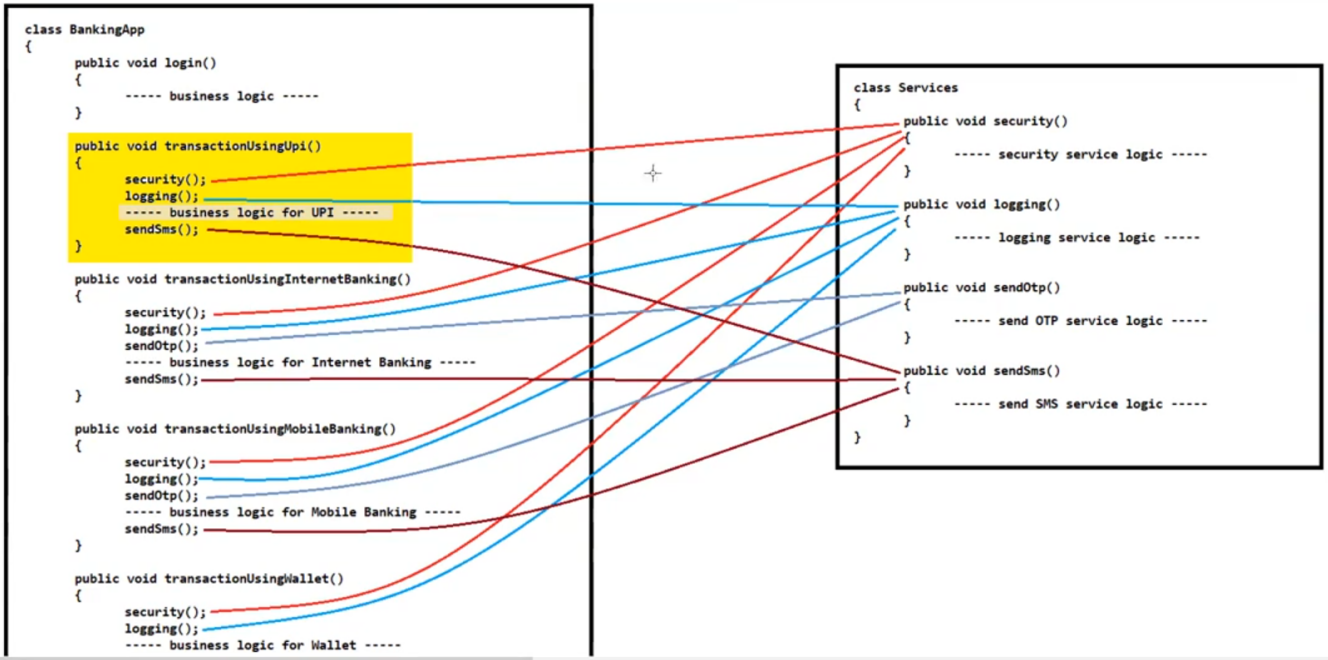
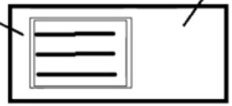

class BankingApp
{
public void login()
{
----- business logic -----
}
public void transactionUsingUpi()
{
security();
logging();
----- business logic for UPI -----
sendSms();
}
public void transactionUsingInternetBanking()
{
security();
logging();
sendOtp();
----- business logic for Internet Banking -----
sendSms();
}
public void transactionUsingMobileBanking()
{
security();
logging();
sendOtp();
----- business logic for Mobile Banking -----
sendSms();
}
public void transactionUsingWallet()
{
security();
logging();
----- business logic for Wallet -----
class Services
{
public void security()
{
----- security service logic -----
}
public void logging()
{
----- logging service logic -----
}
public void sendOtp()
{
----- send OTP service logic -----
}
public void sendSms()
{
----- send SMS service logic -----
}
}

```
class BankingApp
{
	public void login()
	{
		----- business logic -----
	}

	public void transactionUsingUpi()
	{
		----- business logic for UPI -----
	}

	public void transactionUsingInternetBanking()
	{
		----- business logic for Internet Banking -----
	}

	public void transactionUsingMobileBanking()
	{
		----- business logic for Mobile Banking -----
	}

	public void transactionUsingWallet()
	{
		----- business logic for Wallet -----
	}
}
```

```
class Services
{
	public void security()
	{
		----- security service logic -----
	}

	public void logging()
	{
		----- logging service logic -----
	}

	public void sendOtp()
	{
		----- send OTP service logic -----
	}

	public void sendSms()
	{
		----- send SMS service logic -----
	}
}
```



```
-------------------------

=> AOP :-
    -> AOP stands for "Aspect Oriented Programming"
    -> AOP is a "Programming Paradiagm" or "Approach" that focus on modularization and managing the
    cross-cutting concerns in software development
    -> AOP is implemented by a lot of languages like java, python, php, c++ etc

    -> Unlike traditional Object Oriented Programming (OOP) which focus on classes and objects, Aspect
    Oriented Programming (AOP) is more focused on aspects

    -> What is the need of AOP ?
        = Scenerio : Create a banking application having multiple transactions i.e. UPI, internet
        banking, mobile banking, wallet etc. It will also use security, logging, OTP, sms etc
        = Problem : If we create above application using OOP, then we have to face some problems :-
            - our code is not neat and clean because in business logic method we have to provide another
            type of code
            - if we have to change any service logic method, then it will be difficult to change the
            updates in business logic method and maintain the project

        = To resolve above problems we have to use AOP

    -> AOP provides complement to OOP to achieve more and clear modularity

    -> Advantages of AOP :-
        1. It provides more modularity
        2. It improves maintanability and readability of code
        3. It provides loosly coupled design
```

=> Terms used in AOP :-

1. Aspect
2. Advice
3. Join-points
4. Pointcuts
5. Target
6. Proxy
7. Advisor
8. Weaving

=> Aspect :-

-> Aspect is a module or a concept that encapsulates a specific cross-cutting concern such as security, logging, transactions, error handling etc

-> It provides services that can be applied to multiple parts of an application

=> Cross-cutting concerns :-

-> A cross-cutting concern refers to a specific functionality or requirement (logging, security, transactions, error handling etc) in software application that affects multiple parts of the codebase.

```
=> Advice :-
    -> Advice is the actual code that implements a specific aspect's behaviour.
    -> It's the code that runs at designated points in our application typically at join-points to
    achieve the designed cross-cutting concern
    -> For example :-
        = For logging we use Java Logging API, Log4j, Tinylog etc
        = For security we use JAAS (Java Authentication and Autorization Service)
        = For transactions we use JTA (Java Transaction API)
          etc
    -> There are total 5 types of advices :-
        1. Before Advice
            = Runs before the target method's execution
            = Often used for tasks like input validations or setup operations
        2. After Advice
            = Runs after the target method's execution, regardless of its outcome
            = Often used for cleaup tasks or actions that need to occur after the main logic
        3. After Returning Advice
            = Runs after the target method's successfully execution (no runtime exception)
            = Used for tassks that should only happen when a method completes successfully
        4. After Throwing Advice
            = Runs after the target method throws an exception
            = Useful for handling exception cases, logging errors, performing recovery actions etc
        5. Around Advice
            = Runs before and after the method execution
            = Oftens used for tasks that require manipulation before and after method execution
```

=> Join-points :-
-> A join-point is the location in the application where an aspect or advice can be applied or plugged-in
-> A join-point can be before or after executing method, before throwing an exception, before or after modifying an instance variable etc

=> Pointcuts :-
-> It is the join-point or location where aspect or advice is plugged-in or implemented

=> Target :-
-> Target refers to the specific components of the code such as methods or classes where we want to apply the advice

=> Proxy :-
-> Proxy is an object which contains the target object and advice (advisor) details
-> Proxy object is created by AOP framework

=> Advisor :-
    -> Advisor is the group of "advice" and "pointcuts" which is passed to the proxy factory object

=> Weaving :-
    -> It is the process of applying the aspect on the target object in order to generate proxy
    -> Weaving can be achieved at :-
        = compile time
        = load time
        = runtime
    -> NOTE : Spring AOP performs weaving at runtime

File Edit Format View Help

```
//target
class BankingApp
{
        //target
        public void login()
        {
                //join-points
                ----- business logic -----
                //join-points
        }

        //target
        public void transactionUsingUpi()
        {
                //join-points
                security(); //pointcut
                ----- business logic for UPI -----
                //join-points
        }

        //target
        public void transactionUsingInternetBanking()
        {
```

```
=> How to achieve Aspect Oriented Programming in Spring :-
    -> We can achieve AOP by 2 ways :-
        A) Using Spring's built-in AOP Framework
            = Spring provides its own AOP framework that simplifies the implementation of AOP in spring
            = Spring's built-in AOP Framework provides 2 ways to achieve AOP :-
                1. Spring XML Configuration
                    1.1 DTD Based (Older Approach)
                    1.2 XSD Based (Modern Approach)
                2. Java & Annotation-Based Configuration
                    - using @Configuration and @EnableAspectJAutoProxy (Programmatic Configuration)
        B) using AspectJ framework (implementation) in Spring
            = The AspectJ framework provides more advanced AOP capabilities which can be integrated with
            spring applications
            = We can achieve AOP by 2 ways :-
                1. XML Configuration (Advanced)
                2. Annotations (Concise & Widly Used)
```

=> Difference between DTD and XSD :-

1. DTD stands for Document Type Defination
   XSD stands for XML Schema Defination

2. DTD is older approach
   XSD is new approach

3. DTD is less extensible as compared to XSD. It lacks features like data typing, element grouping etc
   XSD is highly extensible. It provides features like data typing, element grouping etc

4. DTD has limited support for namespace
   XSD has highly support for namespace

5. DTD uses a simpler and less strict syntax
   XSD uses more complex and strict syntax

etc

② Agar humko specific method se pehle Aspect provide krna hai to uske lye.

(Name Match Method Pointcut) ~~Mapped Name~~ ka ek method hai. wo

```
<bean class="org.springframework.aop.support.NameMatchMethodPointcut" id="PointcutId">
  <property name="mappedName">   ← Method Name hai
    <list>
      <value> Bank Im Id </value>
      ...
    </list>
  </property>
</bean>.
```

execute hoga.

yaha pr jo jo value / object diya hai list wahi class execute hoga.

Default PointcutAdvisor

```
<bean class="org.springframework.aop.support.DefaultPointcutAdvisor" id="AdvisorId">
  <property name="pointcut" ref="pointcut Id"/>
  <property name="advice" ref="LogAspect Id"/>
</bean>
```

② ①

```
<bean class="org.springframework.aop.framework.ProxyFactoryBean" id="proxy Id">
  <property name="target" ref="bank Impl Id"/>
  <property name="interceptorName">
    <list>
      <value> advisorId </value>
    </list>
</bean>
```

ye advisorId ko hi bolega ki kon kon se class hai execute krna.

then advisor id pointcut Id se le lega to execute ho jayega.

(1)

AOP implementation (DID) based.

Bank Transaction → Target class
↳ It has many methods.

logging Aspect → implements MethodBefore Advice.
↳ It has before, After, etc types of method that executes According to the its behaviour.

App.java → main method.

then Run.

Application Context.xml → we declare the Objects.

target class — ① → Bank Transaction ka bean
Aspect. ② logging Aspect ka bean.

How to achieve AOP.

↳

```
<bean
    class = "org.springframework.aop.framework.
             ProxyFactoryBean"  id = "proxyId" >
<property  name = "target"  ref = "bankTranId" >
<property  name = "interceptorNames" >
    <list >
        <value>  logAspectId </value>
    </list>
</property>
bean
```

target kai

ye target se pehele execute hoga

ye list mein value leta hai,

↳ this is the id of the logging Aspect class that executes before Bank.

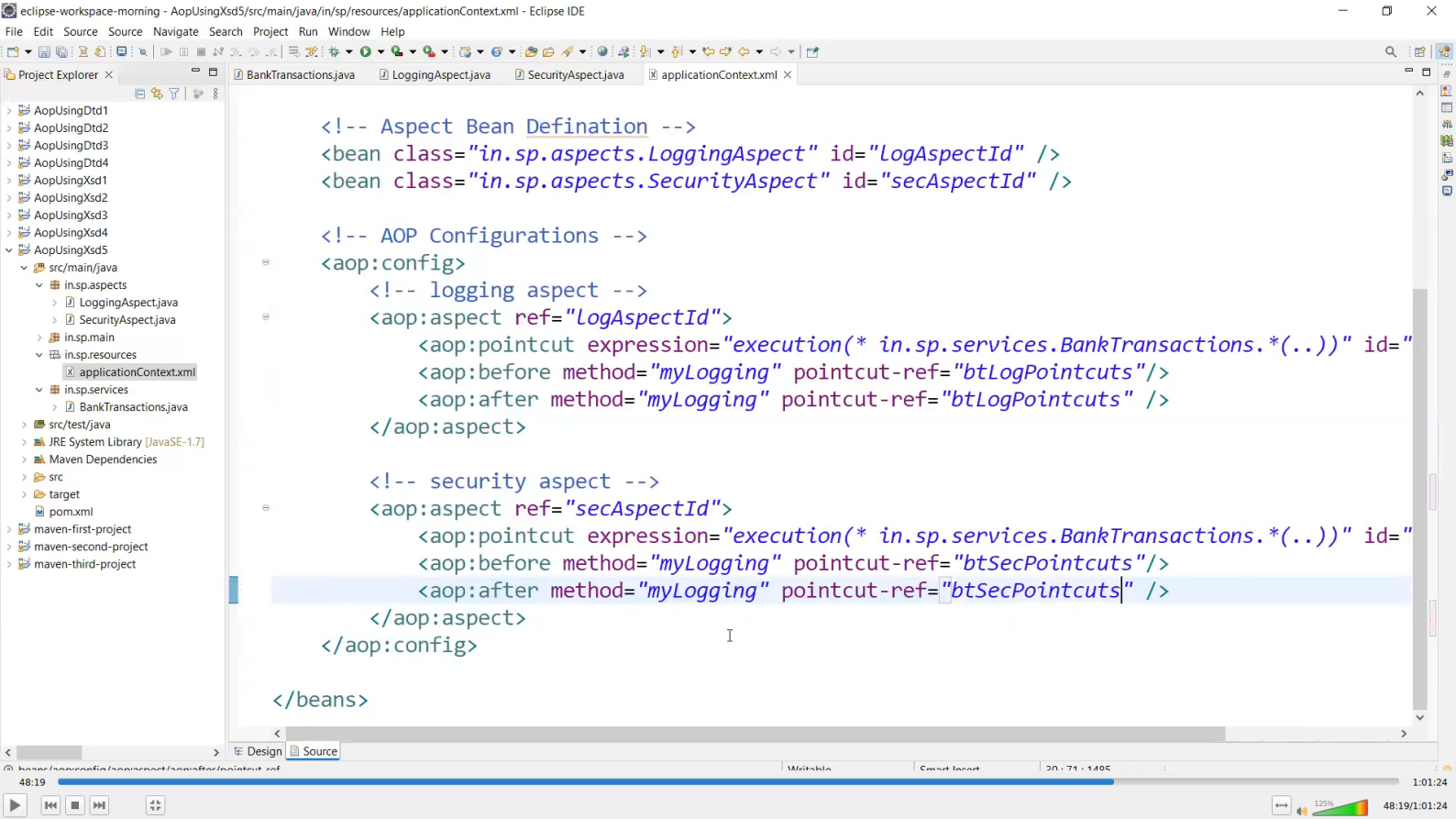
```
<?xml version="1.0" encoding="----" ?>

<beans -------------AOP Schema-----------
       ---------------------------------->

       <bean ------ />

       <aop:config>

              <!-- declaring pointcuts -->
              <aop:pointcuts id="----" expression="-----" />

              <!-- declaring aspects -->
              <aop:aspects id="----" ref="----">

                     <!-- declaring pointcuts -->
                     <aop:pointcuts id="----" expression="-----" />

                     <!-- declaring advices -->
                     <aop:before method="----" pointcut-ref="----" />

              </aop:aspects>

              <!-- declaring aspects -->
              <aop:aspects id="----" ref="----">
                     -----
              </aop:aspects>

       </aop:config>

</beans>
```

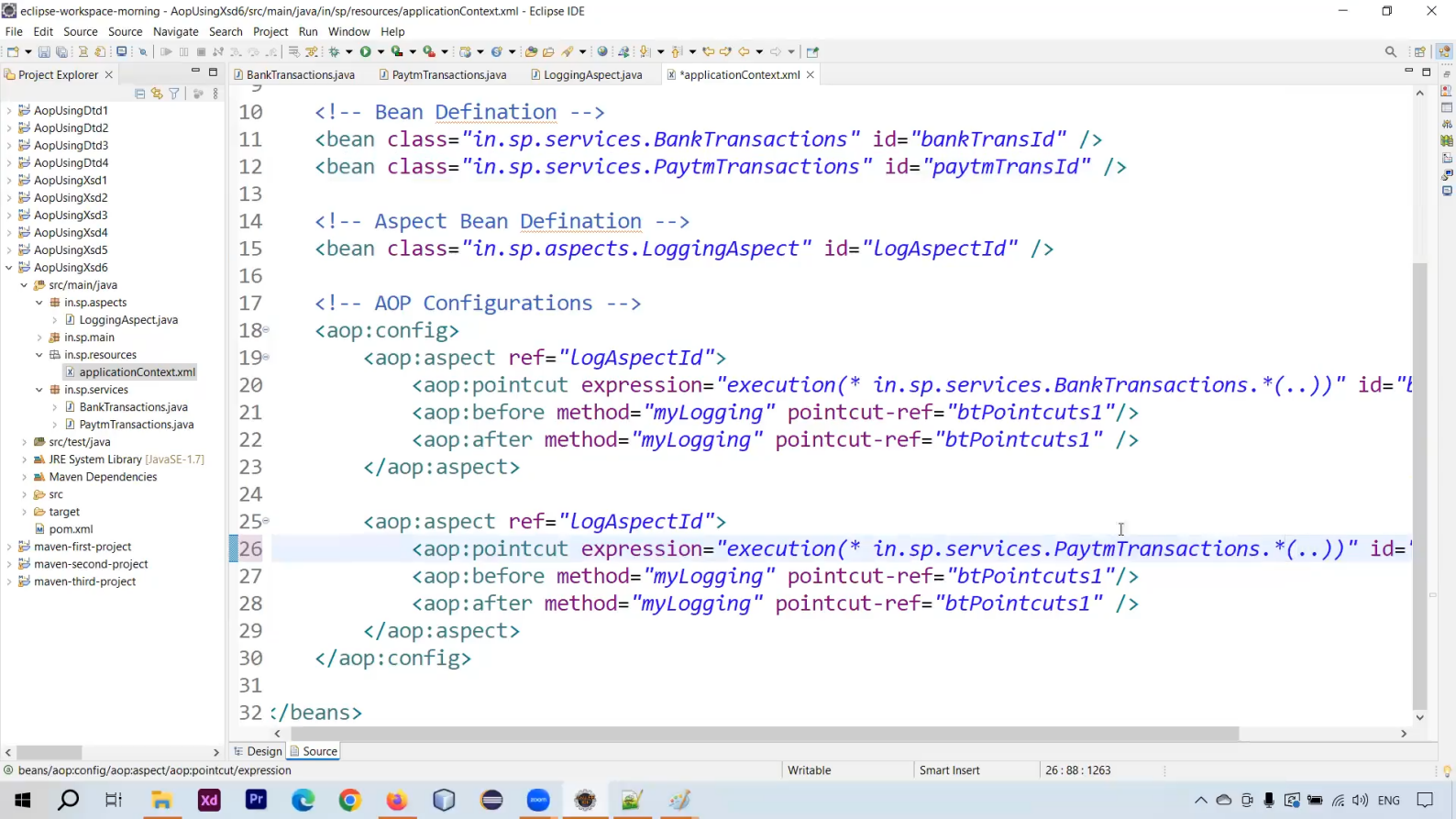
```
=> Spring XML Configuration (XSD Based) :-
    -> In this type of approach we use namespace tags for declaring aspects, advices, pointcuts etc
    -> The parent (main) namespace tag is <aop:config>. All the configurations i.e. aspects, advices,
    pointcuts are placed inside this tag
    -> To use <aop:config> tag, we have to use AOP-Schema in spring.xml file configuration

    -> Syntax :-

    -> What is expression ?
        = Expression is the way to describe the pointcuts programtically
        = We have to provide Pointcut Designators (POD) in expression i.e. execution, within, this, bean
        etc
        = Syntax :-
            expression = "execution(----define expression----)"

        = POD execution syntax :-
            1. execution(* in.sp.services.BankTransaction.*())
            2. execution(* in.sp.services.BankTransaction.*(..))
            3. execution(* in.sp.services.BankTransaction.*(String, ..))
            4. execution(* *(..))

    -> Types of advices :-
        1. <aop:before>
        2. <aop:after>
        3. <aop:after-returning>
        4. <aop:after-throwing>
        5. <aop:around>
```

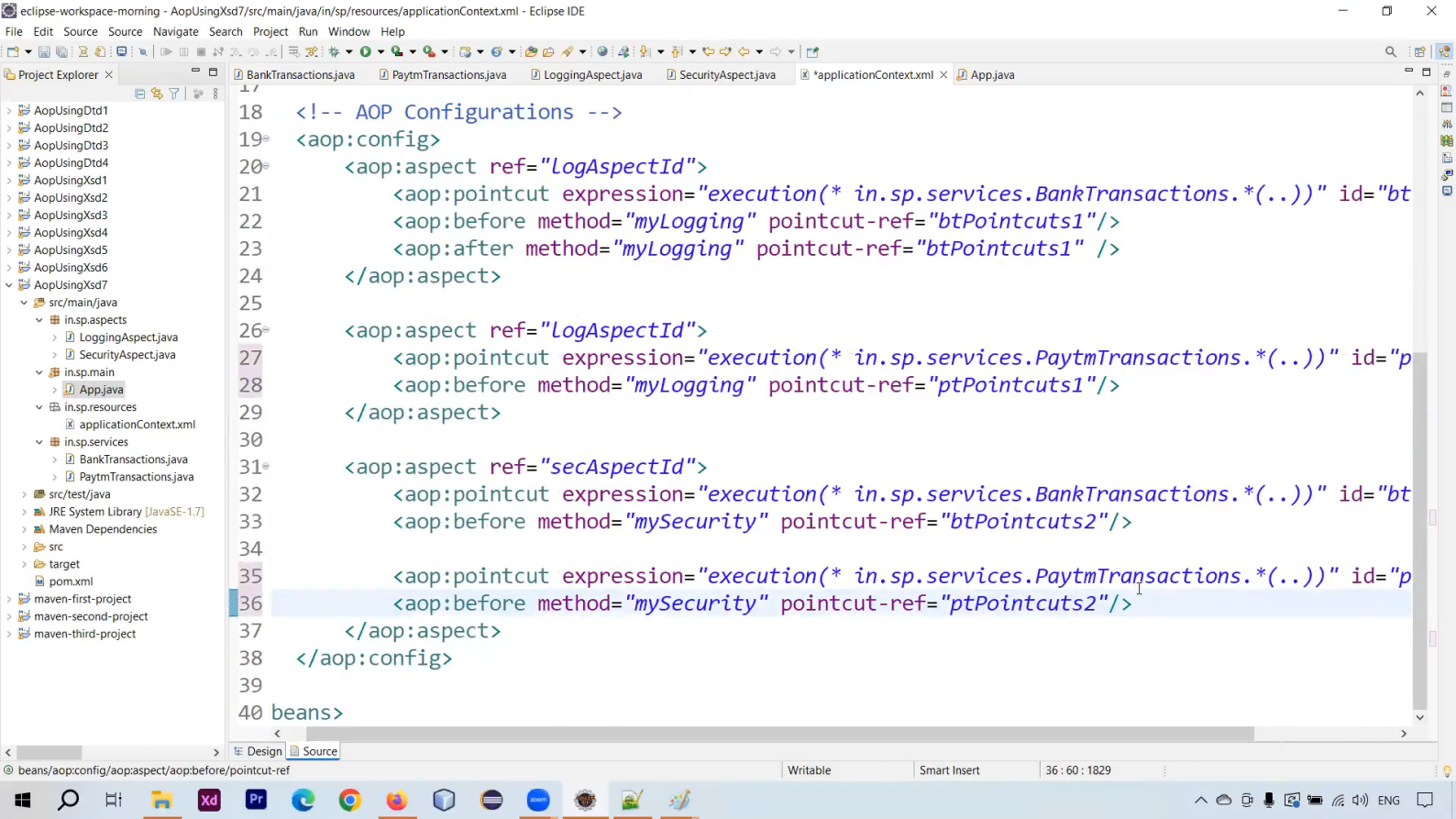
```
t expression="execution(* in.sp.services.BankTransactions.transactionUsingUpi())"
```

```
spectId">
ression="execution(* in.sp.services.BankTransactions.transactionUsingUpi())" id="btPointcuts1"
d="myLogging" pointcut-ref="btPointcuts1"/>
="myLogging" pointcut-ref="btPointcuts1" />

ression="execution(* in.sp.services.BankTransactions.transactionUsingMobileBanking())" id="btP
d="myLogging" pointcut-ref="btPointcuts2"/>
="myLogging" pointcut-ref="btPointcuts2" />
```

```
<aop:config>
    <aop:aspect ref="logAspectId">
        <aop:pointcut expression="execution(* in.sp.services.BankTransactions.transactio
        <aop:before method="myLogging" pointcut-ref="btPointcuts1"/>
        <aop:after method="myLogging" pointcut-ref="btPointcuts1" />

        <aop:pointcut expression="execution(* in.sp.services.BankTransactions.transactio
        <aop:before method="myLogging" pointcut-ref="btPointcuts2"/>
        <aop:after method="myLogging" pointcut-ref="btPointcuts2" />
    </aop:aspect>
</aop:config>
```


eclipse-workspace-morning - AopUsingXsd5/src/main/java/in/sp/resources/applicationContext.xml - Eclipse IDE
File Edit Source Source Navigate Search Project Run Window Help
Project Explorer
AopUsingDtd1
AopUsingDtd2
AopUsingDtd3
AopUsingDtd4
AopUsingXsd1
AopUsingXsd2
AopUsingXsd3
AopUsingXsd4
AopUsingXsd5
src/main/java
in.sp.aspects
LoggingAspect.java
SecurityAspect.java
in.sp.main
in.sp.resources
applicationContext.xml
in.sp.services
BankTransactions.java
src/test/java
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
target
pom.xml
maven-first-project
maven-second-project
maven-third-project
BankTransactions.java
LoggingAspect.java
SecurityAspect.java
applicationContext.xml
<!-- Aspect Bean Defination -->
<bean class="in.sp.aspects.LoggingAspect" id="logAspectId" />
<bean class="in.sp.aspects.SecurityAspect" id="secAspectId" />
<!-- AOP Configurations -->
<aop:config>
<!-- logging aspect -->
<aop:aspect ref="logAspectId">
<aop:pointcut expression="execution(* in.sp.services.BankTransactions.*(..))" id="
<aop:before method="myLogging" pointcut-ref="btLogPointcuts"/>
<aop:after method="myLogging" pointcut-ref="btLogPointcuts" />
</aop:aspect>
<!-- security aspect -->
<aop:aspect ref="secAspectId">
<aop:pointcut expression="execution(* in.sp.services.BankTransactions.*(..))" id="
<aop:before method="myLogging" pointcut-ref="btSecPointcuts"/>
<aop:after method="myLogging" pointcut-ref="btSecPointcuts" />
</aop:aspect>
</aop:config>
</beans>
Design
Source
48:19
1:01:24
48:19/1:01:24


eclipse-workspace-morning - AopUsingXsd6/src/main/java/in/sp/resources/applicationContext.xml - Eclipse IDE
File Edit Source Source Navigate Search Project Run Window Help
Project Explorer
AopUsingDtd1
AopUsingDtd2
AopUsingDtd3
AopUsingDtd4
AopUsingXsd1
AopUsingXsd2
AopUsingXsd3
AopUsingXsd4
AopUsingXsd5
AopUsingXsd6
src/main/java
in.sp.aspects
LoggingAspect.java
in.sp.main
in.sp.resources
applicationContext.xml
in.sp.services
BankTransactions.java
PaytmTransactions.java
src/test/java
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
target
pom.xml
maven-first-project
maven-second-project
maven-third-project
BankTransactions.java
PaytmTransactions.java
LoggingAspect.java
*applicationContext.xml
10 <!-- Bean Defination -->
11 <bean class="in.sp.services.BankTransactions" id="bankTransId" />
12 <bean class="in.sp.services.PaytmTransactions" id="paytmTransId" />
13
14 <!-- Aspect Bean Defination -->
15 <bean class="in.sp.aspects.LoggingAspect" id="logAspectId" />
16
17 <!-- AOP Configurations -->
18 <aop:config>
19 <aop:aspect ref="logAspectId">
20 <aop:pointcut expression="execution(* in.sp.services.BankTransactions.*(..))" id="
21 <aop:before method="myLogging" pointcut-ref="btPointcuts1"/>
22 <aop:after method="myLogging" pointcut-ref="btPointcuts1" />
23 </aop:aspect>
24
25 <aop:aspect ref="logAspectId">
26 <aop:pointcut expression="execution(* in.sp.services.PaytmTransactions.*(..))" id="
27 <aop:before method="myLogging" pointcut-ref="btPointcuts1"/>
28 <aop:after method="myLogging" pointcut-ref="btPointcuts1" />
29 </aop:aspect>
30 </aop:config>
31
32 </beans>
Design
Source
beans/aop:config/aop:aspect/aop:pointcut/expression
Writable
Smart Insert
26 : 88 : 1263
ENG


eclipse-workspace-morning - AopUsingXsd7/src/main/java/in/sp/resources/applicationContext.xml - Eclipse IDE
File Edit Source Source Navigate Search Project Run Window Help
Project Explorer
AopUsingDtd1
AopUsingDtd2
AopUsingDtd3
AopUsingDtd4
AopUsingXsd1
AopUsingXsd2
AopUsingXsd3
AopUsingXsd4
AopUsingXsd5
AopUsingXsd6
AopUsingXsd7
src/main/java
in.sp.aspects
LoggingAspect.java
SecurityAspect.java
in.sp.main
App.java
in.sp.resources
applicationContext.xml
in.sp.services
BankTransactions.java
PaytmTransactions.java
src/test/java
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
target
pom.xml
maven-first-project
maven-second-project
maven-third-project
BankTransactions.java
PaytmTransactions.java
LoggingAspect.java
SecurityAspect.java
*applicationContext.xml
App.java
18 <!-- AOP Configurations -->
19 <aop:config>
20 <aop:aspect ref="LogAspectId">
21 <aop:pointcut expression="execution(* in.sp.services.BankTransactions.*(..))" id="bt
22 <aop:before method="myLogging" pointcut-ref="btPointcuts1"/>
23 <aop:after method="myLogging" pointcut-ref="btPointcuts1" />
24 </aop:aspect>
25
26 <aop:aspect ref="LogAspectId">
27 <aop:pointcut expression="execution(* in.sp.services.PaytmTransactions.*(..))" id="p
28 <aop:before method="myLogging" pointcut-ref="ptPointcuts1"/>
29 </aop:aspect>
30
31 <aop:aspect ref="secAspectId">
32 <aop:pointcut expression="execution(* in.sp.services.BankTransactions.*(..))" id="bt
33 <aop:before method="mySecurity" pointcut-ref="btPointcuts2"/>
34
35 <aop:pointcut expression="execution(* in.sp.services.PaytmTransactions.*(..))" id="p
36 <aop:before method="mySecurity" pointcut-ref="ptPointcuts2"/>
37 </aop:aspect>
38 </aop:config>
39
40 beans>
Design
Source
beans/aop:config/aop:aspect/aop:before/pointcut-ref
Writable
Smart Insert
36 : 60 : 1829
ENG