Présentation

- Ce projet est un Outil de rétroconception, il permet de génerer un Diagramme de classes UML à partir d'un répertoire de classes Java.

- Attention : Lors du chargement d'un projet, les packages ne sont pas pris en compte, toutes les classes doivent être placées à la racine du répertoire selectionné.

Besoin

Ce projet a pour objectif de faciliter la compréhension et la prise en main rapide du code, en particulier lors de la reprise du projet par une personne qui n'en connaît pas le fonctionnement initial.
Il peut notamment être utilisé par des enseignants afin d'illustrer la transformation d'un projet Java en diagrammes UML, permettant ainsi une meilleure visualisation de la structure et de l'architecture du code.

Utilisation

- Une fois le projet compilé et executé, une interface graphique s'ouvrira.

> Pour ajouter un projet, voir la barre du haut : Fichier -> Ouvrir un projet… et enfin ouvrir le répertoire contenant le projet Java votre choix.

> La barre latérale gauche affiche l'entièreté des projets sauvegardés. Cliquer sur un projet l'affichera sur le panneau principal, chargeant le diagramme.

Sur le panneau principal, on peut :

- Déplacer les classes (clic gauche maintenu)
- Zoomer (ctrl + molette), et de se déplacer librement (clic droit maintenu)
- Ajouter des rôles aux associations (double clic gauche sur une classe -> Ajouter rôle)
- Modifier les rôles affichés (double clic gauche sur une classe -> Modifier rôle)
- Modifier les multiplicités affichées (double clic gauche sur une classe -> Modifier multiplicité)

Fonctionnalités accessibles depuis la barre supérieure

- Ouvrir un nouveau projet Java afficher dans la barre des projets (Fichier -> Ouvrir un projet…)
- Exporter en image ce qui est affiché sur le panneau de Diagramme (Fichier -> Exporter en image)
- Sauvegarder l'affichage actuel du schéma présent sur le panneau diagamme (Fichier -> Sauvegarder)
- Fermer l'interface (Fichier -> Quitter)
- Afficher ou non les attributs des classes (Affichage -> Afficher attributs)
- Afficher ou non les méthodes des classes (Affichage -> Afficher méthodes)
- Optimiser les positions des blocs et liaison (Affichage -> Optimiser les positions)
- Optimiser les liaisons seulement (Affichage -> Optimiser les liaisons uniquement)
- Avoir les détails des auteurs (Aide -> A propos)

La liste des projets sauvegardés et leur position est sauvegardée dans le fichier data/projets.xml sous la forme : «chemin absolu» (tabulation) «chemin avec le ficher de sauvegarde du projet en question»
Les diagrammes sont sauvegardés dans data/sauvegarde/ avec toutes les informations relatives aux classes et leurs liaisons.
Les données des diagrammes peuvent être changées directement sur les fichiers .xml dans data, via un éditeur de texte.

Fonctionnalités de la section de projets

- Choisir le projet a afficher dans le panneau diagramme (clique gauche sur le projet voulu)
- Renommer un projet (clique droit sur le projet voulu -> Renommer projet)
- Supprimer un projet (clique droit sur le projet voulu -> Supprimer projet)

Un projet présent dans `projets.xml` mais inexistant sur l'ordinateur est ignoré dans la liste des projet lors du lancement.

Structure

Le projet est organisé en modèle MVC : Métier-Vue-Controleur.
En MVC, le métier gère l'écriture, sauvegarde, le traitement de données. La vue gère l'affichage et l'interface Homme-Machine. Le controleur gère la liaison entre la vue et le métier.

```
C:.
|   compile.list
|   Documentation.md
|   Executer.md
|   run.bat
|   run.sh
|
├───class
|
├───data
|   |
|   ├───donnees
|   |       projets.xml
|   |
|   └───sauvegardes
|
├───src
|   |
|   ├───controleur
|   |       Controleur.java
|   |
|   ├───metier
|   |   |
|   |   ├───lecture
|   |   |       AnalyseurFichier.java
|   |   |       GenerateurAssociation.java
|   |   |       Lecture.java
|   |   |       ParseurJava.java
|   |   |       UtilitaireType.java
|   |   |
|   |   ├───objet
|   |   |       Association.java
|   |   |       Attribut.java
|   |   |       Classe.java
|   |   |       Heritage.java
|   |   |       Interface.java
```

```
|   |   |       Liaison.java
|   |   |       Methode.java
|   |   |       Multiplicite.java
|   |   |       Parametre.java
|   |   |
|   |   ├──────sauvegarde
|   |   |       GestionSauvegarde.java
|   |   |
|   |   └──────util
|   |   |   |   ConstantesChemins.java
|   |   |   |
|   |   |   └──────test_structure_projet
|   |   |           ElementStructureProjet.java
|   |   |           TypeElement.java
|   |   |           VerificationStructureProjet.java
|   |   |
|   ├──────res
|   |       uml_icon.png
|   |
|   └──────vue
|       |   BarreMenus.java
|       |   BlocClasse.java
|       |   FenetrePrincipale.java
|       |   PanneauDiagramme.java
|       |   PanneauProjets.java
|       |
|       ├──────liaison
|       |       CalculateurChemin.java
|       |       DetecteurObstacles.java
|       |       GestionnaireAncrage.java
|       |       GestionnaireIntersections.java
|       |       LiaisonVue.java
|       |       RenduLiaison.java
|       |
|       └──────role_classe
|               FenetreChangementMultiplicite.java
```

| FenetreModifRole.java
| PanneauModif.java
| PanneauModifRole.java

Description du système

Cette section décrit le fonctionnement du système d'ouverture/analyse/sauvegarde/chargement de l'aplication.

Pour une description détaillée de la structure et des classes, voir la section "Description détaillée des classes"

Chargement d'un projet depuis une liste de .java

Lorsque l'utilisateur charge une première fois un projet, le programme va récuperer le chemin absolu de celui ci et l'envoyer au métier (en passant par le controlleur), plus précisément la classe "Lecture".
Lecture et ses classes associées (AnalyseurFicher, GenerateurAssociation, ParseurJava et UtilitaireType) vont instancier différentes listes de Classes, Interfaces, Heritages et Associations.

La lecture se fait via un scanner qui analyse ligne par ligne les classes Java pour en extraire des objets tels que :

- **Classe** : A un `nom` , un `type` (*class*/*abstract*). Elle stocke si elle a un héritage avec `isHeritage` , une interface dans `nomInterface` . Elle stocke une liste d' `attributs` et de `méthodes` .

- **Attribut** : A un `nom` , un `type` , une `visibilité` (*private*/*public*/*protected*/*package*), une `portee` (*instance*/*classe). et `isConstant`

- **Methode** : A un `nom` , une type de `retour` , une `visibilite` (*private*/*public*/*protected*/*package*), un booléen `isAbstract` , et une liste de **Parametre** (un paramètre à un `nom` et un `type`).

- **Liaison** : A une classe d'origine (`classeOrig`) et de destination (`classeDest`). Différents types de "liaisons" entre 2 classes :

    - **Interface** : Hérite de **Liaison**.
    - **Heritage** : Hérite de **Liaison**.

  Si le type d'un attribut correspond au nom d'une autre Classe du projet, il est supprimé et changé en **Association** :

    - **Association** : Hérite de **Liaison**, stocke un booléen si il est `unidirectionnel`, une multiplicité d'origine (`multiOrig`) et de destination (`multiDest`).

- **Multiplicite** : Contient deux int `debut` et `fin`. Exemple : (0..1) et si (0..*), alors "fin" est égal a Integer.MAX_INT (interpreté comme *)

Une fois toute la lecture terminée, les informations sont transférées via le Controleur à la vue (**FenetrePrincipale**), convertit les **Classe** en **BlocClasse** et les **Liaison** en **LiaisonVue**, qui seront utilisées dans la vue :

- **BlocClasse** : A tout les attribut de **Classe**, mais aussi un booléen `estInterface`, les int `x` `y` de sa position et `largeur` et `hauteur`.
- **LiaisonVue** : A tout les atrtibuts de **Association** (et donc de tout les autres types de liaison), mais aussi deux **Point**s `ancrageOrigine` et `ancrageDestination`.

Ces instances de classes sont ensuite envoyées à la vue par le controleur, et sont affichées sur le Diagramme de classe par la vue.

Sauvegarde d'un projet

Lorsque l'utilisateur sauvegarde son diagramme, son contenu sera enregistré dans `data/sauvegardes/«nom du projet».xml`. Voir la classe `GestionSauvegarde`, dans les méthodes `sauvegarderClasses`, `sauvegarderCoordProjet`, `sauvegarderLiaison`.

> Il est important de notifier que le format de fichier XML ne correspond pas au format du contenu, et que ces fichiers auraient du être signés en .data à la place.

Le contenu du diagramme sera sauvegardé sous format texte dans cette forme :

```
<lien du fichier>

-—- Classes -—-
```

`nomBlocClasse abcisse ordonnee largeur hauteur estInterface`

> Les données de classes sont traitées lors de la sauvegarde/chargement. Il peut y avoir autant de classes que possible dans un diagramme. Une classe peut contenir une liste d'attributs et méthodes.
> Exemple :
>
> ```
> Animal 50 50 200 150 false
> ```

`-/+/~ nomAttribut : type {frozen/addOnly/requête}`

`-/+/~ nomMethode ( nomParam : typeParam , ...) : type`

> Les attributs et méthodes sont traités comme des chaînes de caractère. Il peut y avoir autant d'attributs et méthodes dans une classe que possible.
> Exemple :
>
> ```
> - nom : String
> - age : int
> + Animal(nom : String, age : int)
> + getNom() : String
> ```

-—- Liaisons -—-

`typeLiaison id blocOrig coteOrig posRelOrig blocDest coteDest posRelDest roleOrig roleDest multiOrig multiDest`

> Les données de liaisons sont traitées lors de la sauvegarde/chargement. Il peut y avoir autant de liaisons que possible dans un diagramme. Une classe peut contenir une liste de classes et méthodes.
> `typeLiaison` inclut `heritage` / `association_uni` / `association_bi` / `interface`
> `blocOrig` et `blocDest` correspondent aux noms des classes d'origine et de destination de la liaison.
> Exemple :
>
> ```
> heritage 0 Chat TOP 0.5 Animal BOTTOM 0.5 ? ?
> association_uni 1 Chat RIGHT 0.5 Collier LEFT 0.5
> ```

Toutes les informations doivent être séparées par des tabulations, sous les balise "-—- Classes -—-" ou "-—- Liaisons -—-".
Les lignes commencant par `#` ne sont pas prises en compte par le lecteur.

Chargement d'un projet en format XML

Lorsque l'utilisateur sélectionne un projet dans la barre de gauche, si le projet est déjà existant dans `data/sauvegardes/«nom du projet».xml`, le chargement se fera depuis ce XML plutot que sur le projet de classes .java.
La lecture par l'application se basera sur les mêmes critères que la sauvegarde (ci-dessus). Voir la classe `GestionSauvegarde`, dans la méthode `sauvegardeProjetXml`.
Ainsi, les classes seront chargées d'après le principe du premier chargement (**BlocClasse**, **LiaisonVue**...).

Description détaillée des classes

Voici une description détaillée de toutes les classes et leurs attributs & méthodes.

Partie Controleur

Dans le modèle MVC, le controlleur fait le lien entre la vue et le métier.

Présentation

Le **Controleur** est le point central de l'application, assurant la communication entre le **métier** (lecture et analyse des classes Java) et la **vue** (affichage graphique des diagrammes UML). Il gère le chargement des projets, la création des blocs de classes (`BlocClasse`) et des liaisons (`LiaisonVue`), ainsi que la sauvegarde et la récupération des projets.

Structure

Attributs

Attribut	Type	Description
`lecture`	`Lecture`	Instance pour analyser les fichiers Java et extraire les classes, attributs, méthodes, héritages, interfaces et associations.
`fenetrePrincipale`	`FenetrePrincipale`	Référence à la fenêtre principale de l'IHM pour afficher les blocs et liaisons.
`gestionSauvegarde`	`GestionSauvegarde`	Instance pour gérer les sauvegardes des projets en XML.
`lstLiaisons`	`List<LiaisonVue>`	Liste des liaisons entre les blocs (associations, héritages, interfaces).
`lstBlocs`	`List<BlocClasse>`	Liste des blocs représentant les classes et interfaces dans le diagramme.

Constructeur

Controleur()

Initialise le contrôleur, les listes de blocs et de liaisons, ainsi que les instances de `GestionSauvegarde` et `FenetrePrincipale`.

- **Rôle** : Initialise le contrôleur, les listes de blocs et de liaisons, ainsi que les instances de `GestionSauvegarde` et `FenetrePrincipale`.
- **Détails** :
  - `lstLiaisons` et `lstBlocs` sont initialisées comme des listes vides.
  - `gestionSauvegarde` et `fenetrePrincipale` sont initialisées avec `this` pour permettre la communication entre les composants.

Méthodes

static void main(String[] args)

Point d'entrée de l'application.

- **Rôle** : Point d'entrée de l'application.
- **Détails** :
  - Vérifie la structure du projet avec `VerificationStructureProjet`.
  - Crée une instance de `Controleur` et affiche la `FenetrePrincipale`.

void chargerProjet(String cheminProjet)

Charge un projet à partir d'un chemin donné.

- **Rôle** : Charge un projet à partir d'un chemin donné.
- **Paramètres** :
  - `cheminProjet` : Chemin absolu du répertoire contenant les classes Java.
- **Détails** :
  - Vérifie si le projet est déjà sauvegardé dans un fichier XML.
  - Si le projet est sauvegardé, il est chargé depuis le fichier XML (`chargerProjetDepuisXml`).
  - Sinon, il est chargé depuis les fichiers Java (`chargerProjetDepuisJava`).

void chargerProjetDepuisXml(String intituleProjet)

Charge un projet à partir d'un fichier XML de sauvegarde.

- **Rôle** : Charge un projet à partir d'un fichier XML de sauvegarde.
- **Paramètres** :
  - `intituleProjet` : Nom du projet (utilisé pour trouver le fichier XML correspondant).
- **Détails** :
  - Récupère le chemin du projet à partir de l'intitulé.
  - Charge les blocs de classes et les liaisons depuis le fichier XML.
  - Met à jour `lstBlocs` et `lstLiaisons`.

void chargerProjetDepuisJava(String cheminProjet)

Charge un projet à partir des fichiers Java.

- **Rôle** : Charge un projet à partir des fichiers Java.
- **Paramètres** :
  - `cheminProjet` : Chemin absolu du répertoire contenant les classes Java.
- **Détails** :
  - Crée une instance de `Lecture` pour analyser les fichiers Java.
  - Convertit les classes Java en `BlocClasse` et les ajoute à `lstBlocs`.
  - Crée les liaisons (associations, héritages, interfaces) et les ajoute à `lstLiaisons`.
  - Optimise les positions des blocs et des liaisons dans la vue.

BlocClasse créerBlocAPartirDeClasse(Classe classe, int x, int y)

Crée un `BlocClasse` à partir d'une instance de `Classe`.

- **Rôle** : Crée un `BlocClasse` à partir d'une instance de `Classe`.
- **Paramètres** :
  - `classe` : Instance de `Classe` à convertir.
  - `x` : Coordonnée X du bloc.
  - `y` : Coordonnée Y du bloc.
- **Retourne** : Un `BlocClasse` initialisé avec les attributs et méthodes de la classe.
- **Détails** :
  - Convertit les attributs et méthodes de la classe en chaînes de caractères formatées.
  - Définit si le bloc représente une interface.

List<LiaisonVue> créerLiaisonsDepuisAssoc(List<Association> lstAssoc, HashMap<String, BlocClasse> mapBlocsParNom, List<LiaisonVue> lstLiaisons)

Crée des liaisons de type "association" à partir d'une liste d'associations.

- **Rôle** : Crée des liaisons de type "association" à partir d'une liste d'associations.
- **Paramètres** :
  - `lstAssoc` : Liste des associations.
  - `mapBlocsParNom` : HashMap associant les noms de classes aux blocs.
  - `lstLiaisons` : Liste des liaisons existantes.
- **Retourne** : La liste des liaisons mise à jour.
- **Détails** :
  - Pour chaque association, crée une `LiaisonVue` entre les blocs correspondants.

List<LiaisonVue> créerLiaisonsDepuisHerit(List<Heritage> lstHerit, HashMap<String, BlocClasse> mapBlocsParNom, List<LiaisonVue> lstLiaisons)

Crée des liaisons de type "héritage" à partir d'une liste d'héritages.

- **Rôle** : Crée des liaisons de type "héritage" à partir d'une liste d'héritages.
- **Paramètres** :
  - `lstHerit` : Liste des héritages.
  - `mapBlocsParNom` : HashMap associant les noms de classes aux blocs.
  - `lstLiaisons` : Liste des liaisons existantes.
- **Retourne** : La liste des liaisons mise à jour.
- **Détails** :
  - Pour chaque héritage, crée une `LiaisonVue` entre les blocs correspondants.

List<LiaisonVue> créerLiaisonsDepuisInterface(List<Interface> lstInter, HashMap<String, BlocClasse> mapBlocsParNom, List<LiaisonVue> lstLiaisons)

Crée des liaisons de type "interface" à partir d'une liste d'interfaces.

- **Rôle** : Crée des liaisons de type "interface" à partir d'une liste d'interfaces.
- **Paramètres** :
  - `lstInter` : Liste des interfaces.
  - `mapBlocsParNom` : HashMap associant les noms de classes aux blocs.
  - `lstLiaisons` : Liste des liaisons existantes.
- **Retourne** : La liste des liaisons mise à jour.
- **Détails** :
  - Pour chaque interface, crée une `LiaisonVue` entre les blocs correspondants.

void sauvegardeProjetXml(String cheminFichier)

Sauvegarde le projet dans un fichier XML.

- **Rôle** : Sauvegarde le projet dans un fichier XML.
- **Paramètres** :
  - `cheminFichier` : Chemin du fichier à sauvegarder.
- **Détails** :
  - Délègue la sauvegarde à `gestionSauvegarde`.

void sauvegarderClasses(List<BlocClasse> listBlocClasses, List<LiaisonVue> listLiaison, String cheminProjet)

Sauvegarde les blocs et les liaisons dans un fichier - **Rôle** : Sauvegarde les blocs et les liaisons dans un fichier. - **Paramètres** : - `listBlocClasses` : Liste des blocs à sauvegarder. - `listLiaison` : Liste des liaisons à sauvegarder. - `cheminProjet` : Chemin du projet. - **Détails** : - Délègue la sauvegarde à `gestionSauvegarde`.

void ajouterBlockList(BlocClasse block)

Ajoute un bloc à la liste des blocs

- **Rôle** : Ajoute un bloc à la liste des blocs.
- **Paramètres** :
  - `block` : Bloc à ajouter.

Partie Métier

Dans le modèle MVC, le métier gère la logique des entités, des traitements de données et des lectures.

Métier comporte 4 packages.

- **objet** : Contient toutes les entités qui représentent les élements de classe Java.
- **lecture** : Gère l'analyse de classes Java pour les convertir en objets.
- **util** : Contient des outils et constantes utilisées dans tout le projet.
- **sauvegarde** : Contient la gestion de sauvegarde et chargement en format XML.

Package metier.objet

Association

Cette classe permet de créer des objets d'Associations.

Hérite de Liaison.

- **Attributs** : nbAssoc(identifiant unique), multiOrig, multiDest, num
- **Méthodes** : Getters, Setters, isUnidirectionnel(savoir si la classe est Unidirectionnel) et un toString.

Attribut

Cette classe permet de créer des objets d'Attributs.

Elle contient des caractéristiques propres à un attribut.

- **Attributs** : nom, type, visibilite, portee (instance ou classe)
- **Méthodes** : Getters et Setters, ainsi qu'un toString propre.

Methode

Cette classe permet de créer des objets de méthodes.

Elle contient des caractéristiques propres à une méthode.

- **Attributs** : nom, retour(type de retour), visibilite, isAbstract, lstParametre
- **Méthodes** : Getters, Setters, un toString et une méthode pour ajouter un paramètre à la liste

Parametre

Cette classe permet de créer des objets de parametres.

Elle contient des caractéristiques propres à un parametre.

- **Attributs** : nom, type
- **Méthodes** : Getters, Setters et un toString

Classe

Cette classe permet de créer des objets de Classe.

Elle contient des caractéristiques propres à une classe.

- **Attributs** : nom, classeParente(nom de la classe hérité), type, isHeritage, nomInterface, liste d'attributs et de méthode
- **Méthodes** : Getters et des méthodes booléennes qui nous serviront dans les fichiers du dossiers lecture pour détecter de quel type de classe il s'agit.

Liaison

Cette classe permet de créer un objet Liaison qui représente les relations entre deux classes comme par exemple l'héritage ou les interfaces.

- **Attributs** : classeOrigine(classe source), classeSource(classe cible)
- **Méthodes** : Getters

Heritage

Cette classe permet de créer un objet Heritage afin de déterminer quel classe est à l'origine de l'héritage et quel classe hérite de la classe d'origine.

Hérite de la classe Liaison.

- **Attributs** : herite des attributs de Liaison
- **Méthode** : toString

Interface

Cette classe permet de créer un objet Interface afin de déterminer quelle classe est l'interface implémentée et quelle classe implémente l'interface.

Hérite de la classe Liaison.

- **Attributs** : herite des attributs de Liaison
- **Méthode** : toString

Multiplicite

Cette classe permet de créer des objets de multiplicite, qui représente la multiplicité d'une association ou d'une relation entre deux classes.

- **Attributs** : debuts, fin
- **Méthodes** : Deux constructeurs, setters et un toString.

Constructeur 1 : Verifie que les valeurs des multiplicités sont correctes.
Constructeur 2 : Verifie si la valeur est un '*' auquel cas on lui affecte ce caractere.

Package metier.lecture

Classe ParseurJava

Cette classe permet d'analyser un fichier .java, et de créer une classe.

Elle analyse bien sur les attributs, les méthodes etc. C'est ce qui va créer la classe.

Cette classe est appelée dans la classe AnalyserFichier.

ParseurJava (constructeur)

Le constructeur initialise les attributs d'instance.

Classe parser(Scanner scFic, String nomFichierAvExt)

C'est la methode principale de cette classe qui permet d'analyser le fichier, de créer la classe et de la retourner.
Elle fait appel à des sous-méthodes pour gérer les cas des attributs, des méthodes, des records etc.

Attribut parserAttribut(String ligne)

Cette méthode permet de creer et de retourner un Attribut, qu'elle ajoute par la suite dans la liste d'attributs.
Elle extrait la visibilite, les modificateurs, la portée(classe/Instance), le type et le nom.

Methode parserMethode(String ligne, String nomFichier, String typeClasse, boolean ligneCommenceParModificateur, boolean estMethodeInterface, String nomMethode, String nomConstructeur, List<Parametre> lstParametres)

Cette méthode permet de creer et de retourner une Methode.
Elle extrait la visibilite, les modificateurs(si abstract est présent), la portée(classe/Instance), le type et les parametres.
Pour les parametres nous appelons la méthode parserParametre.
Et enfin nous verifions bien si la méthode en question est un constructeur en fonction de la nom de la methode.

Parametre parserParametre(String params)

Cette méthode analyse la chaine qui contient les parametres entre parenthèses. Ensuite elle creer et retourne une liste de parametre avec les éléments de la chaine. Elle extrait donc le nom et le type du parametre puis l'ajoute a la liste.

> On vérifie bien si le type du parametres est un type comme des arraylist ou des hashmap, pour pouvoir découper la chaine.

GenerateurAssociation

Classe responsable de la génération des associations entre classes.

ArrayList<Association> generer()

Permet de gérer les associations à faire avec les classes, en prenant en compte les attributs à retirer du bloc grâce à une liste d'attruts. Elle retourne une Liste d'association

void traiterMultiInstance(Classe classeOrig, ArrayList<String> listeMultiInstance)

Méthode qui traite les associations multi-instance (tableaux, List, Set, Map). Trouve automatiquement la classe de destination.

void traiterSimpleInstance(Classe classeOrig, ArrayList<String> listeMultiInstance)

Méthode qui traite les associations à instance unique. Trouve automatiquement la classe de destination.

boolean estBidirectionnel(Classe classeOrig, Classe classeDest)

Méthode qui vérifie si une association donnée est bidirectionnelle. Renvoie un booléen.

void nettoyerAssociations()

Méthode qui nettoie sa liste des associations pour supprimer les doublons et transformer les doublons en associations bidirectionnelles.

AnalyseurFichier

Classe qui parcourt les classes java données, puis qui délègue l'analyse syntaxique à la classe ParseurJava.

HashMap<String, Classe> analyser(String cheminFichier)

Permet de définir si le document en question choisi par l'utilisateur
- Si c'est un répertoire, on creer une liste de fichiers, qui seront ensuite découpé.
- Si c'est un fichier, on utilise la classe ParseurJava afin de découper le fichier.java

UtilitaireType

Classe utilitaire pour manipuler les types Java.

void nettoyerType(String type)

Nettoie le string de type entré pour retirer des élements en trop comme "List<>", "Set<>" ou "[]" et renvoie le résultat

boolean estMultiInstance(String type)

Retourne true si le type peut contenir plusieurs instances (tableau ou collection).

Package metier.sauvegarde

GestionSauvegarde

List<LiaisonVue> lectureLiaison(String dossierFichSelec, Map<String, BlocClasse> hashMapBlocClass)

Permet de parcourir le fichier de la **fenêtre principale de l'application UML**.
Elle structure l'interface graphique en regroupant les panneaux projets et diagramme.
Elle assure le **rôle de point central entre l'IHM et le contrôleur**.
Permet de lire les liaisons UML (associations, héritages, etc.) depuis un fichier de sauvegarde du projet, puis recréer les objets LiaisonVue correspondants dans l'application.

Map<String, BlocClasse> chargerBlocsClasses(String nomProjet)

La méthode renvoie `Map<String, BlocClasse>` , permettant la création de tous les blocs à créer du fichier Java mis en paramètres.
Chaque bloc créer est alors mis dans une HashMap avec le String : Nom du bloc, puis BlocClasse : Le bloc initialisé à ajouter.

void sauvegarderClasses(List<BlocClasse> listBlocClasses, List<LiaisonVue> listLiaison, String cheminProjet)

Sauvegarder l'état complet d'un projet UML

- les classes UML (positions, dimensions, attributs, méthodes)
- les liaisons entre les classes
- référencer le projet dans le fichier projets.xml s'il n'existe pas encore

void sauvegarderCoordProjet(List<BlocClasse> listBlocClasses, String nomProjet, String cheminProjet)

Permet d'écrire les coordonées des classes dans le .xml du fichier lier au .java.

void sauvegardeProjetXml

Permet de sauvegarder le projet en XML, afin de pouvoir réutiliser les valeurs (Position, Nom, Classes, Méthodes, Attributs...) lorsque l'on réouvrira le fichier après l'avoir sauvegarder

Partie Vue

Dans le modèle MVC, la vue est le contenu de l'interface homme-machine.
La vue comporte 2 packages et des classes sans package.

- **liaison** : Gère le calcul et l'affichage des liaisons
- **role_classe** : Gère les élements de modification/d'ajout de rôles et de classes

FenetrePrincipale

FenetrePrincipale(…) (constructeur)

Initialise la fenêtre, crée les panneaux principaux, configure le layout et installe la barre de menus.

ouvrirProjet(…)

Charge un projet UML dans le panneau diagramme, déclenche la sauvegarde initiale via le contrôleur et force le recalcul graphique des liaisons.

exporterImageDiagramme(…)

Capture le diagramme affiché dans le panneau diagramme et l'exporte au format PNG en gérant temporairement le zoom et l'affichage du texte.

Méthodes secondaires

- **affichageAttributs(...)** : appelle `panneauDiagramme.setAfficherAttributs(...)` pour panneauDiagramme.optimiserPositionsLiaisons() pour recalculer la position des liaisons.
- **setSauvegardeAuto(...)** : appelle `panneauDiagramme.setSauvegardeAuto(...)` pour activer ou désactiver la sauvegarde automatique.
- **actionSauvegarder()** : appelle `panneauDiagramme.actionSauvegarder()` pour sauvegarder l'état courant du diagramme.
- **chargerProjet(...)** : délègue l'appel à `controleur.chargerProjet(...)` .
- **sauvegarderClasses(...)** : délègue la sauvegarde des classes et liaisons à `controleur.sauvegarderClasses(...)` .
- **viderDiagramme()** : appelle `panneauDiagramme.viderDiagramme()` pour réinitialiser l'affichage.

PanneauProjets

Cette classe représente le **panneau latéral de gestion des projets** de l'application.
Elle affiche la liste des projets enregistrés, permet leur sélection et leur gestion (renommer, supprimer).
Elle agit comme une **vue dédiée**, en interaction directe avec `FenetrePrincipale`.

Attributs principaux

- `CHEMIN_SAUVEGARDES` : chemin des sauvegardes.
- `fenetrePrincipale` : référence à la fenêtre principale.
- `cheminDossiers` : chemin des dossiers projets.
- `panelProjets` : panneau contenant les projets.

PanneauProjets (constructeur)

Initialise le panneau, configure l'interface graphique, charge la liste des projets et installe les actions utilisateur.

actualiser

Vide le panneau des projets puis recharge la liste depuis le fichier de configuration et met à jour l'affichage.

chargerProjets

Lit le fichier contenant tous les chemins des projets (`projets.xml`), valide les chemins, récupère les intitulés et crée dynamiquement les boutons correspondant aux projets existants.

Méthodes secondaires

- **créerBoutonProjet(...)** : crée un bouton de projet avec menu contextuel et action d'ouverture via `fenetrePrincipale`.
- **renommerProjet(...)** : demande un nouvel intitulé puis met à jour le projet dans le fichier et les sauvegardes associées.
- **supprimerProjet(...)** : supprime un projet de la liste et efface sa sauvegarde après confirmation utilisateur.
- **intituleExiste(...)** : vérifie si un intitulé de projet existe déjà dans le fichier des projets.
- **modifierProjetDansFichier(...)** : modifie le fichier `projets.xml` pour renommer ou supprimer un projet et gère les fichiers de sauvegarde.

PanneauDiagramme

Cette classe représente le **panneau central du diagramme UML**.
Elle gère l'affichage des blocs de classes, des liaisons et toutes les interactions utilisateur (clic, drag, zoom, pan).
Elle constitue le **cœur graphique et interactif** de l'application.

Attributs principaux

- `lstBlocsClasses` : liste des blocs de classes.
- `lstLiaisons` : liste des liaisons.
- `fenetrePrincipale` : référence à la fenêtre principale.
- `cheminProjetCourant` : chemin du projet courant.
- `zoomLevel` : niveau de zoom.
- `panOffsetX/Y` : décalage du pan horizontal/vertical.

PanneauDiagramme (constructeur)

Initialise le panneau, configure le menu contextuel, les paramètres graphiques et installe tous les listeners d'interaction.

chargerProjet

Charge un projet UML, récupère les blocs et liaisons via `FenetrePrincipale`, initialise leurs dépendances et rafraîchit l'affichage.

optimiserPositionsClasses

Organise automatiquement les blocs en grille, recalcule les ancrages et ajuste les liaisons pour éviter les chevauchements.

optimiserPositionsLiaisons

Recalcule les ancrages de toutes les liaisons en utilisant la liste complète des liaisons puis redessine le diagramme.

Méthodes secondaires

- **ajouterListenersInteraction()** : installe les listeners souris pour clic, drag, zoom, pan et menus contextuels.
- **organiserEnGrille()** : positionne les blocs de classes dans une grille régulière selon leur nombre et leur taille.
- **optimiserAncragesPourLiaison(...)** : calcule dynamiquement les côtés d'ancrage optimaux entre deux blocs.
- **determinerMeilleurCote(...)** : détermine le côté de sortie d'une liaison selon la position relative des blocs.
- **determinerMeilleurCoteDestination(...)** : détermine le côté d'entrée opposé pour la destination d'une liaison.
- **paintComponent(...)** : applique le zoom et le pan puis dessine les liaisons et les blocs du diagramme.
- **dessinerLiaisons(...)** : appelle `liaison.dessiner(...)` pour chaque liaison du diagramme.
- **afficherZoomPercentage(...)** : affiche le pourcentage de zoom dans l'interface graphique.
- **modifieMultiplicite(...)** : modifie la multiplicité d'une liaison en fonction de son origine ou destination.
- **modifierRole(...)** : modifie le rôle d'une liaison (origine ou destination) à partir de son identifiant.
- **rafraichirDiagramme()** : force la mise à jour complète de l'affichage Swing.
- **actionSauvegarder()** : délègue la sauvegarde du diagramme à `fenetrePrincipale.sauvegarderClasses(...)`.
- **actionEffectuee()** : déclenche une sauvegarde automatique si l'option est activée.
- **viderDiagramme()** : supprime tous les blocs et liaisons puis rafraîchit l'affichage.

BlocClasse

Cette classe représente **l'affichage graphique d'une classe UML** dans le diagramme.
Elle gère la représentation visuelle (nom, attributs, méthodes), les modes condensé / plein écran et les interactions géométriques.
Elle constitue l'**unité visuelle de base** manipulée par le panneau de diagramme.

Attributs principaux

- `id` : identifiant du bloc.

- `nom` : nom de la classe UML.
- `attributs` : liste des attributs.
- `methodes` : liste des méthodes.
- `x` : position horizontale.
- `y` : position verticale.
- `largeur` : largeur du bloc.
- `hauteur` : hauteur du bloc.
- `affichagePleinEcran` : mode d'affichage actif.

BlocClasse (constructeur)

Initialise un bloc UML avec un identifiant unique, une position initiale, des dimensions par défaut et des listes vides d'attributs et de méthodes.

dessiner

Dessine intégralement le bloc de classe : fond, en-tête, nom, attributs, méthodes, séparateurs et styles selon l'état (interface, sélection, affichage).

calculerHauteur

Calcule dynamiquement la hauteur réelle du bloc en fonction du contenu affiché, du mode d'affichage et des retours à la ligne.

Méthodes secondaires

- **formatMethodeAvecLargeur(...)** : formate une méthode sur plusieurs lignes selon la largeur disponible et le mode d'affichage.
- **formatMethode(...)** : limite le nombre de paramètres affichés en mode condensé.
- **getAttributsAffichage()** : retourne la liste d'attributs à afficher selon le mode plein écran ou condensé.
- **getMethodesAffichage()** : retourne la liste de méthodes à afficher avec formatage et limitation éventuelle.
- **contient(int px, int py)** : vérifie si un point donné se situe à l'intérieur du bloc.

- **chevaucheTexte(...)** : détecte si un rectangle de texte chevauche la zone du bloc.
- **deplacer(int dx, int dy)** : déplace le bloc en modifiant ses coordonnées X et Y.

Package vue.liaison

Le package **liaison** gère tout ce qui concerne l'affichage et le calcul des liaisons entre les blocs de classes.
Il est composé de plusieurs classes helper qui délèguent les responsabilités spécifiques.

LiaisonVue

Cette classe gère **l'affichage visuel des liens** entre deux `BlocClasse`.
Elle calcule les chemins orthogonaux, gère les ancrages, détecte les intersections et dessine les flèches et multiplicités UML.

Attributs principaux

- `id : UUID` - Identifiant unique de la liaison
- `type : String` - Type de liaison (heritage, association_uni, association_bi, interface)
- `blocOrigine : BlocClasse` - Bloc source de la liaison
- `blocDestination : BlocClasse` - Bloc destination de la liaison
- `ancrageOrigine : Point` - Point d'ancrage sur le bloc origine
- `ancrageDestination : Point` - Point d'ancrage sur le bloc destination
- `roleOrig : String` - Rôle côté origine
- `roleDest : String` - Rôle côté destination
- `multOrig : String` - Multiplicité côté origine
- `multDest : String` - Multiplicité côté destination
- `unidirectionnel : boolean` - Indique si l'association est unidirectionnelle

LiaisonVue(...) (constructeurs)

Initialise une liaison avec les blocs origine/destination, le type et optionnellement les multiplicités.
Appelle automatiquement `chooseBestSides()` pour calculer le meilleur chemin.

chooseBestSides()

Choisit les meilleurs côtés et positions d'ancrage pour minimiser le nombre de segments et éviter les collisions.
Teste toutes les combinaisons de côtés et positions, évalue chaque chemin et sélectionne le meilleur selon plusieurs critères :
- **Nombre de segments** (critère principal)
- **Absence d'accordéons** (allers-retours)
- **Distance totale** du chemin

Pour les liaisons multiples entre les mêmes blocs, applique un décalage pour éviter la superposition.

calculerCheminOptimal()

Recalcule le chemin de la liaison en utilisant `CalculateurChemin` .
Gère la détection des intersections avec les autres liaisons et applique le rendu avec des ponts si nécessaire.

dessiner(Graphics2D g, double zoom, int panX, int panY)

Dessine la liaison sur le panneau diagramme avec zoom et panoramique.
Trace les segments du chemin, ajoute les flèches selon le type (héritage, interface, association), affiche les multiplicités et rôles.

Méthodes secondaires

- **setAncrageOrigine(...)** : définit le point d'ancrage origine
- **setAncrageDestination(...)** : définit le point d'ancrage destination
- **setTousLesBlocs(...)** : met à jour la liste des blocs pour la détection d'obstacles
- **setToutesLesLiaisons(...)** : met à jour la liste des liaisons pour la détection d'intersections
- **getId()** : retourne l'identifiant unique
- **getType()** : retourne le type de liaison
- **getBlocOrigine()**, **getBlocDestination()** : retournent les blocs liés
- **getRoleOrig()**, **getRoleDest()** : retournent les rôles
- **getMultOrig()**, **getMultDest()** : retournent les multiplicités
- **setRoleOrig(...)**, **setRoleDest(...)** : modifient les rôles
- **setMultOrig(...)**, **setMultDest(...)** : modifient les multiplicités
- **isUnidirectionnel()** : indique si la liaison est unidirectionnelle

CalculateurChemin

Cette classe calcule les **chemins orthogonaux** pour les liaisons entre blocs.
Elle génère des chemins composés de segments horizontaux et verticaux uniquement, en évitant les obstacles.

Attribut

- `detecteurObstacles : DetecteurObstacles` - Instance pour détecter les collisions

creerCheminOrthogonal(Point debut, Point fin, int coteDebut, int coteFin)

Crée un chemin orthogonal entre deux points en fonction des côtés de sortie et d'entrée.
Système de côtés : 0=HAUT, 1=DROITE, 2=BAS, 3=GAUCHE

Algorithme :

1. Calcule les points de sortie et d'entrée avec une marge interne (30 pixels)
2. Détecte les côtés opposés (haut-bas, gauche-droite) pour optimiser
3. Si alignés parfaitement, trace une ligne droite
4. Sinon, calcule des points intermédiaires pour créer un chemin en escalier
5. Nettoie les points redondants (3 points alignés → 2 points)

Retourne : Liste de points formant le chemin

Méthodes

calculerLongueurChemin(List<Point> chemin)

Calcule la longueur totale d'un chemin en additionnant les distances entre chaque point.
Retourne : Distance totale en pixels

cheminADesCollisions(List<Point> chemin)

Vérifie si un chemin traverse des blocs obstacles en testant chaque segment.
Retourne : `true` si collision détectée, `false` sinon

Méthodes secondaires

- **calculerPointDecale(Point pt, int cote, int margeInterne)** : calcule un point décalé selon le côté et la marge
- **nettoyerPointsRedondants(List<Point> chemin)** : supprime les points intermédiaires alignés
- **eviterCoin(int coord1, int coord2, int reference, boolean vertical)** : calcule une position qui évite les coins de blocs

DetecteurObstacles

Cette classe détecte les **obstacles** (autres blocs) pour le routage des liaisons.
Elle permet de vérifier si un segment traverse un bloc et d'obtenir la liste des obstacles sur un trajet.

Attributs

- `blocOrigine : BlocClasse` - Bloc source (ignoré dans la détection)
- `blocDestination : BlocClasse` - Bloc destination (ignoré dans la détection)
- `tousLesBlocs : List<BlocClasse>` - Liste de tous les blocs du diagramme

aUnObstacle(boolean estHorizontal, int a1, int a2, int b)

Détecte si un segment horizontal ou vertical traverse des obstacles.

Paramètres :

- `estHorizontal` : true pour segment horizontal, false pour vertical
- `a1, a2` : coordonnées de début et fin sur l'axe principal
- `b` : coordonnée sur l'axe perpendiculaire

Retourne : `true` si obstacle détecté

aObstacleHorizontalStrict(int x1, int x2, int y)

Détection **stricte** : vérifie qu'une ligne horizontale ne traverse **aucun** bloc.
Plus restrictif que `aUnObstacle`, utilisé pour les vérifications finales.

Retourne : `true` si collision détectée

aObstacleVerticalStrict(int x, int y1, int y2)

Détection **stricte** : vérifie qu'une ligne verticale ne traverse **aucun** bloc.

Retourne : `true` si collision détectée

Méthodes secondaires

- **getObstaclesSurLigneHorizontale(int x1, int x2, int y)** : retourne la liste des blocs traversés par une ligne horizontale
- **getObstaclesSurLigneVerticale(int x, int y1, int y2)** : retourne la liste des blocs traversés par une ligne verticale
- **detecterCollisionAvecBloc(...)** : vérifie si un point est à l'intérieur d'un bloc avec marge

GestionnaireAncrage

Cette classe gère les points **d'ancrage** et les positions sur les blocs de classes.
Elle permet de calculer les points pour les liaisons, les multiplicités et les rôles.

Méthodes

getPointSurCote(BlocClasse bloc, int cote, double posRel)

Renvoie un point sur un côté d'un bloc.

Paramètres :

- `bloc` : Bloc cible
- `cote` : Côté (0=HAUT, 1=DROITE, 2=BAS, 3=GAUCHE)
- `posRel` : Position relative sur le côté (0.0 à 1.0)

Retourne : Point exact sur le côté

getCoteLePlusProche(Point souris, BlocClasse bloc)

Renvoie le côté le plus proche d'un point donné (pour les interactions souris).

Retourne : Numéro du côté le plus proche

getPosRelativeDepuisSouris(Point souris, BlocClasse bloc, int cote)

Calcule la position relative d'un point sur un côté du bloc (pour placer un ancrage).

Retourne : Position normalisée entre 0.0 et 1.0

Méthodes secondaires

- **estSurAncrage(Point ancrage, Point souris, ...)** : vérifie si la souris est sur un point d'ancrage
- **calculerPositionMultiplicite(Point a, int cote, ...)** : calcule où afficher la multiplicité
- **calculerPositionRole(Point a, int cote, ...)** : calcule où afficher le rôle
- **calculerPrioriteCentre(double pos)** : détermine la priorité d'une position pour le centrage

GestionnaireIntersections

Cette classe gère la **détection des intersections** entre liaisons.

Elle permet de détecter les croisements et les chevauchements pour gérer le rendu avec des ponts.

getIntersectionSegment(Point a1, Point a2, Point b1, Point b2)

Détecte si deux segments orthogonaux se croisent et retourne le point d'intersection.

Retourne : Point d'intersection ou `null` si pas d'intersection

cheminsPartagentSegments(List<Point> chemin1, List<Point> chemin2)

Vérifie si deux chemins partagent des segments communs (chevauchement).

Retourne : `true` si les chemins se chevauchent

Méthodes secondaires

- **segmentsSeChevauchent(Point a1, Point a2, Point b1, Point b2)** : vérifie si deux segments sont colinéaires et se chevauchent

RenduLiaison

Cette classe gère le **rendu graphique** des liaisons.

Elle dessine les lignes avec ponts aux intersections, les flèches et les symboles UML.

Attribut

- `gestionnaireIntersections : GestionnaireIntersections` - Instance pour gérer les intersections

Méthodes

dessinerLigneAvecPonts(Graphics2D g, Point p1, Point p2, List<Point> intersections, Stroke traitNormal)

Dessine une ligne avec des **ponts** (arcs) aux points d'intersection.
Les ponts permettent de visualiser qu'une liaison passe par-dessus une autre.

dessinerFlecheVide(Graphics2D g, Point a, int s)

Dessine une **flèche vide** (triangle non rempli) pour l'héritage ou l'interface.
Système de côtés : 0=HAUT, 1=DROITE, 2=BAS, 3=GAUCHE

dessinerFlecheAssociation(Graphics2D g, Point a, int s)

Dessine une **flèche d'association** (2 lignes formant un V) pour les associations unidirectionnelles.

BarreMenus

Cette classe représente la **barre de menu principale** de l'application.
Elle fournit un **accès** aux **outils d'affichage**, d'**édition**, de **gestion des fichiers** et à l'**aide**.
Hérite de **JMenuBar** pour être directement intégrée dans la fenêtre principale.

Attributs principaux

- `fenetrePrincipale` : référence à la fenêtre principale.
- items de menu (`JCheckBoxMenuItem`) : options d'affichage et de sauvegarde.

BarreMenus (constructeur)

Initialise la barre de menu avec :
- Couleur de fond personnalisée,
- Menus principaux (**Fichier**, **Affichage**, **Aide**),
- Style graphique (couleurs, police, opacité),
- Liaison des actions aux événements des items de menu.

Méthodes secondaires

- **creerMenuFichier()** : crée le menu "Fichier" avec les options Ouvrir projet, Exporter en image, Sauvegarder, Quitter.
- **creerMenuAffichage()** : crée le menu "Affichage" avec options Afficher attributs, Afficher méthodes, Optimiser positions.
- **creerMenuAide()** : crée le menu "Aide" avec l'item À propos.
- **actionOuvrirProjet()** : ouvre un dialogue pour sélectionner un projet et initie sa vérification et son chargement.
- **verifierFichiersProjet(String cheminFichier)** : affiche un message d'avertissement si des fichiers invalides sont détectés.
- **sauvegardeProjetXml(String cheminFichier)** : transmet le chemin au contrôleur pour charger le projet.
- **actionAffichageAttributs()** : met à jour l'affichage des attributs dans la fenêtre principale.
-

actionAffichageMethodes() : met à jour l'affichage des méthodes dans la fenêtre principale.

- **actionSauvegardeAuto()** : active ou désactive la sauvegarde automatique.
- **actionOptimiser()** : optimise la position des classes et des liaisons dans le diagramme.
- **actionOptimiserLiaisons()** : optimise uniquement la position des liaisons.
- **actionSauvegarder()** : déclenche la sauvegarde manuelle du projet.
- **actionAPropos()** : affiche une boîte de dialogue HTML avec les auteurs et informations sur le projet.

Package vue.role_class

FenetreChangementMultiplicite

Hérite de **JFrame**. Fenêtre pour modifier les multiplicités d'un diagramme UML.

PanneauModif

Cette classe représente le **panneau de modification des multiplicités** d'une classe dans le diagramme UML.

Elle permet à l'utilisateur de sélectionner une liaison, saisir les multiplicités minimum et maximum, puis de valider ou annuler les changements.

Elle agit comme **interface entre l'IHM et les objets LiaisonVue**, en appliquant les changements directement sur le diagramme.

Attributs principaux

- `blocSelectionne` : Bloc actuellement sélectionné.
- `panDiag` : Référence au panneau de diagramme principal.
- `listeLiaisonsIHM` : Liste des liaisons affichées dans l'IHM.
- `txtMultipliciteMin` : Champ de texte pour la multiplicité minimum.
- `txtMultipliciteMax` : Champ de texte pour la multiplicité maximum.

- `btnValider` : Bouton pour valider les modifications.
- `btnAnnuler` : Bouton pour annuler les modifications.

Méthodes secondaires

- **caractereValideMultMin(String min)** : vérifie si min est un entier.
- **caractereValideMultMax(String max)** : vérifie si max est un entier.
- **actionPerformed(ActionEvent e)** : gère les clics sur les boutons valider et annuler.
- **getLiaisonConnectees(BlocClasse blcClasse)** : renvoie la liste des liaisons connectées à la classe donnée.

FenetreModifRole

Cette classe représente la **fenêtre de modification des rôles** dans le diagramme UML.
Elle encapsule le panneau `PanneauModifRole` et sert de conteneur Swing pour permettre à l'utilisateur de modifier les rôles d'une liaison.
Elle constitue la **fenêtre modale pour l'édition des rôles**, centrée sur l'écran et de taille fixe.

Attributs principaux

- `panDiag` : Référence au panneau de diagramme principal.
- `PanneauModifRole` : Instance du panneau de modification des rôles.

FenetreModifRole (constructeur)

Initialise la fenêtre avec un titre, une taille fixe et une position centrée à l'écran. Crée le panneau `PanneauModifRole` et l'ajoute au centre de la fenêtre via `BorderLayout`.

PanneauModifRole (constructeur)

Initialise le panel, configure le titre et le nom du bloc sélectionné, crée le champ de texte pour le rôle, installe les boutons Valider/Annuler et le panel scrollable contenant la liste des liaisons. Remplit la liste des liaisons avec séparation **Associations / Interfaces** et installe les listeners pour les boutons et radio-boutons.

Méthodes

remplirListeLiaisons

Parcourt toutes les liaisons du diagramme et ajoute dans le panel uniquement celles **liées au bloc sélectionné**, séparées en **Associations et Interfaces**. Chaque liaison est représentée par un **radio-bouton** qui remplit le champ du rôle lorsqu'il est sélectionné.

ajouterLiaisonALaListe

Crée une ligne avec un radio-bouton pour une liaison donnée. Si le bloc sélectionné est l'origine, affiche le rôle origine ; si le bloc sélectionné est la destination, affiche le rôle destination. Ajoute le radio-bouton à un `ButtonGroup` pour assurer une sélection unique.

actionPerformed

Gère les actions des boutons Valider et Annuler :
- **Valider** : récupère le rôle saisi et modifie la liaison correspondante via `panDiag.modifierRole(...)`, puis rafraîchit le diagramme et ferme la fenêtre.
- **Annuler** : ferme simplement la fenêtre sans modifier la liaison.

Méthodes secondaires

- **getLiaisonSelectionnee()** : retourne la liaison actuellement sélectionnée.
- **rafraichirPanel()** : force la mise à jour graphique du panel et de ses composants Swing.

Projet académique – IUT du Havre

SAE 3.01 – Outil de rétroconception Java-UML

Romain BARUCHELLO, Jules BOUQUET, Pierre COIGNARD, Paul NOEL, Thibault PADOIS, Hugo VARAO GOMES DA SILVA