INTRODUCTION

L'idée principale de ce projet était d'implémenter une solution pour réparer les différentes erreurs de logique de la base de données : les ubiquités, les interférences et les chevauchements.

Pour résoudre le problème demandé, nous sommes d'abord passés par une phase de conception, elle est très importante et nous a permis d'avoir une structure de données stable et ne pas regretter nos choix dans le futur.

Nous avons découpé ce projet en 7 classes :

Execute : Classe launcher, c'est de la que le code est lancé.

Salle : une salle est composée d'un nom. Il y a une liste en attribut static contenant le nom de toutes les salles

Chirurgien : un chirurgien est composé d'un nom. Il y a une liste en attribut static contenant le nom de tous les chirurgiens.

Chirurgie : Une chirurgie est composée d'un id, d'une date, d'une salle, d'un chirurgien et d'une heure de fin et de début. C'est dans la classe calendrier que sont qu'elles sont instanciées. Toutes les modifications sont effectuées sur cet objet : les changements d'heures, les changements de salles ou des chirurgiens.

Conflit : cette classe est composé de deux chirurgies. Un conflit est instancié uniquement s'il y a un problème entre deux chirurgies. De Plus elle est associée à l'enumération ConflitType qui correspond à une interférence, une ubiquité, ou un chevauchement.

ConflitType : Enumeration contenant contient les types de conflits

Journée : La journée est un ensemble de chirurgies et de conflits correspondant à une date. C'est là où se trouve les différents algorithmes de résolutions. Ils sont appliqués par journée, sur les différents conflits

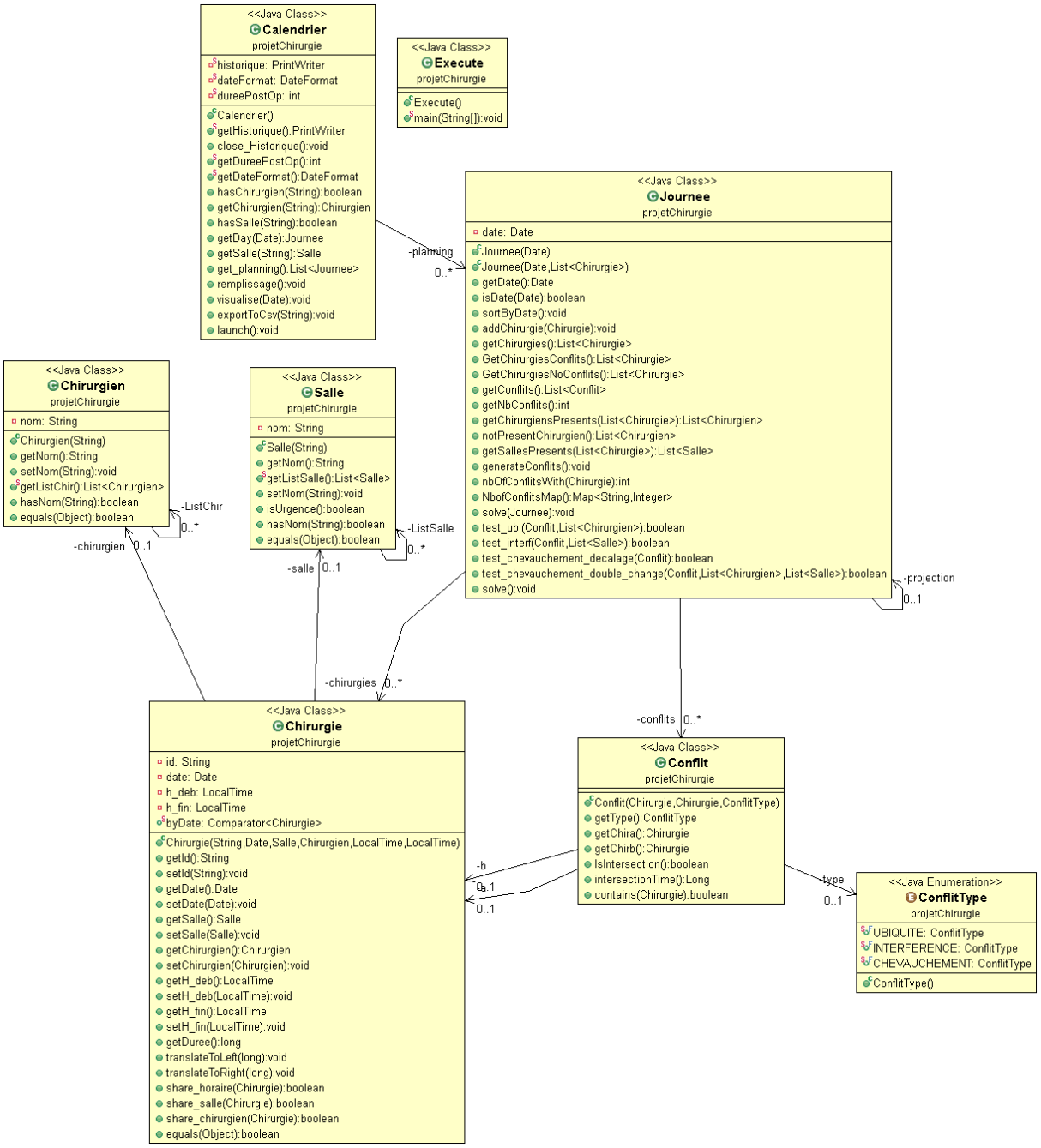
Calendrier : C'est dans cette classe que le remplissage de nos objets est effectué, que toutes les données du CSV fournies sont envoyés pour être réparti dans les différentes classes. Elle est également composée de Journée permettant d'effectuer les algorithmes sur les différentes chirurgies par journées

Diagramme de classe

Le diagramme de classe représentant notre implémentation est sur la page suivante.

Le fichier corrigé mis en input, est traité pour ressortir sous le nom de base_corrections.csv.De plus un historique des corrections est crée sous le nom historique_corrections.

Lors du lancement du code, un File dialog s'ouvre permettant de selectionner la base de données à corriger.


<<Java Class>>
Calendrier
projetChirurgie
historique: PrintWriter
dateFormat: DateFormat
dureePostOp: int
Calendrier()
getHistorique():PrintWriter
close_Historique():void
getDureePostOp():int
getDateFormat():DateFormat
hasChirurgien(String):boolean
getChirurgien(String):Chirurgien
hasSalle(String):boolean
getDay(Date):Journee
getSalle(String):Salle
get_planning():List<Journee>
remplissage():void
visualise(Date):void
exportToCsv(String):void
launch():void
<<Java Class>>
Execute
projetChirurgie
Execute()
main(String[]):void
-planning
0..*
<<Java Class>>
Journee
projetChirurgie
date: Date
Journee(Date)
Journee(Date,List<Chirurgie>)
getDate():Date
isDate(Date):boolean
sortByDate():void
addChirurgie(Chirurgie):void
getChirurgies():List<Chirurgie>
GetChirurgiesConflits():List<Chirurgie>
GetChirurgiesNoConflits():List<Chirurgie>
getConflits():List<Conflit>
getNbConflits():int
getChirurgiensPresents(List<Chirurgie>):List<Chirurgien>
notPresentChirurgien():List<Chirurgien>
getSallesPresents(List<Chirurgie>):List<Salle>
generateConflits():void
nbOfConflitsWith(Chirurgie):int
NbofConflitsMap():Map<String,Integer>
solve(Journee):void
test_ubi(Conflit,List<Chirurgien>):boolean
test_interf(Conflit,List<Salle>):boolean
test_chevauchement_decalage(Conflit):boolean
test_chevauchement_double_change(Conflit,List<Chirurgien>,List<Salle>):boolean
solve():void
-projection
0..1
<<Java Class>>
Chirurgien
projetChirurgie
nom: String
Chirurgien(String)
getNom():String
setNom(String):void
getListChir():List<Chirurgien>
hasNom(String):boolean
equals(Object):boolean
-ListChir
0..*
-chirurgien
0..1
<<Java Class>>
Salle
projetChirurgie
nom: String
Salle(String)
getNom():String
getListSalle():List<Salle>
setNom(String):void
isUrgence():boolean
hasNom(String):boolean
equals(Object):boolean
-ListSalle
0..*
-salle
0..1
-chirurgies
0..*
<<Java Class>>
Chirurgie
projetChirurgie
id: String
date: Date
h_deb: LocalTime
h_fin: LocalTime
byDate: Comparator<Chirurgie>
Chirurgie(String,Date,Salle,Chirurgien,LocalTime,LocalTime)
getId():String
setId(String):void
getDate():Date
setDate(Date):void
getSalle():Salle
setSalle(Salle):void
getChirurgien():Chirurgien
setChirurgien(Chirurgien):void
getH_deb():LocalTime
setH_deb(LocalTime):void
getH_fin():LocalTime
setH_fin(LocalTime):void
getDuree():long
translateToLeft(long):void
translateToRight(long):void
share_horaire(Chirurgie):boolean
share_salle(Chirurgie):boolean
share_chirurgien(Chirurgie):boolean
equals(Object):boolean
-conflits
0..*
<<Java Class>>
Conflit
projetChirurgie
Conflit(Chirurgie,Chirurgie,ConflitType)
getType():ConflitType
getChira():Chirurgie
getChirb():Chirurgie
IsIntersection():boolean
intersectionTime():Long
contains(Chirurgie):boolean
-b
0..1
0..1
-type
0..1
<<Java Enumeration>>
ConflitType
projetChirurgie
UBIQUITE: ConflitType
INTERFERENCE: ConflitType
CHEVAUCHEMENT: ConflitType
ConflitType()

Définition d'un conflit et classification des erreurs

Rappelons qu'un conflit est un couple de chirurgies qui partagent tout ou partie de leur plage horaire ainsi qu'une ou plusieurs ressources parmi le chirurgien et le bloc opératoire.

Nous avons tout d'abord distingué l'intersection de l'inclusion.

Nous pouvons définir une intersection en reprenant les notations de l'énoncé.

Un conflit(x', x'')avec x' la chirurgie qui débute le plus tôt, forme une inclusion si et seulement si,

$t''_f \leq t'_f$.

Sinon, il forme une intersection.

Cette distinction est importante car elle permet d'élargir le champ des correctifs appliqués aux conflits de type chevauchement.

Les journées sont résolues jour par jour. Ainsi, nous retrouvons notre algorithme correctif dans la classe Journee.

Nous allons décrire maintenant quelques méthodes utiles pour la résolution des conflits.

Méthode	Fonction	Type de retour
generateConflits()	Permet de détecter les conflits et de les stocker dans l'attribut : ArrayList<Conflit> conflits.	void
getChirurgiesConflits()	Renvoie la liste des chirurgies de la journée impliquées dans les conflits.	List<Chirurgie>
getChirurgiesnoConflits()	Renvoie la liste des chirurgies de la journée qui ne sont impliqués dans aucun conflit.	List<Chirurgie>
getChirurgiensPresents (List<Chirurgie>)	Renvoie la liste des chirurgiens présents dans la liste de chirurgie passée en argument.	List<Chirurgien>
getSallesPresents (List<Chirurgie>)	Renvoie la liste des salles présentes dans la liste de chirurgie passée en argument.	List<Salle>
test_ubi (Conflit, List<Chirurgien>)	Retourne vrai si le conflit a été résolu par un changement de chirurgien d'une des deux chirurgies. Ce chirurgien appartient à la liste passée en paramètre.	Boolean
test_interf (Conflit, List<Salle>)	Retourne vrai si le conflit a été résolu par un changement de salle d'une des deux chirurgies. Cette salle appartient à la liste passée en paramètre.	Boolean

test_chevauchement_decalage (Conflit)	Retourne vrai si le conflit a été résolu par un décalage d'horaires d'une des chirurgies à droite ou à gauche.	Boolean
test_chevauchement_double_change (Conflit,List<Salle>, List<Chirurgien>)	Retourne vrai si le conflit a été résolu par un double changement de salle et de chirurgien. Ces deux ressources appartiennent aux deux listes en paramètres.	Boolean

La méthode solve(), tente de résoudre l'ensemble des conflits d'une journée.

Nous décrivons son fonctionnement dans le pseudo-code suivant :

- Génération des conflits
- i ← 0
- Tant qu'il reste des conflits ou qu'on ne peut plus résoudre aucun conflit
    - Sélection du i-ème Conflit c
    - Si c est de type UBIQUITE
        - Séquence de tests sur différents ensembles de chirurgiens
    - Si c est de type INTERFERENCE
        - Séquence de tests sur différents ensembles de salles
    - Si c est de type CHEVAUCHEMENT
        - Si c forme une intersection
            - Test de décalage des chirurgies
        - Si c forme une inclusion
            - Séquence de tests sur différents ensembles de chirurgiens et de salles
    - Si c a été résolu : i ← 0
    - Sinon : i ← i + 1
- Fin tant que

<u>Ubiquité :</u>

Pour corriger une ubiquité, il s'agit de modifier un chirurgien d'une des deux chirurgies.

La séquence de tests consiste à appeler la méthode de résolution d'une ubiquité sur des ensembles de chirurgiens du plus restrictif au moins restrictif.

Nous dénombrons trois niveaux de précision :

- getChirurgiensPresents(getChirurgiesNoConflits())

Correspond à l'ensemble des chirurgiens présents ce jour et impliqués dans aucun conflits.

- getChirurgiensPresents(getChirurgiesConflits())

Correspond à l'ensemble des chirurgiens présents ce jour et impliqués dans des conflits.

- GetChirurgiensNoPresents()

Correspond à l'ensemble des chirurgiens non présent le jour donné.

Interférence :

Il s'agit dans ce cas de modifier la salle d'une des deux chirurgies.

La même logique que le cas précédent est appliquée ici.

Chevauchement :

Dans le cas d'une intersection, il s'agit d'effectuer une translation vers la gauche de la 1ère chirurgie ou une translation à droite de la seconde.

Une translation est un décalage de l'horaire de début et de fin d'une chirugie, d'un nombre de minutes égal à la durée de l'intersection auquel on ajoute une durée post opératoire fixée.

Dans le cas d'une inclusion il s'agit de tester l'ensemble des combinaisons de chirurgiens et de salles jusqu'à trouver une solution admissible. Nous avons donc combiné les méthodes d'interférence et d'ubiquité.

Chaque méthode de résolution d'un conflit teste la combinaison donnée en commencant par la 1ère chirurgie du conflit. Celle ci est validée si après génération des conflits, le nombre conflits a diminué strictement.

Si aucune des combinaisons ne fonctionne alors la méthode remet les valeurs d'origines aux variables ayant subies des modifications, et on teste sur la 2ème chirurgie.

De même si aucune combinaison ne fonctionne, les valeurs d'origines sont réaffectées et la méthode renvoie faux.

Les ensembles de chirurgiens et de salles subissent un tri aléatoire dans les méthodes de résolution afin de ne pas affecter toujours le même chirurgien non présent dans un cas d'ubiquité par exemple.

Pistes d'amélioration :

- Nous avions commencé à mettre en place un système de priorité de résolution en triant les chirurgies par nombres d'implications dans différents conflits, mais avons décidé de ne pas l'inclure car un décideur humain commencerait tout d'abord par résoudre les conflits évidents, c'est à dire impliquant des chirurgies peu conflictuelles.

- Il aurait été possible de fixer la durée post-opératoire grâce à une étude statistique sur les chirurgies non conflictuelles.

Toutefois, il est aussi intéressant de voir qu'en modifiant cette durée, le nombre de conflits résolus varie.

- Aussi nous aurions pu attribuer un indice de correction en fonction du niveau de précision dans lequel chaque conflit a été résolu. Cela aurait pu donner un score par journée, chaque niveau de précision apportant de 1 à n points, n étant le plus précis. Il pourrait alors s'agir de fixer un niveau de précision par journée ou pour l'ensemble des journées pour constater le nombre de conflits résolus.

- La 1ère idée que nous avons eu faisait intervenir un système de projection dans une journée. Ainsi résoudre un conflit générerait une projection qui elle même pourrait être utilisée afin de résoudre d'autres conflits, de telle sorte que l'ensemble des projections formerait un arbre dont les feuilles constitueraient l'ensemble des solutions.

- Nous avons aussi pensé à supprimer les chevauchements dont les heures de début et de fin sont identiques. Car il est alors plus probable que la personne chargée de rentrer les chirurgies rentre un doublon plutôt qu'elle fasse deux erreurs sur des chirurgies ayant exactement les mêmes horaires dans la même journée. Toutefois nous ne savions pas s'il était possible de supprimer une chirurgie.

Intérêt du projet

Ce projet nous a permis de consolider nos connaissances techniques en JAVA. Nous ignorions l'utilité d'un attribut en static, notamment sa capacité à être instancié une seule fois et celle à être utilisé n'importe où.

En plus de ça a la suite du projet du premier semestre, Stéphane Airiau nous avez donné des conseils pour rendre le code le plus optimal possible : comme par exemple de laisser nos listes sous forme de List et non pas en ArrayList sur les types de retour, par exemple, nous permettant d'avoir moins de contraintes dans le cas ou nous devrions faire de grosses corrections sur notre implémentation.

Ce projet nous a également permis de familiariser avec Git, qui n'est pas simple d'utilisation en cas d'erreurs. Certain pull nous ont amenés a devoir refaire des parties que nous avions déjà faites.

Ce projet était compliqué par la justification de l'euristique que devait prendre l'algorithme de résolutions