Manuel Programmeur

Rétro Conception Java-UML.

SAE 3.01 Equipe 1

Équipe :

- Matéo CHEVEAU F
- Ethan DAMESTOY F
- Joshua HERMILLY G
- Lucas LAFOSSE G
- Jonathan LECLERC F

Table des Matières

1. De la lecture de .java à leurs création en éléments de classe..... 5
1.1 Principe architectural.....5
1.1.1 Emplacement..... 5
1.1.2 Diagramme UML..... 5
1.2 Le Chef d'Orchestre : Lecture.java.....5
1.2.1 Rôle général..... 6
1.2.2 Méthode principale : lireFichier(String fichier).....6
Étape 1 – Formatage des chaînes de caractères.....6
Étape 2 – Gestion de l'imbrication et filtrage.....6
Etape 3 – Construction des instructions.....6
1.2.4 Méthodes de gestion du contexte..... 6
1.3. L'Usine : Fabrique.java.....7
1.3.1 Rôle général..... 7
1.3.2 Moment et contexte d'appel..... 7
1.3.3 Principe d'appel depuis Lecture..... 7
1.3.4 Responsabilités principales de Fabrique.....7
1.4. Le Nettoyeur : FormateurLigne.java.....8
1.4.1 Rôle général..... 8
1.4.2 Comment elle est utilisée -..... 8
1.4.3 Gestion des commentaires.....8
1.4.4 Formatage des types complexes et chaînes final..... 8
1.4.5 Principe de fonctionnement global..... 8
1.5. Schéma des étapes :.....9
1.5.1 - Le code avant lecture :.....9
1.5.2. Le code après lecture :.....9
2. Exportation et importation de XML du modèle UML.....10
2.1 Principe architectural.....10
2.1.1 Emplacement..... 10
2.1.2 Diagramme UML..... 10
2.2 Organisation générale du document XML.....10
2.3 Représentation des classes..... 11
2.3.1 Classes fantômes (<CLASSE_FANTOME>).....11
2.3.2 Classes complètes (<CLASSE>).....11
2.3.3 Interfaces implémentées (<IMPLEMENTS>).....11
2.4 Attributs de classe.....12
2.4.1 Multiplicité des attributs..... 12
2.5 Méthodes et constructeurs.....12
2.5.1 Paramètres des méthodes..... 13
2.6 Choix architecturaux et prévention des liaisons biaisées.....13
3. Nos objets UML côté métier..... 13
3.1 Les objets UML..... 13
3.1.1 Emplacement..... 13
3.1.2 Diagramme UML..... 13

3.2 Package metier.enums....14
3.2.1 Principe architectural....14
3.2.1.1 Emplacement....14
3.2.1.2 Diagramme UML....14
3.2.2 Fonctionnalités du package....15
3.2.3 Comment sont stockés les ENUMS....15
3.2.3.1 Stocker dans : Classe....15
3.2.3.2 Stocker dans : Attribut....16
3.2.3.3 Stocker dans : Methode....16
3.2.3.4 Stocker dans : Association....17
4. Gestion des associations dans l'application....17
4.1 Principe architectural....17
4.1.1 Emplacement....17
4.1.2 Diagramme UML....18
4.2 Principe général....18
4.3 Détection des associations....18
4.4 Exploitation des associations....18
4.5 Choix de conception....19
5. Classes Utilitaires....19
5.1 Principe architectural....19
5.1.1 Emplacement....19
5.1.2 Diagramme UML....19
5.2 CouleurUtils....20
5.3 FileChooserUtils....20
5.4 ErrorUtils....20
6. IHM - CUI : Package src.vue.CUI....20
6.1 Principe architectural....20
6.1.1 Emplacement....20
6.1.2 Diagramme UML....21
6.2 Rendu UML (Méthode afficherClasse)....21
6.3 Formatage et Style (Méthodes format...)....22
6.4 Interactions Entrée/Sortie....22
7. Panel principaux qui gèrent l''application....22
7.1 Principe architectural....22
7.1.1 Emplacement....22
7.1.2 Diagramme UML des classes utilitaires....22
7.2 Le Sommaire Visuel : (PanelArboressence)....23
7.3 Dessiner les classes : (PanelSchema)....23
7.3.1 Le placement dynamique....23
7.3.3 La gestion des Flèches (Complexe)....23
8. GererSouris.java....24
8.1 Principe architectural....24
8.1.1 Emplacement....24
8.1.2 Diagramme UML....24

8.2 Le role de GererSouris.java..... 24
9. Interface d''Édition UML..... 25
9.1 Principe architectural..... 25
9.1.1 Emplacement..... 25
9.1.2 Diagramme UML..... 25
9.2 Édition des Attributs..... 25
9.2.1 Accès..... 25
9.2.2 Conditions..... 25
9.2.3 Propriétés modifiables..... 26
9.2.4 Fonctionnement..... 26
9.2.5 Validation..... 26
9.3 Édition des Associations..... 26
9.3.1 Accès..... 26
9.3.2 Conditions..... 26
9.3.3 Propriétés modifiables..... 26
9.3.4 Fonctionnement..... 26
9.3.5 Validation..... 26
9.4 Mécanisme général..... 27
9.4.1 Après chaque modification réussie..... 27
9.4.2 En cas d'erreur..... 27
9.5 Points importants..... 27
10. Afficher et Dessiner les Classes UML choisit..... 27
10.1 Principe architectural..... 27
10.1.1 Emplacement..... 27
10.1.2 Diagramme UML..... 28
10.2 DessinerClasse.java..... 28
10.3 DimensionsCalculateur.java..... 28
10.4 FormatteurUML.java..... 29
10.4.1 Méthodes utilitaires :..... 29
11. Sélection des fichiers : PanelChargement..... 29
11.1 Principe architectural..... 29
11.1.1 Emplacement..... 29
11.1.2 Diagramme UML..... 29
11.2 Fonctionnalités..... 30
12. Affichage des flèches..... 30
12.1 Principe Architectural..... 30
12.1.1 Emplacement..... 30
12.2 Les constantes de Fleche.java creer dans son bloc static..... 30
12.3 Attributs d'une flèche..... 31
12.4 Construction d'une flèche..... 31
12.5 Méthodes essentielles..... 32
13. Précision Couleur Uml..... 33
14. Annexe Arborescence du projet..... 34

1. De la lecture de .java à leurs création en éléments de classe

1.1 Principe architectural

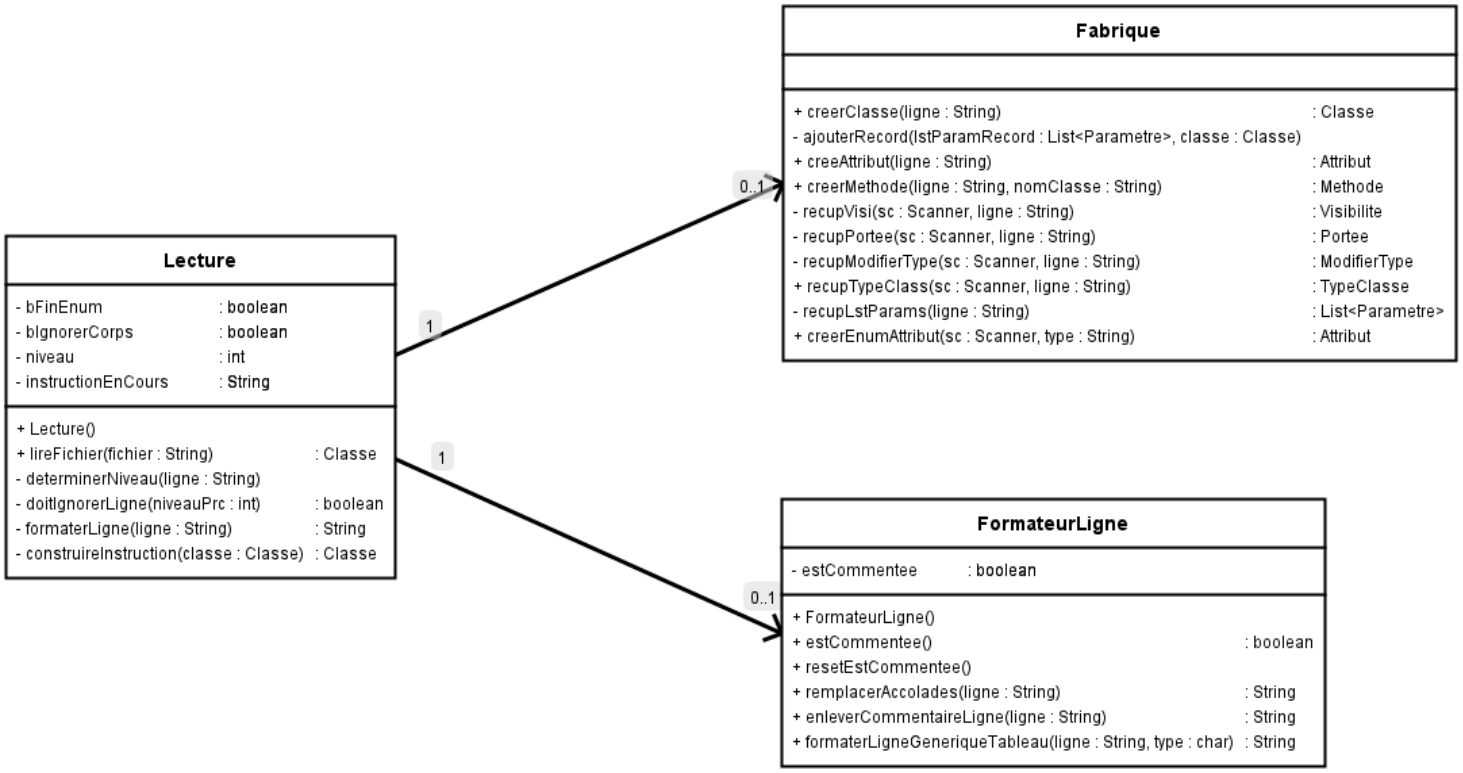
L'architecture repose sur 3 rôles séparé en 3 classe :

- Le Chef d'Orchestre (Lecture.java) : gère le déroulement global
- Le Nettoyeur (FormateurLigne.java) : supprimer les caractères inutiles .
- L'Usine (Fabrique.java) : fabrique les objets du modèle UML

1.1.1 Emplacement

```
src/
└── metier/
    └── lecture/
        ├── Fabrique.java
        ├── FormateurLigne.java
        └── Lecture.java
```

1.1.2 Diagramme UML



1.2 Le Chef d'Orchestre : Lecture.java

1.2.1 Rôle général

La classe Lecture est le point d'entrée de l'analyse. Elle orchestre l'ensemble du processus:

- lecture du fichier source
- gestion du niveau d'imbrication du code (niveau d'accolade)
- construction progressive des instructions
- délégation du nettoyage et de la création des objets UML

1.2.2 Méthode principale : lireFichier(String fichier)

Cette méthode constitue le cœur du programme. Elle parcourt le fichier ligne par ligne et applique des étapes dans un ordre précis.

Étape 1 – Formatage des chaînes de caractères

Pour l'intégrité du calcul du niveau d'imbrication, les caractères susceptibles de poser problème comme : {}, /*, */, et //; sont temporairement masqués lorsqu'ils apparaissent dans les chaînes de constantes finales. Le rôle du formateur, assuré par la classe FormateurLigne.

Étape 2 – Gestion de l'imbrication et filtrage

Le niveau précédent est mémorisé (niveauPrc = this.niveau) afin de déterminer si une ligne doit être ignorée (corps de méthode). Le niveau courant est mis à jour par caractère (this.niveau → this.determinerNiveau(ligne);), garantissant une gestion des accolades, même sur une seule ligne.

Etape 3 – Construction des instructions

Cette étape n'est exécutée que lorsque le niveau est inférieur ou égal à 1. Les lignes sont alors nettoyées, accumulées, alysées pour créer les objets UML. Le rôle du formateur, assuré par la classe FormateurLigne.

1.2.4 Méthodes de gestion du contexte

Méthode	Rôle
determinerNiveau(ligne);	Calcule le niveau d'imbrication via les accolades.
doitIgnorerLigne(niveauPrc);	Détermine si la ligne est dans une méthode.
formaterLigne(ligne);	Nettoie et filtre les parties inutiles ou la ligne.
construireInstruction(classe);	Interprète et déclenche la création UML.

1.3. L'Usine : Fabrique.java

1.3.1 Rôle général

La classe Fabrique joue le rôle d'usine de création des éléments du modèle UML.

Elle intervient après Lecture, une fois qu'une instruction Java complète a été lue, nettoyée et reconstruite. Son rôle est limité à l'interprétation d'une instruction isolée.

1.3.2 Moment et contexte d'appel

Les méthodes de Fabrique sont appelées exclusivement depuis la méthode construireInstruction(...) de la classe Lecture.

Cet appel intervient lorsque :

- le niveau d'imbrication est inférieur ou égal à 1
- une instruction Java est complète (classe, attribut, méthode…)
- La ligne a été nettoyée par FormateurLigne
- Lecture gère la responsabilité du quand, Fabrique celle du quoi créer.

1.3.3 Principe d'appel depuis Lecture

Lecture analyse le contenu de instructionEnCours et décide du type d'élément à créer.

L'ordre d'interprétation est strict :

1. Déclaration de classe / interface / enum / record → creerClasse (...)
2. Constantes d'énumération → creerEnumAttribut (...)
3. Méthodes → creerMethode (...)
4. Attributs → creeAttribut (...)

Fabrique retourne alors un objet UML prêt à être intégré au modèle.

1.3.4 Responsabilités principales de Fabrique

Fabrique est responsable de la création de l'ensemble des éléments UML :

Objet	Éléments pris en charge	Fonction de création/gestion
Classe	Classes (classes, interfaces, énumérations, records)	creerClasse(String ligne)
String // extend List<String> // lstImplement	Classe mère (extends) et classes implémentés (implements)	Gérées au sein de creerClasse(String ligne)
Attribut	Attributs (simples, statiques, constants, etc.)	creeAttribut(String ligne)
Methode	Méthodes (constructeurs,par défaut, normal)	creerMethode(String ligne,String nomClasse)
List<Parametre>	Paramètres de méthodes	recupLstParams(String ligne)

1.4. Le Nettoyeur : FormateurLigne.java

1.4.1 Rôle général

La classe FormateurLigne joue le rôle de nettoyeur syntaxique du programme.
Son unique responsabilité est de produire une ligne propre, exploitable.

1.4.2 Comment elle est utilisée -

FormateurLigne est utilisé exclusivement par la classe Lecture.

Elle intervient avant :

- le calcul du niveau d'imbrication
- la reconstruction des instructions
- tout appel à Fabrique

1.4.3 Gestion des commentaires

FormateurLigne gère l'ensemble des formes de commentaires Java (simplie "//" et multi-lignes "/* */"). Il y a aussi un attribut qui permet de savoir si on est dans un commentaire (boolean estCommentee).

1.4.4 Formatage des types complexes et chaînes final

FormateurLigne assure également le formatage syntaxique de certains types générique ou tableaux:

Les espaces inutiles sont supprimés afin d'obtenir une chaine homogène, par exemple :

- String [ ] tab → String[] tab
- List < Type > lst → List<Type> lst

On a aussi le remplacement de caractères pour les final String.
Ce formatage simplifie l'analyse réalisée par Fabrique car les types sont stocké en String.

1.4.5 Principe de fonctionnement global

Pour chaque ligne lue par Lecture, FormateurLigne peut successivement :

1. supprimer ou tronquer les commentaires
2. assurer des chaînes sans erreurs
3. normaliser les types génériques et tableaux

1.5. Schéma des étapes :

1.5.1 - Le code avant lecture :

```java
import java.util.List;

public class Controleur
{
    private Valderia metier;
    private IhmCui   ihm;


    public Controleur()
    {
        this.metier = new Valderia ();
        this.ihm    = new IhmCui   (this);
    }

    public void lancer()
    {
        int choix;

        choix = this.ihm.menu();

        while ( choix != 5 )
        {
            switch ( choix )
            {
                case 1 -> this.ihm.listerProvinces   ();
                case 2 -> this.ihm.listerTerritoires ('C');  // par code
                case 3 -> this.ihm.listerTerritoires ('P');  // par population
                case 4 -> this.metier.genererHtml    ();

            }

            choix = this.ihm.menu();
        }
    }


    public Province         getProvince        ( String codeProvince ) { return this.metier.getProvince ( codeProvince ); }
    public List<Territoire> getCopieTerritoires()                      { return this.metier.getCopieTerritoires();        }
    public List<Province>   getCopieProvinces  ()                      { return this.metier.getCopieProvinces  ();        }


    Run | Debug
    public static void main(String[] a)
    {
        Controleur ctrl = new Controleur();
        ctrl.lancer();
    }
}
```

1.5.2. Le code après lecture :

```
0 public class Valderia
1       private List<Territoire> lstTerritoires ;
1       private List<Province> lstProvinces ;
1       public Valderia  (  )
1       public Province getProvince  (  String codeProvince  )
1       public List<Territoire> getCopieTerritoires  (  )
1       public List<Province> getCopieProvinces  (  )
1       public void genererHtml  (  )
1       private void initProvince (  )
1       private void initTerritoire (  )
```

2. Exportation et importation de XML du modèle UML

Cette partie présente le format XML utilisé pour sauvegarder et restaurer le modèle UML construit par l'application.

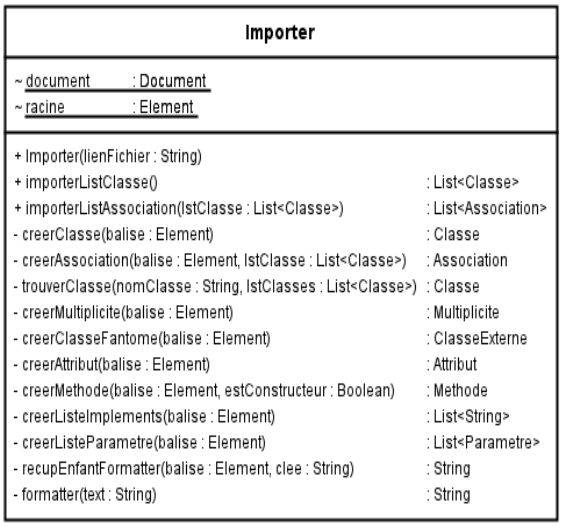
Il ne s'agit pas d'un export du code Java, mais d'une **représentation** du diagramme UML sous forme XML.

2.1 Principe architectural

2.1.1 Emplacement

```
src/
└── metier/
    └── import_export/
        ├── Exporter.java
        └── Importer.java
```

2.1.2 Diagramme UML



2.2 Organisation générale du document XML

Le document est structuré autour d'une racine <SAUVEGARDE>, permettant d'utiliser les méthodes fournies par la librairie **jdom-2.0.6.jar**.

2 grandes sections :

- les classes (classe du projet et externes)
- les relations (associations, traitées séparément)

Cette séparation évite toute dépendance à l'ordre de lecture et la possibilité d'utiliser des getters sur ces 2 balises enfants.

2.3 Représentation des classes

Toutes les classes du diagramme sont regroupées dans une section <CLASSES>.
Deux types de classes peuvent y apparaître : les classes complètes et les classes fantômes.

2.3.1 Classes fantômes (<CLASSE_FANTOME>)

Une classe fantôme représente une classe externe au projet (bibliothèque, dépendance, classe non analysée).
Elle n'a pas de contenu UML interne, mais doit exister pour permettre la représentation graphique et les associations.

Une classe fantôme contient :

- <nom> : nom de la classe externe
- <posX> et <posY> : position graphique sur le diagramme

Aucune information de structure (attributs, méthodes) n'est volontairement stockée.

2.3.2 Classes complètes (<CLASSE>)

Une classe complète correspond à une classe, interface, enum ou record réellement analysé.

Chaque classe est décrite par un ensemble cohérent de balises :

- <visibilite> : visibilité UML (public, protected, private)
- <modifierType> : modificateur de type (abstract, final, etc.)
- <type> : nature de la classe (class, interface, enum, record)
- <nom> : nom de la classe
- <posX> et <posY> : position graphique
- <extend> : classe mère éventuelle (chaîne vide si aucune)

2.3.3 Interfaces implémentées (<IMPLEMENTS>)

Lorsqu'une classe implémente une ou plusieurs interfaces, celles-ci sont listées dans une balise dédiée :

```
<IMPLEMENTS>
    <implement>NomInterface</implement>
</IMPLEMENTS>
```

Chaque interface est stockée indépendamment.

2.4 Attributs de classe

Les attributs sont regroupés dans une balise <ATTRIBUTS>.
Chaque attribut est décrit individuellement par une balise <attribut>.

Pour chaque attribut, sont sauvegardées les informations suivantes :

- <visibilite> : visibilité UML
- <portee> : instance ou classe (statique)
- <type> : type UML de l'attribut
- <nom> : nom
- <contrainte> : contrainte (final ou non)
- <valeur> : valeur éventuelle (simplifiée)

2.4.1 Multiplicité des attributs

Lorsqu'un attribut possède une multiplicité, celle-ci est stockée explicitement :

```
<multiplicite>
        <min>...</min>
        <max>...</max>
</multiplicite>
```

2.5 Méthodes et constructeurs

Les méthodes et constructeurs sont regroupés dans une section <METHODES>.

Deux types de balises peuvent apparaître :

- <methode> : pour les méthodes classiques
- <constructeur> : pour les constructeurs

Cela permet de faciliter la création des composants.

Chaque méthode ou constructeur contient :

- <visibilite> : visibilité UML
- <portee> : portée (instance ou classe)
- <type> : type de retour (vide pour un constructeur)
- <nom> : nom de la méthode
- <estDefault> : indique s'il s'agit d'une méthode default (interfaces)

2.5.1 Paramètres des méthodes

Les paramètres sont regroupés dans une balise <parametres>.

Chaque paramètre est décrit par :

- <type> : type du paramètre
- <nom> : nom du paramètre

Le corps des méthodes n'est jamais sauvegardé.

2.6 Choix architecturaux et prévention des liaisons biaisées

Le format XML a été conçu pour éviter toute liaison prématurée ou incohérente :

- les classes sont créées avant les relations
- les associations sont traitées séparément
- les références se font par nom, jamais par index ou position

3. Nos objets UML côté métier.

Nous gérons les Classes, Attributs, Méthodes … en utilisant des classes ou des records. Il est important de noter que les **classes qui ne sont pas de type Record** sont utilisées afin de **permettre la modification de leurs attributs.**

3.1 Les objets UML

3.1.1 Emplacement

```
src/
└── metier/
    └── classe/
        ├── Association.java
        ├── Attribut.java
        ├── Classe.java
        ├── ClasseExterne.java
        ├── Methode.java    ( record )
        ├── Multiplicite.java
        └── Parametre.java ( record )
```

3.1.2 Diagramme UML

Voir diagramme UML des classes dans : Diagramme UML des objets

3.2 Package metier.enums

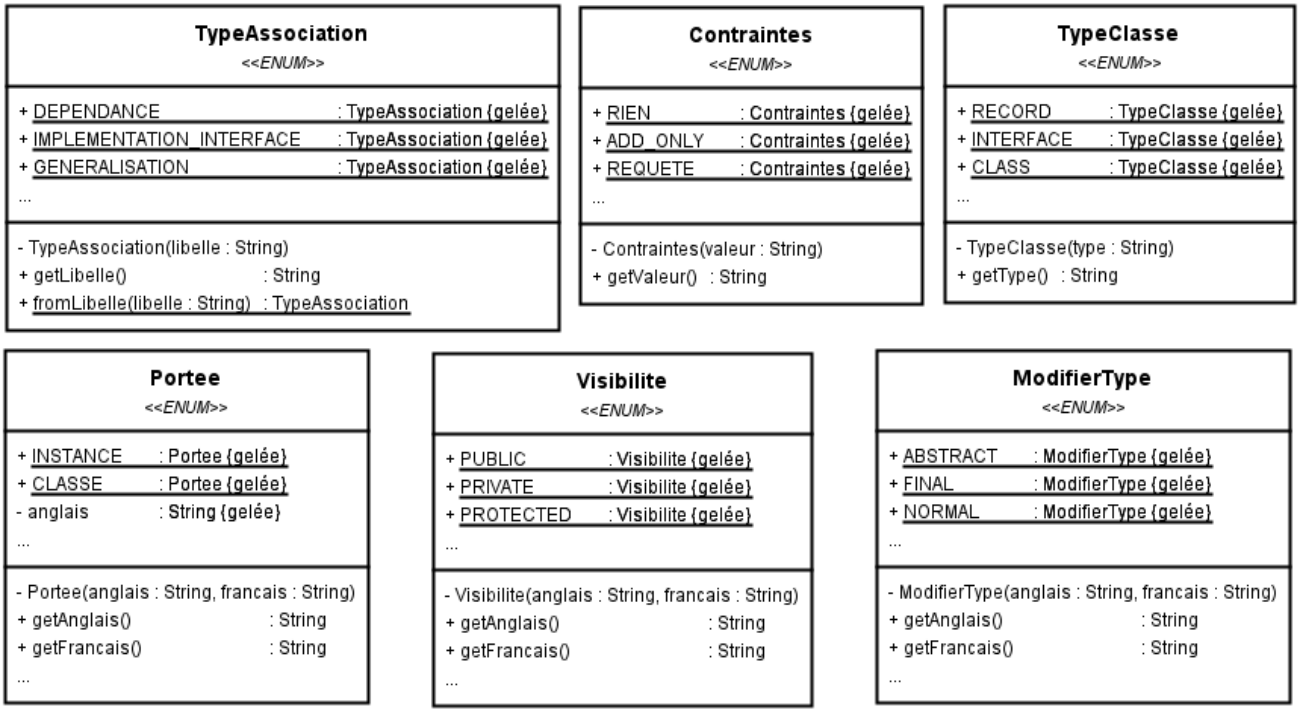
Ce package centralise toutes les définitions constantes et les types énumérés utilisés pour modéliser les concepts UML et Java. Il assure la cohérence des données et la traduction (Anglais et Français) dans l'application.

3.2.1 Principe architectural

3.2.1.1 Emplacement

```
src/
└── metier/
    └── enums/
        ├── Contraintes.java
        ├── ModifierType.java
        ├── Portee.java
        ├── TypeAssociation.java
        ├── TypeClasse.java
        └── Visibilite.java
```

3.2.1.2 Diagramme UML



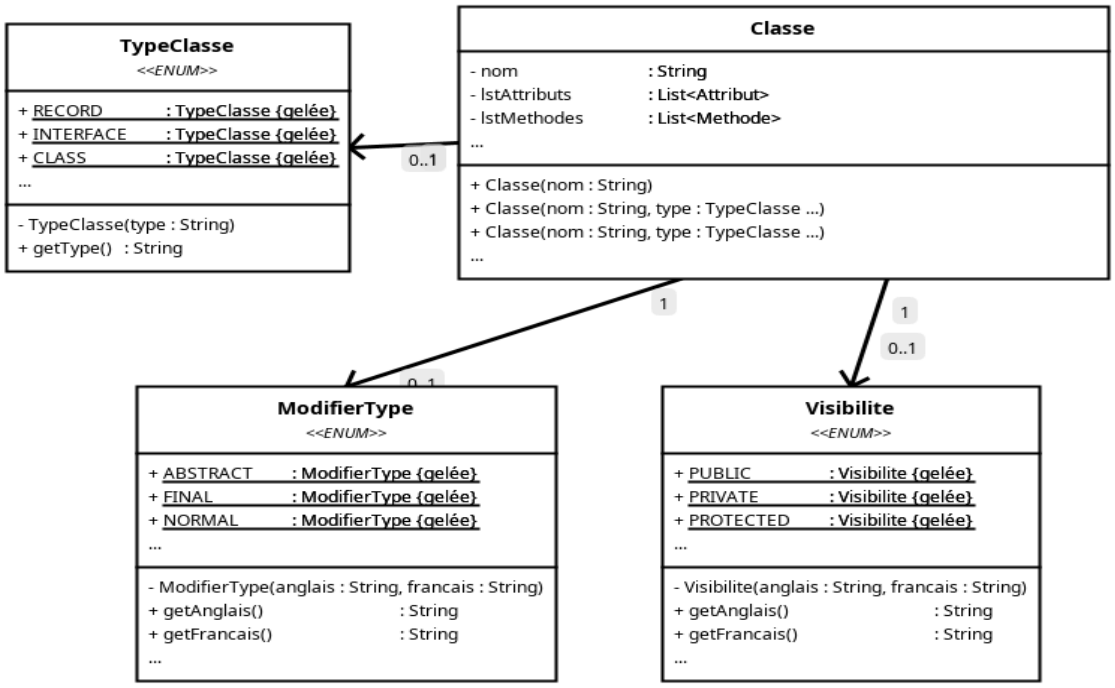
3.2.2 Fonctionnalités du package

Résumé des fonctionnalités par fichier :

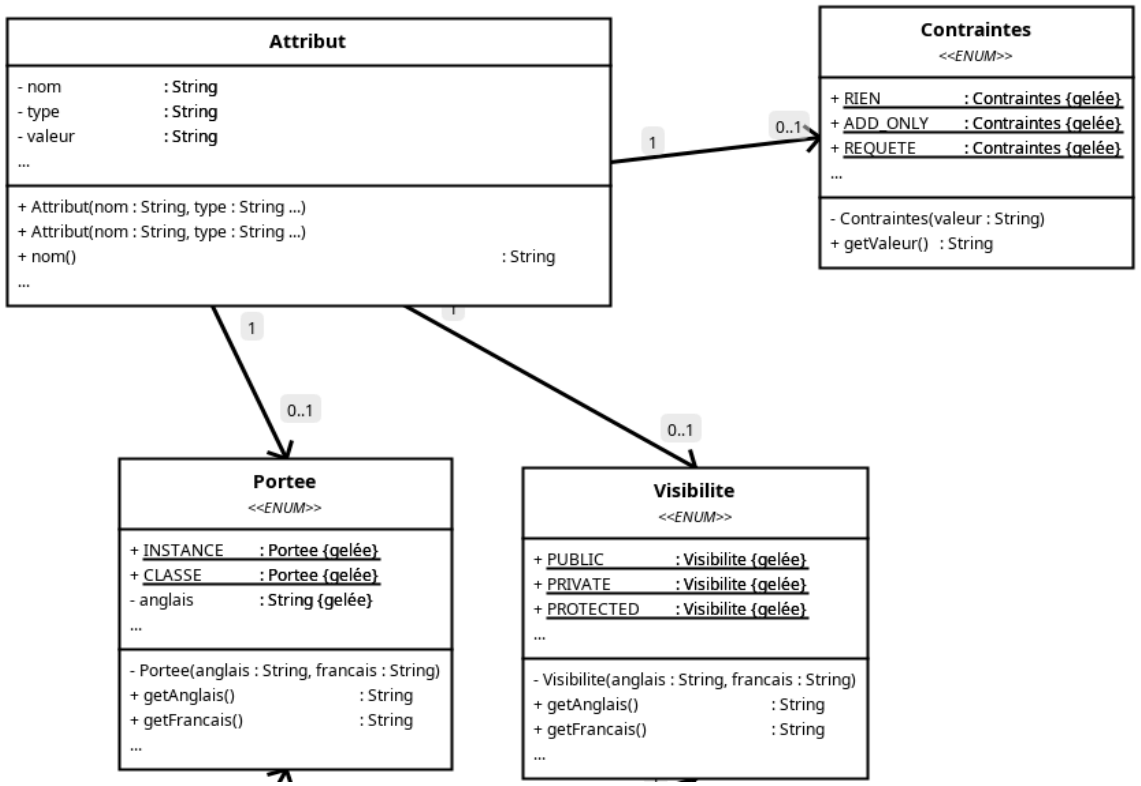
Classe	Rôle Principal	Valeurs Clés
Contraintes	Définit les restrictions applicables aux attributs UML.	{addOnly}, {requête}, {gelée}, RIEN
ModifierType	Gère les modificateurs de statut (classe/méthode) et leur traduction.	ABSTRACT,FINAL, NORMAL
Portee	Distingue l'appartenance d'un membre (Classe vs Instance).	CLASSE (static), INSTANCE
TypeAssociation	Liste les types de relations possibles entre les classes.	DEPENDANCE, HERITAGE, AGREGATION, COMPOSITION
TypeClasse	Enumère les structures Java supportées par l'analyseur.	CLASS, INTERFACE, ENUM, RECORD
Visibilite	Gère les niveaux d'encapsulation et la traduction des mots-clés.	PUBLIC (+), PRIVATE (-), PROTECTED (#), PACKAGE (~)

3.2.3 Comment sont stockés les ENUMS

3.2.3.1 Stocker dans : Classe

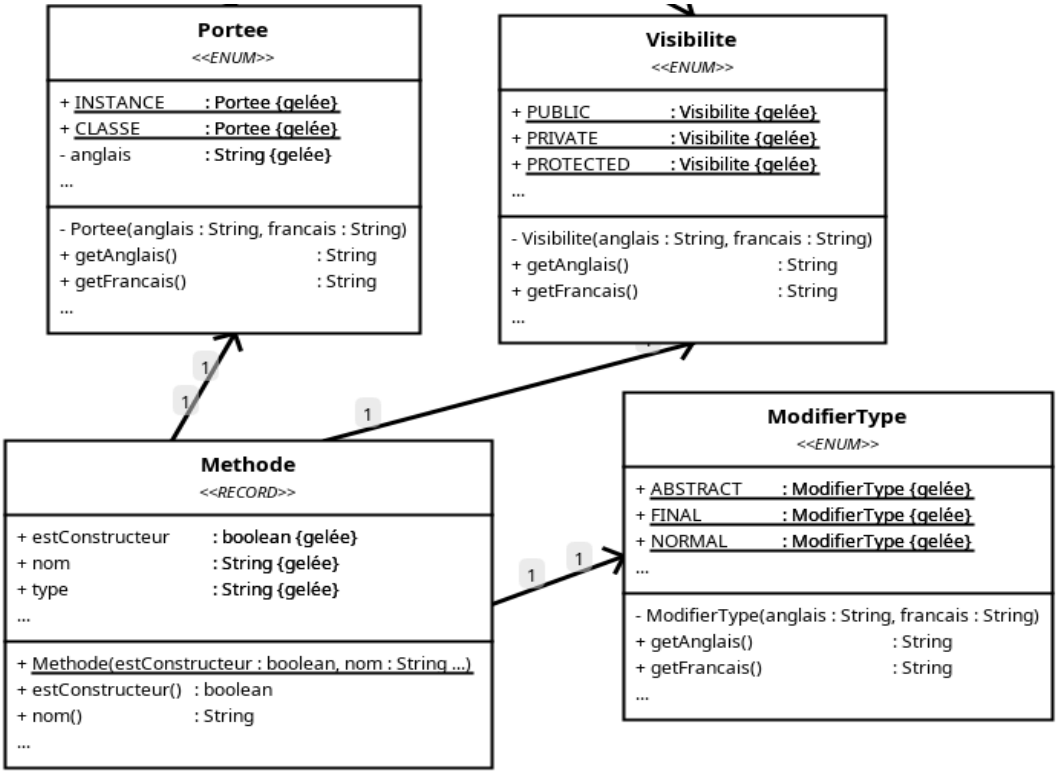


3.2.3.2 Stocker dans : Attribut

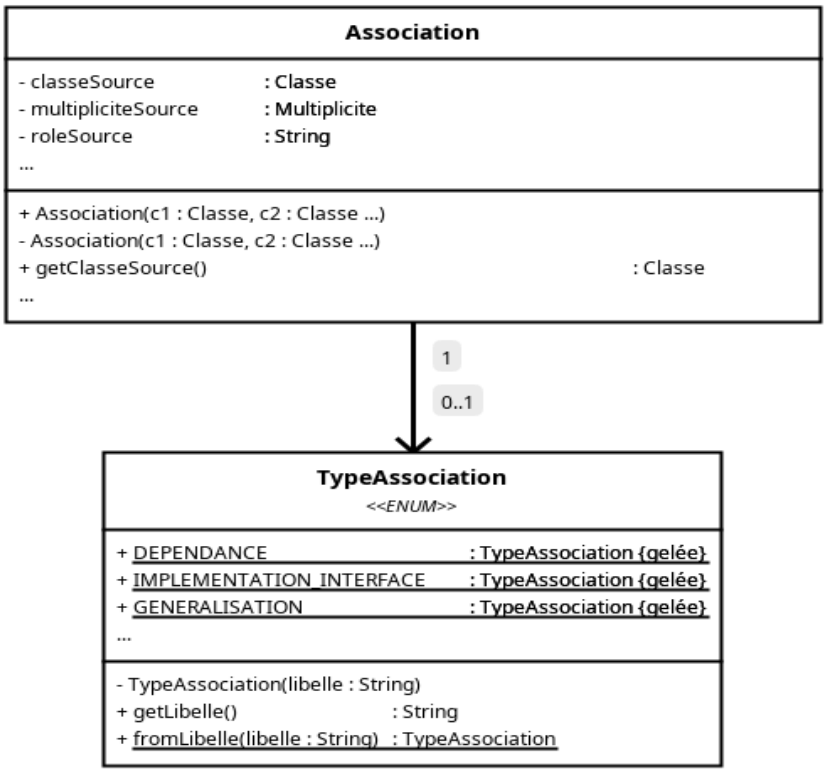


Il est important de noter que la classe << Attributs >> est une classe et non un record. Car pour pouvoir ajouter des contraintes et gérer les multiplicités, il est nécessaire de pouvoir éditer les attributs, ce qui n'est pas possible avec un record.

3.2.3.3 Stocker dans : Methode



3.2.3.4 Stocker dans : Association



4. Gestion des associations dans l'application

Cette partie explique la gestion des associations UML dans l'application. Elles représentent les relations entre les classes, permettant la traduction automatique des liens Java en UML.

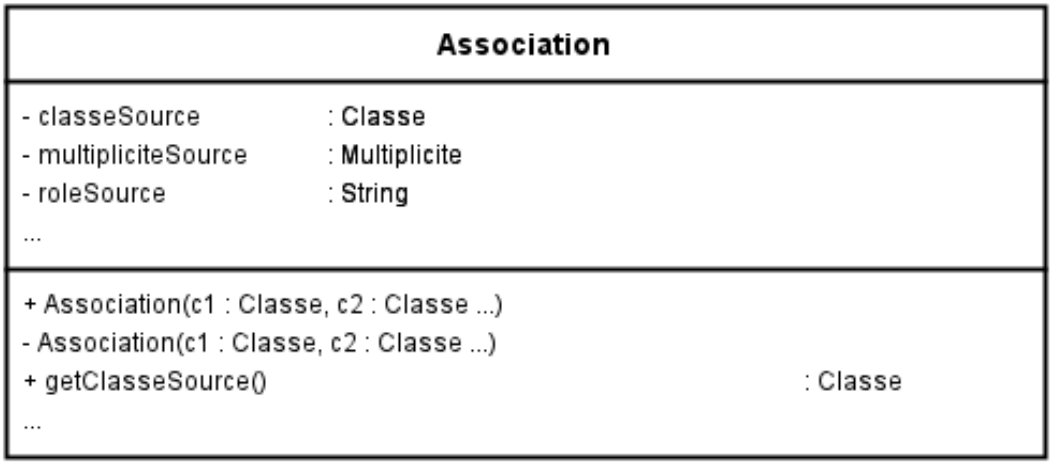
Le système repose sur une détection automatique des relations, suivie de leur stockage dans le modèle métier et de leur utilisation pour l'affichage et la sauvegarde du diagramme.

4.1 Principe architectural

4.1.1 Emplacement

```
src/
└── metier/
    └── classe/
        └── Association.java
```

4.1.2 Diagramme UML



4.2 Principe général

Une association modélise un lien logique entre deux classes.
Lorsqu'une classe utilise une autre classe, l'application identifie cette relation et crée automatiquement l'association UML correspondante.

L'application gère les principaux types d'associations UML : généralisation, implémentation d'interface, agrégation, composition, dépendance et association simple.

4.3 Détection des associations

La détection des associations est effectuée après la lecture des fichiers Java du projet.
Une fois les classes chargées, le contrôleur appelle la méthode statique detecterAssociations de la classe Association (package metier.classe).

Cette méthode parcourt l'ensemble des classes afin d'identifier les relations existantes en regardant les attributs d'une classe avec les autres classes de la liste et détecte :

- l'héritage (extends) génère une généralisation ;
- l'implémentation d'interfaces (implements) génère une implémentation ;
- les attributs typés par une autre classe génèrent des agrégations ou compositions ;

Les attributs de type collection permettent de déduire les multiplicités.

4.4 Exploitation des associations

Les associations détectées sont stockées dans des objets Association du modèle métier.
Elles sont ensuite utilisées pour l'affichage des flèches UML, la modification des relations par l'utilisateur et la sauvegarde du diagramme au format XML.

La détection est réalisée uniquement lors du chargement initial du projet afin de garantir de bonnes performances et la cohérence du modèle dans la méthode detecterAssociations(List<Classe> classes).

4.5 Choix de conception

La détection des associations est séparée de l'affichage graphique et de la persistance.
Ce découpage permet de limiter les dépendances, de simplifier la maintenance et d'assurer une évolution maîtrisée du système.

5. Classes Utilitaires

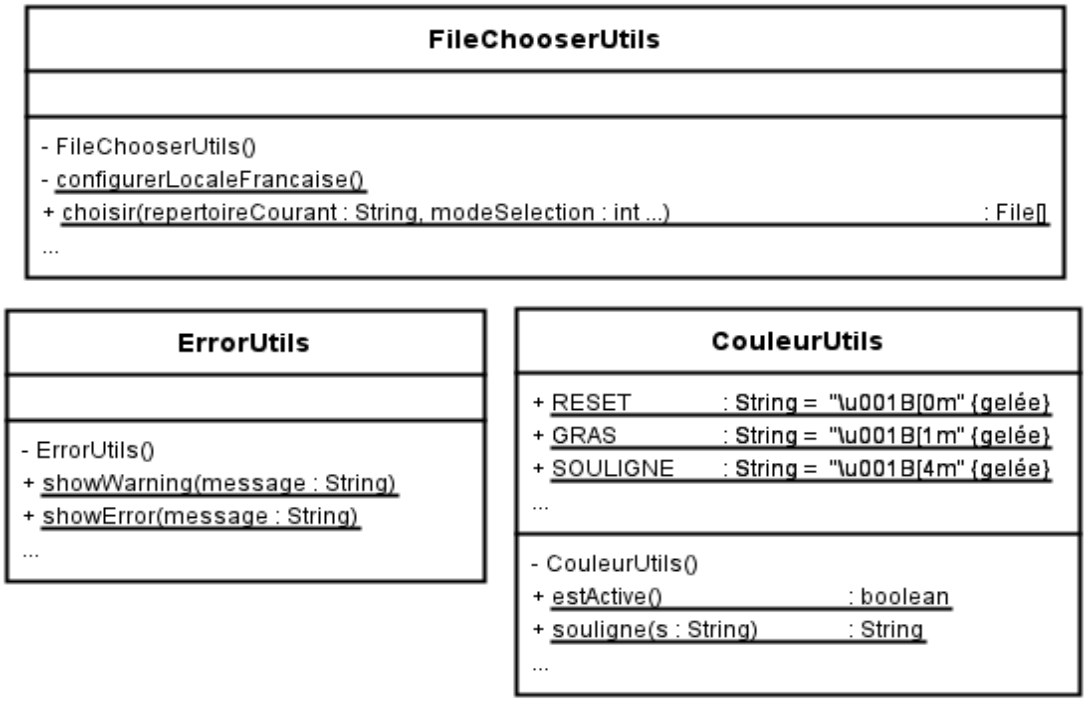
Cette partie explique les différentes classes utilitaires qui nous permettent de gérer les erreurs de l'utilisateur, les choix de fichier ou encore les couleurs. Leur rôle est d'éviter de répéter du code.

5.1 Principe architectural

5.1.1 Emplacement

```
src/
└── utils/
    ├── CouleurUtils.java
    ├── ErrorUtils.java
    └── FileChooserUtils.java
```

5.1.2 Diagramme UML



5.2 CouleurUtils

CouleurUtils gère les couleurs et styles ANSI strictement pour l'affichage en console utilisé par l'ihmCUI.

Elle contient les différents codes ANSI des couleurs pour paramétrer le style d'affichage.

A noter que la classe détecte si la coloration est possible (terminal compatible ou variable d'environnement). Si ce n'est pas le cas, elle retourne simplement des chaînes neutres, sans codes ANSI.

5.3 FileChooserUtils

FileChooserUtils centralise toute la logique liée à la sélection de fichiers et de dossiers via une interface graphique.

Elle est utilisée par les interfaces graphiques (chargement de fichiers, sauvegarde XML).

Elle évite que chaque panneau ou contrôleur reconfigure un sélecteur de fichiers différemment.

5.4 ErrorUtils

ErrorUtils est responsable de l'affichage des messages utilisateurs via des pop ups. Elle est utilisée uniquement par les vues, ne contient aucune logique métier et centralise l'affichage des messages (avertissement, erreur, succès et infos).

6. IHM - CUI : Package src.vue.CUI

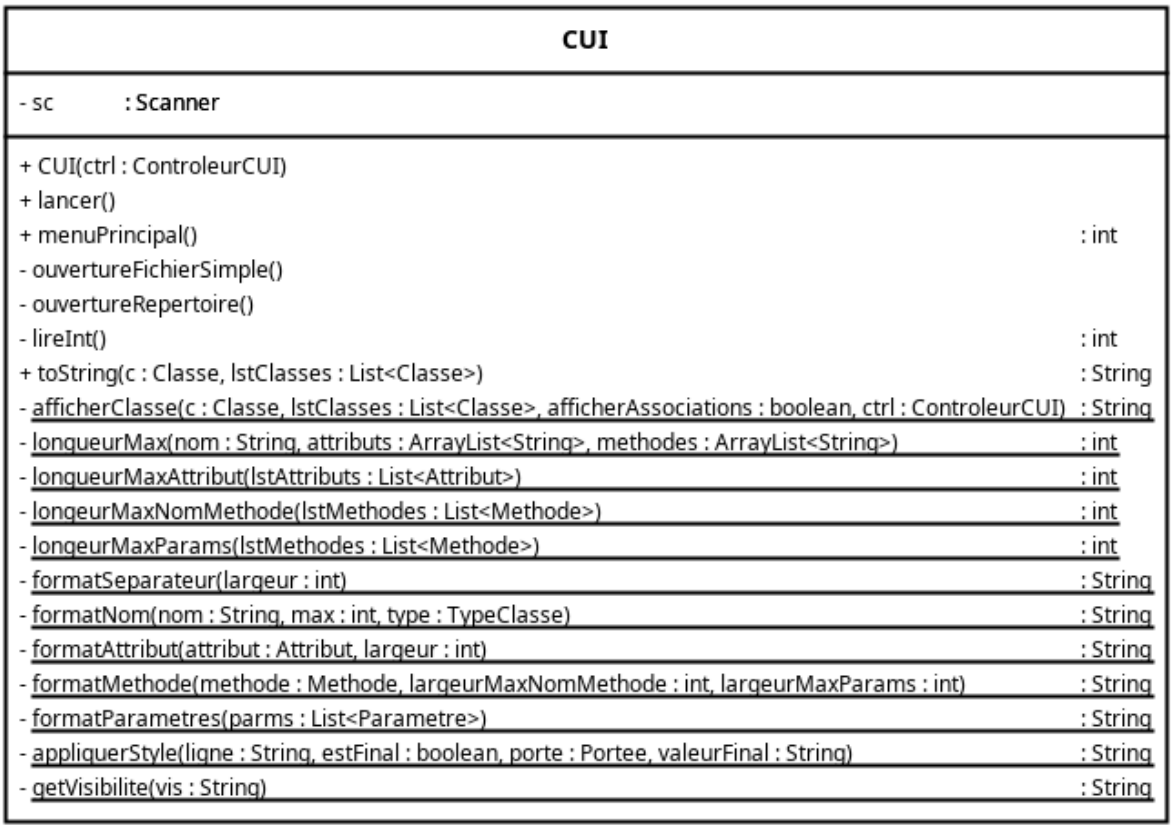
Rôle général : Vue textuelle (Console User Interface) de l'application. Cette classe est responsable de l'affichage d'un diagramme UML directement sur la console. Elle gère le menu de sélection de fichiers/dossiers et la transformation des objets métier en représentation graphique textuelle.

6.1 Principe architectural

6.1.1 Emplacement

```
src/
└── vue/
    └── CUI/
        └── CUI.java
```

6.1.2 Diagramme UML



6.2 Rendu UML (Méthode afficherClasse)

La méthode afficherClasse (appelée via toString des objets) construit dynamiquement l'UML en texte.

Algorithme de dessin :

1. **Calcul des dimensions :** Scanne tous les attributs et méthodes pour déterminer la largeur maximale nécessaire (longueurMax...).
2. **En-tête :** Affiche le nom de la classe centré. Le type (<<interface>> ou <<enum>>) est affiché en rouge au-dessus si pertinent.
3. **Corps :**
   - Trace les séparateurs (----------).
   - Liste les attributs formatés.
   - Liste les méthodes formatées.
4. **Pied de page (Associations) :** Affiche textuellement les relations (Associations, Extends, Implements) sous la boîte principale.

6.3 Formatage et Style (Méthodes format...)

La classe applique une couche de style stricte, utilisant CouleurUtils pour la lisibilité :

- **Visibilité :** Convertie en symboles colorés (+ Cyan, - Vert, # Noir/Gris).
- **Modificateurs Spéciaux :**
  - **Static (Portée Classe) :** Le texte est **souligné**.
  - **Final (Constante) :** Ajout du suffixe {gelé} et de la valeur d'initialisation.
- **Méthodes :** Formatage aligné de la signature : nom (param1 : type) : typeRetour.

6.4 Interactions Entrée/Sortie

- **Entrée :** Utilise java.util.Scanner pour lire les choix de menu et les chemins de fichiers.
- **Sortie :** Tout l'affichage passe par la sortie standard (System.out), avec les codes ANSI de CouleurUtils.

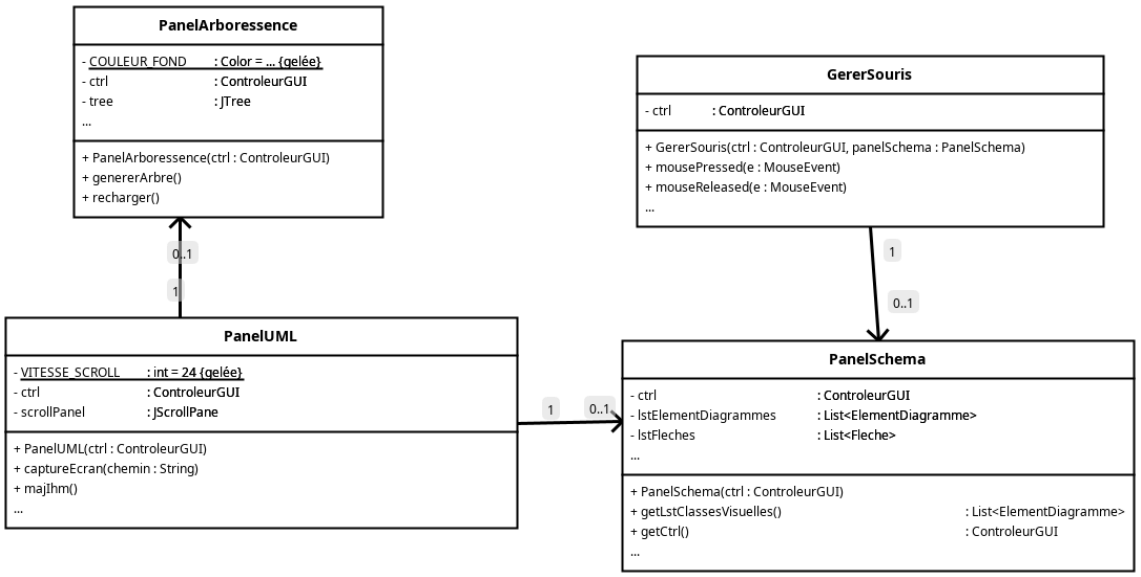
7. Panel principaux qui gèrent l'application

7.1 Principe architectural

7.1.1 Emplacement

```
src/
└── vue/
    └── GUI/
        └── panels.java
            ├── PanelArborescence.java
            ├── PanelSchema.java
            └── PanelUML.java
```

7.1.2 Diagramme UML des classes utilitaires



7.2 Le Sommaire Visuel : (PanelArboressence)

C'est la table des matières du projet, il affiche également le type de chaque fichier (Classe, Interface, Enum, Record) à l'aide d'un JavaTreeRenderer. Il ne prend pas en compte les classes Externe. Enfin il permet de sélectionner une classe à l'aide d'un TreeSelectionListener.

7.3 Dessiner les classes : (PanelSchema)

7.3.1 Le placement dynamique

Au chargement, si les classes n'ont pas de coordonnées fixes, la méthode placerClassesInitiales().

Elle les places selon la conditions suivantes :

```
if      (nbClasses <= 3)  nbColonnes = nbClasses; // Toutes sur une ligne
else if (nbClasses <= 8)  nbColonnes = 4;         // 4 colonnes pour 4-8 classes
else if (nbClasses <= 15) nbColonnes = 5;         // 5 colonnes pour 9-15 classes
else                      nbColonnes = 6;         // 6 colonnes pour plus de 15 classes
```

7.3.2 Le dessin (paintComponent)

À chaque rafraîchissement, mouvement et autres, le panneau redessine tout dans un ordre précis :

1. **Les classes** : Il dessine chaque classe (ElementDiagramme).
2. **Les liens** : Il dessine les flèches (Fleche) par-dessus.
3. **La sélection** : Si une classe est sélectionnée, elle est redessinée en dernier (au premier plan) avec un contour épais pour la mettre en valeur.

7.3.3 La gestion des Flèches (Complexe)

La méthode updatePositionsFleches() évite que toutes les flèches partent du même point.

- Elle regarde combien de liens relient la classe A et la classe B.
- Elle décale les points de départ et d'arrivée sur les bords des boîtes pour que les lignes soient parallèles et lisibles, sans se croiser au même endroit

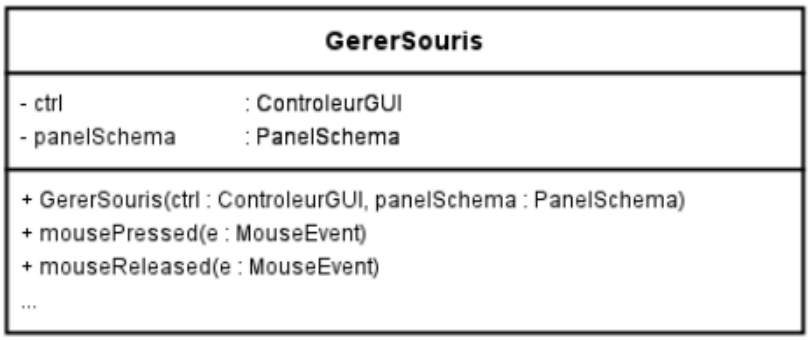
8. GererSouris.java

8.1 Principe architectural

8.1.1 Emplacement

```
src/
└─ vue/
   └─ GUI/
      └─ panels/
         └─ gererSouris/
            └─ GererSouris.java
```

8.1.2 Diagramme UML



8.2 Le role de GererSouris.java

Rôle : Contrôleur unique des interactions utilisateurs sur la zone de dessin (PanelSchema).

Type d'interaction	Action
Clic gauche	Sélectionne une classe ou ouvre l'édition d'une association (flèche).
Clic droit (maintenu)	Affiche temporairement la classe en mode "Complet".
Double-clic	Ouvre la fenêtre d'édition des attributs.
Glisser-Déposer	Déplace les classes (avec défilement automatique de la fenêtre).
Survol	Change le curseur (Main/Défaut) pour indiquer les éléments cliquables

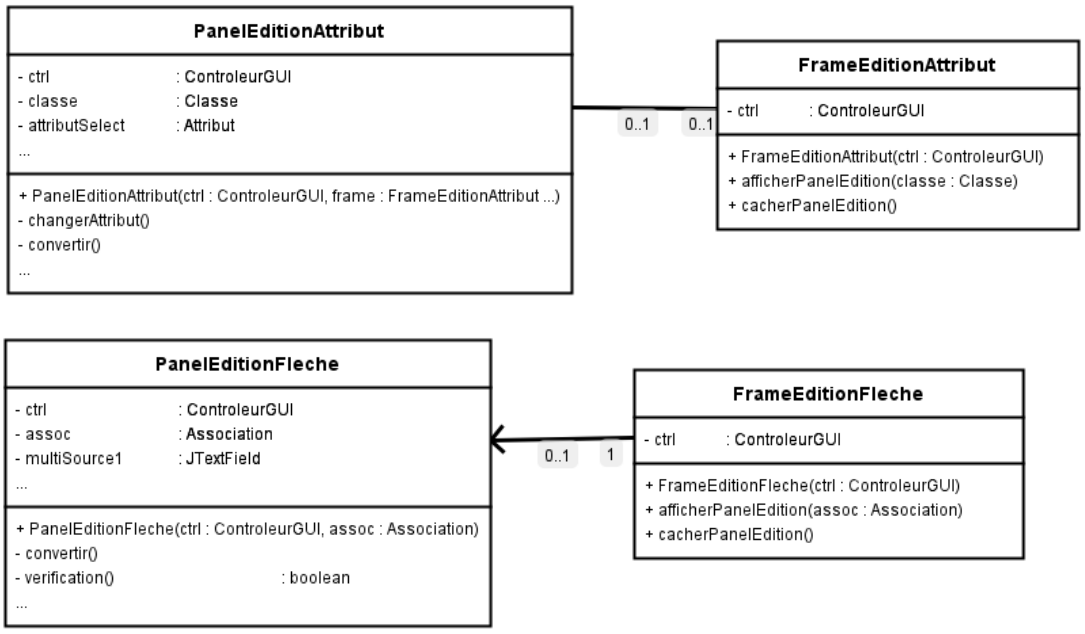
9. Interface d'Édition UML

9.1 Principe architectural

9.1.1 Emplacement

```
src/
└── vue/
    └── GUI/
        └── edition/
            ├── FrameEditionAttribut.java
            ├── FrameEditionFleche.java
            ├── PanelEditionAttribut.java
            └── PanelEditionFleche.java
```

9.1.2 Diagramme UML



9.2 Édition des Attributs

9.2.1 Accès

Double-clic sur une classe dans le diagramme

9.2.2 Conditions

- La classe ne doit pas être externe
- La classe doit avoir au moins un attribut (hors associations)

9.2.3 Propriétés modifiables

Multiplicité :

- Valeur minimale (entier ou *)
- Valeur maximale (entier ou *)
- Exemple : [0..1], [1..], []

Contraintes :

- Rien (aucune contrainte)
- {gelée} (attribut final, non modifiable)
- {addOnly} (ajout seul autorisé)
- {Requête} (méthode de lecture seule)

9.2.4 Fonctionnement

Une liste déroulante qui affiche tous les attributs de la classe. Quand on sélectionne un attribut, les champs se remplissent avec ses valeurs actuelles. On modifie, on valide, et le diagramme se met à jour automatiquement. La modification passe par actionPerformed(ActionEvent e) de PanelEditionAttribut.java.

9.2.5 Validation

- Les multiplicités doivent être cohérentes : Min ≤ Max
- Le caractère * représente l'infini
- Si invalide : message d'erreur, la fenêtre reste ouverte pour correction

9.3 Édition des Associations

9.3.1 Accès

Clic simple sur une flèche du diagramme

9.3.2 Conditions

- La flèche doit être cliquable (zone de collision détectée)
- L'association ne doit pas impliquer de classes externes masquées

9.3.3 Propriétés modifiables

Pour chaque extrémité (Source et Cible) :

- Multiplicité : Min et Max (entier ou *)
- Rôle : Nom textuel qui apparaît sur le diagramme

9.3.4 Fonctionnement

La fenêtre affiche deux colonnes (Source et Cible) avec les valeurs actuelles. On modifie les champs souhaités, on valide, et les deux extrémités de l'association sont mises à jour simultanément. La modification passe par actionPerformed(ActionEvent e) de PanelEditionFleche.java.

9.3.5 Validation

Mêmes règles que pour les attributs. Les rôles peuvent être vides (ils disparaissent du diagramme).

9.4 Mécanisme général

9.4.1 Après chaque modification réussie

1. L'objet métier (Attribut ou Association) est modifié directement
2. Le contrôleur rafraîchit le diagramme
3. Les classes sont redessinées avec les nouvelles valeurs
4. Les flèches sont re-positionnées si nécessaire
5. Un message de succès s'affiche
6. La fenêtre d'édition se ferme

9.4.2 En cas d'erreur

- Message explicite (ex: "Multiplicités invalides")
- La fenêtre reste ouverte
- Les valeurs précédentes sont conservées
- L'utilisateur peut corriger et réessayer

9.5 Points importants

- **Classes externes :** Non éditables. Si on tente de les éditer, un message informatif s'affiche.
- **Persistance :** Les modifications sont en mémoire uniquement. Pour les sauvegarder, il faut exporter le diagramme en XML.
- **Rafraîchissement :** Le diagramme se redessine automatiquement après chaque modification validée, les changements sont immédiatement visibles.
- **Unicité :** Une seule fenêtre d'édition peut être ouverte à la fois par type (attributs ou associations). Si on clique ailleurs, la fenêtre actuelle se ferme et une nouvelle s'ouvre.

10. Afficher et Dessiner les Classes UML choisit

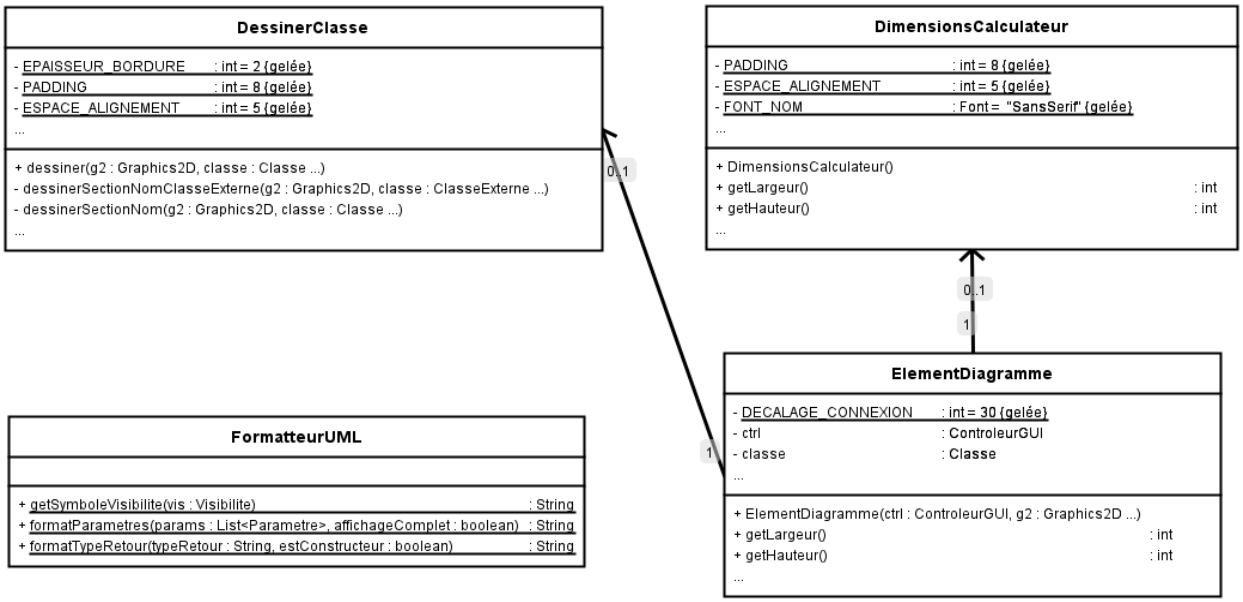
Ce module gère l'affichage visuel des classes (Nom, Attributs, Méthodes), leurs dimensions et leurs points d'ancrage pour les associations.

10.1 Principe architectural

10.1.1 Emplacement

```
src/
└── vue/
    └── GUI/
        └── dessin/
            └── classe/
                ├── DessinerClasse.java
                ├── DimensionsCalculateur.java
                ├── ElementDiagramme.java
                └── FormatteurUML.java
```

10.1.2 Diagramme UML



10.2 DessinerClasse.java

Cette classe effectue le rendu graphique (lignes, texte, rectangles) via l'objet Graphics2D.
Logique : Elle dessine la classe en trois blocs distincts (Nom, Attributs, Méthodes).

- **Alignement** : Elle utilise les données calculées par DimensionsCalculateur pour aligner verticalement les types des attributs et des méthodes
- **Style** : Gère le soulignement pour les membres static, l'italique pour les classes abstract, et un fond gris spécifique pour les ClasseExterne.
- **Troncature** : Si affichageComplet est faux, elle arrête la boucle de dessin après 3 éléments et ajoute "...".

Méthodes importantes :

- **dessiner(...)** : Orchestre qui appelle les sous-méthodes de section (Nom puis Attributs puis Methode).
- **dessinerSectionAttributs(...)** : Gère la concaténation
  - *Symbole + Nom + : Type + {gelé} + [Multiplicité].*

10.3 DimensionsCalculateur.java

Cette classe calcule la taille requise (largeur/hauteur) de la classe avant le dessin. C'est essentiel pour centrer le titre et redimensionner le rectangle de fond.
Logique : Elle parcourt tous les membres (attributs/méthodes) et utilise FontMetrics pour mesurer la largeur des chaînes de caractères en pixels.

- Détermine maxLargeurNomsAttributs et maxLargeurNomsMethodes pour permettre à DessinerClasse d'aligner les types (les ":") sur une même colonne verticale.
- Elle gère les marges (PADDING) et l'espace minimal requis.

Méthode clé :

- **calculer(...)** : Remplit les attributs de dimensions. Doit être appelée à chaque changement de contenu ou de police.

10.4 FormatteurUML.java

Classe statique pour la manipulation de chaînes de caractères. Elle assure la syntaxe UML

10.4.1 Méthodes utilitaires :

- **getSymboleVisibilite(Visibilite)** : Transforme l'enum en +, -, #, ~.
- **formatParametres (...)** : Génère la chaîne (nom : type, ...) pour les signatures de méthodes. Gère l'ajout des "..." si la liste est tronquée.
- **formatTypeRetour (...)** : Gère le cas particulier des constructeurs (pas de retour) et des void (souvent masqués en UML).

11. Sélection des fichiers : PanelChargement

Contenu dans la FrameChargement.java.

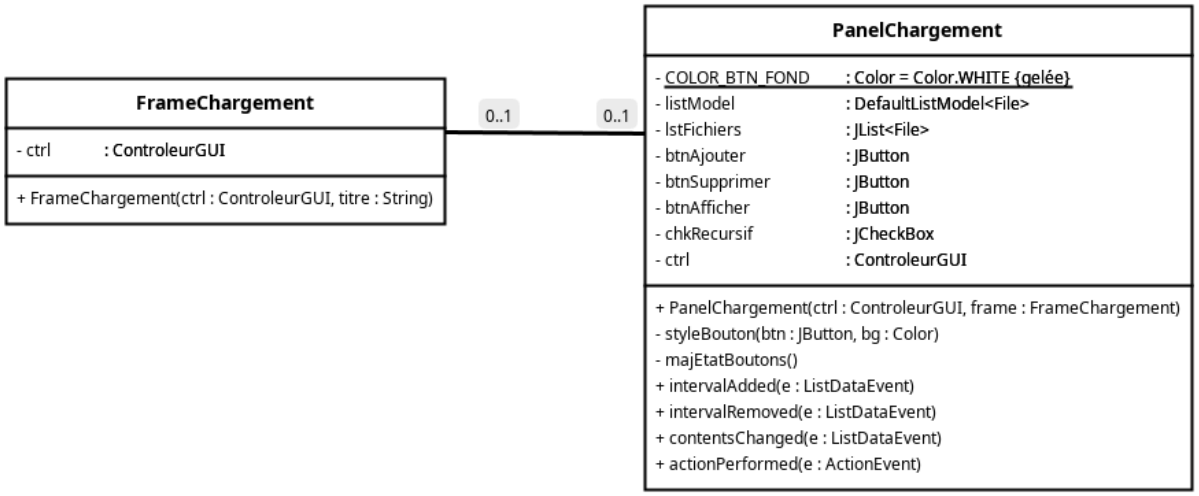
Rôle : Panneau central permettant à l'utilisateur de constituer une liste de fichiers .java ou de dossiers à analyser. Il gère l'ajout, la suppression et le lancement du traitement.

11.1 Principe architectural

11.1.1 Emplacement

```
src/
└── vue/
    └── chargement/
        ├── FrameChargment.java
        └── panels/
            └── PanelChargement.java
```

11.1.2 Diagramme UML



11.2 Fonctionnalités

Ajouter : Appelle SelecteurFichier.choisir(...) en mode *Fichiers et Dossiers*. Filtre les doublons avant l'insertion dans le DefaultListModel.

Supprimer : Retire les éléments sélectionnés de la JList.

Afficher (Lancer l'analyse) :

- Vérifie si la liste contient des éléments (sinon, avertissement via ErrorUtils).
- Récupère l'état de la CheckBox "Récursif".
- Délègue le traitement au contrôleur via ctrl.traiterListeFichiers(...).
- Rafraîchit l'interface principale et ferme la fenêtre de chargement.

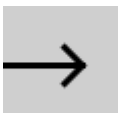
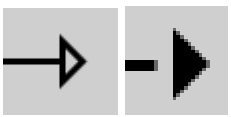
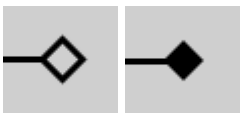
12. Affichage des flèches

12.1 Principe Architectural

12.1.1 Emplacement

```
src/
└── vue/
    └── GUI/
        ├── dessin/
        ├── Fleche.java
        └── fleches/
            ├── FlecheAgregat.java
            ├── FlecheComposition.java
            ├── FlecheDependance.java
            ├── FlecheGeneralisation.java
            ├── FlecheImplementation.java
            ├── FlecheSimple.java
            └── FlecheSousClasse.java
```

12.2 Les constantes de Fleche.java creer dans son bloc static.

Formes de têtes de flèches			Styles de traits	
TETE_V	TETE_TRIANGLE	TETE_CARRE	LIGNE_NORMAL	LIGNE_POINTILLE
				

12.3 Attributs d'une flèche

Attribut	Type	Rôle
xDebut, yDebut	int	Point de départ
xFin, yFin	int	Point d'arrivée
couleur	Color	Couleur de la flèche
bidirectionnel	boolean	si la flèche est réflexive
association	Association	Lien avec un objet Association

12.4 Construction d'une flèche

La classe Fleche **n'est pas instanciée directement** pour représenter une relation UML. Elle joue le rôle de **classe mère**, fournissant une base commune pour la construction de flèches spécialisées.

Son objectif principal est de centraliser :

- les données géométriques (points de départ et d'arrivée),
- les conventions graphiques UML (styles, tailles, transformations),
- ainsi que les comportements partagés (multiplicités, rôles, interactions).

Les constructeurs définis dans Fleche sont donc pensés comme des **outils destinés aux sous-classes**. Chaque type de relation UML est implémenté via une classe héritant de Fleche, par exemple (toutes les flèches ne sont pas forcément utilisée) :

Classe spécialisée	Relation UML représentée
FlecheSimple	Association simple
FlecheAgregat	Agrégation
FlecheComposition	Composition
FlecheDependance	Dépendance
FlecheGeneralisation	Héritage
FlecheImplementation	Implémentation

12.5 Méthodes essentielles.

Méthode draw(Graphics2D)

- dessiner le trait principal,
- appliquer le style adéquat (plein, pointillé, couleur),
- dessiner la ou les têtes de flèches (si réflexive),
- appeler les méthodes d'affichage du texte (multiplicités et rôles).

On utilise la méthode ".draw()" pour les têtes creuses et ".fill()" pour les têtes pleines

Méthode getHitBox()

La méthode getHitBox() calcule et retourne la **zone cliquable** associée à la flèche.

Son objectif est d'améliorer l'ergonomie de sélection : au lieu de se limiter à un simple segment fin, la méthode génère un **rectangle orienté** autour de la ligne.

Méthode transform(Path2D shape, int xd, int yd, int xf, int yf)

Cette méthode utilitaire applique une **transformation géométrique complète** à une forme de tête de flèche (Shape, Path2D).

Elle permet de :

- orienter la forme selon l'angle réel de la relation,
- positionner précisément la tête à l'extrémité souhaitée.

Le calcul repose sur :

- l'angle du segment défini par les points (xd, yd) et (xf, yf),
- une translation vers le point d'arrivée,
- une rotation autour de ce point.

Méthode drawMultiplicite(Graphics2D g)

Cette méthode gère l'affichage des **multiplicités** aux extrémités de la flèche. Elle assure le positionnement dynamique du texte le long de la ligne.

Méthode drawRole(Graphics2D g)

La méthode drawRole(Graphics2D g) affiche les **rôles UML** associés à la relation.

Le positionnement prend en compte :

- l'orientation de la flèche (gauche → droite ou inversement),
- un décalage longitudinal et perpendiculaire contrôlé,
- un fond semi-transparent identique à celui des multiplicités.

13. Précision Couleur Uml

Changement de la couleur de fond principale

Par défaut, l'application utilise un fond blanc, mais pour des raisons de contraste nous l'avions mis en gris Pour le modifier :

1. Ouvrez le fichier PanelSchema.java.
2. Dans le **constructeur**, remplacez la ligne suivante (autour de la ligne 72) :
   `this.setBackground(Color.WHITE);`
   par :
   `this.setBackground(new Color(207,207,207));`

Suppression du fond blanc des multiplicités

Pour améliorer la visibilité, nous avions mis un fond blanc transparent au multiplicité. Pour ajouter cet effet, vous devez décommenter les lignes aux alentours de 325 et 347 de la méthode drawMultiplicite(Graphics2D g) :

- ~Ligne 325 : `g.fillRoundRect(xm, ym - fm.getHeight(), fm.stringWidth(text) + 10, fm.getHeight() + 5, 10, 10);`
- ~Ligne 347 : `g.fillRoundRect(xm, ym - fm.getHeight(), fm.stringWidth(text) + 10, fm.getHeight() + 5, 10, 10);`

Suppression du fond blanc sur les rôles

Pour améliorer la visibilité, nous avions mis un fond blanc transparent au rôles. Pour ajouter cet effet, vous devez décommenter les lignes aux alentours de 394 et 423 de la méthode drawRole(Graphics2D g) :

- ~Ligne 394 : `g.fillRoundRect(xr - 5, yr - texteHauteur + 3, texteLargeur + 10, texteHauteur + 4, 8, 8);`
- ~Ligne 423 : `g.fillRoundRect(xr - 5, yr - texteHauteur + 3, texteLargeur + 10, texteHauteur + 4, 8, 8);`

14. Annexe Arborescence du projet