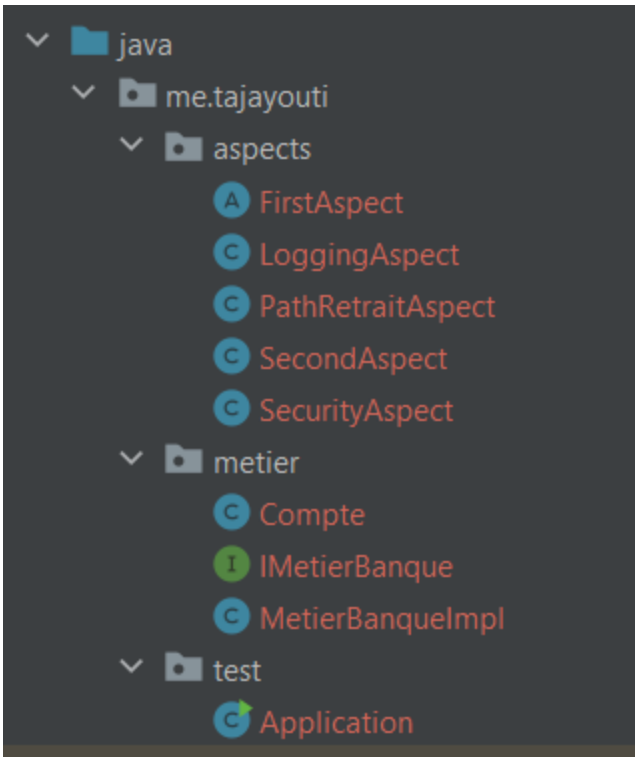
Activité Pratique Programmation Orientée Aspec

Partie I (AspectJ)

- On souhaite créer une application qui permet de gérer des comptes bancaires stockés en mémoire dans une collection de type Map. Chaque compte est défini par son code et son solde.
- Les exigences fonctionnelles de l'application sont :
  - Ajouter un compte
  - Consulter un compte
  - Verser un montant dans un compte
  - Retirer un montant d'un compte
- Les exigences techniques seront implémentées sous formes d'aspects suivants :
  - Un aspect pour la journalisation des appels de toutes les méthodes en affichant la durée d'exécution de chaque méthode
  - Un aspect pour contrôler le montant du retrait
  - Un aspect pour sécuriser l'application



Entités et règles de gestion : 🏷️

- **Une entité "Compte"**

```
public class Compte {
    private Long code;
    private double solde;

    public Compte(Long code, double solde) {
        this.code = code;
        this.solde = solde;
    }

    public Compte() {
    }

    public Long getCode() {
        return code;
    }

    public double getSolde() {
        return solde;
    }

    public void setCode(Long code) {
        this.code = code;
    }
```

```
    public void setSolde(double solde) {
        this.solde = solde;
    }

    @Override
    public String toString() {
        return "Compte{" +
                "code=" + code +
                ", solde=" + solde +
                '}';
    }
}
```

Interface "IMetier"

```
public interface IMetierBanque {
    void addCompte(Compte cp);
    void verser(Long code,double montant);
    void retirer(Long code,double montant);
    Compte consulter(Long code);
}
```

Implémentation de "IMetier"

```
public class MetierBanqueImpl implements IMetierBanque {
   private Map<Long,Compte> compteMap=new HashMap<>();

    @Override
    public void addCompte(Compte cp) {
        compteMap.put(cp.getCode(),cp);
    }

    @Override
    public void verser(Long code, double montant) {
        Compte compte=compteMap.get(code);
        compte.setSolde(compte.getSolde()+montant);
    }

    @Override
    public void retirer(Long code, double montant) {
        Compte compte=compteMap.get(code);
        compte.setSolde(compte.getSolde()-montant);

    }

    @Override
    public Compte consulter(Long code) {
        return compteMap.get(code);
    }
}
```

Aspect 1:

```
public aspect FirstAspect {


    pointcut pc1() : execution(* me..test.Application.main1(..));

    /*//before pointcut pc1 ==> code to be executed before the method "main" is
executed ==> code advice
    before():pc1() {
        System.out.printf("------------------------------------------------------
");
        System.out.println("Before the main method from Aspect with AspectJ syntax");
        System.out.printf("------------------------------------------------------
");
    }

    //after pointcut pc1 ==> code to be executed after the method "main" is executed
==> code advice
    after():pc1() {
        System.out.printf("------------------------------------------------------
");
        System.out.println("After the main method from Aspect with AspectJ syntax");
        System.out.printf("------------------------------------------------------
");
    }*/

    void around():pc1(){
        System.out.println("-----------------------------------------------");
        System.out.println("before main from aspectj with aspectj syntax");
        System.out.println("-----------------------------------------------");

        //Execution de l'operation du pointcut
        //execution(* test.Application.main(..))
        proceed();
        System.out.println("-----------------------------------------------");
        System.out.println("after main from aspectj with aspectj syntax");
        System.out.println("-----------------------------------------------");

    }
}
```

Aspect 2

```
@Aspect
public class SecondAspect {
    @Pointcut("execution(* me..test.*.main1(..))")
    public void pc1(){ }

    //code advice

   /* @Before("pc1()")
    public void beforeMain(){
```

```
        System.out.println("----------***************-----------------------------
-------");
        System.out.println("before main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");

    }
    @After("pc1()")
    public void afterMain(){

        System.out.println("----------***************-----------------------------
-------");
        System.out.println("after main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");

    }*/

    @Around("pc1()")
    public void aroundMain(ProceedingJoinPoint proceedingJoinPoint) throws Throwable
{
        System.out.println("----------***************-----------------------------
-------");
        System.out.println("before main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");
        //execute main
        proceedingJoinPoint.proceed();
        System.out.println("----------***************-----------------------------
-------");
        System.out.println("after main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");

    }
}
```

Aspect de journalisation :

```
@Aspect
public class SecondAspect {
    @Pointcut("execution(* me..test.*.main1(..))")
    public void pc1(){ }

    //code advice

   /* @Before("pc1()")
    public void beforeMain(){

        System.out.println("----------***************-----------------------------
-------");
        System.out.println("before main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");
```

```
    }
    @After("pc1()")
    public void afterMain(){

        System.out.println("----------***************-----------------------------
-------");
        System.out.println("after main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");

    }*/

    @Around("pc1()")
    public void aroundMain(ProceedingJoinPoint proceedingJoinPoint) throws Throwable
{
        System.out.println("----------***************-----------------------------
-------");
        System.out.println("before main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");
        //execute main
        proceedingJoinPoint.proceed();
        System.out.println("----------***************-----------------------------
-------");
        System.out.println("after main from aspectj with class syntax");
        System.out.println("----------***************-----------------------------
--------");

    }
}
```

Aspect de Retrait

```
@Aspect
public class PathRetraitAspect {

    //Pointcut => expression de point de coupage
    @Pointcut("execution(* me..metier.MetierBanqueImpl.retirer(..) )")
    public void pc1(){ }

    //code advice => around the  method retirer
    @Around("pc1() && args(code,montant)")
    public Object autourRetirer(Long code,double montant,ProceedingJoinPoint
proceedingJoinPoint, JoinPoint joinPoint) throws Throwable {
        MetierBanqueImpl metierBanque=(MetierBanqueImpl) joinPoint.getTarget();
        Compte compte=metierBanque.consulter(code);
        if(compte.getSolde()<montant) throw new RuntimeException("solde
insuffisant");
        return  proceedingJoinPoint.proceed();
    }
}
```

Aspect de sécurité

```
@Aspect
```

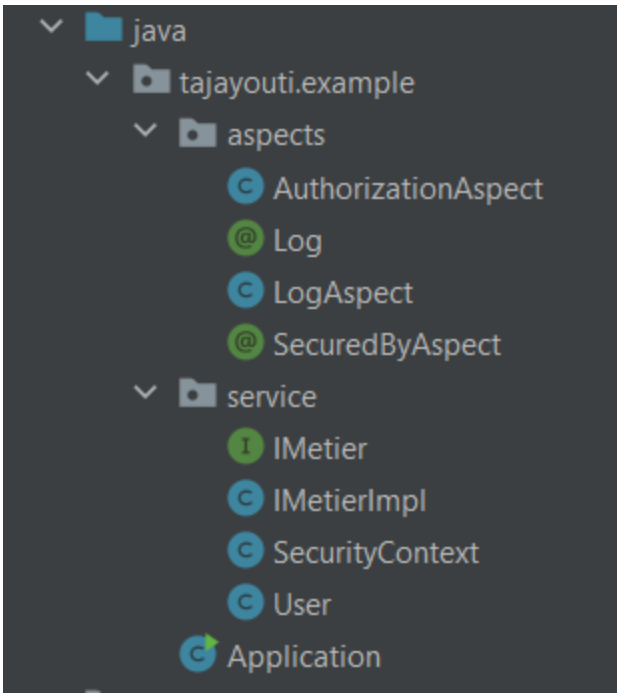
```
public class PathRetraitAspect {

    //Pointcut => expression de point de coupage
    @Pointcut("execution(* me..metier.MetierBanqueImpl.retirer(..) )")
    public void pc1(){ }

    //code advice => around the  method retirer
    @Around("pc1() && args(code,montant)")
    public Object autourRetirer(Long code,double montant,ProceedingJoinPoint
proceedingJoinPoint, JoinPoint joinPoint) throws Throwable {
        MetierBanqueImpl metierBanque=(MetierBanqueImpl) joinPoint.getTarget();
        Compte compte=metierBanque.consulter(code);
        if(compte.getSolde()<montant) throw new RuntimeException("solde
insuffisant");
        return  proceedingJoinPoint.proceed();
    }
}
```

Partie 2 (Spring AOP)

On souhaite créer une application qui offrent deux fonctionnalités métiers basiques:

- Une opération process() permettant d'effectuer un traitement quelconque
- Une opération permettant de retourner un résultat de calcul quelconque.
- Nous définissons dans cette couche métier :
  - Une interface IMetier
  - Une implémentation de cette interface
  - Ensuite nous définissons deux aspects basés sur Spring AOP

- Un Aspect pour la journalisation avec un annotation @Log qui permet de marquer dans la couche la méthode à journaliser
- Un Aspect pour sécuriser l'application avec un authentification basique avec des rôles. Pour sécuriser l'accès à une méthode, nous définissons une annotation @SecuredByAspect(roles=["ADMIN","USER"]) qui sera placée sur les méthodes à sécuriser en spécifiant les rôles requis


java
tajayouti.example
aspects
AuthorizationAspect
Log
LogAspect
SecuredByAspect
service
IMetier
IMetierImpl
SecurityContext
User
Application