Rapport technique – Projet informatique mai 2016
Gestion des emplois du temps

Télécom Lille - INGE1 - Flavien Delangle & Marie Payet

Afin de mieux appréhender les concepts de la programmation orientée objet et plus spécialement du langage JAVA, nous avons travaillé sur un projet d'emplois du temps pour les universités, proche de notre usage quotidien de cet outil. Notre binôme s'est axé sur la gestion de ces emplois du temps. Nous allons rapidement expliquer la structure choisie pour implémenter notre solution, puis nous justifierons les choix principaux réalisés.

I – La structure du projet : diagramme UML :

Le diagramme UML fourni en annexe n°2 représente les différentes classes régissant notre projet et les liens entre chacune d'entre elles. Nous voyons ainsi clairement apparaître le modèle MVC, c'est-à-dire divisé en trois espaces : l'interface utilisateur fournie, le modèle, cœur de notre projet, ainsi que le controller nous permettant de tester et manipuler nos fonctions rédigées dans la partie précédente.

Le point essentiel à dégager de ce diagramme est la concentration de nos fonctions principales dans la classe TimeTableDB : ainsi celle-ci agit comme point central et est la seule entité reliée au controller ce qui nous facilite par la suite l'implémentation.

II – L'organisation interne du code :

- **Les fonctions de base :**

La gestion des fonctions de base se concentre donc comme évoqué précédemment au sein de la classe TimeTableDB. Il a fallu ici faire particulièrement attention à la représentation des attributs pour avoir une cohérence avec les types utilisés pour générer l'interface.

De plus, nous avons fait le choix de factoriser un maximum notre code. En effet, nous avons défini dans les autres classes plusieurs fonctions intermédiaires nous permettant d'avoir un code dans la classe TimeTableDB plus clair et optimisé. Ainsi, nous retrouvons par exemple des fonctions « get » et « set » permettant la gestion interne dans chaque classe de ses attributs et pouvant être appelées de manière explicite dans les fonctions de base.

- **Les fonctions avancées :**

Gestion des emplois du temps des professeurs (annexe n°3) :

Afin de représenter les emplois du temps des professeurs, nous sommes initialement partis sur le principe d'héritage. En effet, nous avions créé une classe générique et abstraite « TimeTable » que nous ne pouvions pas instancier et qui contenait les informations et fonctions communes à des emplois du temps de groupes et de professeurs. Ensuite nous avons généré deux classes filles « TimeTableGroup » contenant en plus un attribut « GroupId » et « TimeTableTeacher » contenant un login (ces deux attributs étant la seule véritable différence entre les deux emplois du temps.) Ensuite, nous réutilisons les fonctions de la classe mère pour créer celles qui nous seraient utiles dans la classe fille.

Cette implémentation nous a semblé la plus immédiate dans un premier temps, cependant, elle nous a posé plusieurs soucis au moment de son utilisation. Nous avons donc modifié notre code afin d'avoir une solution plus efficace directement. Nous sommes repartis sur l'utilisation d'une unique classe « TimeTable » contenant cette fois les deux attributs GroupId et login. Cependant un seul est rempli à la fois et nous indiquera de quel emploi du temps il s'agit.

Gestion à partir d'une base de données MySQL:

Nous avons décidé de faire une intégration complète du SQL dans notre projet. La solution la plus simple aurait alors été de charger le contenu d'une base de données en mémoire au lancement du programme et de la mettre à jour à la fermeture. Ce comportement, proche de celui implémenté en XML est cependant particulièrement peu optimisé. Nous avons donc décidé que chaque action serait immédiatement répercutée en base de données plutôt que dans des objets stockés en mémoire.

Pour cela, vous aurions pu se contenter de créer des fonctions dans TimeTableDB contenant les requêtes SQL. Cependant Java est un langage fortement orienté objet et il nous paraissait dommage de ne pas respecter se paradigme de programmation. Nous avons donc cherché à reproduire le comportement d'un ORM afin de simuler le comportement d'un programme orienté objet. Pour cela nous nous sommes fortement inspirés du fonctionnement de l'ORM de Django et nous avons cherché à obtenir des appels similaires. Notre ORM est bien sûr extrêmement simple comparé à celui de ce Framework et nous avons dû implémenter une classe par type d'objet afin de le faire fonctionner.

Gestion des conflits :

Nous avons géré les conflits au sein de la fonction saveDB. En effet, au moment d'enregistrer notre base de données en XML, nous utilisons une fonction modifiedXML permettant de savoir s'il y a eu un changement depuis la dernière date d'enregistrement. Si oui, nous vérifions le type de modification, et si nous notons un évènement pouvant générer un conflit tel qu'un ajout, nous vérifions systématiquement si c'est le cas. Pour cela nous vérifions lors d'un ajout de Room ou de TimeTable les id. Lorsqu'une réservation est demandée alors qu'une salle de cours n'est pas disponible pour le créneau un message d'erreur apparaît et propose une solution à l'utilisateur. Pour cela, nous avons mis en place une fonction permettant de parcourir toutes les salles pour en trouver une disponible dont la capacité conviendrait. Ainsi, nous retournons la salle ayant la capacité la plus petite pour la proposer en alternative à l'utilisateur. Certains conflits n'ont cependant par été pris en compte tels que la création d'une Room déjà créé, nous avons en effet souhaité nous concentrer sur les conflits présentant le plus d'intérêt en termes de complexité algorithmique.

Génération automatique d'un emploi du temps :

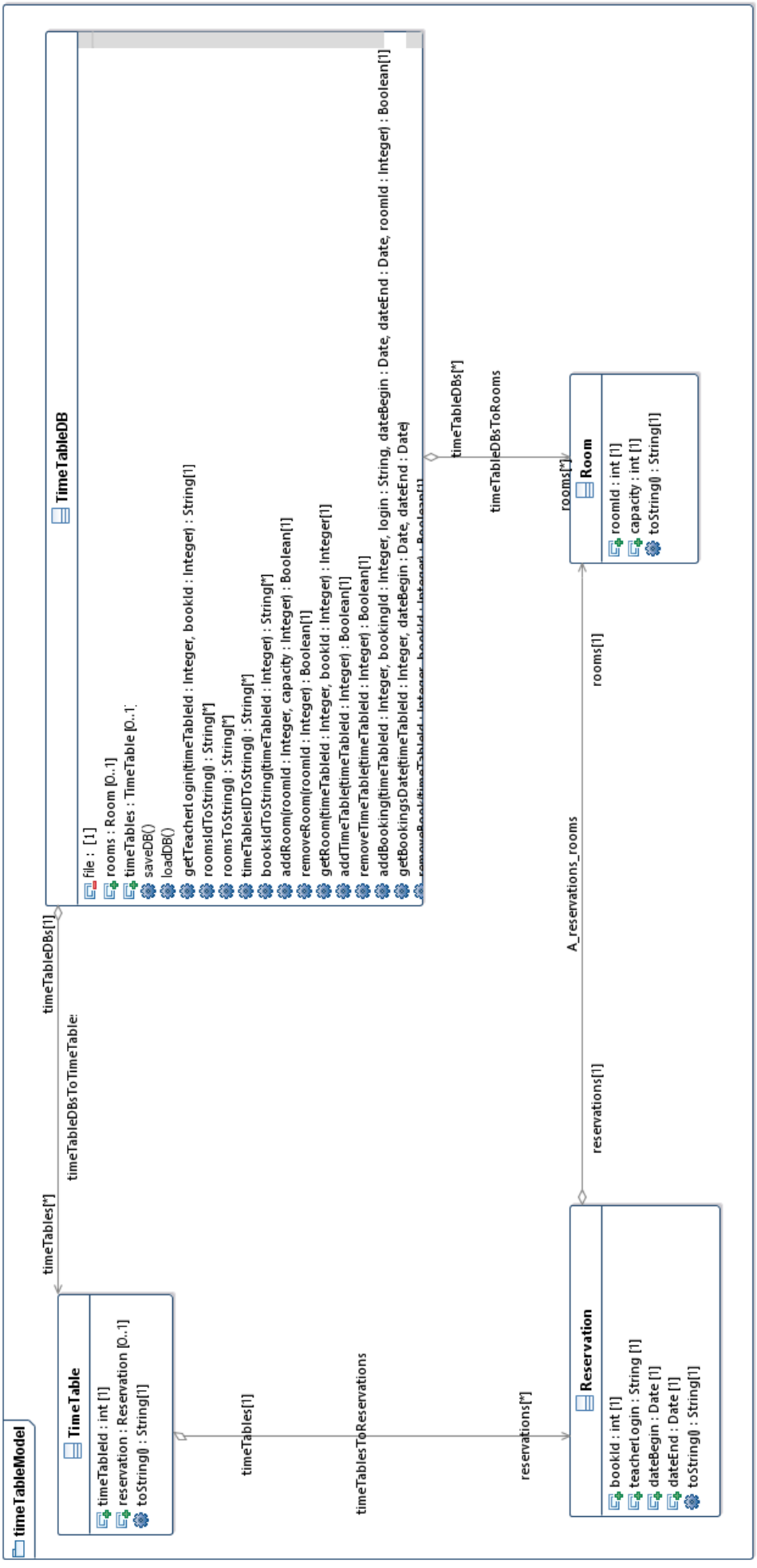
La dernière fonctionnalité que nous avons intégrée est un générateur automatique d'emploi du temps. Ce problème est extrêmement complexe et nous n'avons évidemment pas cherché à proposer une solution complète mais plutôt à découvrir les problématiques liées à cette génération. Nous avons ainsi cherché à répartir une liste de cours sur une plage de temps donnée. Chaque cours comporte une durée ainsi qu'une liste de prérequis. Le fonctionnement de notre programme est décrit en Annexe n°4.

Axes d'amélioration envisageables :

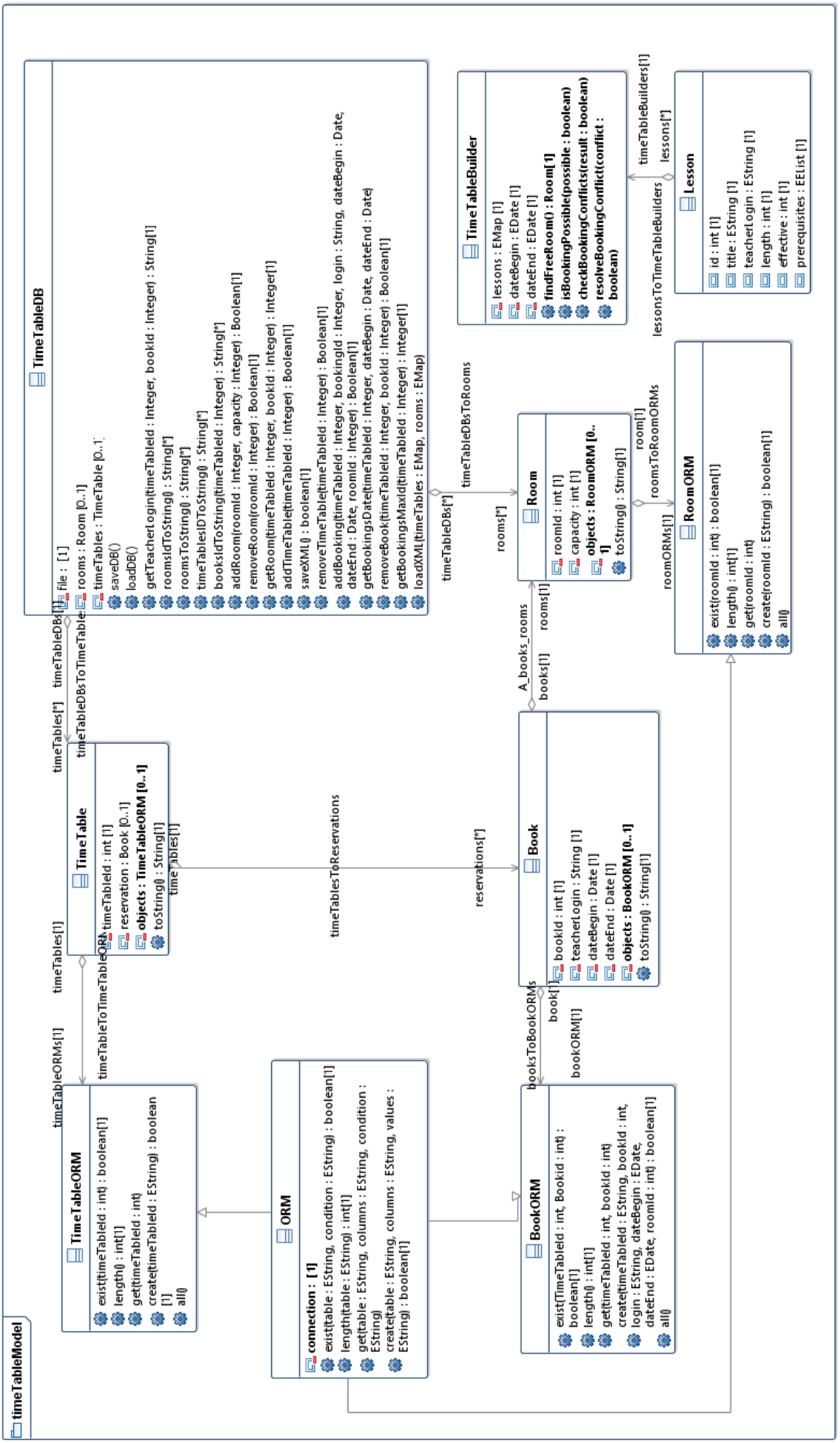
Durant ce projet, nous avons cherché à approfondir au maximum les fonctionnalités qui nous paraissaient les plus intéressantes. Nous n'avons cependant pas eu le temps de toute les réaliser. Ainsi nous aurions souhaité pouvoir gérer plus de conflits, particulièrement en ce qui concerne la création de Room et de TimeTable. Nous aurions également souhaité travailler sur la répartition des cours dans une liste de Room, cette problématique étant particulièrement intéressante algorithmiquement.

Pour conclure, ce travail nous a permis d'aborder un certain nombre de problématiques et de progresser sur la gestion collective d'un projet informatique. En effet, l'utilisation de Git nous a donné l'opportunité de travailler séparément sur le code et d'avoir une meilleure vision au global de l'avancée du projet. De plus, le projet étant relativement libre, nous avons pu progresser chacun à des niveaux différents et à notre rythme.

Annexe n°1 : diagramme UML initial du projet

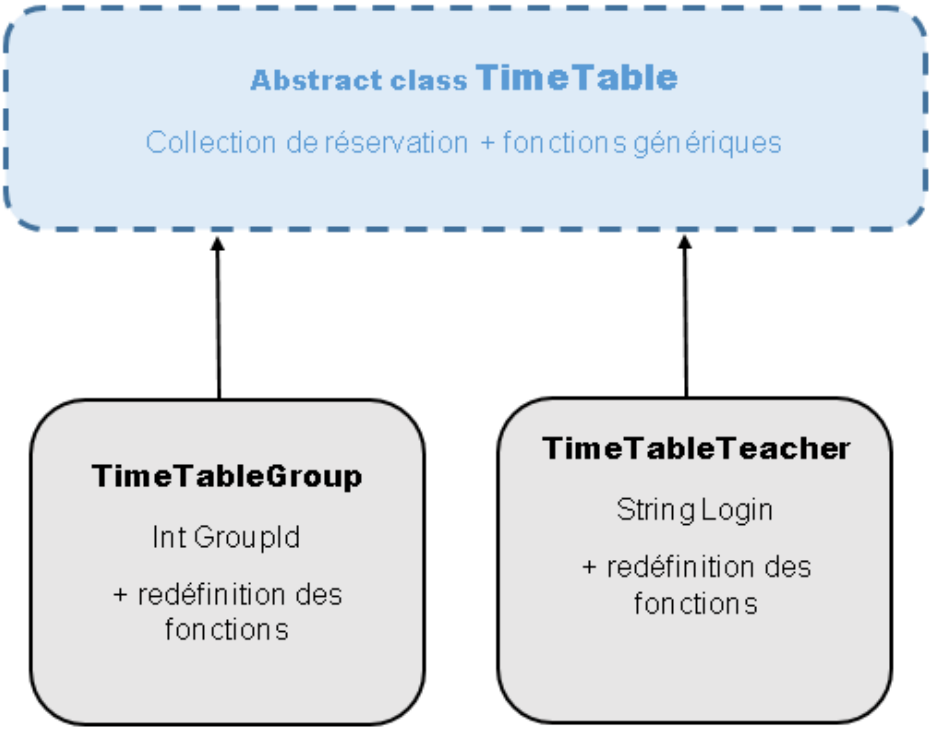


Annexe n°2 : diagramme UML final du projet



Annexe n°3 : conception des emplois du temps des professeurs

Solution initiale : utilisation de l'héritage



Solution finalement adoptée : une classe commune

Annexe n°4 : génération automatique de l'emploi du temps