EdT

Créateur d’emploi du temps

Rémi DUPOUY - Léo MARTINEZ
Gatien PASCAUD - Tom RAVIX

Introduction

EdT est un créateur d’emploi du temps. Il a été conçu pour générer automatiquement des emplois du temps en fonction de paramètres entièrement configurables qui lui sont donnés : groupes d’élèves, salles de classes, matières, enseignants, plages horaires, etc.

Cahier des charges fonctionnel

Note préalable : le cahier des charges a subi des modifications en cours de projet. La version présentée ci-dessous est la dernière en date.

L'intérêt de ce projet réside essentiellement dans la conception d'un algorithme efficace pour réaliser une tâche complexe (création d’un emploi du temps), associé à une interface graphique qui devra être efficace et intuitive, et la gestion de nombreuses contraintes utilisateur.

Le projet sera découpé autant que possible en classes, afin de respecter le modèle de programmation orientée objet.

1. Saisie des paramètres

L’interface graphique devra permettre la saisie et l’enregistrement de paramètres, avant et après la création des emplois du temps. Ces paramètres sont :

- **Enseignant**: possède un nom, prénom et une adresse mail. Chaque enseignant peut enseigner une ou plusieurs matières. De même, un enseignant chercheur a des contraintes horaires qui devront être prises en compte lors de la génération de l’emploi du temps.
- Matière : est d’un type spécifique (cours magistral/TD/TP) et donc peut être dispensée dans un type de salle spécifique (TP physique/TP chimie/Amphi/etc.). Le programme devra également prendre en compte le fait qu’un TP de physique dure 4 heures, et un TD de mathématiques 2 heures maximum.

- Salle de classe : a un numéro et ne peut accueillir qu’un certain nombre d’élèves. Elle devra être localisée sur un plan du premier cycle. De même, il faudra faire en sorte qu’un TP de chimie n’ait pas lieu dans une salle de TD (on pourra donc créer des catégories de salles et certains cours ne pourront avoir lieu que dans certaines catégories de salles).
- Groupe d’élèves : ils seront hiérarchisés. C’est-à-dire que le groupe 46 appartiendra au groupe « Lanière L », qui appartiendra au groupe « 2A », qui appartiendra au groupe « PCC ». Cette hiérarchie permettra d’attribuer un cours à un groupe parent : tous les groupes enfants en hériteront (exemple : cours magistral commun à toute une lanière, ou IE commune à toute une filière). Le groupe d’élèves aura un nombre d’élèves, ainsi qu’une liste des matières qu’il faudra enseigner aux élèves du groupe (avec le nombre d’heures correspondantes pour chaque matière).

2. Génération des emplois du temps

Un algorithme efficace devra prendre en compte tous ces paramètres, et tenter de concevoir les emplois du temps pour chaque groupe/enseignant/salle de manière automatique. Il déterminera alors des contraintes, les classera par ordre de priorité, et tiendra compte des contraintes selon leur priorité.

L'algorithme devra avoir plusieurs comportements lors de la génération de l'emploi du temps :

- Ignorer les erreurs : lorsque l'algorithme ne parvient pas à attribuer un enseignement à un groupe, il affiche un simple message d'avertissement et passe à l'enseignement suivant. L'utilisateur devra alors intervenir pour ces problèmes spécifiques *a posteriori*.
- Tenter de résoudre les erreurs : dans ce mode, l'algorithme tentera de trouver une solution pour attribuer tout de même l'enseignement au groupe.
- S'arrêter en cas d'erreur : un message sera alors affiché à l'utilisateur, qui sera invité à modifier ses paramètres avant de relancer la génération des emplois du temps.

3. Affichage des emplois du temps

L'utilisateur pourra, après génération, afficher l'emploi du temps d'un enseignant ou d'un groupe d'élèves. Celui-ci s'ouvrira dans une nouvelle fenêtre.

4. Modification des emplois du temps *a posteriori*

Cette partie a été abandonnée, faute de temps dû à la complexité des deux premières parties.

Il a été envisagé de permettre la modification des emplois du temps à la volée, depuis l'interface graphique. D'autres outils ont également été prévus, tels que la recherche d'une salle libre pour un créneau horaire précis dans la semaine, ou la recherche d'un enseignant pouvant remplacer son collègue pour un créneau spécifique.

Tout ceci constitue une piste d'amélioration pour notre programme.

A propos de l’algorithme

Nous avons décidé de rassembler tout l'algorithme dans une seule classe, Filler, afin de bien séparer les trois grands blocs du programme : stockage des données (package DATA), interface (package GUI) et moteur (classe Filler).

Le remplissage d'un emploi du temps est une tâche très complexe, car il y a une quasi-infinité de façon de lancer le remplissage, mais seulement un petit nombre de solutions possibles.

Afin de limiter le nombre de degrés de liberté et de permettre au programme de tenter le remplissage plus rapidement et efficacement qu'en testant bêtement toutes les solutions une à une, nous avons décidé d'appliquer un pré-traitement aux données afin de déterminer un ordre de « contrainte » :

Par exemple, pour un professeur, on attribuera la contrainte (nombre d'heures de cours à donner) / (nombre d'heures de présence à l'école)

Ce pré-traitement est concrètement effectué par la méthode ComputeConstraints(), qui attribue à tous les types d'objets une contrainte entre 0 et 1 (si un objet a une contrainte supérieure à 1, l'emploi du temps ne pourra être complété).

Par la suite, on parcourt la liste des objets par ordre de contrainte, et on leur attribue des créneaux horaires jusqu'à remplir les demandes (le plus souvent, on en revient au nombre d'heures par semaine d'une matière pour un groupe).

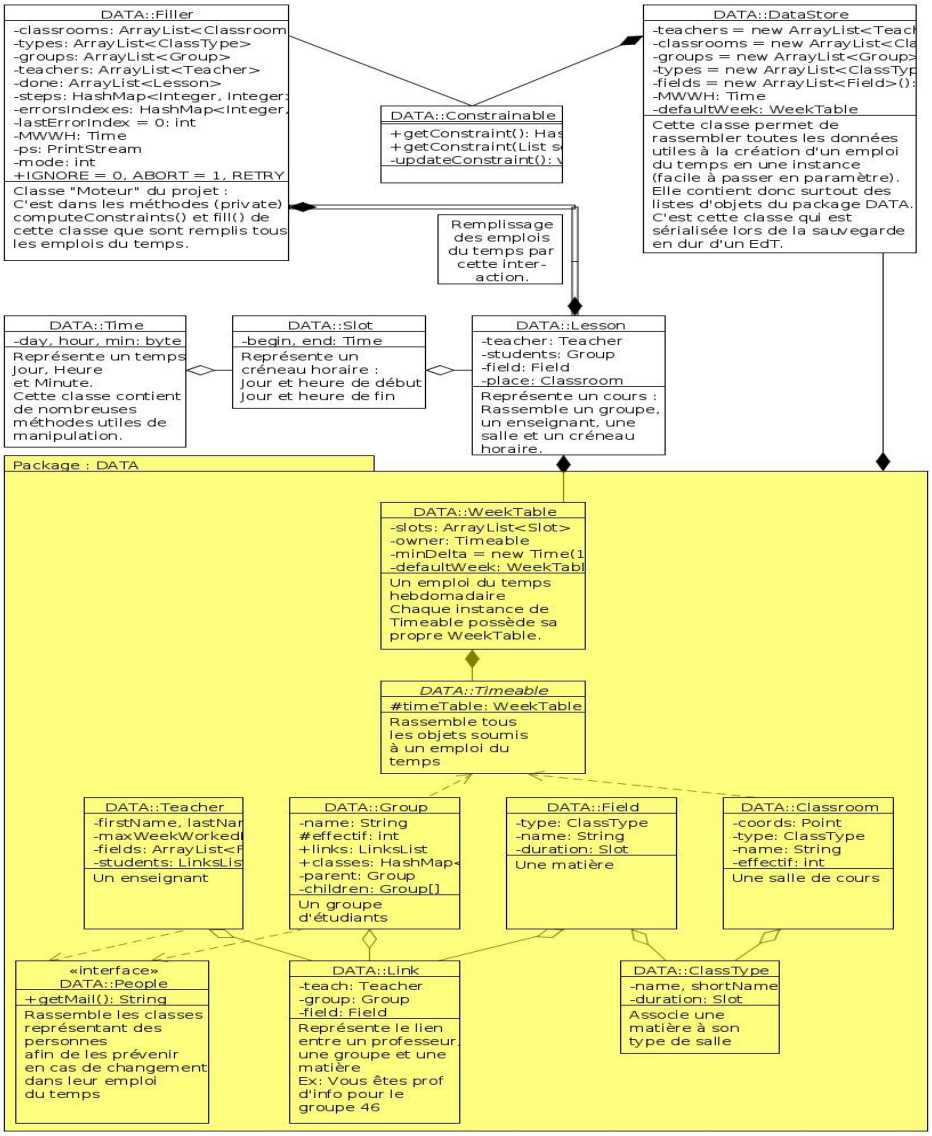
Lorsque le programme arrive dans une impasse (plus de salle disponible pour un créneau donné, un groupe et son professeur n'ont pas de créneau disponible en commun, ...), on a donné trois choix à l'algorithme :

- Annuler : on affiche un message identifiant l'erreur, et le programme quitte, laissant le choix à l'utilisateur de modifier ses paramètres.
- Ignorer : on affiche le message d'erreur, mais on continue le remplissage : l'utilisateur devra revenir manuellement sur l'erreur pour tenter de la résoudre.
- Résoudre : le programme remonte dans l'historique jusqu'à un point critique, annule tout ce qui a été fait après ce point et recommence dans une autre direction (par exemple, deux objets ayant la même contrainte sont traités l'un après l'autre → on annule toute la suite, on échange les deux objets dans l'ordre, et on recommence).

Cette troisième option, bien qu'intéressante du point de vue de l'algorithmie, n'a pas été complètement implémentée : on constate que lorsque le programme rencontre une erreur, il remonte plusieurs fois au même point de l'arbre-historique, par conséquent retombe sur la même erreur à chaque fois, et finit par abandonner pour ne pas créer de boucle infinie. Par manque de temps, nous n'avons pu parachever cette partie.

Structuration des données

Comme expliqué précédemment, le programme est découpé en de nombreuses classes qui interagissent. En voici un schéma UML. Une version PDF du schéma est jointe à l'archive pour plus de lisibilité.



Une classe nommée « Datastore » sert à la persistance des données. Toutes les instances des classes devant être persistées y sont stockées. Cette classe Datastore est ensuite sérialisée puis enregistrée dans un fichier. Ce fichier est lu lorsque le programme est relancé : toutes les instances précédemment sauvegardées sont restaurées.

Bibliographie

La *Javadoc* (http://docs.oracle.com/javase/7/docs/api/) nous a été d'une aide précieuse pour toute la durée du projet, ainsi que les « How-To » d'Oracle, surtout pour la création des `JTree`, `JList` et `JTable`.

Nous nous sommes souvent servis de *stackoverflow* (http://stackoverflow.com), une plateforme d'entraide.

Carnet de route & gestionnaire de version

Pour mener ce projet en équipe, nous avons utilisé *git*, un célèbre logiciel de version. Celui-ci a de nombreux intérêts, parmi eux : la possibilité de travailler à plusieurs en même temps sur un même fichier (grâce à son puissant outil de fusion et de détection des conflits), la possibilité de faire un *rollback* en cas de mauvaise manipulation ou d'erreur de programmation, et enfin et surtout avoir un suivi des modifications effectuées par chacun au cours du projet.

Git a donc fait office de carnet de route, nous vous invitons à consulter les sources de notre projet sur Github pour en savoir plus : https://github.com/Arsaell/EdT