

Rapport de l'Activité Pratique

Filière :

« Génie du Logiciel et des Systèmes Informatiques Distribués »

GLSID 3

**programmation orientée
aspect avec AspectJ et Spring
AOP**

Réalisée par : Meriem TAHIRI

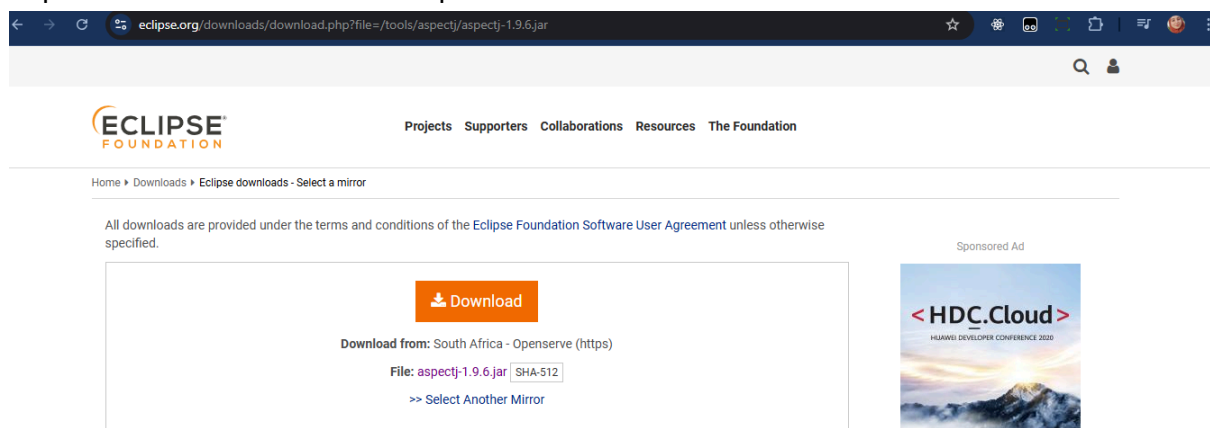
Lien Github : <https://github.com/meriemtahiri/Design-Pattern>

Manipulation avec AspectJ

(Tisseur d'aspects statique)

Étape 1 : Téléchargement de l'outil AspectJ

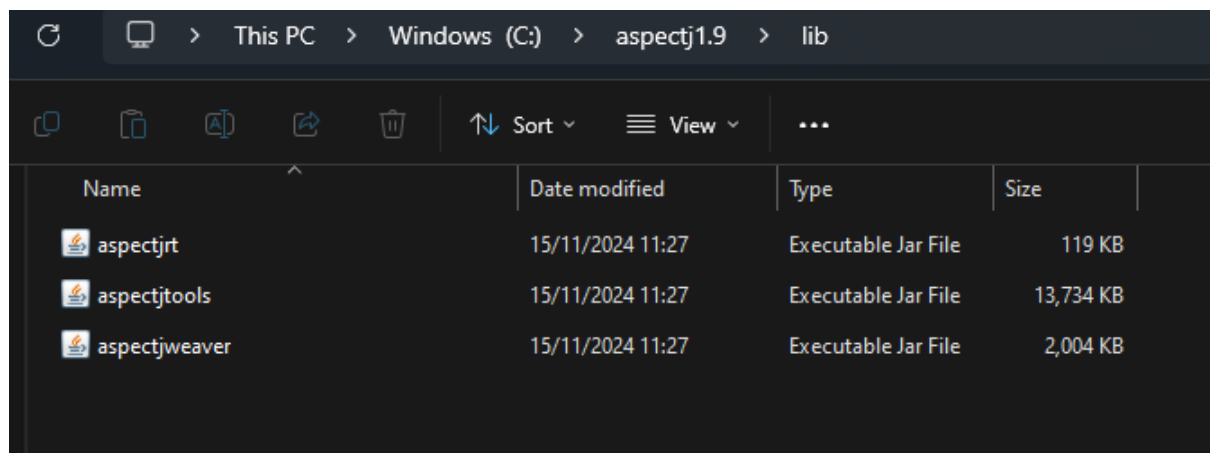
La première étape consiste à télécharger le fichier JAR de l'outil AspectJ. Celui-ci est disponible sur le site officiel d'Eclipse.



l'exécution du fichier : `aspectj-1.9.6.jar`.



Ces fichiers constituent le cœur d'AspectJ et doivent être correctement intégrés au chemin de classe (`classpath`) du projet pour permettre l'utilisation et le tissage des aspects.

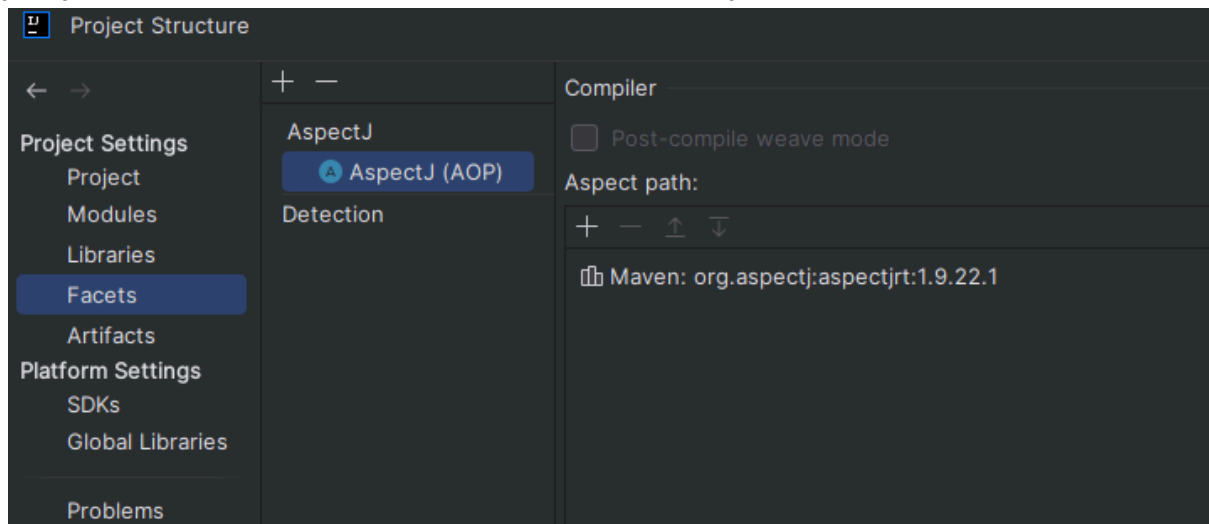


Étape 2 : Configuration de l'environnement

J'ai créé un nouveau projet java avec maven dans lequel j'ai ajouté la dépendance de aspectJ :

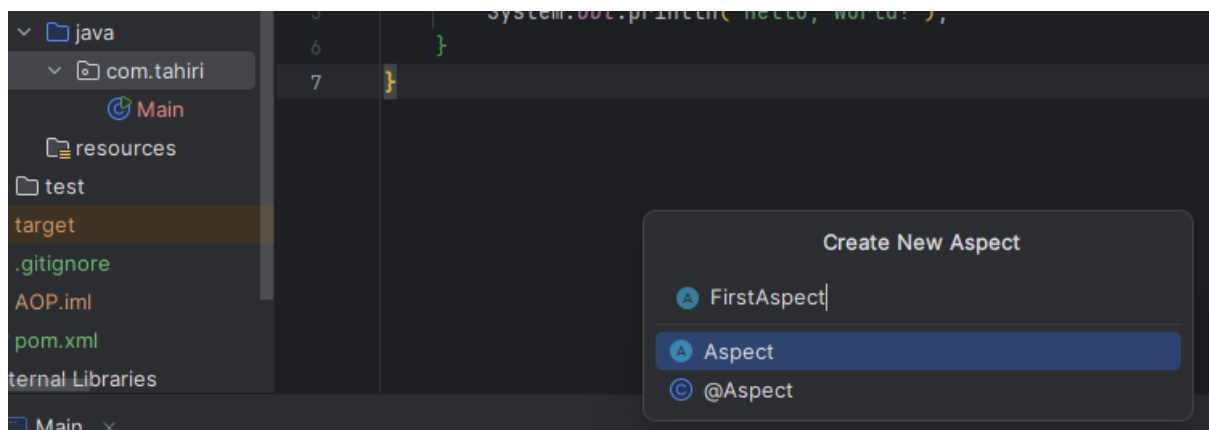
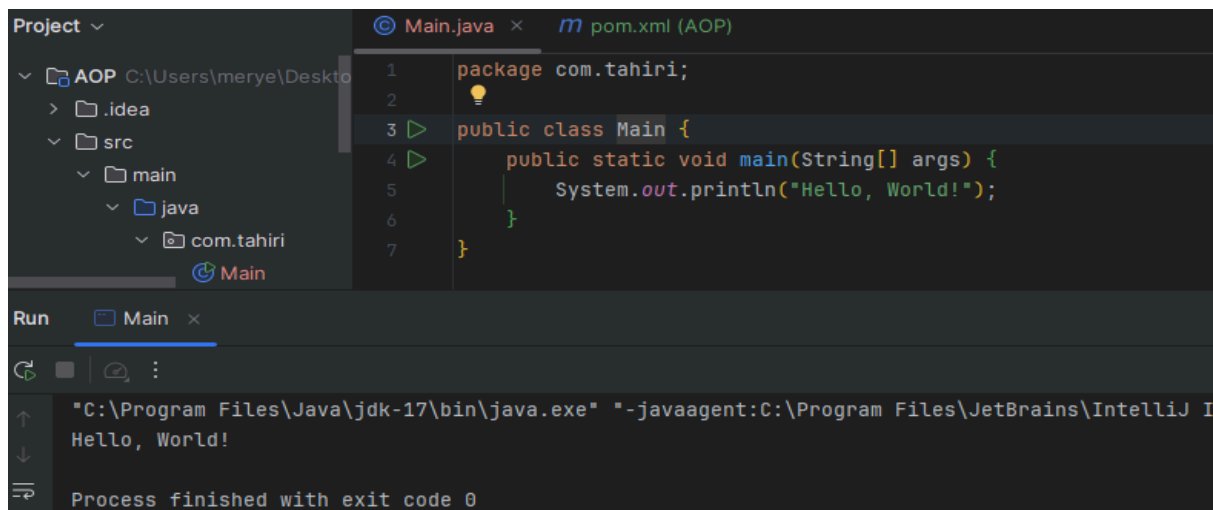
```
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.aspectj/aspectjrt -->
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.9.22.1</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

j'ai ajouter aussi la facette d'AspectJ au structure du projet :



Maintenant nous avons activé un environnement prêt pour le développement avec des aspects.

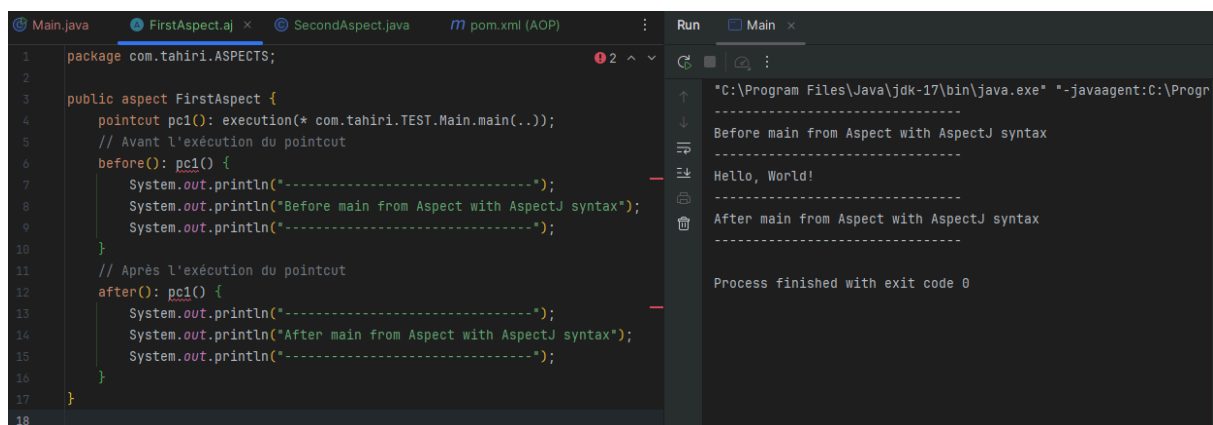
Étape 3 : Créer un aspect avec AspectJ



Il y a deux façons pour créer un Aspect soit l'enseigne comme montré ci-dessus soit avec l'annotation Aspect.

première manipulation “AspectJ Syntax”:

1. La JVM invoque l'aspect.
2. L'exécution de la méthode principale est interceptée par le **before advice**.
3. La méthode principale est appelée.
4. Une fois terminée, le **after advice** s'exécute.



en utilisant **around()** :

Dans ce cas, nous utilisons un **around advice** au lieu des **before** et **after advices** séparés. Cela permet d'exécuter du code avant et après l'exécution du pointcut intercepté, tout en contrôlant explicitement l'appel de la méthode d'origine via **proceed()**.

```
3 public aspect FirstAspect {
11 // // Après l'exécution du pointcut
12 // after(): pc1() {
13 //     System.out.println("-----");
14 //     System.out.println("After main from Aspect with AspectJ syntax");
15 //     System.out.println("-----");
16 // }
17 Object around(): pc1() {
18     System.out.println("-----");
19     System.out.println("Before main from Aspect with AspectJ syntax");
20     System.out.println("-----");
21
22     Object result = proceed(); // Appelle la méthode originale
23
24     System.out.println("-----");
25     System.out.println("After main from Aspect with AspectJ syntax");
26     System.out.println("-----");
27
28     return result; // Retourne la valeur originale (si applicable)
29 }
30 }
```

Run console output:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program F:
-----
Before main from Aspect with AspectJ syntax
-----
Hello, World!
-----
After main from Aspect with AspectJ syntax
-----
Process finished with exit code 0
```

5. La JVM invoque l'aspect.
6. L'exécution de la méthode principale est interceptée par le **before advice**.
7. La méthode principale est appelée via la fonction **proceed()**.
8. Une fois terminée, le **after advice** s'exécute.

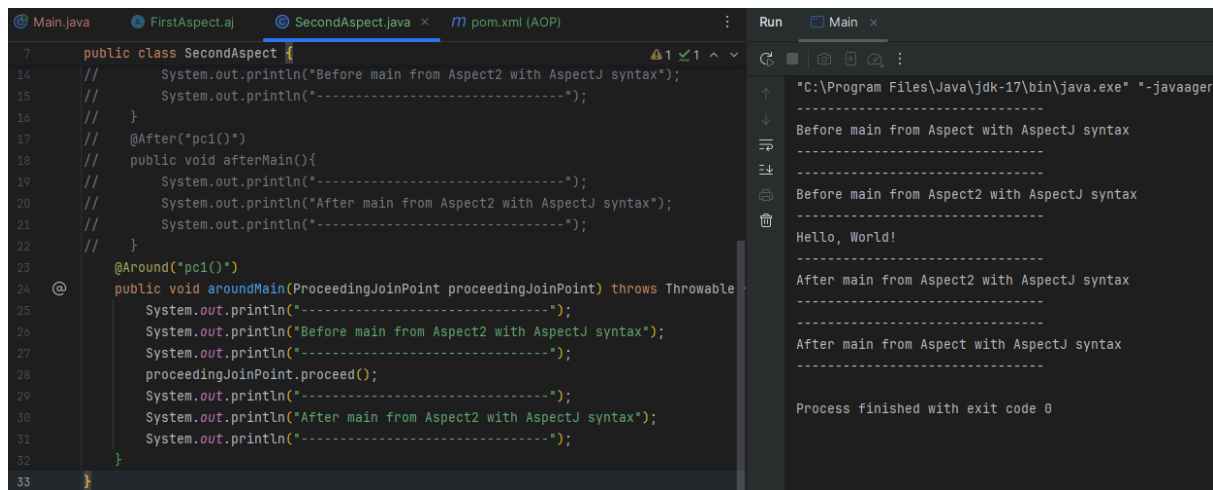
deuxième manipulation “Classe Syntax”:

```
1 package com.tahiri.ASPECTS;
2
3 import org.aspectj.lang.annotation.After;
4 import org.aspectj.lang.annotation.Aspect;
5 import org.aspectj.lang.annotation.Before;
6 import org.aspectj.lang.annotation.Pointcut;
7
8 @Aspect
9 public class SecondAspect {
10     @Pointcut("execution(* com.tahiri.TEST.Main.main(..)*)")
11     public void pc1(){
12
13         @Before("pc1()")
14         public void beforeMain() {
15             System.out.println("-----");
16             System.out.println("Before main from Aspect2 with AspectJ syntax");
17             System.out.println("-----");
18         }
19
20         @After("pc1()")
21         public void afterMain(){
22             System.out.println("-----");
23             System.out.println("After main from Aspect2 with AspectJ syntax");
24             System.out.println("-----");
25         }
26     }
27 }
```

Run console output:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Progrn
-----
Before main from Aspect1 with AspectJ syntax
-----
Before main from Aspect2 with AspectJ syntax
-----
Hello, World!
-----
After main from Aspect2 with AspectJ syntax
-----
After main from Aspect1 with AspectJ syntax
-----
Process finished with exit code 0
```

en utilisant **around()** :



```
7 public class SecondAspect {
14 //     System.out.println("Before main from Aspect2 with AspectJ syntax");
15 //     System.out.println("-----");
16 // }
17 // @After("pc1()")
18 // public void afterMain(){
19 //     System.out.println("-----");
20 //     System.out.println("After main from Aspect2 with AspectJ syntax");
21 //     System.out.println("-----");
22 // }
23
24 @Around("pc1()")
25 public void aroundMain(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
26     System.out.println("-----");
27     System.out.println("Before main from Aspect2 with AspectJ syntax");
28     System.out.println("-----");
29     proceedingJoinPoint.proceed();
30     System.out.println("-----");
31     System.out.println("After main from Aspect2 with AspectJ syntax");
32     System.out.println("-----");
33 }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent
-----
Before main from Aspect with AspectJ syntax
-----
Before main from Aspect2 with AspectJ syntax
-----
Hello, World!
-----
After main from Aspect2 with AspectJ syntax
-----
After main from Aspect with AspectJ syntax
-----
Process finished with exit code 0
```

Use case AspectJ

Contexte :

Cette manipulation concerne la gestion des comptes bancaires à l'aide d'une application en Java. L'application permet de créer des comptes, effectuer des dépôts et des retraits, ainsi que de consulter les soldes des comptes. Ce programme utilise une approche orientée aspect pour gérer la sécurité par une vérification des accès à l'application via un mécanisme de login.

L'objectif principal de ce projet est de simuler un système bancaire simple en Java en utilisant la programmation orientée aspect avec l'utilisation de la bibliothèque **AspectJ** pour gérer les logs et la sécurité des opérations bancaires.

Structure du projet :

Le projet est organisé en plusieurs classes et interfaces, chacune ayant une fonction spécifique :

- **Compte** : Classe représentant un compte bancaire.
- **IMetierBanque** et **IMetierBanqueImpl** : Interface et implémentation pour gérer les opérations bancaires comme l'ajout de compte, le virement, le retrait et la consultation des comptes.
- **LoginAspect**, **PatchRetraitAspect**, **SecurityAspect** : Classes d'aspects qui utilisent AspectJ pour ajouter des fonctionnalités transversales comme la gestion des logs, des vérifications de solde avant un retrait et la gestion de la sécurité.
- **Application** : La classe principale qui permet à l'utilisateur d'interagir avec l'application via la ligne de commande.

Explication du Code :

Classe **Compte** :

La classe **Compte** est utilisée pour stocker les informations d'un compte bancaire et fournir des méthodes pour manipuler ces informations.

```
1  package com.tahiri.METIER;
2
3  public class Compte {
4      private Long code;
5      private double solde;
6
7      public Compte(Long code, double solde) {
8          this.code = code;
9          this.solde = solde;
10     }
11
12     public Compte() {
13
14     }
15
16     > public Long getCode() { return code; }
17
18
19     > public double getSolde() { return solde; }
20
21
22
23     > public void setCode(Long code) { this.code = code; }
24
25
26
27     > public void setSolde(double solde) { this.solde = solde; }
28
29
30
31
```

Interface **IMetierBanque** et Classe **IMetierBanqueImpl** :

IMetierBanque définit les opérations principales sur les comptes bancaires, tandis que **IMetierBanqueImpl** fournit l'implémentation de ces opérations avec une collection (**Map**) pour stocker les comptes.

```
package com.tahiri.METIER;

public interface IMetierBanque {
    void addCompte(Compte cp);

    void verser(Long code, double montant);

    void retirer(Long code, double montant);
}
```

```
Compte consulter(Long code);  
}
```

```
package com.tahiri.METIER;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class IMetierBanqueImpl implements IMetierBanque {  
    private final Map<Long, Compte> comptes = new HashMap<>();  
  
    @Override  
    public void addCompte(Compte c) {  
        comptes.put(c.getCode(), c);  
    }  
  
    @Override  
    public void verser(Long code, double montant) {  
        Compte compte = comptes.get(code);  
        compte.setSolde(compte.getSolde() + montant);  
    }  
  
    @Override  
    public void retirer(Long code, double montant) {  
        Compte compte = comptes.get(code);  
        compte.setSolde(compte.getSolde() - montant);  
    }  
  
    @Override  
    public Compte consulter(Long code) {  
        return comptes.get(code);  
    }  
}
```

Gestion des Aspects avec AspectJ :

Les classes `LoginAspect`, `PatchRetraitAspect` et `SecurityAspect` utilisent AspectJ pour intercepter les méthodes des classes métiers et ajouter des fonctionnalités supplémentaires comme la journalisation des opérations et la vérification des accès.

```
package com.tahiri.ASPECTS;  
  
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.Pointcut;  
  
import java.util.Scanner;  
  
@Aspect
```



```

public class SecurityAspect {

    @Pointcut("execution(* com.tahiri.TEST.Application.start(..))")
    public void loginAspect(){};

    @Around("loginAspect()")
    public void autourStart(ProceedingJoinPoint proceedingJoinPoint, JoinPoint
joinPoint) throws Throwable {
        Scanner scanner=new Scanner(System.in);
        System.out.println("username : ");
        String username = scanner.next();
        System.out.println("password : ");
        String password = scanner.next();
        if(username.equals("meriem") && password.equals("meriem")){
            proceedingJoinPoint.proceed();
        }else{
            System.out.println("Access Denied ..... ");
        }
    }
}

```

Cet aspect ajoute un mécanisme de sécurité **basé sur une authentification simple** avant de lancer l'application. Si l'utilisateur entre des identifiants corrects, l'application continue ; sinon, elle refuse l'accès.

```

package com.tahiri.ASPECTS;

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.Pointcut;

import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Logger;

@Aspect
public class LoginAspect {

    Logger logger=Logger.getLogger(LoginAspect.class.getName());

    public LoginAspect() throws IOException {
        logger.addHandler(new FileHandler("log.xml"));
        // this logger shouldn't send its output to its parent Logger
        logger.setUseParentHandlers(false);
    }

    @Pointcut("execution(* com.tahiri.METIER.IMetierBanqueImpl.*(..) ) ")
    public void pc1(){ }

    /*
    @Before("pc1() ")
    public void avant(JoinPoint joinPoint){
        t1=System.currentTimeMillis();

```

```

logger.info("-----");
        logger.info("Avant execution de la methode
"+joinPoint.getSignature());
    }
    @After("pc1()")
    public void apres(JoinPoint joinPoint){
        logger.info("apres execution de la methode
"+joinPoint.getSignature());
        t2=System.currentTimeMillis();
        logger.info("Duree d'execution de la methode :"+ (t2-t1));

logger.info("-----");
    }*/
    @Around("pc1()")
    public Object autour(ProceedingJoinPoint proceedingJoinPoint,JoinPoint
joinPoint) throws Throwable {
        long t1=System.currentTimeMillis();
        logger.info("-----");
        logger.info("Avant execution de la methode
"+joinPoint.getSignature());
        Object resultat=proceedingJoinPoint.proceed();
        logger.info("apres execution de la methode
"+joinPoint.getSignature());
        long t2=System.currentTimeMillis();
        logger.info("Duree d'execution de la methode :"+ (t2-t1));
        logger.info("-----");
        return resultat;
    }
}

```

Cette classe journalise le temps d'exécution des méthodes de l'interface **IMetierBanque** on stockons les résultats dans le fichier **log.xml** .Elle utilise **@Around** pour intercepter l'exécution des méthodes et ajouter des comportements avant et après l'exécution.

```

package com.tahiri.ASPECTS;

import com.tahiri.METIER.Compte;
import com.tahiri.METIER.IMetierBanque;
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.Pointcut;

@Aspect
public class PatchRetraitAspect {
    @Pointcut("execution(* com.tahiri.METIER.IMetierBanqueImpl.retirer(..) )")
    public void pc1(){ }
    @Around("pc1() && args(code,montant)")
    public Object autourRetirer(Long code,double montant,ProceedingJoinPoint
proceedingJoinPoint, JoinPoint joinPoint) throws Throwable {
        IMetierBanque metierBanque=(IMetierBanque) joinPoint.getTarget();

```

```

        Compte compte=metierBanque.consulter(code);
        if(compte.getSolde()<montant) throw new RuntimeException("Balance not
sufficient ");
        return proceedingJoinPoint.proceed();
    }
}

```

Cet aspect implémente une règle métier importante : empêcher les retraits si le solde du compte est insuffisant. Cela évite d'avoir une logique de validation directement dans la classe métier, en séparant les préoccupations grâce à l'AOP.

Classe Application (main) :

```

package com.tahiri.TEST;

import com.tahiri.METIER.Compte;
import com.tahiri.METIER.IMetierBanque;
import com.tahiri.METIER.IMetierBanqueImpl;

import java.util.Scanner;

public class Application {
    public static void main(String[] args) {
        new Application().start();
    }

    public void start() {
        System.out.println("Application is starting ... 🤖");
        Scanner scanner = new Scanner(System.in);
        System.out.println("Account code : ");
        Long code = scanner.nextLong();
        System.out.println("initial balance : ");
        double solde = scanner.nextDouble();

        IMetierBanque metierBanque = new IMetierBanqueImpl();
        metierBanque.addCompte(new Compte(code, solde));
        while (true) {
            try {
                System.out.println("Operation Type: ");
                String type = scanner.next();
                if (type.equals("q")) break;
                System.out.println("Amount : ");
                double montant = scanner.nextDouble();
                if (type.equals("v")) {
                    metierBanque.verser(code, montant);
                } else if (type.equals("r")) {
                    metierBanque.retirer(code, montant);
                }
                Compte c = metierBanque.consulter(code);
                System.out.println("Account Balance = $" + c.getSolde());
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

```

```
    }  
    }  
    System.out.println("end of Application,Thanks for being here ❤️");  
    }  
}
```

Résultats et Tests

Si les informations d'authentification est incorrect :

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program F  
username :  
meriem  
password :  
123  
Access Denied .....  
  
Process finished with exit code 0  
|
```

pour faire appelle aux différentes méthodes et générer le fichier log.xml:

```
Run Application x
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Progra
username :
meriem
password :
meriem
Application is starting ... 🤖
Account code :
123
initial balance :
10000
Operation Type:
v
Amount :
2000
Account Balance = $12000.0
Operation Type:
r
Amount :
1000
Account Balance = $11000.0
Operation Type:
q
end of Application,Thanks for being here ❤️

Process finished with exit code 0
```

la journalisation est bien enregistrer dans log.xml et l'appelle à l'aspect pc1() est bien fait :

```
</> log.xml x
3   <log>
4   <record>
11  <method>addCompte_aroundBody1$advice</method>
12  <thread>1</thread>
13  <message>-----</message>
14  </record>
15  <record>
16  <date>2024-11-21T21:47:09.041875200Z</date>
17  <millis>1732225629041</millis>
18  <sequence>1</sequence>
19  <logger>com.tahiri.ASPECTS.LoginAspect</logger>
20  <level>INFO</level>
21  <class>com.tahiri.METIER.IMetierBanqueImpl</class>
22  <method>addCompte_aroundBody1$advice</method>
23  <thread>1</thread>
24  <message>Avant execution de la methode void com.tahiri.METIER.IMetierBanqueImpl.addCompte(Compte)</message>
25  </record>
26  <record>
27  <date>2024-11-21T21:47:09.043866700Z</date>
28  <millis>1732225629043</millis>
29  <sequence>2</sequence>
30  <logger>com.tahiri.ASPECTS.LoginAspect</logger>
31  <level>INFO</level>
32  <class>com.tahiri.METIER.IMetierBanqueImpl</class>
33  <method>addCompte_aroundBody1$advice</method>
34  <thread>1</thread>
35  <message>apres execution de la methode void com.tahiri.METIER.IMetierBanqueImpl.addCompte(Compte)</message>
36  </record>
```

si le solde ne suffit pas pour faire un retrait :

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Pr
username :
meriem
password :
meriem
Application is starting ... 🤖
Account code :
123
initial balance :
10000
Operation Type:
r
Amount :
100000
Balance not sufficient
Operation Type:
q
end of Application,Thanks for being here ❤️

Process finished with exit code 0
```

Manipulation avec Spring AOP

(Tisseur d'aspects dynamique)

Étape 1 : Créer un aspect avec Spring AOP

```
@Component
@Aspect
@EnableAspectJAutoProxy
public class LoggingAspect {
    Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @Around("execution(* com.tahiri.SERVICES..*(..))")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        Object result = null;
        long t1 = System.currentTimeMillis();
        logger.info(msg: "Logging Aspect ==> Before execution ");
        result = joinPoint.proceed();
        logger.info(msg: "Logging Aspect ==> After execution");
        long t2 = System.currentTimeMillis();
        logger.info(msg: "Execution Duration : " + (t2-t1));
        return result;
    }
}
```

Cette fonctionnalité intercepte les appels des méthode **process()** et **compute()** dans IMitierImpl :

```
@Service
public class IMetierImpl implements IMetier {

    @Override
    public void process() {
        System.out.println("Business processing ...");
    }

    @Override
    public double compute() {
        double data = 80;
        System.out.println("Business Computing and returning result ....");
        return data;
    }
}
```

le test :

```

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
        IMetier metier = context.getBean(IMetier.class);
        System.out.println(metier.getClass().getName());
        metier.process();
        System.out.println(metier.compute());
    }
}

```

le résultat :

```

"C:\Program Files\Java\jdk-17\bin\java.exe" ...
jdk.proxy2.$Proxy21
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> Before execution
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> After execution
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Execution Duration : 36
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> Before execution
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> After execution
Nov 30, 2024 9:35:08 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Execution Duration : 3
Business processing ...
Business Computing and returning result ....
80.0

Process finished with exit code 0

```

On constate ici que, lorsqu'on injecte l'implémentation de l'interface `IMetier`, c'est plutôt un proxy qui est appelé. Et ce proxy il est généré dynamiquement au moment de l'exécution.

avec spring AOP on peut également créer une annotation liée à l'aspect de logging. Alors tous les méthodes qui sont précédés par cette annotation vont être ciblées par cette aspect.

Annotation Log liée à l'aspect Logging:

création de l'annotation :

```

@Retention(RetentionPolicy.RUNTIME) //It is processed at runtime.
@Target(ElementType.METHOD) //apply it on methods where logging behavior is desired.
public @interface Log {
}

```

pour lier l'annotation à l'aspect de logging :


```

@Component
@Aspect
@EnableAspectJAutoProxy
public class LoggingAspect {
    Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @Around("@annotation(com.tahiri.ASPECTS.Log)")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        long t1 = System.currentTimeMillis();
        logger.info(msg: "Logging Aspect ==> Before execution ");
        Object result = joinPoint.proceed();
        logger.info(msg: "Logging Aspect ==> After execution");
        long t2 = System.currentTimeMillis();
        logger.info(msg: "Execution Duration : " + (t2-t1));
        return result;
    }
}

```

Pour appliquer l'aspect sur une méthode il suffit de la précéder maintenant par l'annotation Log.

```

@Service
public class IMetierImpl {
    @Override
    @Log
    public void compute() {
        // ...
    }

    @Override
    @Log
    public double compute(double data) {
        // ...
    }
}

```

et on va avoir le même résultat de tout à l'heure.

Annotation SecuredByAspect liée à l'aspect Security:

j'ai ajouté donc l'annotation qui va s'occuper des autorisations à une telle méthode.

```

@Service
public class IMetierImpl implements IMetier {
    @Log
    @SecuredByAspect(roles = {"ADMIN", "USER"})
    public void process() {
        System.out.println("Business processing ...");
    }

    @Override
    @Log
    @SecuredByAspect(roles = {"ADMIN"})
    public double compute() {
        double data = 80;
    }
}

```

la définition de l'annotation :

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface SecuredByAspect {
    String[] roles();
}

```

la définition de l'aspect liée à l'annotation :

```

@Component
@Aspect
@EnableAspectJAutoProxy
public class SecurityAspect {

    @Around(value = "@annotation(securedByAspect)", argNames =
        "proceedingJoinPoint,securedByAspect")
    public Object log(ProceedingJoinPoint proceedingJoinPoint, SecuredByAspect
        securedByAspect) throws Throwable {
        String[] roles = securedByAspect.roles();
        boolean authorized = false;
        for (String r : roles) { //pour comparer les rôles d'accès à la méthode avec ceux qui sont fournis.
            if (SecurityContext.hasRole(r)) {
                authorized = true;
                break;
            }
        }
        if (!authorized) {
            throw new RuntimeException("Not Authorized");
        } else {
            return proceedingJoinPoint.proceed();
        }
    }
}

```

Cette classe pour définir d'une manière simple les utilisateurs avec leurs rôles :

```

public class SecurityContext {
    private static String username = "";
    private static String password = "";
    private static String[] roles = {};

    public static void authenticateUser(String u, String p, String[] rs) {
        if ((u.equals("tahiri")) && (p.equals("tahiri"))) {
            username = u;
            password = p;
            roles = rs;
        } else throw new RuntimeException("Access Denied...");
    }

    public static boolean hasRole(String role) {
        for (String r : roles) {
            if (r.equals(role)) return true;
        }
        return false;
    }
}

```

test avec un utilisateur qui n'existe pas :

alors l'exception de Access Denied va être élevée :

```

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
        SecurityContext.authenticateUser(u: "noSot", p: "1234", new String[]{"ADMIN"});
        Metier metier = context.getBean(Metier.class);
        metier.process();
        System.out.println(metier.compute());
    }
}

```

```

"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Exception in thread "main" java.lang.RuntimeException Create breakpoint : Access Denied...
    at com.tahiri.SERVICES.SecurityContext.authenticateUser(SecurityContext.java:13)
    at com.tahiri.Main.main(Main.java:11)

Process finished with exit code 1

```

test avec un utilisateur existe est qui a le rôle USER :

alors une exception va être élevée parce que USER n'est pas autorisé pour l'appel de la méthode compute() :

```

Nov 30, 2024 11:31:48 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Execution Duration : 38
Nov 30, 2024 11:31:48 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> Before execution
Exception in thread "main" java.lang.RuntimeException: Create breakpoint : Not Authorized @ Explain with AI
    at com.tahiri.ASPECTS.SecurityAspect.log(SecurityAspect.java:28) <4 internal lines>
    at org.springframework.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethodWithGivenArgs(AbstractAspectJAdvice.java:634)
    at org.springframework.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethod(AbstractAspectJAdvice.java:624)
    at org.springframework.aop.aspectj.AspectJAroundAdvice.invoke(AspectJAroundAdvice.java:72)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:175)
    at org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint.proceed(MethodInvocationProceedingJoinPoint.java:89)
    at com.tahiri.ASPECTS.LoggingAspect.log(LoggingAspect.java:22) <4 internal lines>
    at org.springframework.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethodWithGivenArgs(AbstractAspectJAdvice.java:634)
    at org.springframework.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethod(AbstractAspectJAdvice.java:624)
    at org.springframework.aop.aspectj.AspectJAroundAdvice.invoke(AspectJAroundAdvice.java:72)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:175)
    at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:97)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
    at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:215)
    at jdk.proxy2/jdk.proxy2.$Proxy21.compute(Unknown Source)
    at com.tahiri.Main.main(Main.java:15)
Business processing ...

Process finished with exit code 1

```

test avec un utilisateur existe est qui a le rôle ADMIN et USER :
alors les deux méthodes vont être appelés :

```

"C:\Program Files\Java\jdk-17\bin\java.exe" ...
jdk.proxy2.$Proxy21
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> Before execution
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> After execution
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Execution Duration : 40
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> Before execution
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Logging Aspect ==> After execution
Nov 30, 2024 11:36:23 PM com.tahiri.ASPECTS.LoggingAspect log
INFO: Execution Duration : 2
Business processing ...
Business Computing and returning result ....
80.0

Process finished with exit code 0

```

Conclusion

Cette activité pratique a permis de mettre en évidence la puissance des approches orientées aspect, qu'il s'agisse de **AspectJ** (tissage statique) ou de **Spring AOP** (tissage dynamique), dans la gestion des préoccupations transversales.

Avec **AspectJ**, nous avons démontré comment séparer des fonctionnalités comme :

- **La journalisation (logging)** : en enregistrant les temps d'exécution des méthodes et en sauvegardant ces logs dans un fichier.
- **La sécurité** : en ajoutant des mécanismes d'authentification et des validations métier (par exemple, empêcher un retrait si le solde est insuffisant).
- **La validation métier** : en interceptant les appels pour s'assurer que les opérations respectent les règles définies.

Avec **Spring AOP**, les concepts d'**aspects**, de **pointcuts**, et d'**advice**s ont été utilisés pour :

- Gérer la **sécurisation des accès** à travers une annotation dédiée (@SecuredByAspect) qui compare les rôles des utilisateurs aux permissions nécessaires.
- Implémenter un **proxy dynamique** qui agit comme un intermédiaire entre les appels et les méthodes ciblées, offrant ainsi une flexibilité accrue.
- Ajouter une **annotation personnalisée de journalisation** (@Log) pour cibler dynamiquement les méthodes spécifiques.

Ces manipulations ont permis de constater que :

1. **AspectJ** fournit un contrôle plus direct et détaillé grâce à son tissage statique, mais nécessite une configuration explicite.
2. **Spring AOP** offre une solution plus intégrée et dynamique et même y a une implémentation de AspectJ d'une manière indirecte, simplifiant la mise en œuvre tout en permettant une extensibilité rapide.

Grâce à ces outils, nous avons pu isoler les préoccupations secondaires (sécurité, logs) des préoccupations principales (logique métier), rendant ainsi le code plus **modulaire, facile à maintenir, et réutilisable**.

Ainsi, cette expérience met en avant l'efficacité des outils comme AspectJ et Spring AOP pour séparer les préoccupations transversales des exigences métier, tout en complétant Spring IoC afin de fournir une solution middleware performante. Alors, la programmation orientée aspect (AOP) enrichit la programmation orientée objet (OOP) en proposant une nouvelle manière de structurer le programme.

Thanks for reading! ❤️