
...
}
```

Sada je moguće traženje tačaka dodira u klasama koje su obeležene anotacijom @LoggingRequired.

```
@within(com.metropolitan.calc.LoggingRequired)
```

NAPOMENA:

Nove verzije Springa obezbeđuju šablone za nazive zrna. Na primer, sledeći izraz presečne tačke traži zrna čiji se naziv završava stringom *Calculator.* Ovaj tip presečne tačke je podržan isključivo XML baziranim Spring AOP konfiguracijama, nije podržan originalnim AspectJ anotacijama.

```
bean(*Calculator)
```

KOMBINOVANJE IZRAZA PRESEČNIH TAČAKA

Izrazi mogu biti povezani logičkim operatorima.

Primenom logičkih operatora **&&** *(and)*, || *(or)* i **!** *(not)* moguće je vršiti kombinovanje izraza presečnih tačaka u složenije izraze. Dat je sledeći primer kojim će ovo biti ilustrovano.

```
within(ArithmeticCalculator+) || within(UnitCalculator+)
```

Na ovaj način traže se tačke dodira unutar klasa koje implementiraju jedan od interfejsa *ArithmeticCalculator* ili ***UnitCalculator*** .

Operandi ovih operatora mogu biti bilo koje presečne tačke ili reference na druge presečne tačke.

```
package com.metropolitan.calc;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class CalculatorPointcuts {
@Pointcut("within(ArithmeticCalculator+)")
public void arithmeticOperation() {}
@Pointcut("within(UnitCalculator+)")
public void unitOperation() {}
@Pointcut("arithmeticOperation() || unitOperation()")
public void loggingOperation() {}
}
```

DEKLARISANJE PARAMETARA PRESEČNIH TAČAKA

Jedan način pristupanja informacijama presečne tačke je refleksija.

Jedan način pristupanja informacijama presečne tače je refleksija (na primer, putem argumenta tipa ***org.aspectj.lang.JoinPoint*** u metodi saveta). Na drugi način, do ovih informacija je moguće doći i korišćenjem specijalnih izraza tačaka prekida. Na primer, izrazi ***target()*** i ***args()*** uzimaju ciljni objekat i vrednosti argumenata tekuće tačke dodira i prikazuju ih kao parametre presečne tačke. Ovi parametri će biti prosleđeni metodi saveta putem argumenata istog naziva.

```
package com.metropolitan.calc;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class CalculatorLoggingAspect {
...
@Before("execution(* *.*(..)) & target(target) && args(a,b)")
public void logParameter(Object target, double a, double b) {
log.info("Target class : " + target.getClass().getName());
log.info("Arguments : " + a + ", " + b);
}
}
```

Prilikom deklarisanja nezavisne tačke prekida koja otkriva parametre, neophodno je izvršiti njihovo uključivanje u listu parametara metode presečne tačke.

```
package com.metropolitan.calc;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class CalculatorPointcuts {
...
@Pointcut("execution(* *.*(..)) &  target(target) && args(a,b)")
public void parameterPointcut(Object target, double a, double b) {}
}
```

Svaki savet koji ukazuje na ovu parametrizovanu presečnu tačku, može da pristupi parametrima presečne tačke pomoću argumenata metode koji nose identične nazive kao i parametri.

```
package com.metropolitan.calc;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class CalculatorLoggingAspect {
...
@Before("CalculatorPointcuts.parameterPointcut(target, a, b)")
```

```
public void logParameter(Object target, double a, double b) {
log.info("Target class : " + target.getClass().getName());
log.info("Arguments : " + a + ", " + b);
}
}
```

ZADATAK ZA SAMOSTALNI RAD 2

Pokušajte sami da implementirate prikazani kod!!!

Prikazani kod iz ovog objekta učenja pokušajte da samostalno implementirate u postojeći primer koji ste preuzeli preko aktivnosti *Shared Resources*.

Poglavlje 7

"Predstavljanje" ponašanja i stanja zrnu

SAVET TIPA INTRODUCTION

Introduction je specijalan tip saveta u AOP.

PROBLEM: Ponekad je prisutno više klasa koje dele zajedničko ponašanje. U OOP, tada klase nasleđuju zajedničku roditeljsku klasu ili implementiraju isti interfejs. Ovaj problem je zapravo jedna od stavki zamršenog koda koja može biti modularizovana primenom AOP.

Takođe, mehanizmi nasleđivanja u JAVA jeziku omogućavaju nasleđivanje samo jedne klase pa, otuda, nije moguće naslediti ponašanje više klasa od jednom.

REŠENJE:***Introduction*** je specijalan tip saveta u AOP. On dozvoljava da objekti mogu dinamički da implementiraju interfejse pomoću implementacione klase za te interfejse. Dalje, moguće je implementirati više interfejsa pomoću više implementacionih klasa istovremeno, a time se postižu isti efekti kao da se radi o višestrukom nasleđivanju.

Za podršku problemu, biće uveden konkretan primer. Kreirani su interfejsi ***MaxCalculator*** i ***MinCalculator*** za kojima su definisane operacije (metode) ***max()*** *i* ***min()*** , respektivno.

```
package com.metropolitan.calc;
public interface MaxCalculator {
public double max(double a, double b);
}
-----------------------------------------------------------------------
package  com.metropolitan.calc;
public interface MinCalculator {
public double min(double a, double b);
}
```

Implementacija interfejsa je podržana naredbom ***println*** zbog lakšeg praćenja da li je metoda izvršena ili ne.

```
package com.metropolitan.calc;
public class MaxCalculatorImpl implements MaxCalculator {
public double max(double a, double b) {
double result = (a >= b) ? a : b;
System.out.println("max(" + a + ", " + b + ") = " + result);
return result;
}
}
-------------------------------------------------------------------------------
```

```
---------
package com.metropolitan.calc;
public class MinCalculatorImpl implements MinCalculator {
public double min(double a, double b) {
double result = (a <= b) ? a : b;
System.out.println("min(" + a + ", " + b + ") = " + result);
return result;
}
}
```

Ako je dat zadatak klasi *ArithmeticCalculatorImpl* da obavi izračunavanja *min()* i *max()*, javlja se problem. Zbog JAVA jednostrukog nasleđivanja nije moguće da ova klasa nasledi istovremeno klase *MaxCalculatorImpl* i ***MinCalculatorImpl*** . Mogući scenariji su nasleđivanje jedne klase i odgovarajućeg drugog interfejsa ili implementacija dva odgovarajuća interfejsa. Posledica bi bila ponavljanje deklaracija metoda.

UVOĐENJE SAVETA INTRODUCTION

Introduction dozvoljava da objekti mogu dinamički da implementiraju interfejse.

Primenom saveta tipa Introduction ***ArithmeticCalculatorImpl*** klasa može dinamički da implementira interfejse ***MaxCalculator*** i ***MinCalculato*** *r* koristeći implementacione klase ***MaxCalculatorImpl***
i ***MinCalculatorImpl*** . Još jedna značajna stvar je da nije potrebno modifikovati klasu *ArithmeticCalculatorImpl* predstavljanjem novih metoda. To praktično znači, da je ovde moguće predstavljati metode klasama čak iako kod nije dostupan.

Predstavljanje (Introducton*)*, kao savet, mora da bude deklarisano u okviru aspekta. Moguće je kreirati novi aspekt ili ponovo upotrebiti kreirani za ovu svrhu. Za deklarisanje predstavljanja moguće je koristiti anotaciju @DeclareParents.

```
package com.metropolitan.calc;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorIntroduction {

    @DeclareParents(
            value = "com.metropolitan.calc.ArithmeticCalculatorImpl",
            defaultImpl = MaxCalculatorImpl.class)
    public MaxCalculator maxCalculator;
    @DeclareParents(
```

```
            value = "com.metropolitan.calc.ArithmeticCalculatorImpl",
            defaultImpl = MinCalculatorImpl.class)
    public MinCalculator minCalculator;
}
```

Vrednost value opisuje ciljne klase, a ***defaultImpl*** označava implementacione klase za interfejse. Kroz ova dva predstavljanja, dinamički je moguće klasi *ArithmeticCalculatorImpl* dodeliti ponašanja određena parom interfejsa. Takođe, za vrednost ***value*** moguće je definisati *AspectJ* izraz preklapanja tipova kojim bi bilo moguće predstaviti neki interfejs većem broju klasa. Na kraju, neophodno je deklarisati instancu aspekta u kontekstu aplikacije.

```
<beans ...>
...
<bean class="com.metropolitan.calc.CalculatorIntroduction" />
</beans>
```

Konačno, predstavljanjem interfejsa *MaxCalculator* i *MinCalculator* aritmetičkom kalkulatoru, moguće je realizovati mehanizam koji će obaviti operacije, a primer klasu ***Main*** .

```
package com.metropolitan.calc;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 *
 * @author Vladimir Milicevic
 */
public class Main {

    public static void main(String[] args) {
        ApplicationContext context
                = new ClassPathXmlApplicationContext("classpath:/META-INF/
beans.xml");
        ArithmeticCalculator arithmeticCalculator
                = (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        MaxCalculator maxCalculator = (MaxCalculator) arithmeticCalculator;
        maxCalculator.max(1, 2);
        MinCalculator minCalculator = (MinCalculator) arithmeticCalculator;
        minCalculator.min(1, 2);
    }
}
```

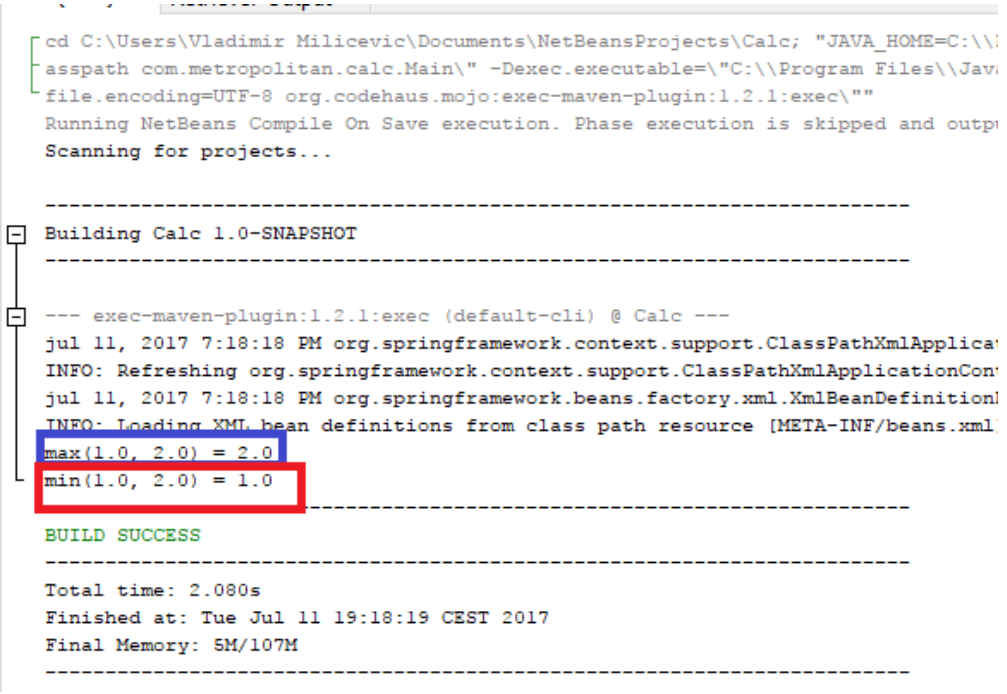
"PREDSTAVLJANJE" PONAŠANJA ZRNU - DEMO

Test koda sa primenom saveta Introduction

Sa ciljem provere valjanosti kreiranog koda, primene aspekata i urađenih konfiguracija, neophodno je pokrenuti urađeni projekat. Jednostavno u razvojnom okruženju *NetBeans IDE*,

izborom opcije *Run Project*, i ako je sve urađeno na prikazani način, trebalo bi da se dobije izlaz prikazan sledećom slikom.

Ovaj primer, kao i većina pokaznih primera u ovim materijalima, priložen je kompletno urađen na kraju ove secije kao dodatna akivnost Shared Resources.



Slika 7.1.1 "Predstavljanje" ponašanja zrnu - demo [izvor: autor]

7.1 Predstavljanje stanja zrnima

UPRAVLJANJE STANJIMA

Moguće je predstaviti novi interfejs objektima, implementacionom klasom koja čuva polja stanja.

PROBLEM: Ponekad je neophodno dodati nova stanja u grupu postojećih objekata za praćenje njihove primene. Stanja su, na primer, broj pojavljivanja, poslednji modifikovani datum i slično. Ovo ne predstavlja problem ukoliko svi objekti pripadaju istoj osnovnoj klasi. Međutim, problem se komplikuje kada je neophodno dodati ova stanja većem broju klasa koje, još, i ne pripadaju istoj hijerarhiji.

REŠENJE: Moguće je predstaviti novi interfejs objektima, odgovarajućom implementacionom klasom koja čuva polja stanja. Tada je moguće napisati novi savet za promenu stanja u skladu sa zahtevima.

Primerom se prati broj poziva svakog objekta tipa ***calculator*** . Prvo će biti kreirani interfejs za operacije brojača.

```
package com.metropolitan.calc;

/**
 *
 * @author Vladimir Milicevic
 */
public interface Counter {

    public void increase();

    public int getCount();
}
```

Dalje, biće kreirana implementaciona klasa za interfejs. Klasa će da sadrži polje *count* za čuvanje vrednosti brojača.

```
package com.metropolitan.calc;

/**
 *
 * @author Vladimir Milicevic
 */
public class CounterImpl implements Counter {

    private int count;

    public void increase() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

Predstavljanje *Counter* interfejsa svim *Calculator* objektima moguće je realizovati konkretnim izrazom podudaranja tipova.

```
package com.metropolitan.calc;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

/**
 *
 * @author Vladimir Milicevic
 */
@Aspect
public class CalculatorIntroduction {
```

```
    @DeclareParents(
            value = "com.metropolitan.calc.*CalculatorImpl",
            defaultImpl = CounterImpl.class)
    public Counter counter;
}
```

Iako je interfejs predstavljen svim *calculator* objektima, to još uvek nije dovoljno da se prati broj njihovih pozivanja. Neophodno je uvećati vrednost brojača za svaki poziv neke metode kalkulatora. To je moguće realizovati ***After*** savetom. Još je neophodno napomenuti da je neophodno preuzeti ***this***, a ne ***target*** objekat jer jedino *proxy* objekti implementiraju *Counter* interfejs.

```
package com.metropolitan.calc;
...
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class CalculatorIntroduction {
...
@After("execution(* com.metropolitan.calc.*Calculator.*(..))"
            + " &&  this(counter)")
    public void increaseCount(Counter counter) {
        counter.increase();
    }
```

UPRAVLJANJE STANJIMA - DEMO

Neophodno je kreirati glavnu klasu za kompletiranje primera.

Klasom *Main* moguće je prikazati izlaz - vrednost brojača za svaki objekat tipa *calculator* konvertovan u tip ***Counter*** .

```
package com.metropolitan.calc;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 *
 * @author Vladimir Milicevic
 */
public class Main {

    public static void main(String[] args) {
        ApplicationContext context
                = new ClassPathXmlApplicationContext("classpath:/META-INF/
beans.xml");
        ArithmeticCalculator arithmeticCalculator
                = (ArithmeticCalculator) context.getBean("arithmeticCalculator");
```

```
        UnitCalculator unitCalculator
                = (UnitCalculator) context.getBean("unitCalculator");
        Counter arithmeticCounter = (Counter) arithmeticCalculator;
        System.out.println("Arithmetic counter = " + arithmeticCounter.getCount());
        Counter unitCounter = (Counter) unitCalculator;
        System.out.println("Unit counter = " +unitCounter.getCount());
    }
}
```

```
cd C:\Users\Vladimir Milicevic\Documents\NetBeansProjects\Calc; "JAVA_HOME=C:\\Progr
asspath com.metropolitan.calc.Main\" -Dexec.executable=\"C:\\Program Files\\Java\\jd
file.encoding=UTF-8 org.codehaus.mojo:exec-maven-plugin:1.2.1:exec\""
Running NetBeans Compile On Save execution. Phase execution is skipped and output di
Scanning for projects...

------------------------------------------------------------------------
Building Calc 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Calc ---
jul 11, 2017 8:16:18 PM org.springframework.context.support.ClassPathXmlApplicationC
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@
jul 11, 2017 8:16:19 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReade
INFO: Loading XML bean definitions from class path resource [META-INF/beans.xml]
Arithmetic counter = 0
Unit counter = 0
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.713s
Finished at: Tue Jul 11 20:16:19 CEST 2017
Final Memory: 5M/107M
------------------------------------------------------------------------
```

Slika 7.2.1 Upravljanje stanjima - demo [izvor: autor]

Poglavlje 8

Deklarisanje aspekata XML konfiguracijama

XML UMESTO ANOTACIJA

Spring podržava deklarisanje aspekata u XML konfiguracionim datotekama.

PROBLEM: Upotreba anotacija za deklarisanje aspekata funkcioniše jako dobro u većini slučajeva i u novim verzijama JVM i Springa. Međutim, ukoliko se radi sa starom verzijom JAVA jezika (1.4 i niža), što je malo verovatno, ili ako se ne želi da aplikacija ima zavisnosti od AspectJ, anotacije nije moguće koristiti.

REŠENJE: Spring podržava deklarisanje aspekata u XML konfiguracionim datotekama. Ovakav tip deklaracije se obavlja primenom XML elemenata u okviru aop šeme. Ova šema mora da se nalazi u korenskom *<beans>* elementu iz razloga što su svi XML elementi za AOP konfiguraciju definisani ovom šemom.

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
<!--
<aop:aspectj-autoproxy />
-->
...
</beans>
```

DEKLARISANJE ASPEKATA

Sve AOP konfiguracije moraju da budu definisane u okviru elementa aop:config

U konfiguracionom fajlu zrna, sve AOP konfiguracije moraju da budu definisane u okviru elementa *<aop:config>*. Za svaki aspekt je neophodno kreirati element *<aop:aspect>* koja ukazuje na pozadinsku instancu zrna za implementaciju konkretnog aspekta. Zato, aspektno zrno mora da poseduje identifikator koji će ukazivati na njega u elementu *<aop:aspect>.*

```
<beans ...>
<aop:config>
<aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
</aop:aspect>
<aop:aspect id="validationAspect" ref="calculatorValidationAspect">
</aop:aspect>
<aop:aspect id="introduction" ref="calculatorIntroduction">
</aop:aspect>
</aop:config>
<bean id="calculatorLoggingAspect"
class="com.metropolitan.calc.CalculatorLoggingAspect" />
<bean id="calculatorValidationAspect"
class="com.metropolitan.calc.CalculatorValidationAspect" />
<bean id="calculatorIntroduction"
class="com.metropolitan.calc.CalculatorIntroduction" />
...
</beans>
```

DEKLARISANJE PRESEČNIH TAČAKA

Presečna tačka može biti definisana u okviru elemenata aop:pointcut na dva načina

Presečna tačka može biti definisana u okviru elemenata *<aop:aspect>* ili *<aop:config>*. U prvom slučaju, presečna tačka će biti vidljiva samo za deklarisanje aspekta. Drugi slučaj kaže da se radi o globalnoj definicije presečne tačke koja će biti dostupna svim aspektima. Za razliku od anotacija, XML konfiguracije ne dozvoljavaju referenciranje na druge presečne tačke po nazivu, unutar izraza presečne tačke. To znači da je neophodno iskopirati izraz i ugraditi ga direktno.

```
<aop:config>
<aop:pointcut id="loggingOperation" expression=
"within(com.metropolitan.calc.ArithmeticCalculator+) ||
within(com.metropolitan.calc.UnitCalculator+)" />
<aop:pointcut id="validationOperation" expression=
"within(com.metropolitan.calc.ArithmeticCalculator+) ||
within(com.metropolitan.calc.UnitCalculator+)" />
...
</aop:config>
```

Prilikom korišćenja anotacija, u izrazima presečnih tačaka dozvoljena je upotreba operatora *&&*. U XML, karakter *&* označava referencu entiteta pa operator *&&* u izrazima presečnih tačaka nije moguće koristiti. Umesto njega koristi se ključna reč *and*.

DEKLARISANJE SAVETA

U aop šemi konkretan element odgovara svakom tipu saveta.

U *aop* šemi konkretan element odgovara svakom tipu saveta. Elementi zahtevaju atribut ***pointcut-ref*** koji ukazuje na presečnu tačku ili atribut presečne tačke za direktno ugrađivanje izraza presečne tačke. Atribut ***method*** određuje naziv metode saveta u aspektnoj klasi.

```
<aop:config>
...
<aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
<aop:before pointcut-ref="loggingOperation"
method="logBefore" />
<aop:after-returning pointcut-ref="loggingOperation"
returning="result" method="logAfterReturning" />
<aop:after-throwing pointcut-ref="loggingOperation"
throwing="e" method="logAfterThrowing" />
<aop:around pointcut-ref="loggingOperation"
method="logAround" />
</aop:aspect>
<aop:aspect id="validationAspect" ref="calculatorValidationAspect">
<aop:before pointcut-ref="validationOperation"
method="validateBefore" />
</aop:aspect>
</aop:config>
```

DEKLARISANJE PREDSTAVLJANJA

Intoduction koncept saveta ima svoju XML reprezentaciju.

Konačno, predstavljanje je moguće realizovati unutar aspekta primenom elementa *<aop:declare-parents>*.

```
<aop:config>
...
<aop:aspect id="introduction" ref="calculatorIntroduction">
<aop:declare-parents
types-matching=
"com.metropolitan.calc.ArithmeticCalculatorImpl"
implement-interface=
"com.metropolitan.calc.MaxCalculator"
default-impl=
"com.metropolitan.calc.MaxCalculatorImpl" />
<aop:declare-parents
types-matching=
"com.metropolitan.calc.ArithmeticCalculatorImpl"
implement-interface=
"com.metropolitan.calc.MinCalculator"
```

```
default-impl=
"com.metropolitan.calc.MinCalculatorImpl" />
<aop:declare-parents
types-matching=
"com.metropolitan.calc.*CalculatorImpl"
implement-interface=
"com.metropolitan.calc..Counter"
default-impl=
"com.metropolitan.calc.CounterImpl" />
<aop:after pointcut=
"execution(* com.metropolitan.calc.*Calculator.*(..)) ?
and this(counter)"
method="increaseCount" />
</aop:aspect>
</aop:config>
```

Poglavlje 9

Upredanje AspectJ aspekata u Springu

UPREDANJE TOKOM UČITAVANJA

Upredanje (weaving) je proces primene aspekata na ciljne objekte.

PROBLEM: *Spring AOP* okvir podržava ograničene tipove *AspectJ presečnih tačaka* i dozvoljava primenu aspekata za zrna deklarisana u okviru *Spring IoC kontejnera*. Ukoliko je cilj primena dodatnih tipova presečnih tačaka, ili aspekata na zrna koja su kreirana izvan IoC kontejnera, neophodno je koristiti AspectJ okvir u konkretnoj Spring aplikaciji.

REŠENJE: *Upredanje (*weaving*) je* proces primene aspekata na ciljne objekte. U Spring AOP, upredanje se dešava tokom vremena izvršavanja kroz dinamički ***proxy*** . Na suprot ovome, AspectJ okvir podržava upredanja tokom kompajliranja (compile-time), kao i tokom učitavanja (load-time).

AspectJ compile-time upredanje se dešava kroz specijalni AspectJ kompajler koji se naziva ***ajc*** . On je u mogućnosti da uprede aspekte u JAVA izvorne datoteke i da proizvede *upredene* binarne datoteke klasa. Takođe, on može da uprede aspekte u datoteke prevedenih klasa ili JAR datoteke. Upredanje nije u domenu Springa i za više informacija neophodno je pogledati AspectJ dokumentaciju.

AspectJ load-time upredanje (LTW) se dešava kada je ciljna klasa učitana u JVM. Da bi klasa bila utkana, neophodan je specijalni čitač klasa kojim će biti unapređen *bytecode* ciljne klase.

PRIMER UPREDANJA TOKOM UČITAVANJA

Upredanje je opisano primerom.

Za razumevanje procesa *AspectJ* upredanja tokom učitavanja, neophodno je uvesti konkretan primer. Kalkulator dobija mogućnost rada sa kompleksnim brojevima koji će biti reprezentovani klasom ***Complex*** . Takođe, za ovu klasu će biti definisana metoda *toString()* kojom će kompleksni broj imati string reprezentaciju *(a + bi)*.

```
package com.metropolitan.calc;

/**
 *
 * @author Vladimir Milicevic
 */
```

```
public class Complex {

    private int real;
    private int imaginary;

    public Complex(int real, int imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public int getReal() {
        return real;
    }

    public void setReal(int real) {
        this.real = real;
    }

    public int getImaginary() {
        return imaginary;
    }

    public void setImaginary(int imaginary) {
        this.imaginary = imaginary;
    }

public String toString() {
        return "(" + real + " + " + imaginary + "i)";
    }
}
```

Dalje, biće definisan jednostavan interfejs za operacije sa kompleksnim brojevima.

```
package com.metropolitan.calc;

/**
 *
 * @author Vladimir Milicevic
 */
public interface ComplexCalculator {

    public Complex add(Complex a, Complex b);

    public Complex sub(Complex a, Complex b);
}
```

Sledi implementacioni kod za interfejs. Rezultat je uvek *Complex* objekat.

```
package com.metropolitan.calc;

/**
 *
```

```
 * @author Vladimir Milicevic
 */
public class ComplexCalculatorImpl implements ComplexCalculator {

    public Complex add(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() + b.getReal(),
                a.getImaginary() + b.getImaginary());
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public Complex sub(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() - b.getReal(),
                a.getImaginary() - b.getImaginary());
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }
}
```

Pre upotrebe, kalkulator mora da bude deklarisan kao zrno u Spring IoC kontejneru.

```
<bean id="complexCalculator"
class="com.metropolitan.calculator.ComplexCalculatorImpl" />
```

Primer je zaokružen i može biti testiran klasom *Main.*

```
package com.metropolitan.calc;
...
public class Main {
public static void main(String[] args) {
...
ComplexCalculator complexCalculator =
(ComplexCalculator) context.getBean("complexCalculator");
complexCalculator.add(new Complex(1, 2), new Complex(2, 3));
complexCalculator.sub(new Complex(5, 8), new Complex(2, 3));
}
}
```

DODAVANJE NAPREDNIJIH FUNKCIONALNOSTI

Primer može da dobije i naprednije funckionalnosti.

Kompleksni kalkulator, za sada, funkcioniše korektno. Međutim, ukoliko se želi dodati nova funkcionalnost, na primer keširanje objekata kompleksnih brojeva, primer će morati da se unapredi. Keširanje je problem *zamršenog koda* i moguće je izvršiti modularizaciju pomoću aspekta.

```
package com.metropolitan.calc;

import java.util.Collections;
```

```
import java.util.HashMap;
import java.util.Map;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;

/**
 *
 * @author Vladimir Milicevic
 */
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public ComplexCachingAspect() {
        cache = Collections.synchronizedMap(new HashMap<String, Complex>());
    }

    @Around("call(public Complex.new(int, int)) && args(a,b)")
    public Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b)
            throws Throwable {
        String key = a + "," + b;
        Complex complex = cache.get(key);
        if (complex == null) {
            System.out.println("Cache MISS for (" + key + ")");
            complex = (Complex) joinPoint.proceed();
            cache.put(key, complex);
        } else {
            System.out.println("Cache HIT for (" + key + ")");
        }
        return complex;
    }
}
```

U ovom aspektu, objekti kompleksnih brojeva su keširani mapom u kojoj kao ključevi figurišu realne i imaginarne vrednosti. Da bi mapa bila *tread-safe* neophodno je da bude obmotana sinhronizovanom mapom. Najpogodniji trenutak za *uvid* u keš jeste kada se konstruktorom kreira kompleksni objekat. Koristi se AspectJ izraz presečne tačke *call* za preuzimanje svih presečnih tačaka pozivom konstruktora ***Complex(int, int)*** .

Dalje je neophodan *savet* da prilagodi prikazivanje povratne vrednosti. Ukoliko je objekat iste vrednosti pronađen u kešu, biće direktno prosleđen pozivaču. U suprotnom, biće nastavljeno pozivanje originalnog konstruktora za kreiranje novog kompleksnog objekta. Pre prosleđivanja pozivaču, on se kešira u mapu za dalju upotrebu.

Ova presečna tačka nije podržana Spring AOP pa o njoj ranije nije ni bilo govora. Zato će aspekt biti implementiran AspectJ okvirom. Konfiguracija AspectJ okvira je obavljena datotekom *aop.xml* u *META-INF* direktorijumu za *classpath.*

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
    <weaver>
```

```
        <include within="com.metropolitan.calc.*" />
    </weaver>
    <aspects>
        <aspect
            name="com.metropolitan.calc.ComplexCachingAspect" />
    </aspects>
</aspectj>
```

LOAD-TIME UPREDANJE PRIMENOM ASPECTJ WEAVER

AspectJ obezbeđuje agenta za Load-Time upredanje.

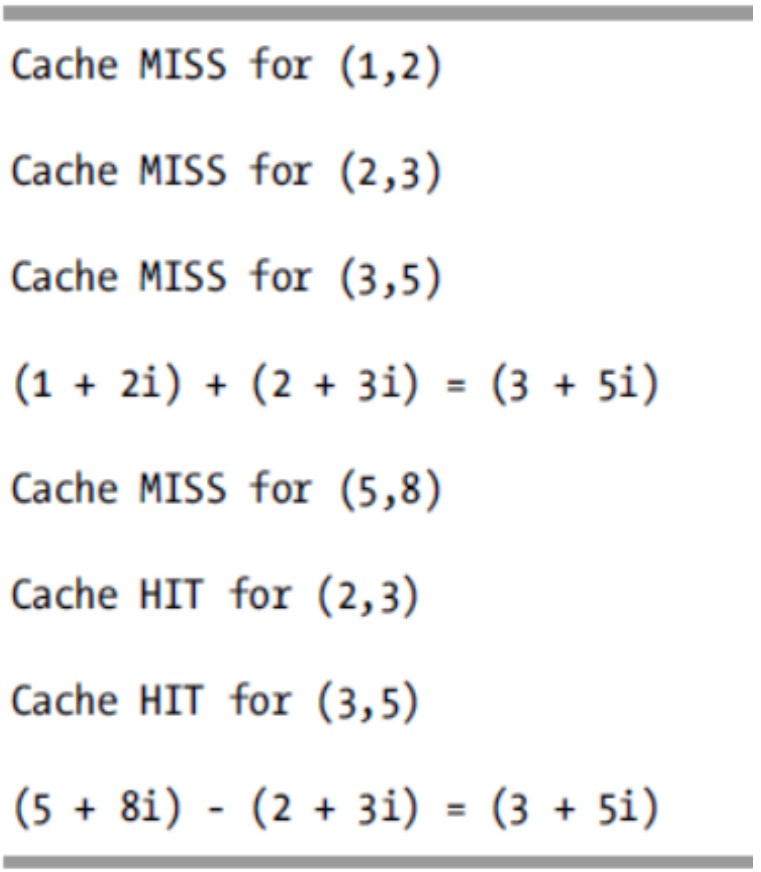
Load-Time upredanje primenom AspectJ Weaver omogućeno je posebnim agentom kojeg obezbeđuje AspectJ. Neophodno je dodati samo odgovarajući argument virtuelne mašine u komandnu liniju za pokretanje aplikacije.

```
java -javaagent:c://lib/aspectjweaver.jarcom.metropolitan.calculator.Main
```

Za korišćenje AspectJ Weaver-a, u pozivu je neophodno uključiti ***aspectjweaver.jar*** . Odgovarajuća zavisnost u putanju klasa *Maven* projekta na sledeći način:

```
<dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.10</version>
        </dependency>
```

Ukoliko se pokrene aplikacija primenom navedenog argumenta biće prikazan sledeći izlaz i keš status. AspectJ agent primenjuje savet na svaki poziv metode konstruktora ***Complex(int, int)*** .



Slika 9.1 Izlaz primera Load-Time upredanje primenom AspectJ Weaver [izvor: autor]

SPRING LOAD-TIME WEAVER

Spring podržava nekoliko Load-Time Weaver za različita okruženja izvršavanja aplikacije.

Spring podržava nekoliko*Load-Time Weaver* za različita okruženja izvršavanja aplikacije. Da bi odgovarajući bio uključen za Spring aplikaciju, neophodno je deklarisati prazan XML element *<context:load-time-weaver>*. Ovaj element je definisan u okviru *context* šeme.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
          http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring-aop-4.3.xsd
```

```
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
          http://www.springframework.org/schema/util http://www.springframework.org/
schema/util/spring-util-4.3.xsd
">
   ****
    <context:load-time-weaver />
    ****
</beans>
```

Spring će sada biti u mogućnosti da detektuje odgovarajući Load-Time Weaver za različita okruženja izvršavanja aplikacije. Izvesni JAVA EE serveri aplikacija podržavaju *Spring load-time weaver* mehanizme i nije potrebno specificirati JAVA agenta u komandnoj liniji. Za proste JAVA aplikacije ovo je, ipak, neophodno uraditi.

```
java -javaagent:c:/lib/spring-instrument.jarcom.apress.springrecipes.calculator.Main
```

Moguće je primetiti da je, za korišćenje AspectJ weaver-a, u pozivu uključen ***spring-instrument.jar*** .

U Maven projektu odgovarajuća zavisnost se dodaje na sledeći način.

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-instrument</artifactId>
<version>${spring.version}</version>
</dependency>
```

Ukoliko se pokrene aplikacija primenom navedenog argumenta biće prikazan sledeći izlaz i keš status. Različito od prethodnog slučaja, *Spring agent* primenjuje savete samo na pozive konstruktora *Complex(int, int)* iz konteksta *Spring IoC kontejnera*.

```
Cache MISS for (3,5)

(1 + 2i) + (2 + 3i) = (3 + 5i)

Cache HIT for (3,5)

(5 + 8i) - (2 + 3i) = (3 + 5i)
```

Slika 9.2 Izlaz primera Load-Time upredanje primenom Spring Weaver-a [izvor: autor]

ZADATAK ZA SAMOSTALNI RAD 3

Pokušajte sami!!!

Prikazani kod iz ovog objekta učenja pokušajte da samostalno implementirate u postojeći primer koji ste preuzeli preko aktivnosti *Shared Resources*.

Poglavlje 10

Podešavanje AspectJ aspekata u Springu

DEFINISANJE PROBLEMA

Svaki AspectJ aspekt obezbeđuje statičku produkcionu metodu pod nazivom aspectOf().

PROBLEM: *Spring AOP* aspekti su deklarisani u konfiguracionoj datoteci i lako je izvršiti njihovo podešavanje. Međutim, aspekti koje koristi *AspectJ* okvir instancirani su upravo ovim okvirom. Da bi bili podešeni, neophodno je prihvatiti instance aspekata iz *AspectJ* okvira.

REŠENJE: Svaki AspectJ aspekt obezbeđuje statičku produkcionu metodu pod nazivom ***aspectOf()*** koja dozvoljava pristup tekućoj aspektnoj instanci. U Spring IoC kontejneru, moguće je deklarisati zrno, kreirano produkcionom metodom, specifikacijom atributa *factory-method.*

Na primer, dozvoljeno je da keš mapa za *ComplexCachingAspect* bude podešena pomoću seter metode i da briše svoje instanciranje iz konstruktora.

```
package com.metropolitan.calc;
...
import java.util.Collections;
import java.util.Map;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class ComplexCachingAspect {
private Map<String, Complex> cache;
public void setCache(Map<String, Complex> cache) {
this.cache = Collections.synchronizedMap(cache);
}
...
}
```

Za podešavanje ove osobine u Spring IoC kontejneru, moguće je deklarisati zrno kreirano produkcionom metodom *aspectOf().*

```
<bean class="com.metropolitan.calc.ComplexCachingAspect"
factory-method="aspectOf">
<property name="cache">
<map>
<entry key="2,3">
<bean class="com.metropolitan.calc.Complex">
```

```
<constructor-arg value="2" />
<constructor-arg value="3" />
</bean>
</entry>
<entry key="3,5">
<bean class="com.metropolitan.calc.Complex">
<constructor-arg value="3" />
<constructor-arg value="5" />
</bean>
</entry>
</map>
</property>
</bean>
```

ZADATAK ZA SAMOSTALNI RAD 4

Pokušajte sami!!!

Prikazani kod iz ovog objekta učenja pokušajte da samostalno implementirate u postojeći primer koji ste preuzeli preko aktivnosti *Shared Resources*.

Poglavlje 11

Umetanje Spring zrna u objekte domena

PROBLEM SPRING ZRNA I OBJEKATA DOMENA

Objekti kreirani izvan IoC kontejnera su najčešće objekti domena.

PROBLEM: Zrna deklarisana u Spring IoC kontejneru mogu međusobno da se povezuju pomoću Spring umetanja zavisnosti. Međutim, objekti kreirani izvan kontejnera ne mogu da budu povezani sa Spring zrnima preko konfiguracija. Povezivanje se realizuje ručno, pomoću programskog koda.

REŠENJE: Objekti kreirani izvan IoC kontejnera su najčešće objekti domena. Oni su kreirani primerom operatora ***new*** ili pomoću rezultata upita nad bazom podataka. Za umetanje Spring zrna u objekat domena, neophodna je AOP asistencija. Spring podržava specijalizovani AspectJ aspekt za ovu svrhu. Njega je moguće obezbediti pomoću AspectJ okvira.

Uvodi se konkretan primer. Pretpostavka je da postoji globalni formater za kompleksne brojeve koji prihvata šablone formatiranja.

```
package com.metropolitan.calc;
public class ComplexFormatter {
private String pattern;
public void setPattern(String pattern) {
this.pattern = pattern;
}
public String format(Complex complex) {
return pattern.replaceAll("a", Integer.toString(complex.getReal()))
.replaceAll("b", Integer.toString(complex.getImaginary()));
}
}
```

Sada, formater će biti konfigurisan u IoC kontejneru sa specifikacijom šablona formatiranja.

```
<bean id="complexFormatter"
class="com.metropolitan.calc.ComplexFormatter">
<property name="pattern" value="(a + bi)" />
</bean>
```

U klasi ***Complex***, formater se koristi u metodi *toString()* za konvertovanje kompleksnog broja u string. Klasa uvodi seter metodu za ***ComplexFormatter***.

```
package com.metropolitan.calc;
public class Complex {
private int real;
private int imaginary;
...
private ComplexFormatter formatter;
public void setFormatter(ComplexFormatter formatter) {
this.formatter = formatter;
}
public String toString() {
return formatter.format(this);
}
}
```

Međutim, kompleksni objekti nisu kreirani u IoC kontejneru i ne mogu biti konfigurisani za umetanje zavisnosti. Neophodno je napisati odgovarajući kod za umetanje *ComplexFormatter* instance u svaki kompleksni objekat.

ANNOTATIONBEANCONFIGURERASPECT

Spring sadrži aspektnu biblioteku za podešavanje zavisnosti svih objekata.

Nastavljajući se na prethodno izlaganje, trebalo bi napomenuti da Spring uključuje AnnotationBeanConfigurerAspect u vlastitoj aspektnoj biblioteci za podešavanje zavisnosti svih objekata, bez obzira da li potiču iz IoC kontejnera ili izvan njega. Prvo, neophodno je obeležiti tip objekta anotacijom ***@Configurable*** da bi se istaklo da se radi o *podesivom* tipu objekata.

```
packagecom.metropolitan.calc;
import org.springframework.beans.factory.annotation.Configurable;
@Configurable
public class Complex {
...
}
```

Spring definiše pogodan XML element *<context:spring-configured>* za omogućavanje pomenutog aspekta.

```
<beans ...>
...
<context:load-time-weaver />
<context:spring-configured />
<bean class="com.metropolitan.calc.Complex"
scope="prototype">
<property name="formatter" ref="complexFormatter" />
</bean>
</beans>
```

Kada je kreiran objekat klase obeležene pomenutom anotacijom, aspekt će potražiti prototipsku definiciju zrna čiji je tip identičan klasi. Tada će konfigurisati nove instance na osnovu definicije zrna.

Ukoliko postoje osobine deklarisane u definiciji zrna, nove instance će takođe posedovati te osobine postavljene aspektom.

Konačno, aspekt mora biti omogućen AspectJ okvirom da bi postao funkcionalan. Moguće ga je *upresti* u konkretnu klasu tokom vremena učitavanja, pomoću Spring agenta.

```
java -javaagent:c:/lib/spring-instrument.jarcom.apress.springrecipes.calculator.Main
```

POVEZIVANJE POMOĆU ID ZRNA

Povezivanja podesive klase sa definicijom zrna omogućeno je pomoću ID zrna.

Drugi način povezivanja *podesive* klase sa definicijom zrna jeste pomoću ID zrna. Moguće je prezentovati ID zrna kao vrednost u anotaciji ***@Configurable***.

```
packagecom.metropolitan.calc;
import org.springframework.beans.factory.annotation.Configurable;
@Configurable("complex")
public class Complex {
...
}
```

Dalje je neophodno dodati ID atribut u odgovarajuću definiciju zrna za povezivanje sa *podesivom* klasom.

```
<bean id="complex" class="com.metropolitan.calc.Complex"
scope="prototype">
<property name="formatter" ref="complexFormatter" />
</bean>
```

Slično običnim Spring zrnima, podesiva zrna takođe mogu da podrže automatsko povezivanje i proveru zavisnosti.

```
package com.metropolitan.calc;

import org.springframework.beans.factory.annotation.Autowire;
import org.springframework.beans.factory.annotation.Configurable;
@Configurable(
value = "complex",
autowire = Autowire.BY_TYPE,
dependencyCheck = true)
public class Complex {
...
}
```

Kada je logički atribut ***dependencyCheck*** podešen na ***true***, postiže se isti efekat kao u slučaju *dependency-check="objects"* - provera neprimitivnih i nekolekcijskih objekata. Sa omogućenim automatskim povezivanjem, ne postoji više potreba za eksplicitnim podešavanjem osobine ***formatter***.

```
<bean id="complex" class="com.metropolitan.calc.Complex"
scope="prototype" />
```

U novim verzijama Springa nema više potrebe za ***auto-wiring*** konfigurisanjem i proverom zavisnosti na nivou klase za anotaciju ***@Configurable***. Umesto toga, moguće je obeležiti formaterovu seter metodu sa anotacijom ***@Autowired***.

```
packagecom.metropolitan.calc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
@Configurable("complex")
public class Complex {
...
private ComplexFormatter formatter;
@Autowired
public void setFormatter(ComplexFormatter formatter) {
this.formatter = formatter;
}
}
```

Konačno, neophodno je omogućiti *<context:annotation-config>* element u datoteci konfiguracije zrna za obradu metoda sa ovim anotacijama.

```
<beans ...>
<context:annotation-config />
...
</beans>
```

ZADATAK ZA SAMOSTALNI RAD 5

Pokušajte sami da implementirate nov kod iz ovog objekta učenja!!!

Prikazani kod iz ovog objekta učenja pokušajte da samostalno implementirate u postojeći primer koji ste preuzeli preko aktivnosti *Shared Resources*.

Poglavlje 12

Pokazna vežba 3

OSNOVE ASPEKTNO - ORJENTISANOG PROGRAMIRANJA (45 MINUTA)

Upoznavanje studenta sa osnovama AOP programiranja, funkcijama i terminologijom neophodnom da isprati vežbu

Počnimo definisanjem osnovnih pojmova vezanim za aspektno-orjentisano programiranje - AOP i terminologije koja se koristi.

Napomena: Ovi termini nisu specifično vezani za Spring već se pojavljuju svuda gde se koristi AOP.

Aspekt je način modularizacije koji prolazi kroz razne klase i eliminiše zamršeni kod. Koristi se anotacija *@Aspect* kako bi se *bean* definisao kao aspekt.
Presečna tačka: AspectJ deklariše presečnu tačku kao prostu metodu obeleženu anotacijom *@Pointcut*. Telo metode je obično prazno jer je nelogično mešanje definicije presečne tačke sa programskom logikom. Modifikator pristupa metodi presečne tačke kontroliše, takođe, i vidljivost presečne tačke. Ostali saveti se obraćaju ovoj presečnoj tački pomoću naziva metode.

Savet je akcija koju aspekt obavlja u određenoj tački preseka. Postoje razne vrste saveta:

@Before, *@After*, *@AfterReturning*, *@AfterThrowing i @Around.*

Pri dolasku do presečne tačke, prvo će se izvršiti metoda sa anotacijom *@Before*, potom sama metoda, ako metoda baci grešku izvršiće se metoda *@AfterThrowing*, ako metoda nema povratnu vrednost izvršiće se metoda *@After*, a ako ima povratnu vrednost izvršiće se *@AfterReturning* metoda. U slučaju da treba da se preskoči neki deo koda, AOP aspekt obilazi tu metodu sa *@Around* savetom.

Ciljni objekat je objekat na koji utiču saveti jednog ili više aspekata.

Pošto Spring AOP koristi runtime proksije, ovaj objekat će uvek biti pod uticajem proksija.

AOP proxy je objekat koji kreira sam okvir kako bi moglo da se implementuje aspektno orjentisano programiranje (presretanje i savetovanje metoda).

Upredanje je povezivanje aspekata sa ostatkom aplikacije i ono se dešava tokom vremena izvršavanja kroz dinamički proxy.

KONFIGURACIJA SPRING AOP U MAVENU

Pokazni primer konfigurisanja AOP biblioteka za rad u Spring frameworku

Kako bi bilo moguće da se koriste sve anotacije i metode iz Spring AOP frameworka, potrebno je da se biblioteke za Spring AOP framework učitaju u *pom.xml* fajl. U postojećem projektu sa prethodnih predavanja otvoriti *pom.xml* fajl i u delu gde se nalazi tag **dependencies** kopirati sledeći kod.

```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.6.11</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.6.11</version>
        </dependency>
```

Nakon ovoga, desnim klikom na projekat otvoriti meni sa opcijama, izabrati ***Build with Dependencies*** opciju kako bi se biblioteke učitale i sačuvale u projektu.

KONFIGURACIJA SPRING AOP

Pokazni primer Spring AOP konfiguracije

AOP **saveti** rade isključivo sa interfejsima.

Kako bismo mogli da vidimo kako oni rade, potrebno je da napravimo novi *interfejs* i njegovu *implementaciju* i da ga deklarišemo kao bean.

Potrebno je napraviti novi paket **com.metropolitan.it355.dao** i u njemu novi interfejs **StudentDao**. U samom interfejsu kopirati sledeći kod:

```
package com.metropolitan.it355.dao;

public interface StudentDao {

    public void addStudent();
```

```
    public String addStudentRet();

    public void addStudentThrowException() throws Exception;

    public void addStudentAround(String name);
}
```

Nakon toga, u istom paketu napraviti novi paket **com.metropolitan.it355.dao.impl** i u njemu novu *klasu* koja implementira *StudentDao interfejs* i u njoj kopirati sledeći kod:

```
package com.metropolitan.it355.dao.impl;

import com.metropolitan.it355.dao.StudentDao;

public class StudentDaoImpl implements StudentDao {

    @Override
    public void addStudent() {
        System.out.println("Adding student...");
    }

    @Override
    public String addStudentRet() {
        System.out.println("Adding student and returning value...");
        return "Added succesfully!";
    }

    @Override
    public void addStudentThrowException() throws Exception {
        System.out.println("Adding student and throwing exception...");
        throw new Exception("Generic Exception");
    }

    @Override
    public void addStudentAround(String name) {
        System.out.println("Adding student with name: " + name);
    }
}
```

Kako bi Spring Framework mogao da učita ovaj *servis* potrebno je da se on deklariše kao *bean* pod nazivom **studentDao**.

U folderu **WEB-INF** u **dispatcher-servlet** fajlu dodati sledeći kod

```
<bean id="studentDao" class="com.metropolitan.it355.dao.impl.StudentDaoImpl" />
```

KREIRANJE I DEFINISANJE ASPEKATA

Kako bismo mogli da koristimo aspekte, prvo moramo da ih kreiramo.

Kako bismo mogli da koristimo povoljnosti AOP, potrebno je da imamo bar jedan aspekt koji ćemo koristiti da *presreće* metode i *savetuje* ih.

Potrebno je napraviti novi paket **com.metropolitan.it355.aspect** i u njemu novu klasu **InterceptorLog**. U njoj kopirati sledeći kod:

```
package com.metropolitan.it355.aspect;

import java.lang.reflect.Method;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.aop.MethodBeforeAdvice;

@Aspect
public class InterceptorLog implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] os, Object o) throws Throwable {
        System.out.println("logBefore() is running!");
        System.out.println("hijacked : " + method.getName());
    }
}
```

Ovaj **aspekt** sadrži **before** koji se poziva kada u **studentDao** interfejsu pozovemo metodu **addStudent**. Kako bismo mogli da koristimo ovaj aspekt, u fajl **dispatcher-servlet** treba dodati sledeći kod kako bismo ga definisali kao **bean**:

```
<bean id="logAspect" class="com.metropolitan.it355.aspect.InterceptorLog" />

    <bean id="logPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice" ref="logAspect" />
        <property name="mappedName">
            <value>*</value>
        </property>
    </bean>

    <bean id="testServiceProxy"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames" value="studentDao" />
        <property name="interceptorNames">
            <list>
                <value>logPointcutAdvisor</value>
            </list>
        </property>
    </bean>
```

Radi testiranja ovog servisa, u **StudentController** izmeniti kod dodavanjem sledećeg koda:

```
package com.metropolitan.it355.controller;

import com.metropolitan.it355.dao.StudentDao;
import com.metropolitan.it355.model.Student;
```

```
import java.util.Locale;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {

    @Autowired
    private MessageSource messageSource;

    @Autowired
    private StudentDao studentDao;

    // GET method
    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() {
        System.out.println("Calling MessageSource");
        System.out.println(messageSource.getMessage("name", null, new
Locale("sr-Latn-RS")));
        System.out.println("Calling studentDao");
        studentDao.addStudent();
        return new ModelAndView("student", "command", new Student());
    }

    // POST method
    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute Student student, ModelMap model) {
        model.addAttribute("name", student.getName());
        model.addAttribute("number", student.getNumber());
        model.addAttribute("year", student.getYear());
        System.out.println("Calling studentDao");
        studentDao.addStudent();
        // viewStudent.jsp
        return "viewStudent";
    }
}
```

Pokrenuti projekat i otići na URL: http://localhost:8084/IT355/student kako biste proverili rad ovog saveta.

SAVETI: VEŽBANJE

Cilj je da student vidi kako rade svi saveti u AOP frameworku.

U nastavku se nalazi kod izmenjenog **StudentController**-a koji treba da kopirate:

```
package com.metropolitan.it355.controller;

import com.metropolitan.it355.dao.StudentDao;
import com.metropolitan.it355.model.Student;
import java.util.Locale;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {

    @Autowired
    private MessageSource messageSource;

    @Autowired
    private StudentDao studentDao;

    // GET method
    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() {
        System.out.println("Calling MessageSource");
        System.out.println(messageSource.getMessage("name", null, new
Locale("sr-Latn-RS")));
        System.out.println("Calling studentDao");
        studentDao.addStudent();
        studentDao.addStudentAround("Illegal argument");
        try {
            studentDao.addStudentThrowException();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return new ModelAndView("student", "command", new Student());
    }

    // POST method
    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute Student student, ModelMap model) {
        model.addAttribute("name", student.getName());
        model.addAttribute("number", student.getNumber());
        model.addAttribute("year", student.getYear());
        // viewStudent.jsp
        return "viewStudent";
    }
}
```

U nastavku se nalazi kod izmenjene klase InterceptorLog koji treba da kopirate:

```
@Aspect
public class InterceptorLog implements MethodBeforeAdvice {

    @Override
    public void before(Method arg0, Object[] arg1, Object arg2) throws Throwable {
         System.out.println("logBefore() is running!");
            System.out.println("hijacked : " + arg0.getName());
            System.out.println("******");
    }

     @After("execution(* com.it355.dao.impl.StudentDaoImpl.addStudent(..))")
        public void logAfter(JoinPoint joinPoint) {
            System.out.println("logAfter() metoda je pozvana!");
            System.out.println("presretanje : " +
joinPoint.getSignature().getName());
            System.out.println("******");
        }

           @AfterThrowing(
                       pointcut = "execution(*
com.it355.dao.impl.StudentDaoImpl.addStudentThrowException())",
                       throwing= "error")
                     public void logAfterThrowing(JoinPoint joinPoint, Throwable
error) {

                     System.out.println("logAfterThrowing()  metoda je pozvana!");
                     System.out.println("presretanje : " +
joinPoint.getSignature().getName());
                     System.out.println("Exception : " + error);
                     System.out.println("******");
           }

           @Around("execution(* execution(*
com.it355.dao.impl.StudentDaoImpladdStudentAround(..))")
           public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {

            System.out.println("logAround()  metoda je pozvana!");
            System.out.println("presretanje metode : " +
joinPoint.getSignature().getName());
            System.out.println("presretanje : " +
Arrays.toString(joinPoint.getArgs()));

            System.out.println("Around before  metoda je pozvana!");
            joinPoint.proceed();
            System.out.println("Around  metoda je pozvana!");

            System.out.println("******");

           }
}
```

```
}
```

Poglavlje 13

Individualna vežba 3

ZADATAK ZA SAMOSTALNI RAD (90 MIN)

Zadaci za samostalni rad koji studenti rade na individualnim vežbama.

Zadatak 1 (45 min):

Primer sa klasom Fakultet iz prethodne lekcije izmeniti dodavanjem AOP zavisnosti.

Zadatak 2 (45 min):

Kreirati servis za Fakultete i interceptor koji presreće sve metode iz oba servisa.

U interceptor klasi treba da postoje Before, After, AfterThrowing, AfterReturning i Around saveti koji štampaju poruku o imenu presretnute metode i štampaju poruku o tipu saveta.

VIDEO TUTORIJALI 1

Cilj je da student samostalno izvežba AOP metode i savete

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO TUTORIJALI 2

Cilj je da student samostalno izvežba AOP metode i savete uz pomoć tutorijala

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 14

Domaći zadatak 3

DOMAĆI ZADATAK (120 MIN)

Cilj ove sekcije je da prikaže studentu šta treba da uradi za domaći zadatak 3.

Na Spring projekat iz prethodnog domaćeg zadatka dodati primer korišćenja aspektno orijentisanog programiranja.

Na primer, možete koristiti AOP za logging ili za exception handling.
U interceptor klasi treba da postoje Before, After, AfterThrowing, AfterReturning i Around saveti koji štampaju poruku o imenu presretnute metode i štampaju poruku o tipu saveta.

Spring projekat treba izraditi koristeći materijal sa vežbi i potrebno je poslati mejl asistentu sa zipovanim Spring projektom.

Nakon obaveznog zadatka, studenti dobijaju različite zadatke na email od predmetnog asistenta.

Poglavlje 15

Zaključak

ZAKLJUČAK

U četvrtoj lekciji obrađene su teme koje govore o Spring AOP i AspectJ podrsci.

U lekciji je naučeno kako se pišu aspekti, pomoću AspectJ anotacija i XML konfiguracija. Naučeno je kako se aspekti registruju u Spring IoC kontejneru. Posebno je akcentovano da Spring AOP podržava pet tipova saveta: ***before, after, after returning, after throwing*** i ***around***.

Naučeni su i različiti tipovi presečnih tačaka za podudaranje tačaka dodira po *potpisu* metode, tipu potpisa i nazivu zrna. Pokazano je da Spring podržava izvršavanje tačaka dodira samo za zrna koja su deklarisana unutar Spring IoC kontejnera. Ako se koristi presečna tačka izvan ove oblasti biće izbačen izuzetak.

U lekciji je obrađen i specijalni tip saveta ***predstavljanje*** . On dozvoljava objektima da implementiraju interfejs, dinamički, primenom implementacionih klasa. Na ovaj način se postižu isti efekti kao u slučaju višestrukog nasleđivanja.

Na kraju je diskutovano o mogućnosti primene presečnih tačaka koje nisu direktno podržane Springom. Takođe, pomenuta je i mogućnost korišćenja aspekata na objekte koji su kreirani izvan IoC kontejnera. U tom slučaju je neophodno koristiti AspectJ okvir u Spring aplikacijama. Pokazano je, u posebnom delu lekcije, kako aspekti mogu biti *utkani* u klase pomoću ***load-time weaver-a*** . Spring poseduje i neke veoma korisne AspectJ aspekte koji se čuvaju u aspektnoj biblioteci. Jedan od njih omogućava umetanje zrna u domenske objekte kreirane izvan Spring IoC kontejnera.

LITERATURA

Za pripremu Lekcije 04 upotrebljena je najnovija literatura.

Za pripremu lekcije korišćena je najnovija pisana i elektronska literatura:
1. Gary Mak, Josh Long, and Daniel Rubio, Spring Recipes Third Edition, Apress
2. Spring Framework Reference Documentation - https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/spring-framework-reference/
3. Craig Walls, Spring in Action, Manning
4. Craig Walls, Spring Boot in Action, Manning

Dopunska literatura:

1. http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/spring-tutorials/
2. http://www.tutorialspoint.com/spring/
3. http://www.javatpoint.com/spring-tutorial

LEKCIJA 03 - VIDEO MATERIJALI

Lekcija o Spring i AspectJ AOP biće završena odgovarajućim video materijalima.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.