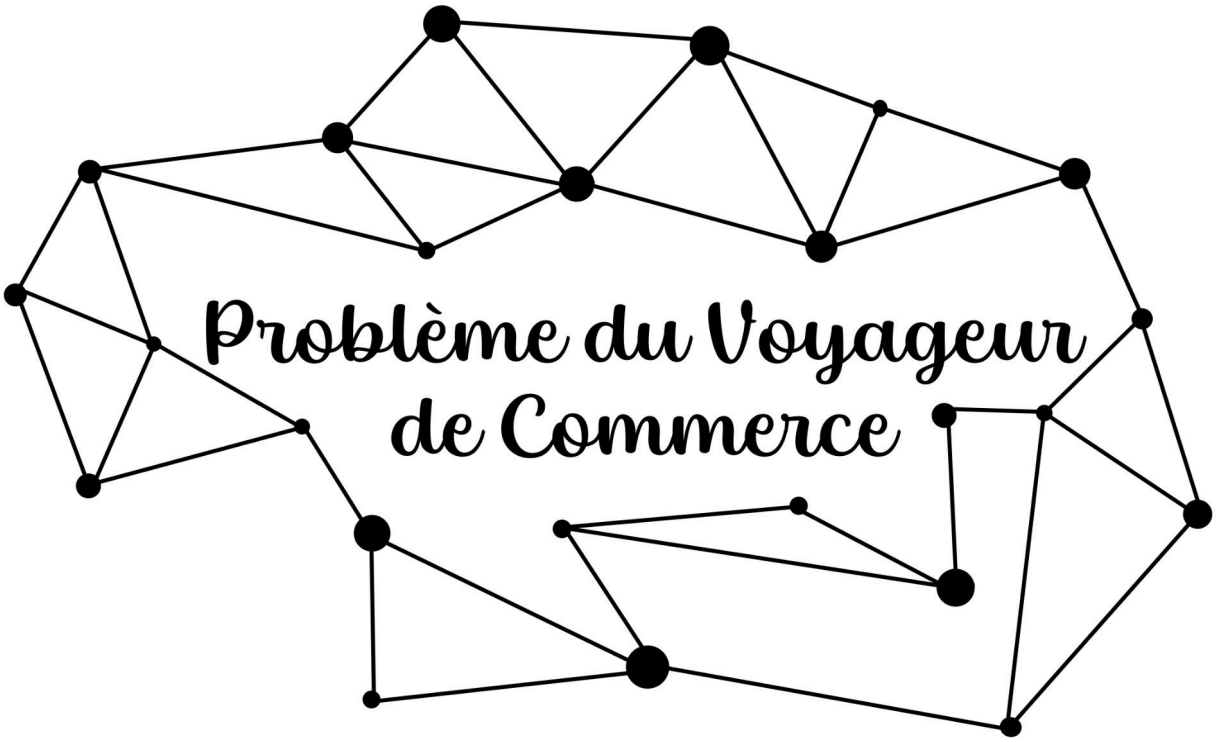
Ekaterina Galkina et Lou Toubiana

RAPPORT

Projet pour Voyageurs et Voyageuses



Sommaire

Structure du code

Convention

Pour la meilleure lisibilité du rapport :

nomPackage - gras vert

NomClasse - gras

méthode - Italique violet

Programme principal

Le package **main** contient uniquement la classe **Main** avec le programme principal.

Implémentation des personnes et des villes

Gestion des personnes

Pour gérer les personnes présentes dans notre base de données, nous avons créé un package nommé **peuple**. Ce package contient les classes suivantes :

1. **Personne** :
- Il s'agit d'une classe abstraite qui constitue la classe mère de toutes les autres classes représentant des personnes dans la base de données.
- Elle contient des attributs communs à toutes les personnes, ainsi que des méthodes générales applicables à l'ensemble des personnes.
2. **Titulaire** :
- C'est également une classe abstraite qui hérite de **Personne**.
- Elle ajoute deux attributs spécifiques :

- discipline : la spécialité académique du titulaire.

- numeroDeBureau : le numéro du bureau attribué.

    - Elle contient aussi des méthodes spécifiques liées aux titulaires.

3. **MCF** (Maître de Conférences) et **Chercheur** :

- Ces deux classes héritent de **Titulaire** et implémentent des comportements spécifiques.
- La différence entre ces classes :

    - Un **MCF** peut encadrer **un seul étudiant** au maximum.

    - Un **Chercheur** peut encadrer **un nombre illimité d'étudiants**.

4. **Etudiant** :

- Cette classe hérite directement de **Personne**. Elle représente les étudiants de la base de données.

5. **Discipline** :

- Il s'agit d'une classe d'énumération qui contient la liste des disciplines possibles pour les personnes de notre base de données.
- Elle fournit également des méthodes spécifiques pour manipuler et représenter ces disciplines.

Vous pouvez retrouver dans les documents annexes le schéma qui met en lumière les relations entre ces différentes classes.

Gestion du territoire

Pour traiter les villes, nous avons créé un package nommé **territoire**. Ce package contient :

1. **Ville** :

- Cette classe représente une ville avec ses attributs spécifiques (nom, code de département, etc.).
- Elle fournit également des méthodes pour récupérer les distances entre villes, notamment une méthode qui renvoie un dictionnaire de distances décrit précédemment dans le cadre de l'implémentation de l'algorithme génétique.

2. **Region** :

- Il s'agit d'une énumération représentant les régions de France.

- Cette énumération contient des méthodes spécifiques pour manipuler et représenter les régions.

Procédure de stockage de la base de données des personnes et des villes

Dans notre projet, nous avons un package nommé **baseDeDonnees** qui contient les classes nécessaires pour charger et stocker les données des personnes et des villes.

Chargement des données

1. Pour les personnes :
- Les données sont stockées dans un fichier sous forme d'un objet Liste de Personnes, sérialisé en flux d'octets.
- Une classe nommée **ChargementDonnees** dans ce package permet de charger ces données depuis le fichier et de les manipuler.

2. Pour les villes :
- Les données sont récupérées directement depuis une base de données PostgreSQL.

Liste spécifique pour éviter les warnings

Le package **baseDeDonnees** contient également une classe nommée **ListeEcosysteme**, qui est une liste chaînée spécifique au type **Personne**.

Pourquoi avons nous créé cette classe ?
Même si nous utilisions principalement des ArrayList**<Personne>** fournies par Java dans notre code, ces objets génériques posent un problème lors du passage en flux d'octets pour la sérialisation. En effet, lors de la lecture des données depuis un fichier, le type générique s'efface, ce qui génère des warnings.

Pour éviter ce problème, nous avons implémenté notre propre structure de données **ListeEcosysteme**. Cette liste chaînée spécifique au type **Personne** propose les méthodes suivantes :

1. *charger* les données depuis une ArrayList**<Personne>** vers notre structure **ListeEcosysteme**.
2. *stockerEcosysteme* (ListeÉcosystème) dans un fichier nommé **ecosysteme**.
3. *recupererEcosysteme* depuis le fichier sérialisé, en transformant l'objet **ListeEcosysteme** en ArrayList**<Personne>**.

Implémentation de l'algorithme génétique.

Afin de calculer le meilleur chemin à partir d'une liste de villes à visiter, nous avons utilisé l'algorithme génétique proposé dans le sujet.
Toute la logique de cet algorithme est implémentée dans un package nommé **genetique**.
Ce package contient les éléments suivants :

- **Chemin** - classe qui représente une liste de villes à parcourir dans un ordre précis
- **Individu<E>** - classe abstraite, représentant l'individu dans la logique de l'algorithme génétique
- **Mutation<E>** - interface qui implémente la logique de la mutation d'un individu
- **Croisement<E>** - interface qui implémente la logique du croisement.
- **Population<E>** - classe qui représente la population dans le cadre de l'algorithme génétique
- **Selection<E>** - interface qui implémente l'algorithme génétique lui-même.

Vous pouvez retrouver dans les documents annexes le schéma qui met en lumière les relations entre ces différentes classes.

Voici la description plus précise des ces différentes classes :

Tout d'abord, nous avons une classe abstraite appelée **Individu<E>**, de type générique.
Cette classe représente un individu à l'aide de deux attributs principaux :

1. Gene : une liste de valeurs de type générique, représentant les caractéristiques de l'individu.
2. Score : une valeur qui mesure la qualité de cet individu.

Cette classe est abstraite, car à ce niveau, il est impossible de calculer le score, puisque nous n'avons pas encore d'informations précises sur le type générique de l'individu.

Cependant, nous pouvons y définir quelques méthodes générales, applicables à tous les individus pour la conception de l'algorithme :

- Un comparateur par score pour classer les individus en fonction de leur qualité.
- Les opérations de croisement, qui consistent à obtenir deux individus enfants à partir de deux parents.
- Les opérations de mutation, qui modifient la gène d'un individu donné.
- Le lancement de l'algorithme génétique à partir d'un individu : cela consiste à générer une population initiale à partir d'un individu donné, puis à lancer l'algorithme génétique sur cette population (qui est défini dans l'interface **Selection<E>**).

Cette structure permet de séparer les méthodes propres à la population de ceux liée à un individu spécifique.

La classe **Individu<E>** implémente également deux interfaces de type générique également : **Croisement<E>** et **Mutation<E>**. Comme leurs noms l'indiquent, ces interfaces définissent les méthodes nécessaires pour effectuer les opérations de croisement et de mutation sur un individu.

Par ailleurs, nous avons l'interface **Selection<E>**, de type générique. Elle décrit la procédure générale de l'algorithme génétique et impose à toute classe qui l'implémente de définir ses propres méthodes suivantes :

- Meilleur individu pour identifier le meilleur individu de la population.
- Fonction qui renvoie la fitness de la population qui mesure la qualité de la population.

- Remplacement qui remplace une certaine partie des moins bons individus de l'ancienne population, par les meilleurs de la nouvelle population générée.
- Sélection aléatoire qui sélectionne deux parents à partir d'une population (les parents avec un meilleur score ont plus de chances d'être tirés).

Cette interface a pour principal objectif de séparer la logique de l'algorithme génétique de l'implémentation spécifique de la population.

Ensuite, nous avons une classe **Population<E>** (qui implémente l'interface **Selection<E>**), également de type générique, cette classe implémente des méthodes abstraites de son interface, permettant ainsi d'exécuter l'algorithme génétique. La population est représentée par :

1. Une liste d'individus de type **Individu<E>** (la classe décrite plus haut).
2. Une valeur de fitness, mesurant la qualité globale de la population, dans notre cas, c'est le score du meilleur individu de la population.

Adaptation à notre problème : recherche du plus court chemin

Dans notre cas, pour résoudre le problème du plus court chemin, nous avons besoin d'une classe qui hérite de **Individu<E>**, mais avec un type générique spécifique, à savoir **Ville**. Cette classe, appelée **Chemin**, définit toutes les méthodes abstraites de la classe **Individu<E>** :

- *getScore()* : Cette méthode calcule le score comme étant la distance totale du chemin, incluant la distance entre la dernière ville et la première, car nous cherchons à parcourir toutes les villes et revenir à la ville initiale.
- *cloner()* : Pour cloner un individu (comme la classe Individu est abstraite on ne peut pas instancier un objet de ce type, donc on ne pouvait pas définir cette méthode avant)
- *InitialiserPopulation()* : Pour initialiser une population d'individus à partir d'un individu donné. Cette méthode est en soi un générateur d'un certain nombre de permutations des éléments d'une gêne d'un individu donné. Dans notre cas, nous

avons décidé de choisir d'utiliser la méthode par insertion aléatoire pour obtenir ces permutations, qui est simple à implémenter.

Cette méthode consiste à générer des permutations aléatoires des villes en procédant comme suit :

- Le premier élément est placé directement.
- Chaque élément suivant est inséré soit avant, entre, ou après les éléments déjà présents, selon un choix aléatoire.

La classe **Chemin** redéfinit également l'algorithme génétique pour le lancement correct. Voici les étapes principales :

- Initialisation d'un **Chemin** à partir d'un ensemble de villes à visiter.
- Initialisation de deux chemins enfants comme espaces de stockage pour les individus générés lors du croisement. Étant donné que **Individu<E>** est abstrait, il est impossible d'en instancier directement une instance : les chemins enfants sont donc passés en paramètre pour servir de stockage temporaire.
- Remplissage du dictionnaire des distances entre les villes grâce à une méthode statique définie dans la classe **Ville** : *HashMap<Couple<Ville, Ville>, Double> setDictDistances(ArrayList<Ville> ensembleVilles)* (nous allons revenir sur le type **Couple** plus tard dans le rapport) qui récupère les données d'une base de données et calcule les distances entre chaque paire de villes à visiter.
- Lancement de l'algorithme génétique défini dans la classe mère **Individu<E>**, avec adaptation pour récupérer le meilleur chemin trouvé.

Attributs supplémentaires de la classe Chemin

La classe **Chemin** possède un attribut statique spécifique :

- **distance** : un dictionnaire qui stocke les distances entre chaque paire de villes. Ce dictionnaire utilise comme clé un objet de type **Couple<Ville, Ville>** (défini dans le package **objetsAuxiliaires**) et comme valeur un **Double** représentant la distance entre cette paire de villes. La classe **Couple<A, B>** a été conçue pour contenir exactement deux objets, sans tenir compte de l'ordre des éléments : c'est idéal pour

représenter des distances symétriques entre deux villes, cependant comme en Java il n’existe pas d’un type comme ceci, nous avons décidé de l'implémenter nous mêmes.

Implémentation de filtrage des personnes

Pour faciliter la sélection des personnes que l'utilisateur souhaite visiter, nous avons mis en place plusieurs méthodes de tri basées sur différents critères, tels que l'âge, l'identifiant (ID), la ville, etc. Ces méthodes offrent une flexibilité accrue dans le filtrage des données, permettant ainsi une navigation efficace dans la base de données.

La classe **Trie** regroupe les différentes méthodes de tri. La plupart d'entre elles prennent en paramètre un ArrayList<Personne> et renvoient une liste filtrée contenant uniquement les personnes répondant aux critères spécifiés. Cette approche centralise les opérations de filtrage.

La classe **TrieTerminal** quant à elle, implémente trois méthodes principales qui exploitent celles de la classe **Trie** tout en offrant une interaction directe avec l'utilisateur via le terminal. Grâce à ces méthodes, l'utilisateur peut sélectionner des personnes en fonction des critères définis, tout en bénéficiant d'une expérience interactive.

Pour assurer la robustesse des interactions dans le terminal, les méthodes de **TrieTerminal** s'appuient sur celles de la classe **VerificationTrieTerminal**, qui a pour rôle de valider les entrées utilisateur. Par exemple, elle vérifie si les valeurs saisies sont cohérentes avec les attentes (vérification qu’un nombre est entré là où attendu, qu’un nom de ville existe dans la base de données, etc.). Cette vérification réduit les risques d'erreur et améliore la fiabilité globale du système.

Modification de la base de données des personnes

Nous avons mis en place des mécanismes permettant à l’utilisateur de modifier la base de données tout en garantissant la cohérence des données.

Modification via le terminal

Une classe dédiée a été créée pour permettre la modification de la base de données à travers le terminal. Cette classe propose les méthodes statiques suivantes :

- *modifier* : Cette méthode interagit avec l'utilisateur pour lui donner la possibilité de modifier la base de données. Elle repose sur plusieurs méthodes auxiliaires :

    - *ajouter* : Permet d'ajouter une ou plusieurs personnes.

    - *changer* : Permet de modifier une ou plusieurs caractéristiques d'une ou plusieurs personnes.

    - *supprimer* : Permet de supprimer une ou plusieurs personnes de la base de données.

- *actualiser* : Permet de sauvegarder la base de données dans un fichier dès que tous les changements ont été apportés.

Règles de cohérence des données

Afin d'assurer la cohérence des données, les règles suivantes doivent être respectés :

1. Discipline d'un étudiant :
- Chaque étudiant possède exactement une seule discipline.
- Cette discipline doit obligatoirement appartenir à l'ensemble des disciplines de son titulaire.
2. Relation entre étudiant et titulaire :
- Un étudiant doit obligatoirement avoir un titulaire (soit un MCF, soit un chercheur).
3. Nombre d'étudiants par titulaire :
- Un MCF peut avoir zéro ou un étudiant au maximum.
- Un chercheur peut avoir zéro ou plusieurs étudiants (il n'y a pas de limite).
4. Disciplines d'un titulaire :

- Un titulaire peut avoir entre une et deux disciplines.

Gestion des règles lors de la modification des données

Pour garantir le respect des règles de cohérence des données, nous avons mis en place des procédures strictes lors de l'ajout, la modification et la suppression des personnes dans la base de données.

Ajout d'une personne

1. Ajout d'un titulaire :

L'utilisateur peut attribuer des étudiants au titulaire immédiatement après l'ajout, mais cela reste optionnel.

2. Ajout d'un étudiant :

L'utilisateur doit obligatoirement choisir un titulaire disponible tel que la discipline de l'étudiant soit incluse dans l'ensemble des disciplines du titulaire.

Si aucun titulaire compatible n'existe dans la base de données, l'ajout de l'étudiant est bloqué.

Modification des caractéristiques d'une personne

1. Règles générales :

ID et fonction (ex. MCF, Étudiant) d'une personne ne peuvent pas être modifiés. Dans ce cas, pour changer un MCF en Étudiant, il faut supprimer la personne existante et la réajouter avec la nouvelle fonction.

Les valeurs spécifiques sont contrôlées :

- Année de thèse : doit être compris entre 1 et 3.
- Numéro de bureau : doit être un entier strictement positif.

2. Subtilités pour les disciplines :

Modification de la discipline d'un étudiant :

- Le programme réattribue automatiquement l'étudiant à un autre titulaire compatible si la nouvelle discipline n'est plus dans l'ensemble des disciplines de son titulaire actuel.
- Si aucun titulaire compatible n'est trouvé, le changement de discipline est bloqué.

Modification des disciplines d'un titulaire :

- Si les nouvelles disciplines ne sont plus compatibles avec certains de ses étudiants, ces derniers sont réattribués automatiquement aux titulaires disponibles et compatibles dans la base de données.
- Si une réattribution n'est pas possible, le changement des disciplines est bloqué.

3. Modification des étudiants en thèse d'un titulaire :

Si l'utilisateur souhaite remplacer un étudiant par un autre ou juste tout simplement de le retirer de sa liste des étudiants en thèse, l'ancien étudiant est automatiquement réattribué à un autre titulaire compatible.

Si aucun titulaire compatible n'existe, le programme bloque le changement.

4. Changement d'encadrant pour un étudiant :

Le processus est plus simple : comme le programme propose que les titulaires compatibles et qui peuvent prendre en thèse encore un étudiant sans suppression de l'ancien étudiant, il suffit d'ajouter l'étudiant dans la liste des étudiants du nouveau titulaire.

Suppression d'une personne

1. Suppression d'un étudiant :

Il suffit de retirer l'étudiant de la liste des étudiants en thèse de son encadrant.

2. Suppression d'un titulaire :

Cette opération est plus délicate, car il faut s'assurer que tous les étudiants en thèse du titulaire peuvent être réattribués à d'autres titulaires compatibles.

Si cela n'est pas possible, la suppression est bloquée.

3. Cas spécifique des étudiants d'un chercheur :

Étant donné qu'un chercheur peut avoir plusieurs étudiants, le processus est complexe puisque nous ne pouvons pas savoir directement si nous pouvons attribuer tous ses étudiants en thèse aux autres titulaires.

Première approche que nous avons implémentée, mais elle n'est plus utilisée : Pour chaque étudiant, une liste des titulaires disponibles et compatibles est proposée à l'utilisateur afin qu'il fasse un choix.

- Problème : L'utilisateur n'ayant pas une vue globale de la base de données, il peut arriver qu'au final, certains étudiants n'aient plus de titulaire. Cela bloque la suppression en cours.

Solution implémentée :

- Une méthode *essaieSupprimerTousEtudiants* tente de réattribuer tous les étudiants d'un titulaire à d'autres titulaires compatibles.
- Si la réattribution réussit, la méthode renvoie true et réalise les changements.
- Si elle échoue, elle renvoie false, n'effectue aucune modification et stocke dans des listes temporaires tous les associations étudiant-titulaire trouvées.

4. Choix de l'utilisateur en cas d'échec :

Si la suppression d'un chercheur n'est pas possible, l'utilisateur a le choix entre :

- Garder la même liste des étudiants pour le chercheur.
- Supprimer le maximum d'étudiants possibles, si au moins une association étudiant-titulaire a été trouvée à l'aide de la dernière fonction.

Résolution de l’attribution des étudiants aux titulaires

Ce problème consiste à attribuer un ensemble d’étudiants à un ensemble de titulaires tout en respectant les contraintes de compatibilité des disciplines.

Méthode de résolution

1. Parcours initial des titulaires :

Nous parcourons la liste des titulaires pour identifier :

- Les chercheurs compatibles avec les disciplines des étudiants.

  -> Pour chaque chercheur trouvé, nous stockons :

  - Le chercheur dans une liste (ArrayList<**Titulaire**>).

  - Les étudiants compatibles dans un ensemble associé (ArrayList<Set<**Etudiant**>>).

  - Les MCF disponibles et compatibles avec les disciplines des étudiants non encore attribués.

2. Si pas tous les étudiants ont pas été attribués aux chercheurs:

Si à la fin du parcours, il reste des étudiants sans chercheurs compatibles, nous essayons de les attribuer aux MCF disponibles stockés.

Cela se ramène à un problème de couplage maximal dans un graphe biparti non orienté :

- Premier ensemble : étudiants restants à attribuer.
- Deuxième ensemble : MCF disponibles.
- Arêtes : relations de compatibilité des disciplines entre les étudiants et les MCF.

Ce graphe est représenté à l’aide d’un schéma (voir annexe) qui illustre un bon couplage et un mauvais couplage.

Implémentation de l’algorithme

Pour résoudre ce problème, nous avons utilisé un algorithme vu en cours d'algorithmes dans les graphes, qui repose sur le fait de sélectionner des couples pour les sommets ayant le moins de voisins.

L'algorithme est implémenté dans la classe **GrapheBiparti** située dans le package **objetsAuxiliaires**. Cette classe possède les attributs suivants :

- Premier ensemble : stable de type générique.
- Deuxième ensemble : stable de type générique.
- Dictionnaire des voisins : associe à chaque sommet du premier ensemble l'ensemble des sommets voisins dans le deuxième ensemble.

Les méthodes principales de cette classe sont :

- *triParSuccesseur* : trie les sommets selon leur nombre de successeurs.
- *couplageMaximal* : tente de trouver le maximum de couples compatibles dans le graphe biparti donné.

Modification de la base de données via l'interface graphique

Dans la version du programme utilisant une interface graphique, les mêmes règles sont appliquées et les mêmes algorithmes sont utilisés pour garantir la cohérence et la compatibilité des données, juste les fichiers sont organisés différemment. Le package **affichage** contient un sous-package **affichageEcosysteme**, où se trouvent les classes permettant à l'utilisateur d'observer et de modifier la base de données.

Ce sous-package contient quatre classes principales :

1. **PanelEcosysteme** :

- Cette classe définit un panneau permettant d'observer la base de données sous la forme d'un tableau des personnes.
- En bas de ce tableau, une section d'informations supplémentaires affiche les détails de la personne sélectionnée.
- Cette section d'informations est définie dans une autre classe appelée **InfoPanel**.

2. **InfoPanel** :

- Ce panneau affiche des informations détaillées sur la personne sélectionnée.
- Boutons d'action : À gauche du tableau et des informations supplémentaires, l'utilisateur dispose de trois boutons : Ajouter, Modifier, Supprimer

Activation/désactivation des boutons :

- Pour effectuer des opérations de modification ou de suppression, l'utilisateur doit sélectionner une personne dans la base de données.
- Si aucune personne n'est sélectionnée, les boutons restent désactivés.

3. **ModificationPersonne** :

- Cette classe définit une petite fenêtre de dialogue permettant de modifier les informations d'une personne sélectionnée.

4. **AjoutPersonne** :

- L'ajout d'une nouvelle personne est géré par un autre panneau de dialogue organisé en plusieurs étapes. Comme, la saisie des informations dépend de la fonction de la personne et de ses disciplines. Les disciplines saisies influencent les titulaires ou les étudiants proposés lors de l'ajout d'une personne avec la fonction correspondante.

Contrôle des entrées de l'utilisateur

Pour contrôler l'entrée de l'utilisateur à travers le terminal, nous avons mis en place un package nommé **verificationEntree**. Ce package contient deux classes principales :

- **Coherence** : permet de vérifier la cohérence des données saisies par l'utilisateur.
- **VerificationTriTerminale** : permet de corriger les entrées de l'utilisateur lors du tri dans le terminal.

Implémentation de l'Interface Graphique

Pour rendre notre projet plus intuitif et accessible, nous avons ajouté une version avec interface graphique en complément de la version terminal. Cette interface offre une meilleure lisibilité et une expérience utilisateur plus pratique. De plus, cette démarche nous a permis de découvrir et d'explorer les fonctionnalités de la bibliothèque Swing.

Nous avons conçu une architecture basée sur une classe principale, **Frame**, qui hérite de la classe **JFrame** de Swing. Cette classe constitue la fenêtre principale de l'application. En complément, nous avons implémenté plusieurs classes héritant de **JPanel** , facilitant la gestion de différents panneaux et améliorant la lisibilité ainsi que la modularité du code.

La classe **PanelTop**, qui hérite donc de la classe **JPanel**, représente un panneau fixe affiché en haut de la fenêtre. Il est présent sur toutes les autres interfaces, offrant une navigation cohérente et constante.

L'interface graphique est organisée dans un package nommé **affichage**, subdivisé en deux sous-packages :

- **affichageEcosystem** : Ce sous-package gère les fonctionnalités liées à la modification de la base de données. Ces aspects sont détaillés dans la section *Modification de la base de données via l'interface graphique* (p.15).
- **affichageAlgo** : Ce sous-package traite de la sélection des personnes à visiter et de l'affichage de l'itinéraire calculé à l'aide de l'algorithme génétique. Il inclut également deux sous-packages supplémentaires :
    - **choixPersonne** : Permet à l'utilisateur de sélectionner les personnes qu'il souhaite visiter.
    - **choixVille** : Permet de choisir les personnes en fonction de leur ville.

Certaines classes, comme **PanelNombre** et **PanelChoix**, jouent un rôle essentiel dans le tri des personnes. Bien qu'elles soient utilisées dans les deux sous-packages mentionnés, elles sont situées en dehors pour favoriser leur réutilisation et maintenir une structure organisée.

La classe **PanelFin** est dédiée à l'affichage des résultats finaux. Elle présente les informations des personnes sélectionnées ainsi que l'itinéraire optimal calculé. Cette classe exploite **PanelPersonneVilleDisplay**, un panneau interactif qui affiche les informations détaillées d'une personne (âge, fonction, etc.) lorsque l'utilisateur clique sur son nom. En bas de la fenêtre, l'itinéraire le plus court est également affiché.

Enfin, pour clarifier la navigation entre les différents panneaux, un schéma détaillant le passage d'un panneau à un autre est fourni en annexe.

Représentation des solutions

Nous avons également un package nommé **representationSolution**, qui contient la classe **FichierTexteGenerateur** et la classe **FichierHTMLGenerateur**.

Comme leur nom l'indique, ces classes permettent de représenter le chemin trouvé grâce à l'algorithme génétique et de le stocker dans un fichier texte ou html respectivement.

Tests JUnit

Afin d'assurer la robustesse et la fiabilité de notre application, nous avons mis en place une série de tests JUnit. Ces tests sont regroupés dans un package dédié nommé **TestsJUnit**, ce qui facilite leur identification et leur exécution.

Ce package contient trois classes principales de tests :

1. **CouplageMaximalTests**
   Cette classe teste une méthode qui, à partir de deux ensembles stables, calcule le couplage maximal entre les sommets. Le test vérifie que la méthode retourne correctement les paires optimales de sommets connectés et respecte les contraintes des ensembles stables.
2. **DisciplineTests**
   Cette classe teste la validation des disciplines saisies par les utilisateurs. Elle s'assure que seules les disciplines saisis correctement (par exemple, *informatique*,

mathématiques, etc.) sont acceptées. Ces tests permettent de valider la robustesse des vérifications dans les interactions avec l'utilisateur.

3. **TrieTests**
   Cette classe teste certaines méthodes de la classe **Trie**, notamment les fonctionnalités de tri en fonction de différents critères (âge, ville...). Les tests garantissent que les résultats retournés sont conformes aux attentes, même dans des cas particuliers (par exemple, des listes vides ou des critères non pertinents).

Malheureusement, lors de la génération de la documentation Javadoc, nous avons rencontré un problème : Javadoc ne reconnaît pas les modules JUnit. En conséquence, la génération échoue pour les classes de test. Ce problème est lié à une limitation de Javadoc avec les dépendances externes. Cependant, en commentant les extensions JUnit ajoutées dans le fichier module-info.java, ainsi que les fichiers du package **tests**, il est possible de générer le Javadoc.

Difficultés rencontrées

La condition d'arrêt pour l'algorithme génétique

Nous avons également rencontré des difficultés à définir une condition d'arrêt autre que le nombre d'itérations maximal atteint, pour déterminer à quel moment un individu peut être considéré comme « suffisamment bon » dans l'algorithme génétique. En l'absence d'une limite clairement définie, il était compliqué de savoir quelle valeur de seuil utiliser pour évaluer la fitness de la population.

Nous avons tenté de trouver des solutions en effectuant des recherches sur Internet, mais sans succès.

Finalement, nous avons décidé de définir une condition d'arrêt basée sur la fitness initiale de la population générée. Plus précisément, nous avons pris la valeur de score du meilleur individu de la première population générée. Ensuite, si un individu dans la population nouvellement générée atteint une valeur de fitness inférieure ou égale à cette fitness initiale divisée par deux (à une tolérance epsilon près), alors l'algorithme s'arrête.

Mise en place de la possibilité de changer la base de données des personnes

Comme décrit précédemment dans le rapport, de nombreux cas subtils de modification de la base de données ont été traités.

Et comme nos premières approches pour la résolution des certains problèmes, étaient fonctionnelles, mais n'étaient pas toujours optimales, nous avons donc dû repenser certains algorithmes afin d'obtenir de meilleures performances. Cette transition vers des solutions plus efficaces a pris du temps, mais nous a permis de proposer une version robuste de notre projet.

Implémentation de filtrage

Lors de l'implémentation du filtrage, l'un des principaux défis a été d'identifier tous les types de tri nécessaires et pertinents pour répondre aux besoins des utilisateurs. Un exemple concret de difficulté rencontrée est la mise en place d'un tri basé sur le code postal. Bien que cette idée semblait initialement intéressante, nous nous sommes rendu compte que les objets de type **Personne** dans notre base de données ne disposaient pas d'un attribut directement lié au code postal. Ce constat nous a conduit à réévaluer la pertinence de ce critère de tri et, finalement, à supprimer cette fonctionnalité.

La difficulté du projet

Le projet en lui-même n'était pas particulièrement difficile ; au contraire, il s'est révélé très intéressant et enrichissant. Cependant, nous avons choisi de le complexifier en ajoutant une interface graphique, ce qui a représenté un défi supplémentaire.

Cependant, cette décision nous a permis non seulement de maîtriser la création d'interfaces graphiques en Java, mais également de nous familiariser avec les subtilités de ce langage et obtenir des compétences pratiques pour nos futurs projets, donc c'était bien !

Les sources

- Permutations - Algorithmes et leurs performances (Utilisé pour trouver l'algorithme de génération de la population - permutations par insertions)

  http://villemin.gerard.free.fr/aNombre/MOTIF/PermutAl.htm

- Le cours des algorithmes dans les graphes de cette année pour la résolution du couplage maximal
- Différents forum pour swing :
  - https://www.jmdoudoux.fr/java/dej/chap-swing.htm
  - https://waytolearnx.com/2020/05/afficher-une-image-dans-un-jframe.html
  - https://waytolearnx.com/2020/05/jtextfield-java-swing.html
  - https://waytolearnx.com/2020/05/jcheckbox-java-swing.html
  - https://waytolearnx.com/2020/05/jdialog-java-swing.html
  - https://stackoverflow.com/questions/19780300/mouse-click-java
  - https://www.geeksforgeeks.org/java-jscrollpane/
  - https://waytolearnx.com/2020/05/les-boites-de-dialogue-joptionpane-java-swing.html
  - https://waytolearnx.com/2020/05/jlist-java-swing.html

Annexe

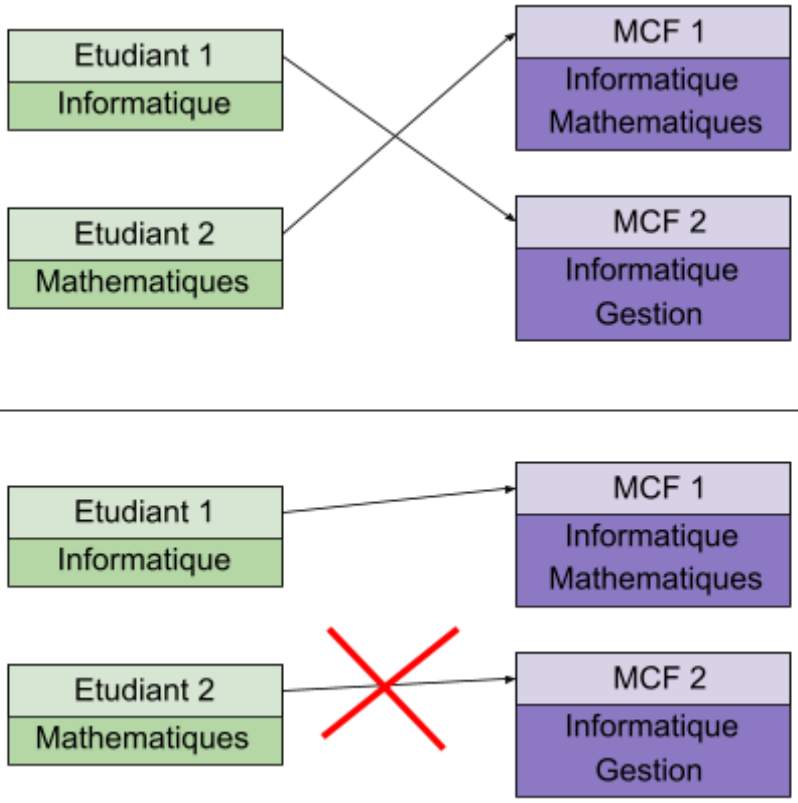


Schéma représentant un exemple d'un mauvais couplage

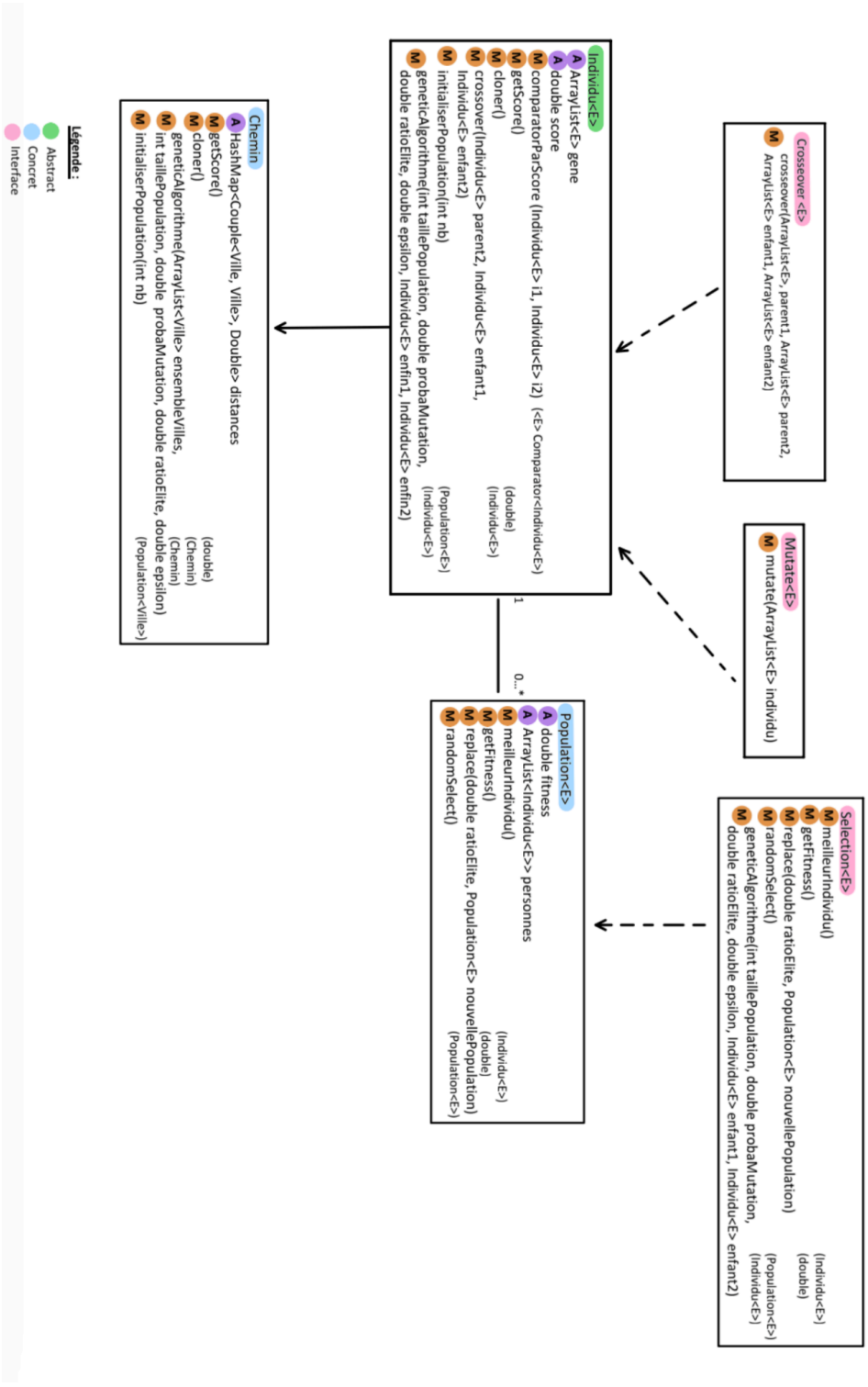


Schéma représentant la hiérarchie des classes pour l'implémentation de l'algorithme génétique

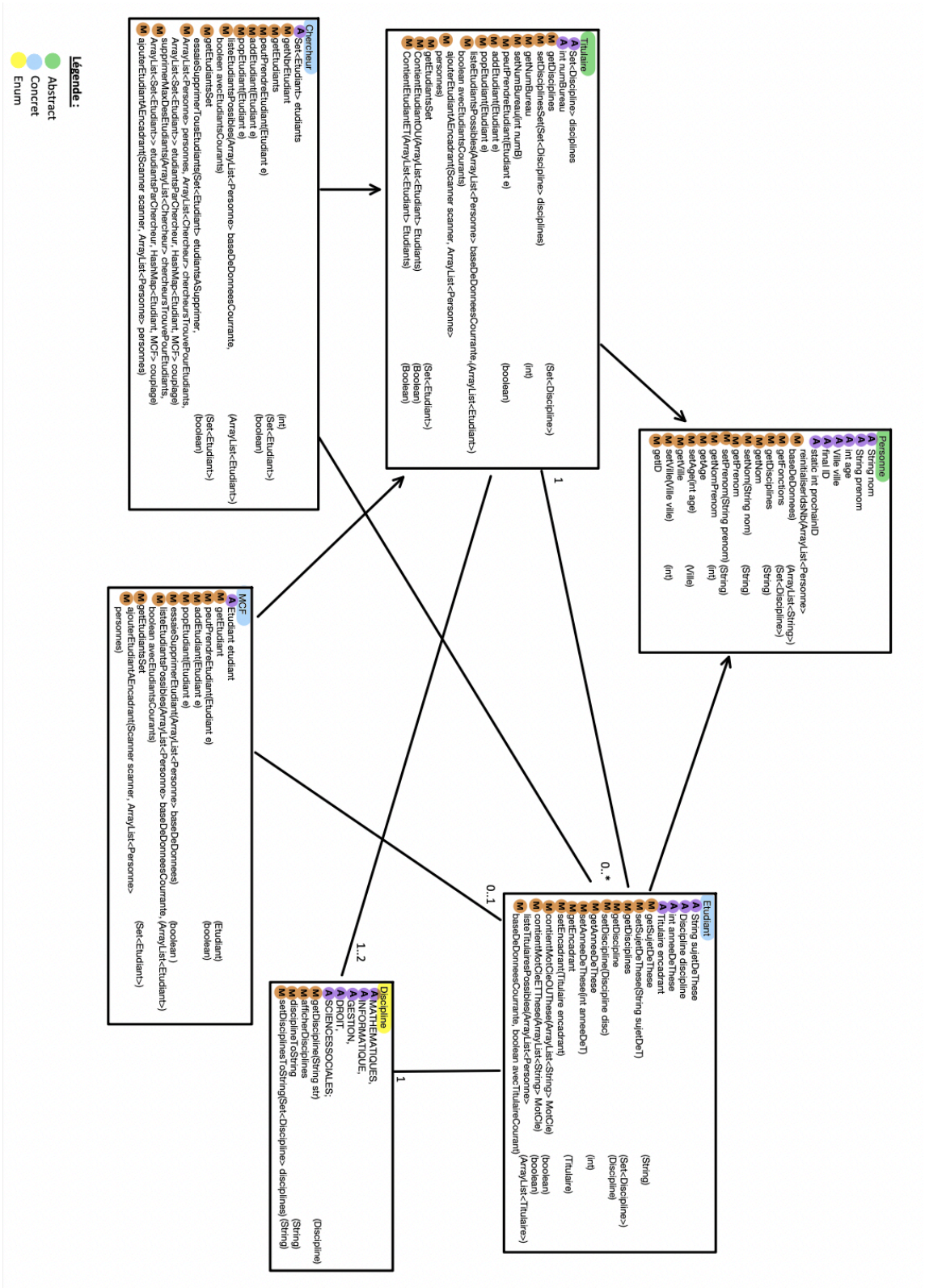


Schéma représentant la hiérarchie des classes du package peuple

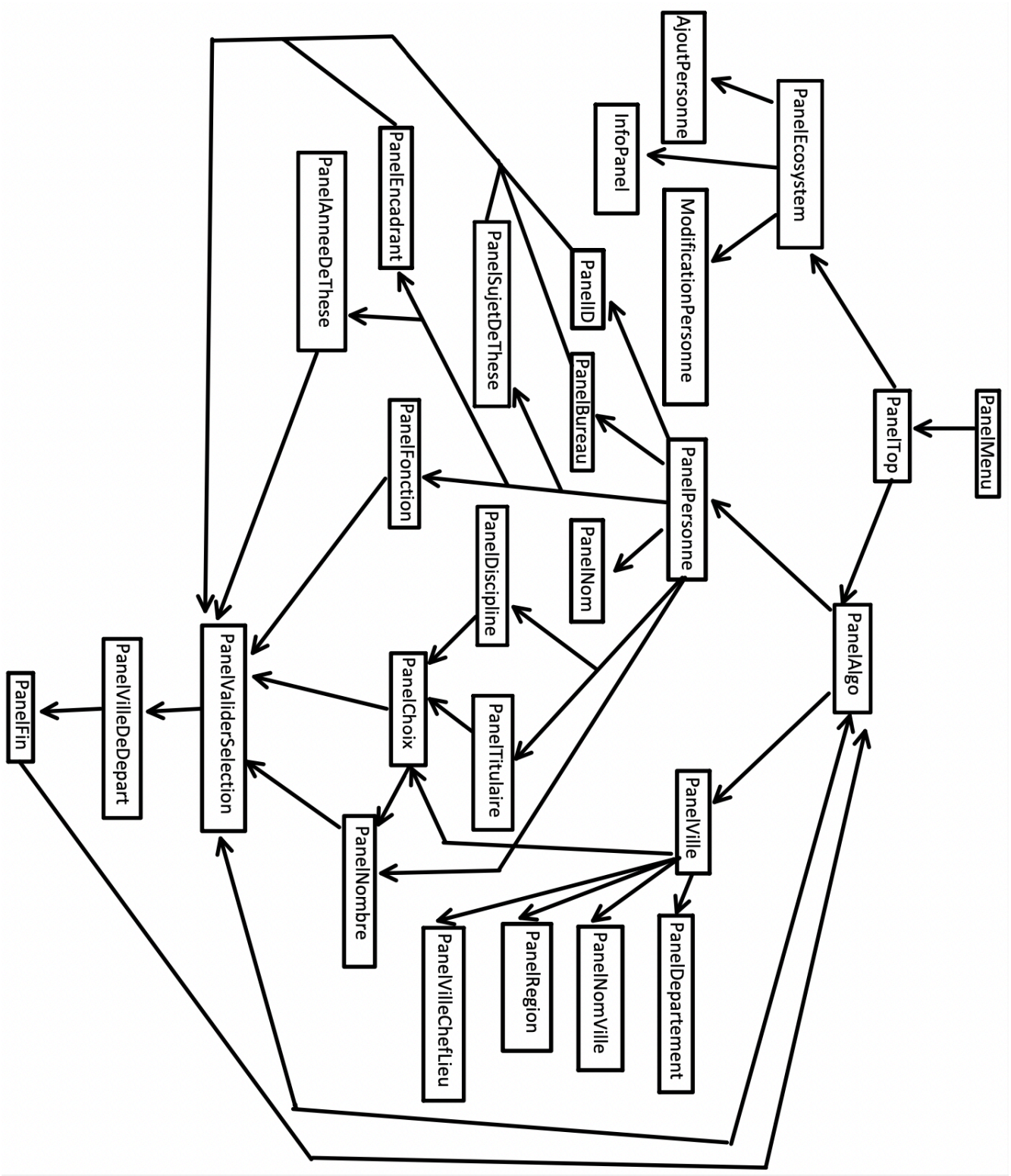


Schéma représentant comment on passe d'un panneau à un autre dans le package Affichage