

DEPARTEMENT MATHEMATIQUES ET INFORMATIQUE

Filière :
« Génie du Logiciel et des Systèmes Informatiques Distribués »
GLSID3

Examen final design pattern

Module : Design Pattern

Élaboré par :

ELMAJNI Khaoula

Encadré par :

M. EL YOUSSEFI Mohammed

Année Universitaire : 2022-2023

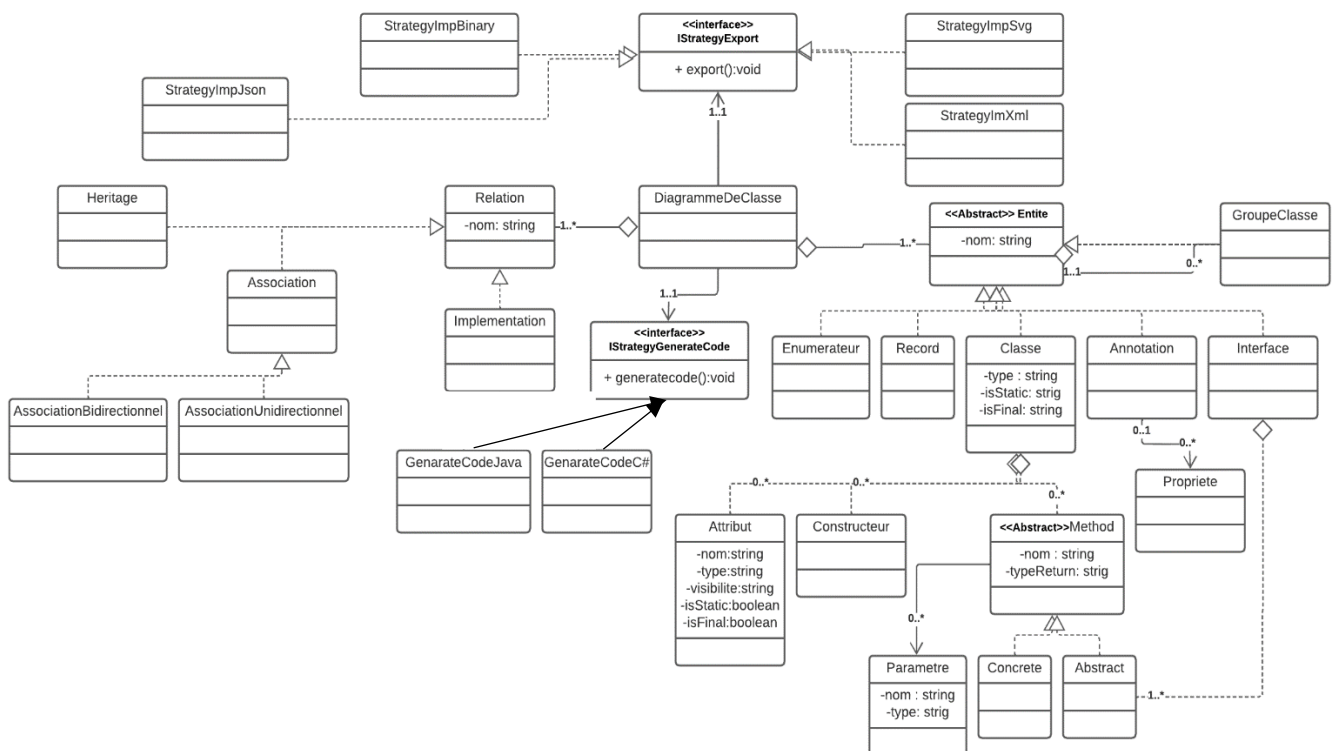
1. Établir un diagramme de classes de ce Framework en utilisant les design patterns les plus appropriés. En plus des designs patterns exigés par l'énoncé du problème, vous pouvez proposer d'autres design patterns que vous voyez pertinents pour améliorer la qualité du Framework :

Diagramme de classe :

J'ai utilisé les Design patterns suivants : **Strategy, Composite, Adapter, Observer, Template Method**

J'ai utilisé un outil en ligne «**Lucidchart**», la création d'un diagramme atteint sa limite, j'étais obligée de découper le diagramme en 2 morceaux :

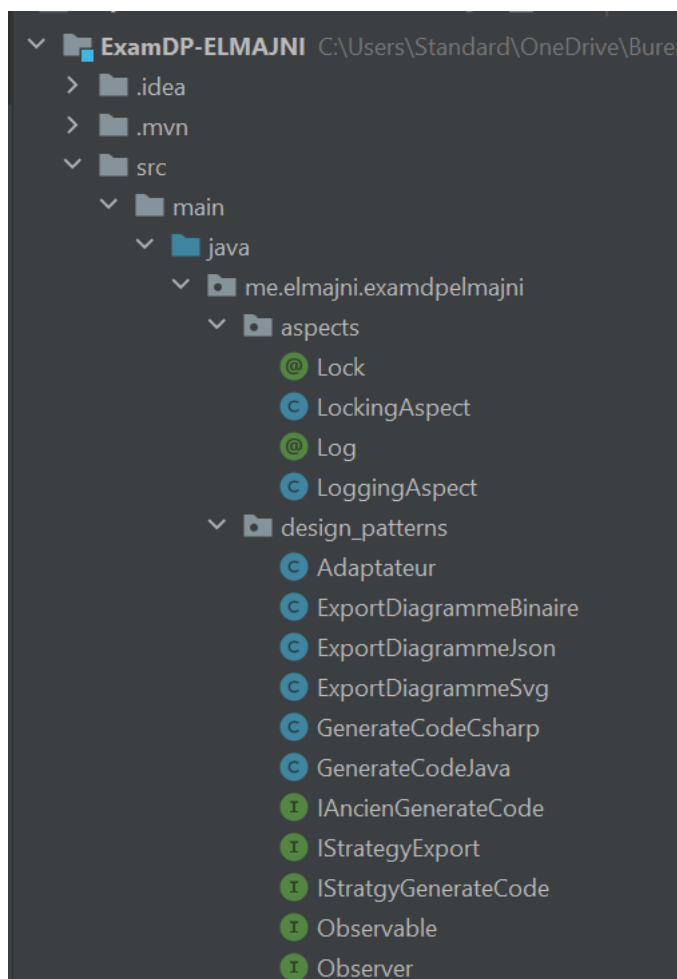
La première partie :

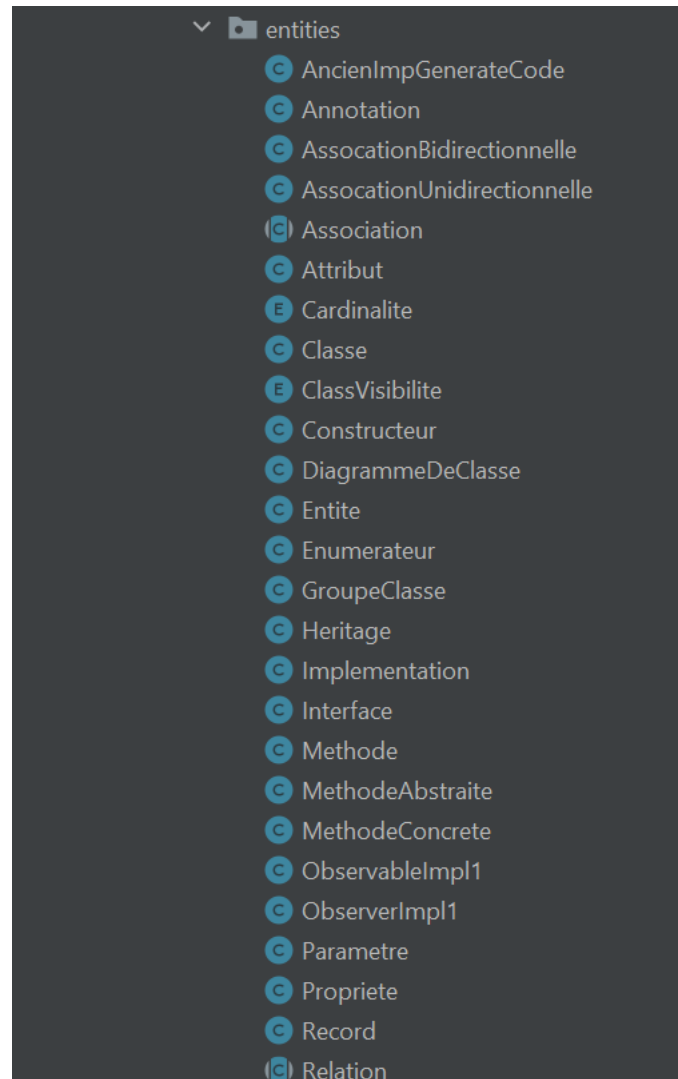


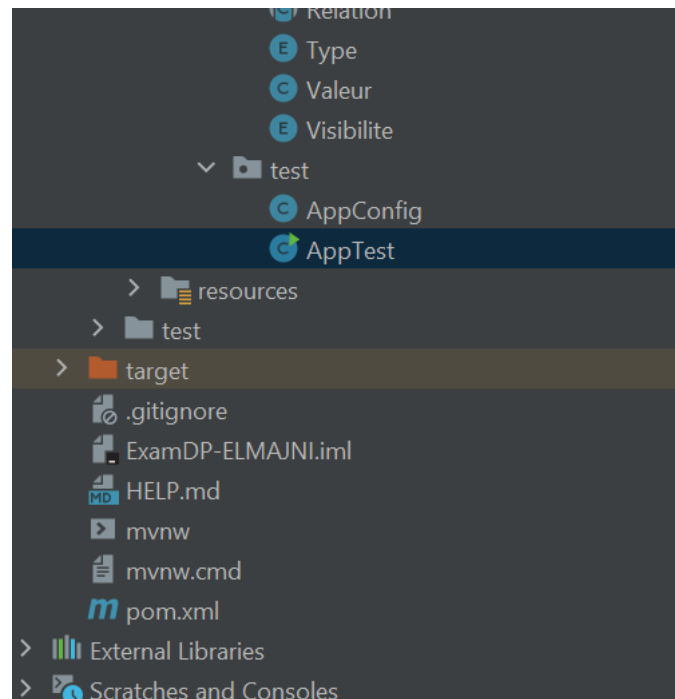
La deuxième partie :



3







La partie 'Aspects' :

J'ai utilisé le tisseur d'aspect « Spring AOP »

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
```

Annotation @Lock :

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Lock {
}
```

```
@Component
@Aspect
@EnableAspectJAutoProxy
public class LockingAspect {

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

    @Around("@annotation(me.elmajni.exampelmajni.aspects.Lock)")
    public Object lock(ProceedingJoinPoint proceedingJoinPoint) throws
```

```
Throwable {
    logger.info("From Locking Aspect ... Before
"+proceedingJoinPoint.getSignature());
    Object result = proceedingJoinPoint.proceed();
    logger.info("From Locking Aspect ... After
"+proceedingJoinPoint.getSignature());
    return result;
}

/*public Object log(ProceedingJoinPoint joinPoint) {
    System.out.println("verrouillée Methode ");
    return null;
}*/
}
```

Annotation @Log:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Log {
}
```

```
@Component
@Aspect
@EnableAspectJAutoProxy
public class LoggingAspect {

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

    @Around("@annotation(me.elmajni.exampelmajni.aspects.Log)")
    public Object log(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        long t1 = System.currentTimeMillis();
        logger.info("From Logging Aspect ... Before
"+proceedingJoinPoint.getSignature());

        Object result = proceedingJoinPoint.proceed();
        logger.info("From Logging Aspect ... After
"+proceedingJoinPoint.getSignature());
        long t2 = System.currentTimeMillis();
        logger.info("Duration : "+(t2-t1));
        return result;
    }
}
```

La partie des Design patterns :

```
public interface IStrategyExport {

    public void exporter();
}
```

implémentations :

```
public class ExportDiagrammeBinaire implements IStrategyExport {
    @Log
    @Override
    public void exporter() {
        System.out.println("Exporter le Diagramme de Classe");
    }
}
```

```
public class ExportDiagrammeJson implements IStrategyExport {
    @Log
    @Override
    public void exporter() {
        System.out.println("Export diagramme json");
    }
}
```

```
public class ExportDiagrammeSvg implements IStrategyExport {
    @Log
    @Override
    public void exporter() {
        System.out.println("Export diagramme svg");
    }
}
```

```
public interface IStratgyGenerateCode {
    public void genererCode();
}
```

```
public class GenerateCodeJava implements IStratgyGenerateCode {
    @Log
    @Override
    public void genererCode() {
        System.out.println("Générer le code JAVA");
    }
}
```

```
public class GenerateCodeCsharp implements IStratgyGenerateCode {
    @Log
    @Override
    public void genererCode() {
        System.out.println("Generate code Csharp");
    }
}
```

```
public interface IAncienGenerateCode {
    void genererCode();
}
```

```
public class AncienImpGenerateCode implements IAncienGenerateCode {
    @Log
    public void genererCode() {
```



```
System.out.println("Ancienne Implmentation Génération du code");  
}  
}
```

```
public interface Observable {  
    public void addObserver(Observer observer);  
    public void removeObserver(Observer observer);  
    public void notifyObservers();  
}
```

```
public interface Observer {  
    public void update(Observable o);  
}
```

La partie entities :

```
public class Annotation extends Entite{  
  
    private List<Propriete> proprietes = new ArrayList<>();  
  
    public Annotation(String name) {  
        super(name);  
    }  
  
    @Log  
    public List<Propriete> getProperties() {  
        return proprietes;  
    }  
  
    public void setProperties(List<Propriete> properties) {  
        this.proprietes = properties;  
    }  
  
    @Log  
    public void addProperty(Propriete property) {  
        proprietes.add(property);  
    }  
  
    @Log  
    public void removeProperty(Propriete property) {  
        proprietes.remove(property);  
    }  
}
```

```
public abstract class Association extends Relation{  
    private Classe entiteSrc;  
    private Classe entiteDst;  
    @Log  
    public Classe getEntiteSrc() {  
        return entiteSrc;  
    }  
}
```

```
@Log
public void setEntiteSrc(Classe entiteSrc) {
    this.entiteSrc = entiteSrc;
}

@Log
public Classe getEntiteDst() {
    return entiteDst;
}

@Log
public void setEntiteDst(Classe entiteDst) {
    this.entiteDst = entiteDst;
}

public Association(String name) {
    super(name);
}

public Association(Classe entiteSrc, Classe entiteDst) {
    super("Association");
    this.entiteSrc = entiteSrc;
    this.entiteDst = entiteDst;
}
}
```

```
public class AssociationBidirectionnelle extends Association{
    public AssociationBidirectionnelle(String name) {
        super(name);
    }
}
```

```
public class AssociationUnidirectionnelle extends Association {
    public AssociationUnidirectionnelle(String name) {
        super(name);
    }
}
```

```
public class Attribut {
    private String nom;
    private Type type;
    private Visibilite visibilite;
    private boolean isStatique;
    private boolean isFinal;

    public Attribut() {
    }

    public Attribut(String nom,
                    Type type,
                    Visibilite visibilite,
                    boolean isStatique,
                    boolean isFinal) {
        this.nom = nom;
        this.type = type;
    }
}
```

```
        this.visibilite = visibilite;
        this.isStatique = isStatique;
        this.isFinal = isFinal;
    }

    @Log
    public String getNom() {
        return nom;
    }

    @Log
    public void setNom(String nom) {
        this.nom = nom;
    }

    @Log
    public Type getType() {
        return type;
    }

    @Log
    public void setType(Type type) {
        this.type = type;
    }

    @Log
    public Visibilite getVisibilite() {
        return visibilite;
    }

    @Log
    public void setVisibilite(Visibilite visibilite) {
        this.visibilite = visibilite;
    }

    @Log
    public boolean isStatique() {
        return isStatique;
    }

    @Log
    public void setStatique(boolean statique) {
        this.isStatique = statique;
    }

    @Log
    public boolean isFinal() {
        return isFinal;
    }

    @Log
    public void setFinal(boolean aFinal) {
        this.isFinal = aFinal;
    }
}
```

```
public enum Cardinalite {  
    ZERO,  
    UN,  
    PLUSIEURS  
}
```

```
public class Classe extends Entite implements Observable {  
    private List<Attribut> attributs = new ArrayList<>();  
    private List<Methode> methodes = new ArrayList<>();  
    private List<Constructeur> constructeurs = new ArrayList<>();  
    private ClassVisibilite visibilite;  
    private Cardinalite[] cardinalites = new Cardinalite[2];  
    private List<Observer> observers = new ArrayList<>();  
  
    public Classe(String nom) {  
        super(nom);  
    }  
    public Classe(String nom,  
        List<Attribut> attributs,  
        List<Methode> methodes,  
        List<Constructeur> constructeurs,  
        List<Observer> observers,  
        ClassVisibilite visibilite) {  
        super(nom);  
        this.attributs = attributs;  
        this.methodes = methodes;  
        this.constructeurs = constructeurs;  
        this.observers = observers;  
        this.visibilite = visibilite;  
    }  
  
    @Override  
    @Log  
    public void addObserver(Observer observer) {  
        observers.add(observer);  
    }  
  
    @Log  
    @Override  
    public void removeObserver(Observer observer) {  
        observers.remove(observer);  
    }  
  
    @Log  
    @Override  
    public void notifyObservers() {  
        for(Observer o:observers){  
            o.update(this);  
        }  
    }  
}
```

```
public enum ClassVisibilite {  
    PUBLIC,  
    ABSTRACT,  
    STATIC,
```

```
    FINAL  
}
```

```
public class Constructeur {  
}
```

```
@Component  
public class DiagrammeDeClasse {  
    private IStratgyGenerateCode generateCode;  
    private IStrategyExport Export;  
    private List<Entite> entites = new ArrayList<>();  
    private List<Methode> methodes= new ArrayList<>();  
  
    @Log  
    public void setGenerateCode(IStratgyGenerateCode generateCode) {  
        this.generateCode = generateCode;  
    }  
  
    @Log  
    public void setExport(IStrategyExport export) {  
        this.Export = export;  
    }  
    @Log  
    public void genererCode(){  
        generateCode.genererCode();  
    }  
    @Log  
    public void exporterDiagramme(){  
        Export.exporter();  
    }  
}
```

```
@Component  
public class Entite implements Observer {  
    protected String nom;  
    protected int level;  
  
    public Entite(String nom) {  
        this.nom = nom;  
    }  
  
    @Log  
    @Override  
    public void update(Observable o) {  
    }  
}
```

```
public class Enumerateur extends Entite{  
  
    private List<Valeur> valeurs = new ArrayList<>();  
  
    public Enumerateur(String nom) {
```

```
        super(nom);  
    }  
}
```

```
public class GroupeClasse extends Entite{  
    private List<Entite> children = new ArrayList<>();  
  
    public GroupeClasse(String nom) {  
        super(nom);  
    }  
  
    public Entite add(Entite c){  
        c.level=this.level+1;  
        children.add(c);  
        return c;  
    }  
}
```

```
public class Heritage extends Relation {  
    public Heritage(String name) {  
        super(name);  
    }  
}
```

```
public class Implementation extends Relation{  
    public Implementation(String name) {  
        super(name);  
    }  
}
```

```
public class Interface extends Entite{  
    List<MethodeAbstraite> methodeAbstraite = new ArrayList<>();  
    public Interface(String nom) {  
        super(nom);  
    }  
}
```

```
public class Methode {  
    private String typeRetour;  
    private Visibilite visibilite;  
    private String nom;  
    private List<Parametre> parametres = new ArrayList<>();  
    private boolean isAbstract;  
    private boolean isStatic;  
    private boolean isFinal;  
  
    @Log  
    public boolean isAbstract() {  
        return isAbstract;  
    }  
  
    @Log  
    public void setAbstract(boolean anAbstract) {
```

```
        isAbstract = anAbstract;
    }

    @Log
    public boolean isStatic() {
        return isStatic;
    }

    @Log
    public void setStatic(boolean aStatic) {
        isStatic = aStatic;
    }

    @Log
    public boolean isFinal() {
        return isFinal;
    }

    @Log
    public void setFinal(boolean aFinal) {
        isFinal = aFinal;
    }

    @Log
    public List<Parametre> getParametres() {
        return parametres;
    }

    @Log
    public void setParametres(List<Parametre> parametres) {
        this.parametres = parametres;
    }

    @Log
    public String getTypeRetour() {
        return typeRetour;
    }

    @Log
    public void setTypeRetour(String typeRetour) {
        this.typeRetour = typeRetour;
    }

    @Log
    public Visibilite getVisibilite() {
        return visibilite;
    }

    @Log
    public void setVisibilite(Visibilite visibilite) {
        this.visibilite = visibilite;
    }

    @Log
    public String getNom() {
        return nom;
    }
}
```

```
@Log
public void setNom(String nom) {
    this.nom = nom;
}
}
```

```
public class MethodeAbstraite extends Methode{
}
```

```
public class MethodeConcrete extends Methode{
}
```

```
public class ObservableImpl1 implements Observable {
    private List<Observer> observers;

    @Log
    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Log
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Log
    @Override
    public void notifyObservers() {
        for(Observer o:observers){
            o.update(this);
        }
    }
}
```

```
public class ObserverImpl1 implements Observer {

    @Log
    @Override
    public void update(Observable o) {
    }
}
```

```
public class Parametre {
    private String nom;
    private Type type;

    public Parametre() {
    }
    public Parametre(String nom, Type type) {
        this.nom = nom;
    }
}
```



```
        this.type = type;
    }

    @Log
    public String getNom() {
        return nom;
    }

    @Log
    public void setNom(String nom) {
        this.nom = nom;
    }

    @Log
    public Type getType() {
        return type;
    }

    @Log
    public void setType(Type type) {
        this.type = type;
    }
}
```

```
public class Propriete extends Entite {

    private String type;
    private String valeur;

    @Log
    public String getType() {
        return type;
    }

    @Log
    public void setType(String type) {
        this.type = type;
    }

    @Log
    public String getValeur() {
        return valeur;
    }

    @Log
    public void setValeur(String valeur) {
        this.valeur = valeur;
    }

    public Propriete(String nom) {
        super(nom);
    }

    public Propriete(String nom, String type, String valeur) {
        super(nom);
        this.type = type;
        this.valeur = valeur;
    }
}
```

```
}  
}
```

```
public class Record extends Entite {  
    public Record(String nom) {  
        super(nom);  
    }  
}
```

```
public abstract class Relation {  
    protected String name;  
    public Relation(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
public enum Type {  
    STRING,  
    INT,  
    FLOAT,  
    DOUBLE,  
    BOOLEAN,  
    CHAR,  
    LONG,  
    SHORT,  
    VOID  
}
```

```
public class Valeur {  
    private String name;  
  
    public Valeur() {  
    }  
    public Valeur(String name) {  
        this.name = name;  
    }  
    @Log  
    public String getName() {  
        return name;  
    }  
    @Log  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

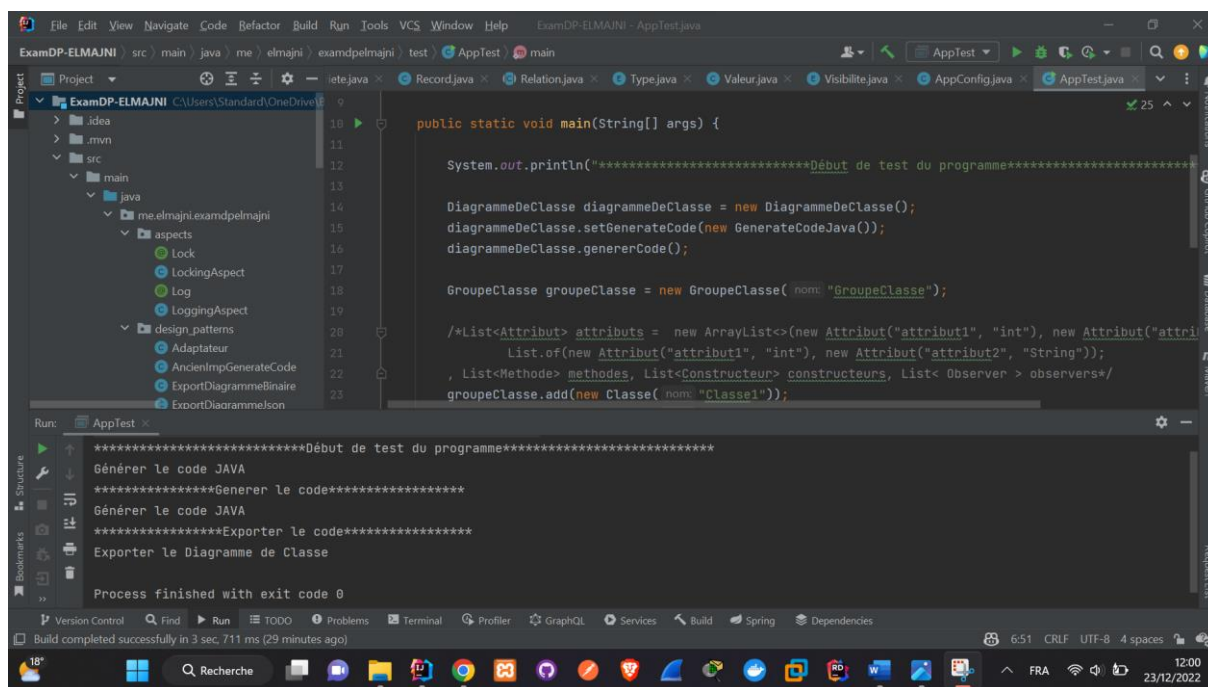
```
public enum Visibilite {  
    PUBLIC,  
    PRIVATE,  
    PROTECTED,  
    DEFAULT  
}
```

3. Créer une application de Test du Framework en choisissant un use case :

Configuration des aspects :

```
@Configuration  
@ComponentScan(value = {"me.elmajni"})  
public class AppConfig {  
}
```

```
public class AppTest {  
  
    public static void main(String[] args) {  
  
        System.out.println("*****Début de test du  
programme*****");  
  
        DiagrammeDeClasse diagrammeDeClasse = new DiagrammeDeClasse();  
        diagrammeDeClasse.setGenerateCode(new GenerateCodeJava());  
        diagrammeDeClasse.genererCode();  
  
        GroupeClasse groupeClasse = new GroupeClasse("GroupeClasse");  
  
        /*List<Attribut> attributs = new ArrayList<>(new  
Attribut("attribut1", "int"), new Attribut("attribut2", "String"));  
        List.of(new Attribut("attribut1", "int"), new  
Attribut("attribut2", "String"));  
        , List<Methode> methodes, List<Constructeur> constructeurs, List<  
Observer > observers*/  
        groupeClasse.add(new Classe("Classe1"));  
        groupeClasse.add(new Classe("Classe3"));  
        groupeClasse.add(new Classe("Classe2"));  
        groupeClasse.add(new Annotation("annotation1"));  
  
        System.out.println("*****Generer le  
code*****");  
        diagrammeDeClasse.setGenerateCode(new GenerateCodeJava());  
        diagrammeDeClasse.genererCode();  
  
        System.out.println("*****Exporter le  
code*****");  
        diagrammeDeClasse.setExport(new ExportDiagrammeBinaire());  
        diagrammeDeClasse.exporterDiagramme();  
  
        GroupeClasse groupeClasse1 = new GroupeClasse("GroupeClasse");  
        groupeClasse1.add(new Classe("Classe1"));  
        groupeClasse1.add(new Annotation("annotation1"));  
  
    }  
}
```



```
public static void main(String[] args) {  
  
    System.out.println("*****Début de test du programme*****");  
  
    DiagrammeDeClasse diagrammeDeClasse = new DiagrammeDeClasse();  
    diagrammeDeClasse.setGenerateCode(new GenerateCodeJava());  
    diagrammeDeClasse.genererCode();  
  
    GroupeClasse groupeClasse = new GroupeClasse( nom: "GroupeClasse");  
  
    /*List<Attribut> attributs = new ArrayList<>(new Attribut("attribut1", "int"), new Attribut("attribut2", "String"));  
    List.of(new Attribut("attribut1", "int"), new Attribut("attribut2", "String"));  
    , List<Methode> methodes, List<Constructeur> constructeurs, List<Observer> observers*/  
    groupeClasse.add(new Classe( nom: "Classe1"));  
}
```

Run: AppTest

```
*****Début de test du programme*****  
Générer le code JAVA  
*****Generer le code*****  
Générer le code JAVA  
*****Exporter le code*****  
Exporter le Diagramme de Classe  
Process finished with exit code 0
```

Lien du repository au github :

<https://github.com/KhaoulaElmajni/Examen-Design-Pattern-et-Programmation-Orientee-Aspect>