

Gestion Emploi du temps

Livrable 0 - Projet

Matthieu Aubry
Jeremie Guidoux
Quentin Marguet
Mathieu Vénisse-Garson

Projet Multimodule

01/11/08

Encadrant : Gerson Sunyé

2008-2009 – Projet Multimodule -

Matthieu AUBRY : mas1.aubry@gmail.com

Jéréemie GUIDOUX : darkjer1@gmail.com

Quentin MARGUET : zorbitos@gmail.com

Mathieu VENISSE-GARSON : mathieu.venisse@gmail.com



UNIVERSITE DE NANTES

Sommaire

Projet.....3

1- Présentation.....3

2- Problématique 4

Projet

1- Présentation

L'objectif de ce projet est de mettre en pratique des concepts vu dans trois matières différentes : *génie logiciel*, *objets distribués* et *interface homme machine* (IHM). Pour cela, nous devons concevoir un système permettant la gestion distribuée d'un emploi du temps. Le travail que nous devons réaliser se divise en quatre étapes distinctes :

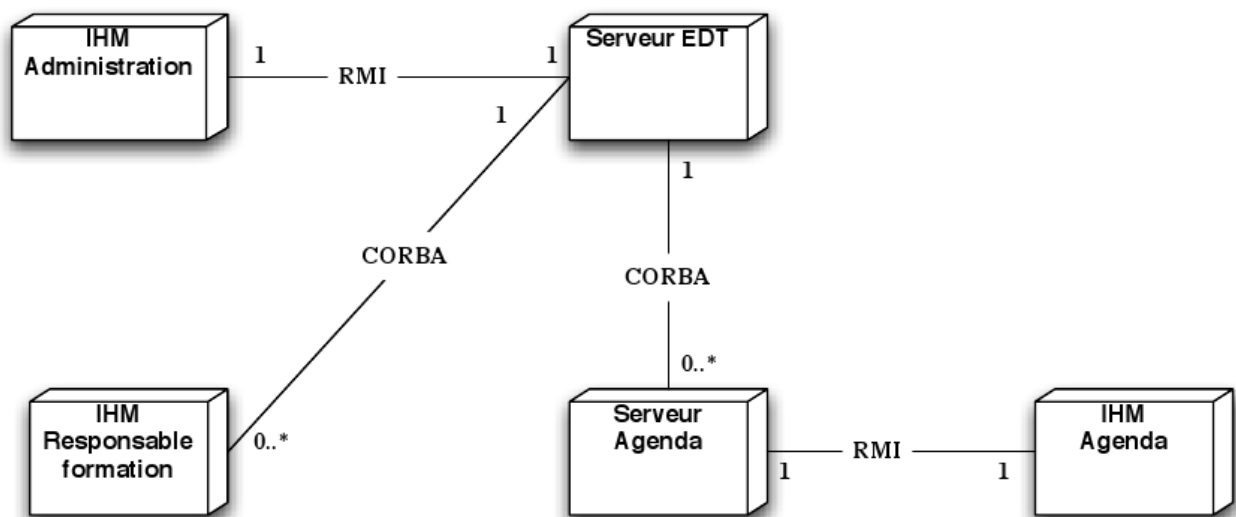
- Spécification des besoins (*cf 2*)
- Spécification des composants (*fait dans le livrable 2*)
- Architecture (*fait dans le livrable 3*)
- Conception détaillée (*fait dans le livrable 4*)

Ces quatre étapes présentes la démarche de conception complète d'une application, la première étape présente une étude de haut niveau de l'application à concevoir, la seconde présente les briques logiciels abstraites qui seront nécessaire à l'application et les différentes technologies utilisées pour les faire communiquer entre elles. La troisième étape présente une architecture globale de l'application ainsi que les schéma de base de donnée. Enfin la dernière étape est celle située juste avant l'implémentation, elle présente une description détaillée de tous les composants et permet d'implémenter le système sans avoir à ce poser de nouvelles questions. A chaque passage d'un niveau à un autre, la vision du système descend d'un niveau d'abstraction de manière à voir l'application de la manière la plus globale qui soit puis de plus en plus précisément.

2- Problématique

Comme expliqué brièvement dans la précédente partie, nous avons pour but la conception d'un système permettant la gestion distribuée d'un emploi du temps. Cet outils est un outils d'aide à la gestion, en conséquence il ne devra pas permettre l'optimisation automatique au niveau de l'affectation des enseignements. Le système sera composés de cinq nœuds logiques : deux serveurs et trois interfaces graphiques. Pour les communication entre nœuds logiques nous devrons utiliser deux technologies différentes : *Corba* et *RMI*.

Schéma explicitant l'interaction entre les différents acteurs qui composent le système :



Architecture

Gestion Emploi du temps

Livrable 1 – Spécification des besoins

Matthieu Aubry
Jeremie Guidoux
Quentin Marguet
Mathieu Vénisse-Garson

Projet Multimodule
[01/11/08](#)
Encadrant : Gerson Sunyé

2008-2009 – Projet Multimodule -

Matthieu AUBRY : mas1.aubry@gmail.com

Jéréemie GUIDOUX : darkjer1@gmail.com

Quentin MARGUET : zorbitos@gmail.com

Mathieu VENISSE-GARSON : mathieu.venisse@gmail.com



Sommaire

Spécification des besoins.....	3
1- Introduction.....	3
1.1- Présentation.....	3
1.2- Problématique.....	3
2- Dictionnaire des données.....	4
3- Cas d'utilisation.....	6
3.1- Présentation.....	6
3.2- Représentation graphique.....	7
3.3- Présentation des acteurs.....	7
3.4- Cas d'utilisation – Gestion agenda personnel.....	8
3.5- Cas d'utilisation – Gestion système EDT.....	9
3.6- Cas d'utilisation – Affectation des ressources.....	10
4- Cartes heuristiques.....	11
4.1- Présentation.....	11
4.2- Enseignant.....	11
4.3- Responsable.....	12
5- Diagrammes de séquences.....	13
5.1- Présentation.....	13
5.2- Diagramme de séquence – Authentification.....	13
5.3- Diagramme de séquence – Affecter une matière à un créneau horaire.....	15
5.4- Diagramme de séquence – Affecter un enseignant à une matière.....	17
5.5- Diagramme de séquence – Gestion de l'agenda personnel.....	18
5.6- Diagramme de séquence – Création d'une matière.....	19
6- Diagrammes d'activités.....	20
6.1- Présentation.....	20
6.2- Diagramme d'activité – Administrateur.....	21
6.3- Diagramme d'activité – Enseignant.....	22
6.4- Diagramme d'activité – Responsable de formation.....	23
7- Diagramme de classes niveau analyse.....	25
7.1- Présentation.....	25
7.2- Le diagramme.....	26
8- Ce que nous avons fait.....	28
9- Conclusion.....	28

Spécification des besoins

1- Introduction

1.1- Présentation

Cette première partie dans la démarche de conception de notre logiciel présente approche très abstraite du système dans son intégralité. Pour effectuer cette analyse, nous utilisons le langage *UML (Unified Modeling Language)* et plus particulièrement les diagrammes suivants :

- Un ensemble de cas d'utilisation, comprenant avec pour chacun leur description textuelle associé selon le canevas de *Cockburn* vu en cours
- Des instantanés illustrant l'impact des cas d'utilisation sur les objets
- Des diagrammes de séquences qui permettrons de présenté de manière général les étapes importantes d'exécution de notre système
- Un diagramme de classe de niveau analyse qui permettra de mettre en évidence les concepts clé qui seront ensuite étudiés plus en détails lors des phases suivantes

Nous présenterons aussi un dictionnaire de données qui permettra de détailler précisément tous les concepts que nous avons identifiés.

1.2- Problématique

Le système d'emploi du temps que nous devons modéliser doit permettre de mettre en relation trois ressources distinctes : des *enseignants*, des *salles* et des *étudiants*. Trois acteurs agissent sur le système : un *administrateur*, des *responsable de formation*, et des *enseignants*. La fonction de l'administrateur est de *gérer les ressources*, c'est lui qui les crée, les modifie et les supprime. Il s'occupe aussi de la *gestion des date* afin de déterminer les semestres et les jours fériés. Le responsable de formation réalise *l'affectation des enseignements*, c'est lui qui sera chargé de validé ou non un événement sur le calendrier. Enfin l'enseignant peut *gérer son emploi du temps personnel*, avec la possibilité de créer, modifier et supprimer ses évènements personnels, et *consulter les enseignements* auxquels il est affecté.

2- Dictionnaire des données

Le dictionnaire de données permet de présenter explicitement tous les termes utilisés dans ce projet et de faire le lien avec les concepts informatiques les modélisant.

Notion	Définition	Traduit en ...	Nom informatique
Gestion d'un emploi du temps	Créer les cours, supprimer les cours, créer un emploi du temps.	Classe	Responsable formation
Gestion des ressources	Créer une salle, une matière, un professeur, une formation	Classe	Administrateur
Emploi du temps	Regroupe les informations concernant une ou plusieurs personnes sur ce qu'elles ont à faire tel jour à telle heure.	Classe	EDT
Agenda personnel	Regroupe les informations d'une seule personne sur ses activités propre à elle.	Classe	AgendaPersonnel
Cours	Regroupe les informations d'un cours comme la salle, le professeur et la matière	Classe	Cours
Événement personnel	Regroupe les informations d'un événement personnel	Classe	Événement
Système	Comprend l'ensemble des données d'un établissement, comme les formations, les responsables formations, etc.	Package	Système
Jour	Permet de situer précisément un jour de l'année, cela comprend donc l'année, le mois et le numéro du jour, ainsi que la semaine	Classe	Jour
Créneau horaire	Définit les informations relative à un élément de l'emploi du temps, comme la durée, l'heure du début.	Classe	Créneau horaire
Enseignant	Il fait des cours, consulte son agenda, se connecte au système pour cela, et peut modifier son agenda personnel	Classe	Enseignant
Administrateur	Il enregistre des ressources (matières, professeurs, salles, etc...) dans une base de donnée persistante.	Classe	Administrateur

Notion	Définition	Traduit en ...	Nom informatique
Responsable de formation	Il affecte les matières à des créneaux horaires, les enseignants à des cours... Finalement, il construit l'emploi du temps.	Classe	Responsable
Formation	Regroupe les informations telles que les matières, les responsables de formations	Classe	Formation
Salle	Contient le nom de la salle telle qu'elle est désignée dans l'établissement	Classe	Salle
Matière	Contient le nom de la matière ainsi que son code unique au sein de l'établissement	Classe	Matière
Ressource	Cela correspond soit à une matière, une salle ou un enseignant...	Classe	
Conflit	Indique une incohérence au niveau de l'emploi du temps.		

Action	Définition	Traduit en ...	Nom informatique
Connexion	Permet à un utilisateur de se connecter au système	Opération	connexion
Créer cours	Permet d'affecter à un cours un professeur, une salle et une matière	Opération	créer_cours
Créer un emploi du temps	Permet de créer un emploi du temps et de lui donner un nom	Opération	créer_EDT
Supprimer un cours	Permet de supprimer un cours	Opération	supprimer_cours
Consulter	Permet de consulter un emploi du temps ou un agenda personnel	Opération	consulte

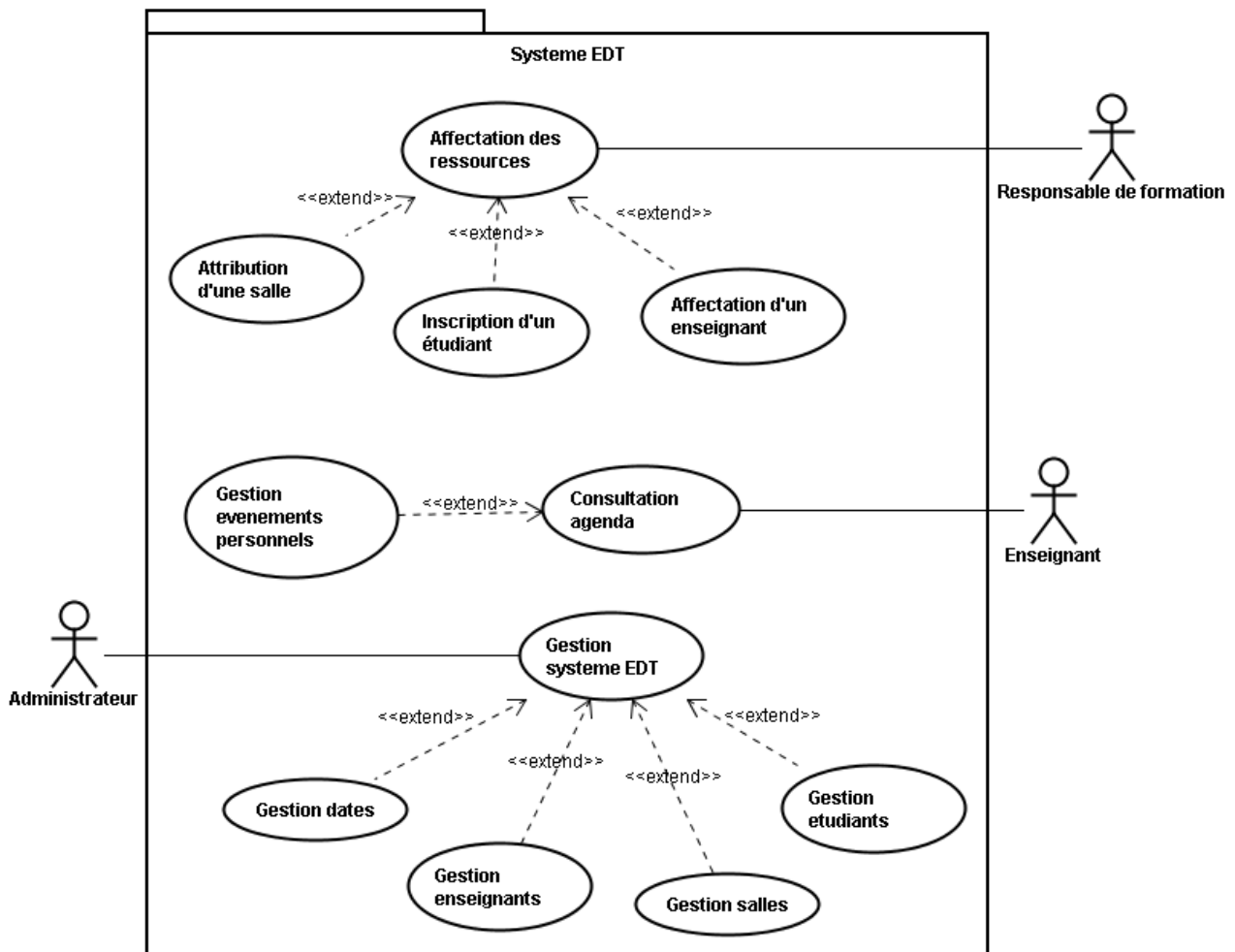
3- Cas d'utilisation

3.1- Présentation

Les cas d'utilisation permettent de décrire les objectifs et les interactions entre le système et ses acteurs. Chaque cas d'utilisation est accompagné d'une description textuelle permettant de définir de façon précise les différentes notions introduites. Nous avons choisi pour ce projet d'utiliser le format de représentation d'*Alistair Cockburn*.

Cette étape représente le départ de toute réflexion concernant la conception d'application informatique, elle présente une vue abstraite du système entier. Dans le paragraphe décrivant la problématique générale (*cf* 2.2.2) nous avons pris soin de mettre en italique certains termes. Ce sont, pour nous, les concepts essentiels de l'application et la base de notre raisonnement pour la réalisation de ces cas d'utilisation. En effet, ce texte représente le dialogue entre le client et les concepteurs du système, c'est donc le point de départ. A partir de l'étude de ce texte, nous avons pu établir le diagramme présenté dans la prochaine partie.

3.2- Représentation graphique



3.3- Présentation des acteurs

Notre cas d'utilisation se compose de trois acteurs :

- L'administrateur : il alimente la base de données du système en introduisant les ressources nécessaires (dates, enseignants, salles, étudiants, ...)
- Le responsable de formation : il s'occupe de l'affectation des enseignements et gère les conflits au niveau de l'emploi du temps
- L'enseignant : il gère son emploi du temps personnel et indique ses disponibilités

3.4- Cas d'utilisation – Gestion agenda personnel

Cas d'utilisation	Gestion agenda personnel		
Acteur	Enseignant		
Parties prenantes et intérêts	<u>Enseignant</u> : il veut consulter son emploi du temps (les enseignements) et/ou créer, modifier, supprimer des événements personnels ¹ . <u>Système EDT</u> : les informations qui sont présentes doivent être à jour afin que les autres acteurs (notamment le responsable de formation) interagissent avec des données contractuelles.		
Niveau	Objectif utilisateur		
Portée	Système EDT		
Pré-conditions	L'enseignant possède ses identifiants		
Post-conditions	Les informations ont été enregistrées.		
Scénario nominal	Étapes	Action	
	1	L'enseignant s'identifie au système avec ses identifiants.	
	2	Le système présente les informations actuellement présentes (l'agenda).	
	3	L'enseignant se déconnecte du système	
Extensions	Étapes	Condition	Action
	*	A tout moment	L'enseignant peut quitter l'action en cours
	1.a	L'identification échoue	Une message d'erreur apparaît et retour en 1
	2.a	L'enseignant veut ajouter/modifier/supprimer ses évènements personnels	Sous cas : gestion évènements personnels
	2.b	Modification des données échouées	Un message d'erreur apparaît et retour
Contraintes	Type	Description	
	Temps de réponse	L'identification ne doit pas dépasser plus de 10 secondes	
Priorité	Élevée (3/5)		
Performance			
Fréquences	Dès que l'enseignant souhaite consulter son agenda ou souhaite signaler une indisponibilité ou une tranche horaire réservé à une activité extra-scolaire.		

¹ Activité extra-scolaire qui indique une indisponibilité de l'enseignant pour dispenser un cours.

3.5- Cas d'utilisation – Gestion système EDT

Cas d'utilisation	Gestion système EDT		
Acteur	Administrateur		
Parties prenantes et intérêts	<u>Administrateur</u> : il veut créer, modifier et supprimer une ressource ² qu'il a préalablement choisit. Cela afin d'avoir des informations en concordance avec la réalité. <u>Système EDT</u> : les informations qui sont présentes doivent être à jour afin que les autres acteurs interagissent avec des données contractuelles.		
Niveau	Objectif utilisateur		
Portée	Système EDT		
Pré-conditions	L'administrateur possède ses identifiants.		
Post-conditions	Les informations ont été enregistrées.		
Scénario nominal	Étapes	Action	
	1	L'administrateur s'identifie au système avec ses identifiants.	
	2	Le système présente les informations actuellement présentes (ressources).	
	3	L'administrateur gère les ressources (création, modification, suppression).	
	4	L'administrateur soumet les modifications au système.	
	5	Le système enregistre les modifications et signale le succès de l'opération.	
Extensions	Étapes	Condition	Action
	*	A tout moment	L'administrateur peut quitter l'action en cours
	1.a	L'identification échoue	Une message d'erreur apparaît et retour en 1
	3.a	Nouvelle salle	Sous cas : Gestion salles
	3.a	Nouvel étudiant	Sous cas : Gestion étudiants
	3.a	Nouveau jour important ³	Sous cas : Gestion dates
	3.a	Nouvel enseignant	Sous cas : Gestion enseignants
	4.a	Modification des données échouées	Un message d'erreur apparaît et retour en (3 ou 4)
Contraintes	Type	Description	
	Temps de réponse	L'identification ne doit pas dépasser plus de 10 secondes	
Priorité	Très importante (5/5)		
Performance			
Fréquences	Dès que l'administrateur estime que la réalité diffère des informations contenues dans le système.		

² Une formation, un enseignement, un enseignant, une salle, etc.

³ Jour de début ou de fin d'un semestre, jour férié

3.6- Cas d'utilisation – Affectation des ressources

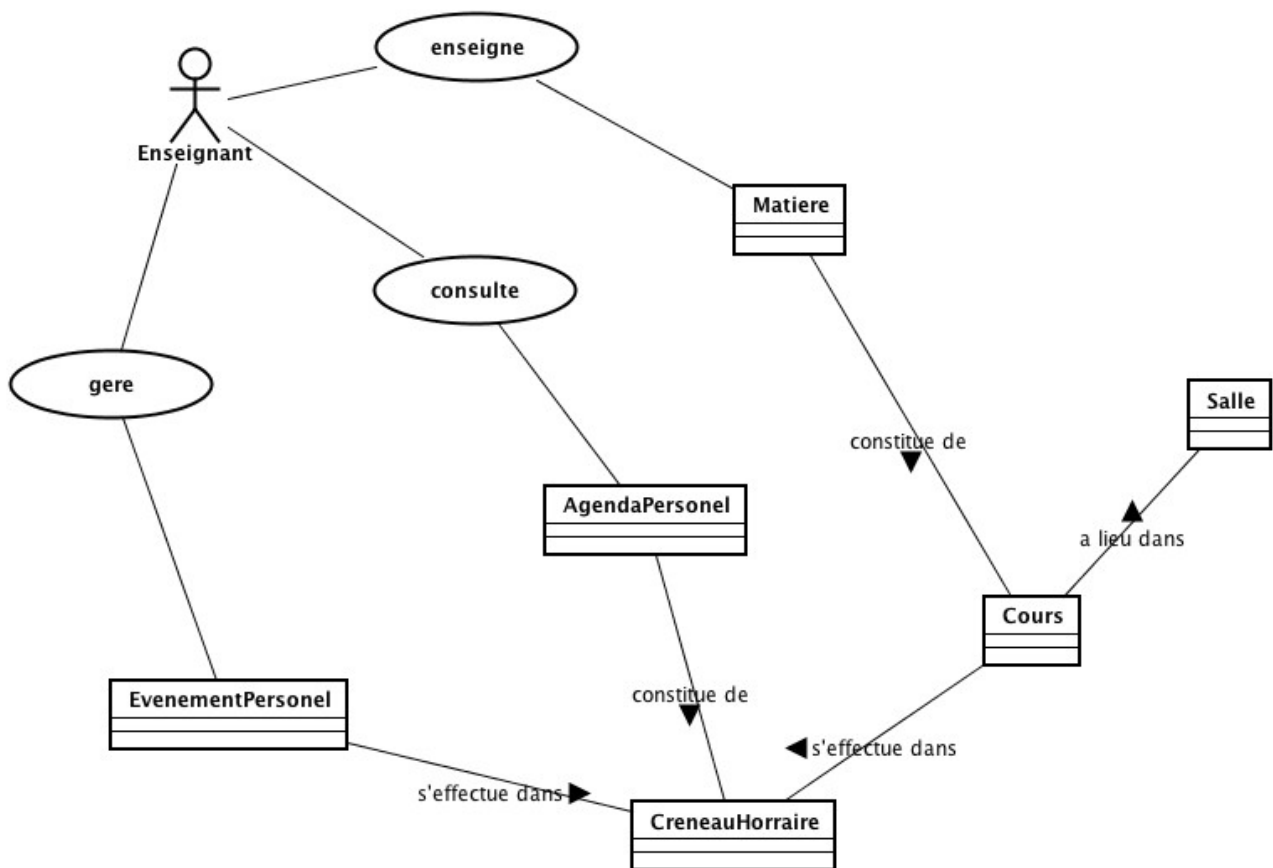
Cas d'utilisation	Affectation des ressources		
Acteur	Responsable de formation		
Parties prenantes et intérêts	Administrateur : il veut affecter et/ou mettre à jour les ressources créées par l'administrateur afin d'avoir des informations en concordance avec la réalité. Système EDT : les informations qui sont présentes doivent être à jour afin que les autres acteurs interagissent avec des données contractuelles.		
Niveau	Objectif utilisateur		
Portée	Système EDT		
Pré-conditions	Le responsable de formation possède ses identifiants et des ressources existent.		
Post-conditions	Les informations ont été enregistrées.		
Scénario nominal	Étapes	Action	
	1	L'administrateur s'identifie au système avec ses identifiants.	
	2	Le système présente les informations actuellement présentes (ressources).	
	3	L'administrateur gère les ressources (création, modification, suppression).	
	4	L'administrateur soumet les modifications au système.	
	5	Le système enregistre les modifications et signale le succès de l'opération.	
Extensions	Étapes	Condition	Action
	*	A tout moment	L'administrateur peut quitter l'action en cours
	1.a	L'identification échoue	Une message d'erreur apparaît et retour en 1
	3.a	Attribution d'une salle	Sous cas : Attribution d'une salle
	3.a	Nouvel étudiant	Sous cas : Inscription d'un étudiant
	3.a	Nouvel enseignant	Sous cas : Affectation d'un enseignant
	4.a	Modification des données échouées	Un message d'erreur apparaît et retour en 5
Contraintes	Type	Description	
	Temps de réponse	L'identification ne doit pas dépasser plus de 10 secondes	
Priorité	Très importante (5/5)		
Performance			
Fréquences	Dès que le responsable de formation considère que le système doit être mis à jour		

4- Cartes heuristiques

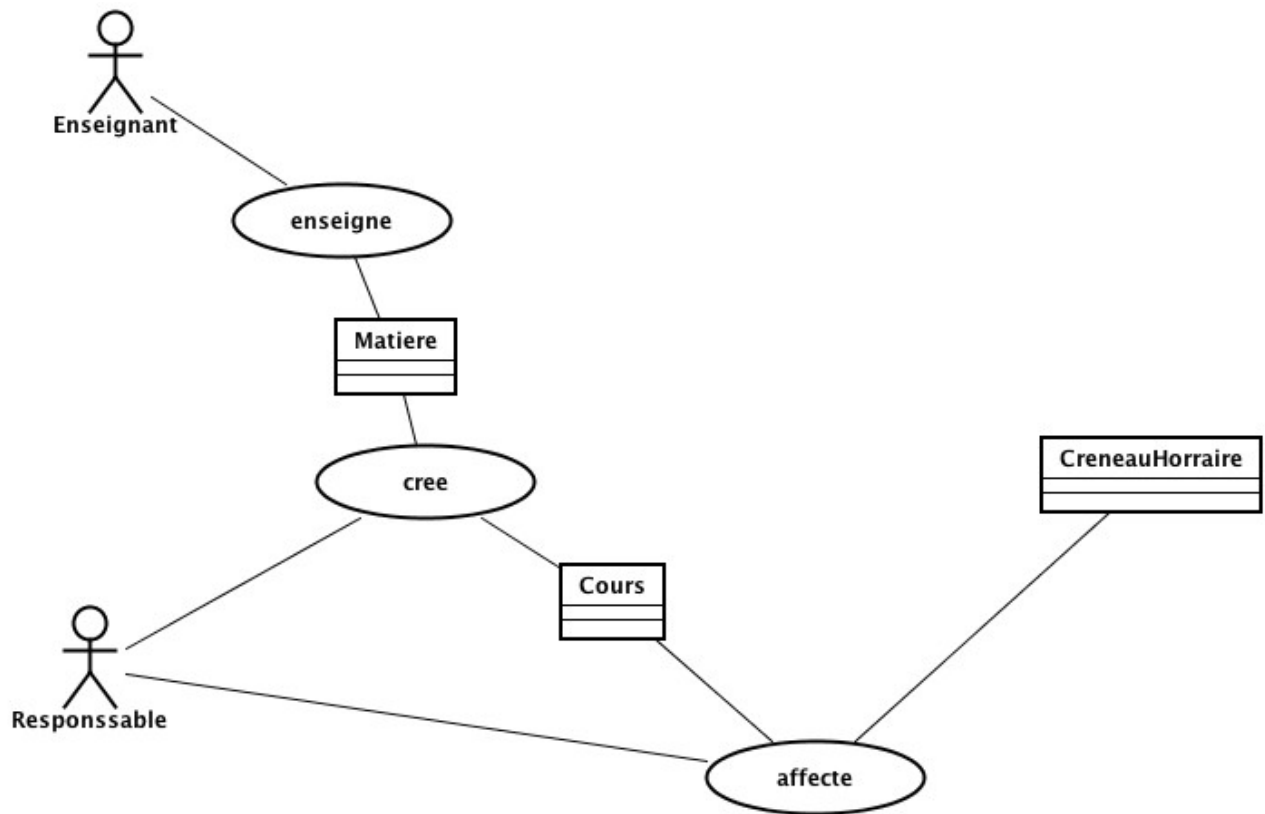
4.1- Présentation

Les deux diagrammes qui suivent sont des représentation informelle de l'organisation des données. L'interaction entre les différentes données et/ou acteurs du système est mise en valeur, ils permettent de mieux comprendre la modélisation du système et complète les connaissances définies dans les cas d'utilisation et le dictionnaire de données.

4.2- Enseignant



4.3- Responsable



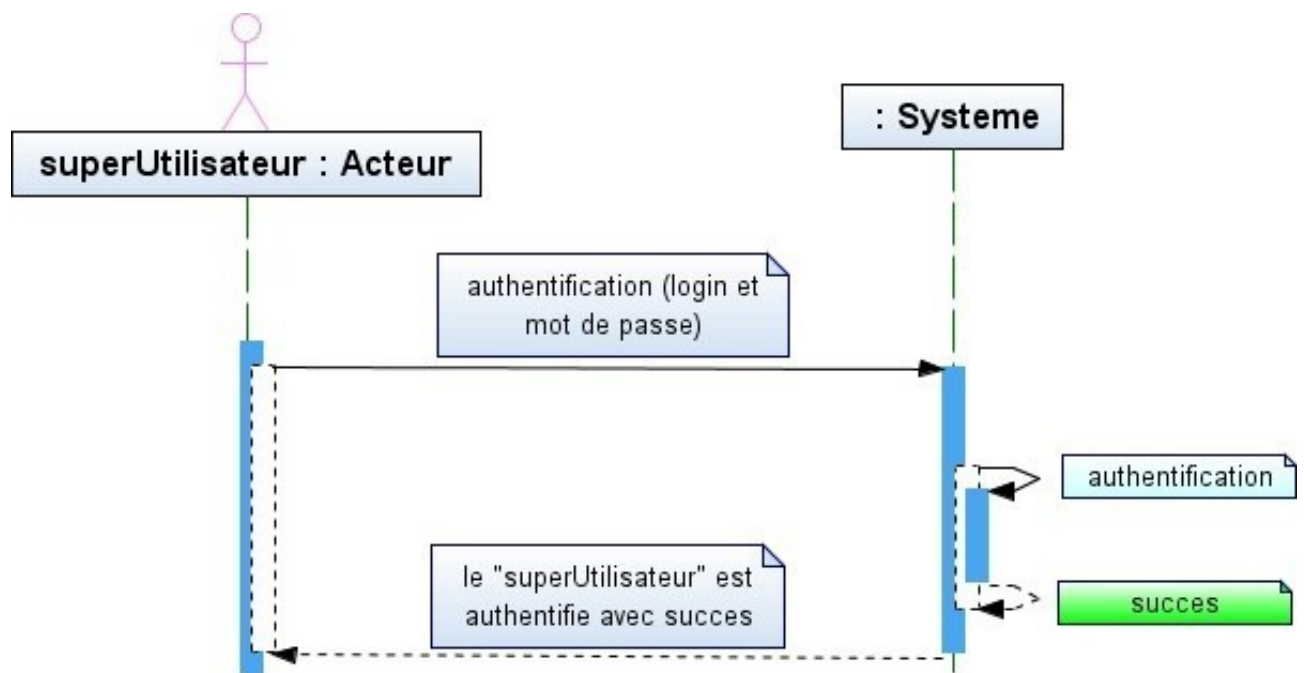
5- Diagrammes de séquences

5.1- Présentation

Les *diagrammes de séquences* sont la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique, il permettent de définir plus précisément que dans les cas d'utilisation le principe de fonctionnement de certaines phases de l'application. De plus, ils mettent en valeur certain choix de modélisation, nous en feront la remarque lorsque cela sera nécessaire.

5.2- Diagramme de séquence – Authentification

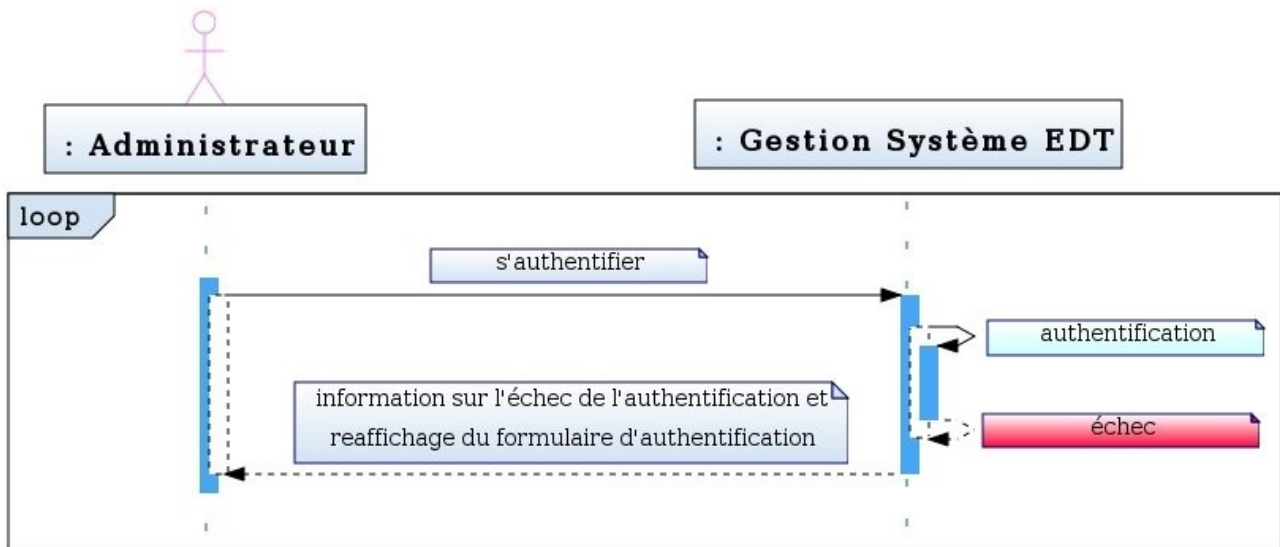
Succès :



Remarque :

- Ce diagramme est très simple à comprendre, il présente l'authentification d'un acteur par rapport au système. L'acteur peut être soit l'administrateur, soit un enseignant, soit un responsable de formation. La procédure est la même pour tous et utilise un système de login/mot de passe.

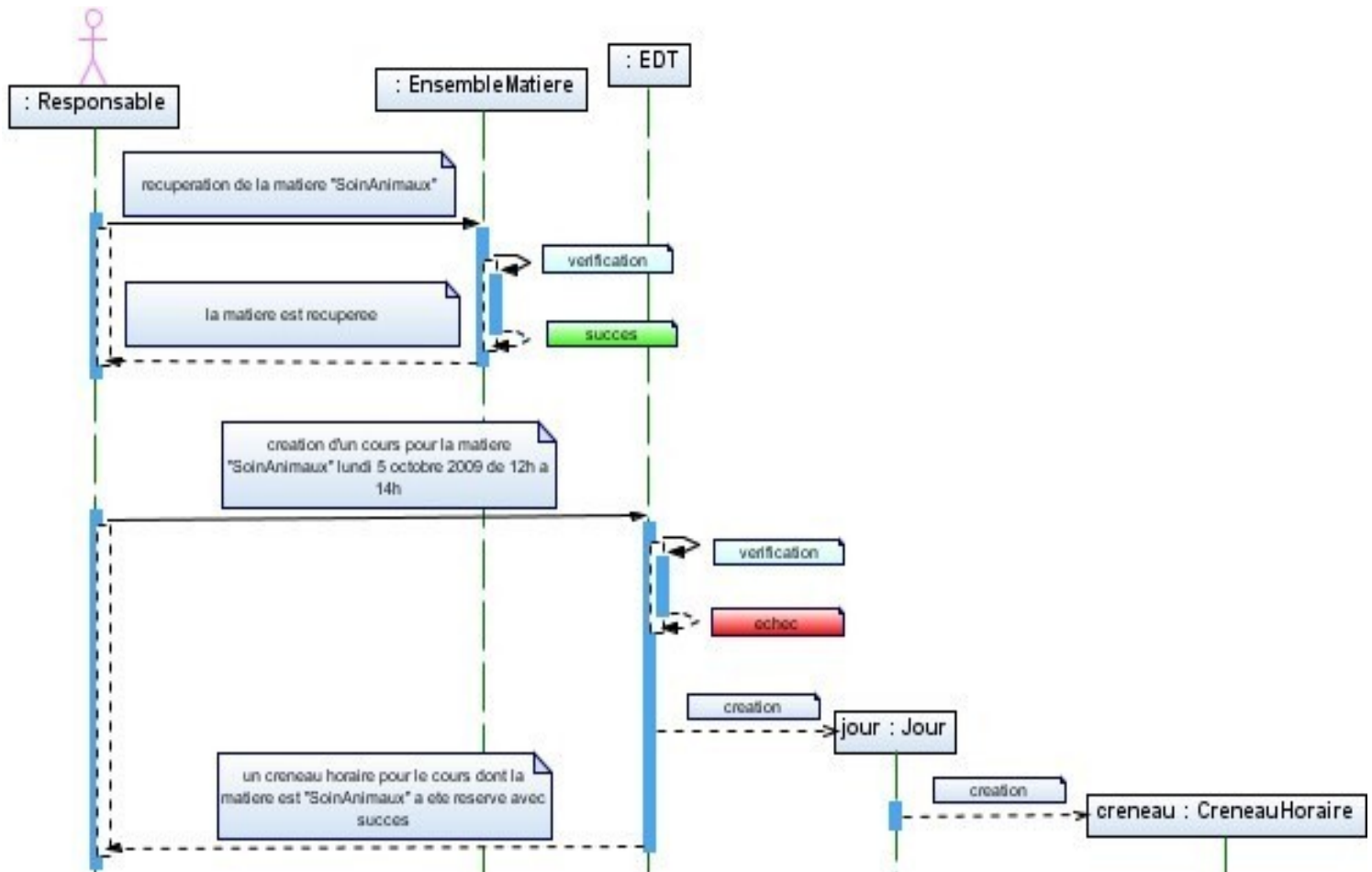
Échec :



Remarques :

- Le diagramme de séquence ci-dessus montre une erreur lorsque l'administrateur souhaite s'authentifier. Il permet notamment de mettre en avant que cette étape peut être infinie si l'erreur se répète. Une sécurité pour éviter/limiter les possibilités de piratage de compte pourrait être de mettre une temporisation après un certain nombre d'échecs.
- Ce diagramme est aussi valable pour le responsable de formation et l'enseignant.

5.3- Diagramme de séquence – Affecter une matière à un créneau horaire



Remarques :

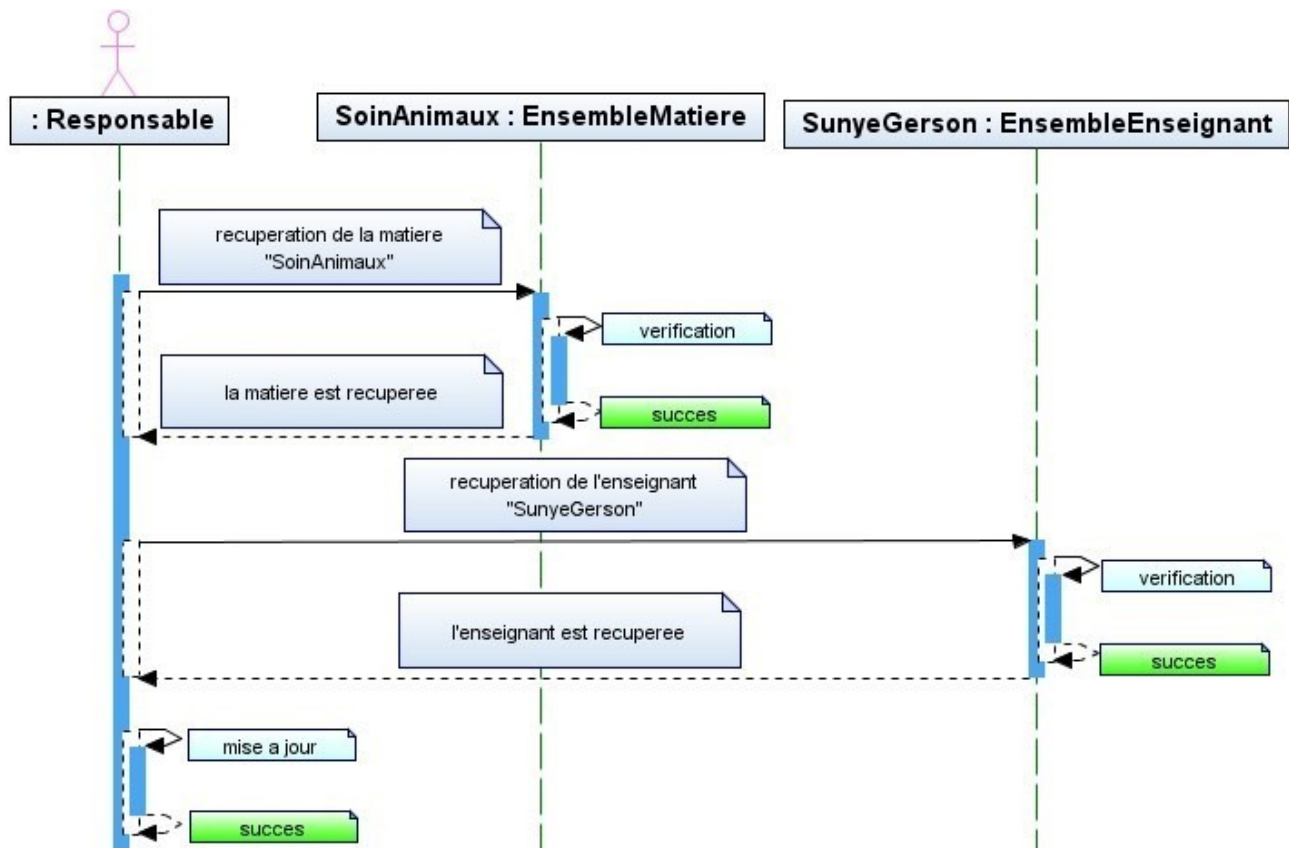
Dans ce diagramme de séquence, le responsable de formation va prévoir une date pour un cours dont la matière est « SoinAnimaux ». Ce cas d'utilisation fait apparaître la complexité du système que nous devons modéliser. Nous avons des données complexes et liées à manipuler :

- une matière
- une date
- un enseignant
- une salle

Nous avons choisi de ne pas traiter tout d'un coup. Le responsable va donc d'abord s'occuper de choisir un créneau horaire pour une matière. Le choix de la salle et de l'intervenant interviendra après. Nous dissociions ainsi les contraintes matérielles (ressources) des contraintes liées à l'enseignement. Nous remarquons également ici que le jour est créé dynamiquement (s'il n'existe pas déjà) ainsi que le créneau horaire. L'échec montre que le jour n'existait pas encore.

Le fait que les jours soient créés dynamiquement laisse penser qu'il y a des risques d'accès à des jours inexistants. De ce fait (nous nous avançons peut-être un peu à ce niveau de l'analyse), mais le patron *Null Object* devrait permettre de gérer l'accès à des jours inexistants.

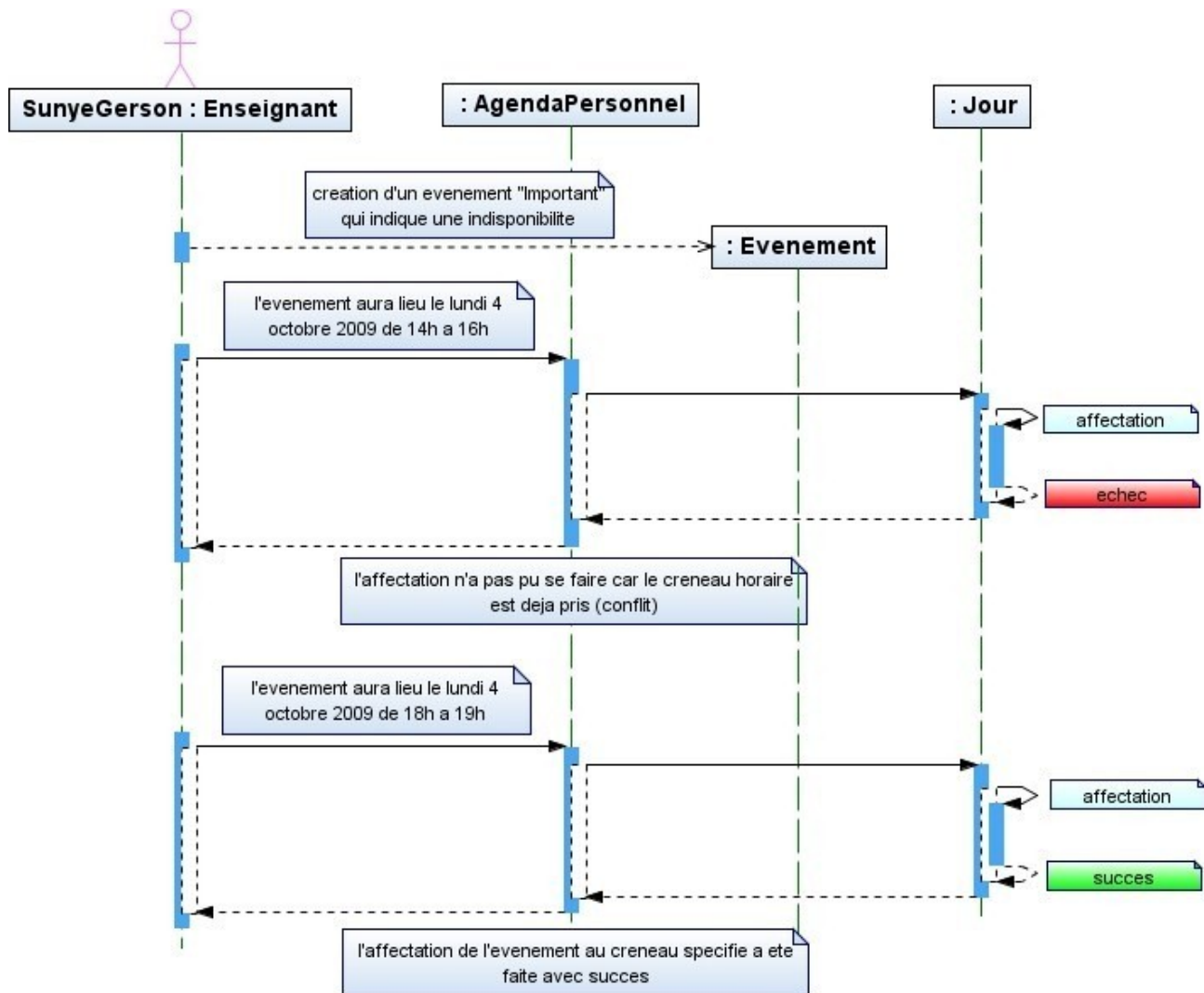
5.4- Diagramme de séquence – Affecter un enseignant à une matière



Remarque :

Le responsable de formation veut affecter une matière à un enseignant. Il doit donc récupérer l'instance de la matière et celle de l'enseignant pour les lier. A l'heure actuelle, nous estimons que la liaison doit être croisée (les deux instances peuvent se retrouver). Ce diagramme de séquence bien que simpliste fait apparaître une subtilité de notre modélisation. L'enseignant étant lié à la matière, il serait envisageable qu'il enseigne tous les cours de celle-ci. Cela serait simple à mettre en place mais particulièrement rigide. Notre modélisation permet d'éviter ce problème. En effet, plusieurs enseignants peuvent être liés à une matière et en conséquence les cours pourront être repartis entre différents enseignants.

5.5- Diagramme de séquence – Gestion de l'agenda personnel

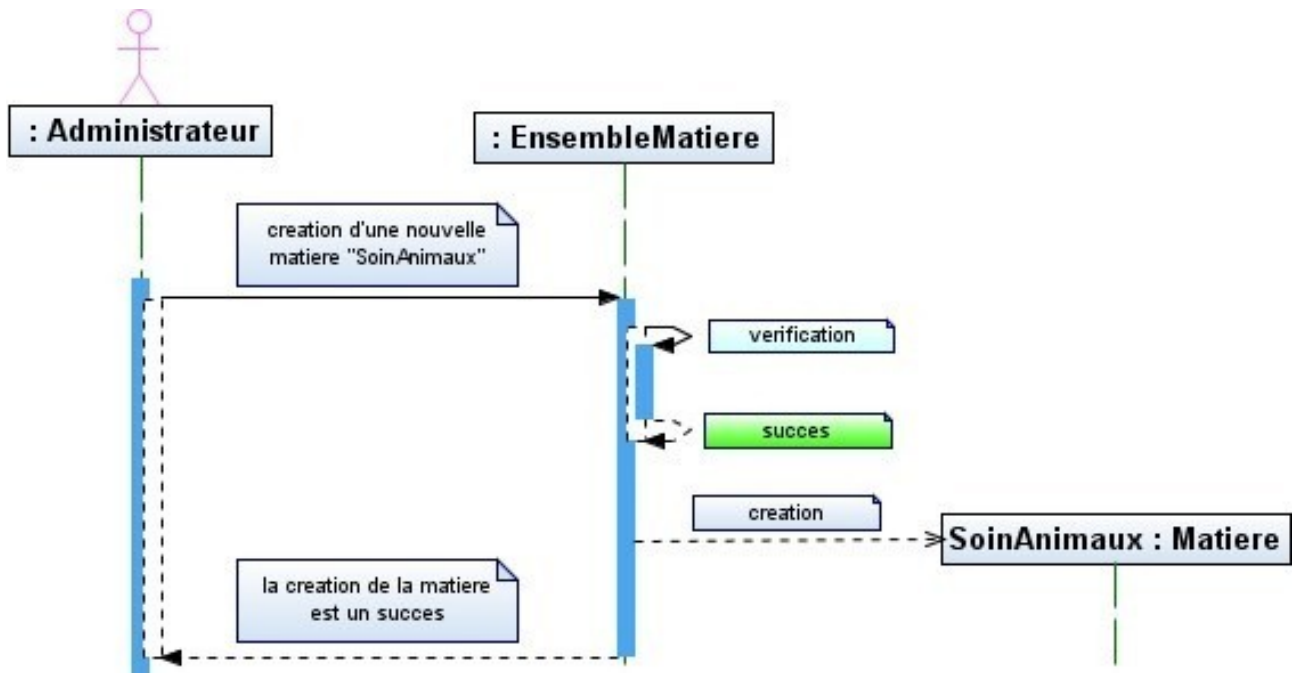


Remarque :

L'enseignant veut indiquer qu'il sera indisponible à une date donnée. Pour ce faire, il doit préalablement créer un événement qui contient le « pourquoi » de son absence. Il spécifie ensuite la date avec notamment les heures durant lesquels, il sera indisponible. Le système va ensuite (essayer) d'enregistrer l'événement. Dans ce cas d'utilisation, la première affectation échoue car il existe un autre événement (cours ou événement personnel) qui rentre en conflit. Les spécifications actuelles ne permettent pas à l'enseignant de signaler son absence alors que le créneau est occupé par un cours. Il faudra donc trouver un moyen de gérer cela. Le livrable 2 devait corriger ce problème. Finalement, l'enseignant réussit à affecter son événement à un créneau horaire.

La gestion des conflits intervient aussi un peu de la même manière du côté du responsable de formation.

5.6- Diagramme de séquence – Création d'une matière



Remarque :

L'administrateur souhaite créer une nouvelle matière, tout d'abord le système vérifie si la matière n'existe pas déjà, si tel est le cas la matière est créée.

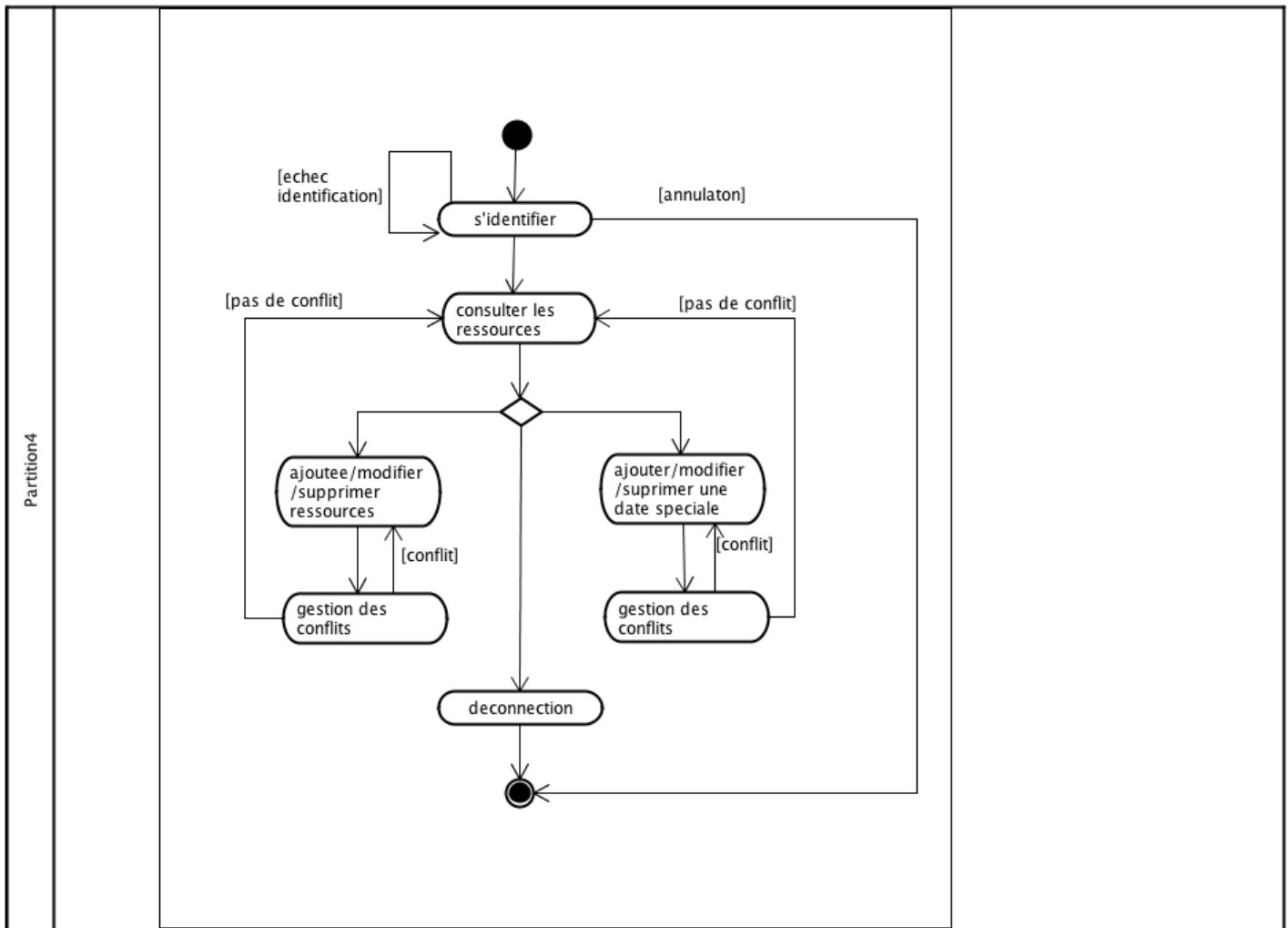
6- Diagrammes d'activités

6.1- Présentation

Les *diagrammes d'activité* permettent de modéliser un processus interactif, global ou partiel pour un système donné, ils mettent en valeur les différents état du système et leur enchaînement d'un point de vue temporel. Ils présentent une vision macroscopique et temporelle du système modélisé. Nous présenterons trois diagrammes différents, un pour chaque acteur de nos cas d'utilisation : *administrateur*, *enseignant* et *responsable de formation*.

6.2- Diagramme d'activité – Administrateur

Représentation graphique :



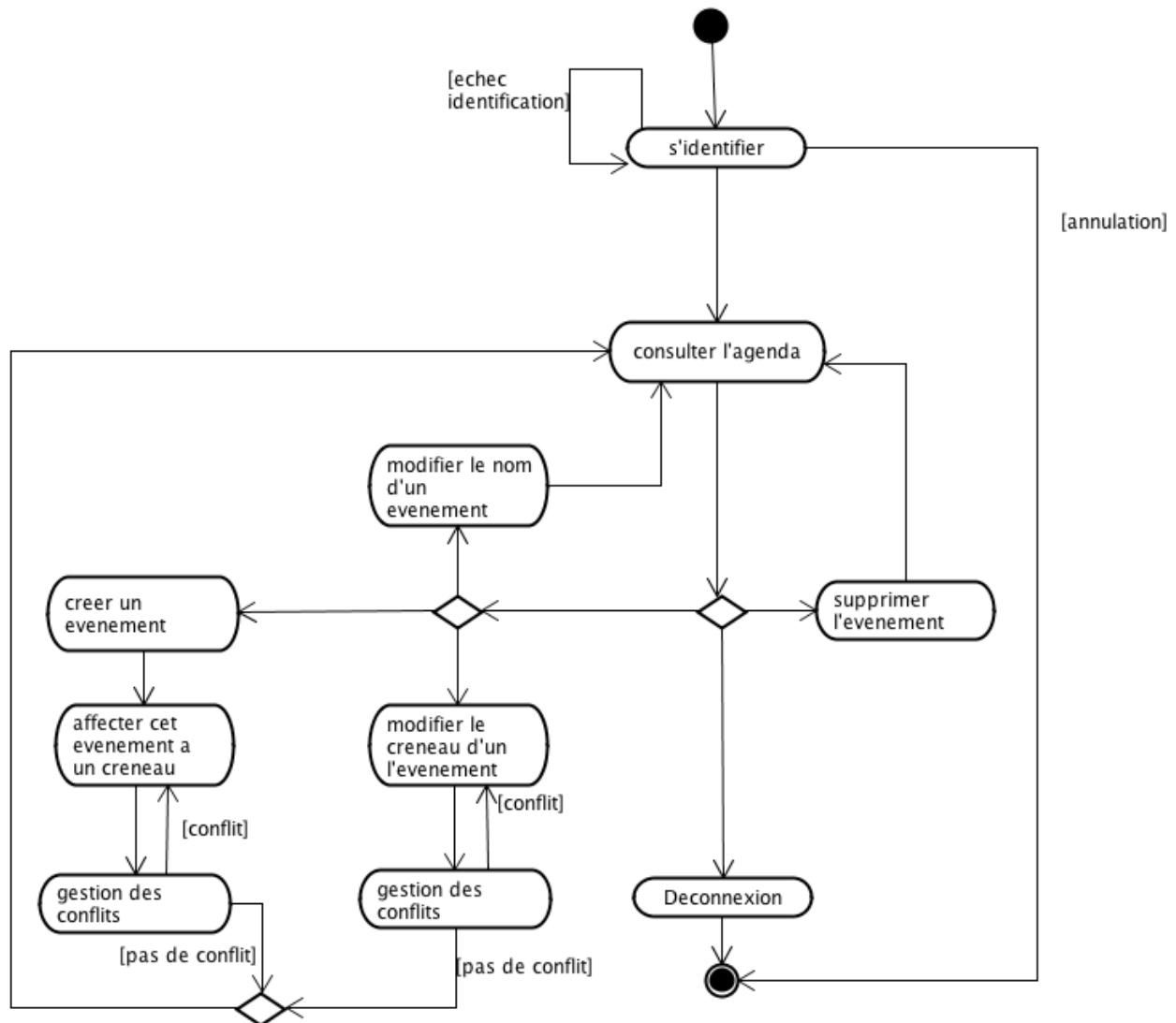
Commentaire :

Ce diagramme présente les actions de l'administrateur par rapport au système. L'administrateur est celui qui ajoute à la base de données les différentes ressources (matière, enseignant, date, étudiants).

Après s'être identifier sur le système à accès automatiquement aux ressources et peut ensuite ajouter, modifier ou supprimer une ressources. Chacune de ses actions entraîne une gestion des conflits : créer une salle, un enseignant ou une matière qui existe déjà, supprimer une salle, un enseignant ou une matière qui n'existe pas.

6.3- Diagramme d'activité – Enseignant

Représentation graphique :

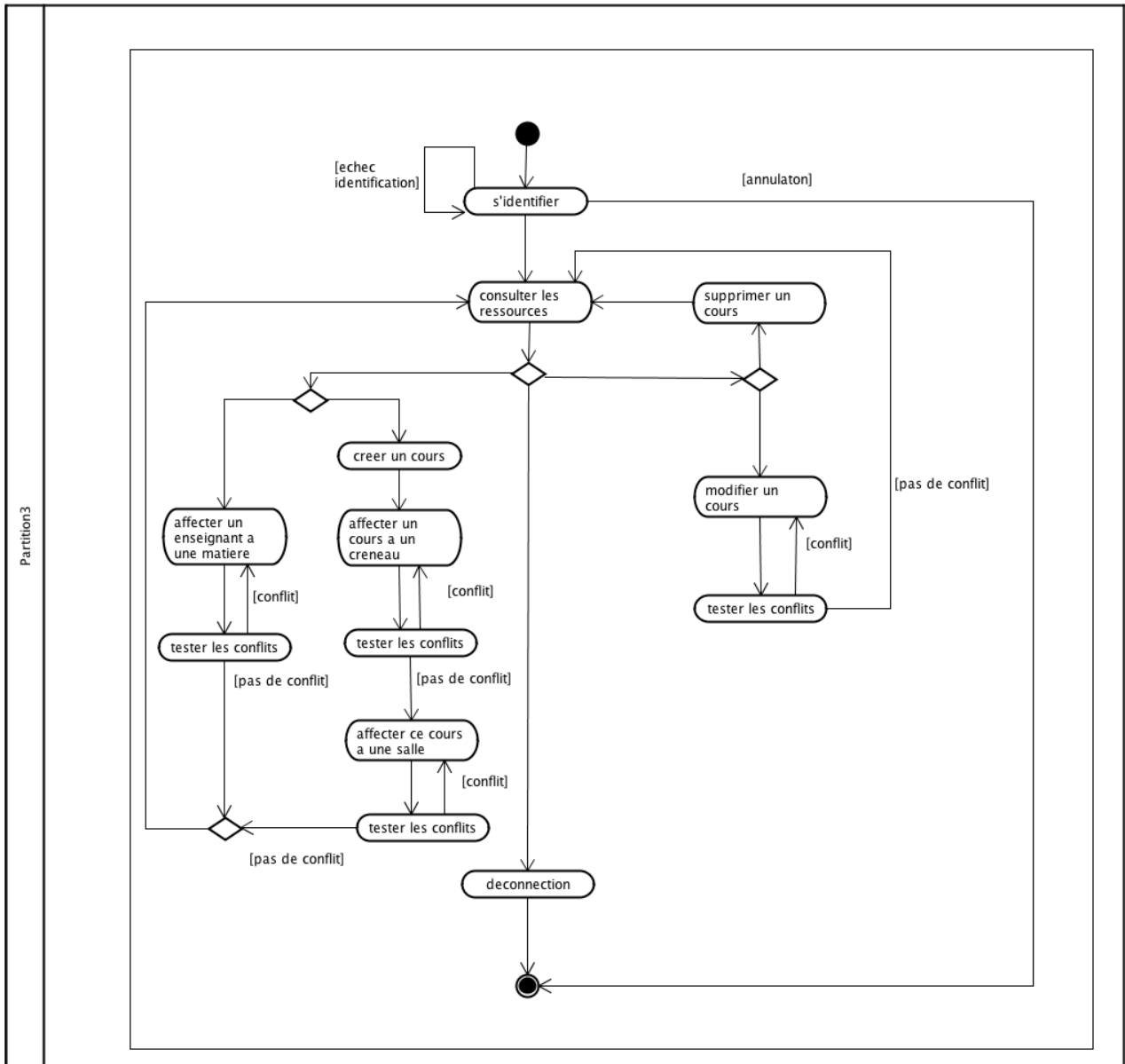


Commentaire :

Ce diagramme présente les actions d'un enseignant par rapport au système. Après s'être identifié, il peut créer ou supprimer un événement personnel, modifier son nom ou le changer de créneau horaire. Lorsque qu'un événement est créé, il faut l'affecter à un créneau horaire. Un identifiant étant généré automatiquement lors de la création d'un événement, les gestion des conflits n'est pas nécessaire à ce niveau là. C'est seulement lorsque l'on affecte l'évènement personnel à un créneaux horaires qu'il faut vérifier qu'il n'y a pas de conflits : qu'aucun autre événement n'a (personnel ou non) n'a lieu sur le même créneau.

6.4- Diagramme d'activité – Responsable de formation

Représentation graphique :



Commentaire :

Ce diagramme présente les actions d'un responsable de formation par rapport au système. C'est l'acteur le plus important du système, en effet l'administrateur ne fait que créer les ressources et l'enseignant ne fait que les consulter (excepté les événements personnels). Le responsable de formation est donc celui qui crée, modifie, supprime le plus fréquemment des ressources dans le système.

Après s'être identifié par rapport au système, il accède aux ressources automatiquement afin de pouvoir les consulter. À partir de là il peut créer ou modifier ou supprimer un cours ou affecter un enseignant à une matière. Par modifier un cours nous nous entendons en réalité deux actions : modifier les informations liées directement au cours comme son nom par exemple et affecter le cours à un nouveau créneau horaire. Cette dernière action entraîne une gestion des conflits afin de savoir si le créneau n'est pas déjà utilisé. La création d'un cours entraîne deux tests de conflits, l'un testant si le créneau auquel est affecté le cours n'est pas déjà utilisé et l'autre testant si la salle dans laquelle le cours doit se passer n'est pas déjà réservée à cette heure-là.

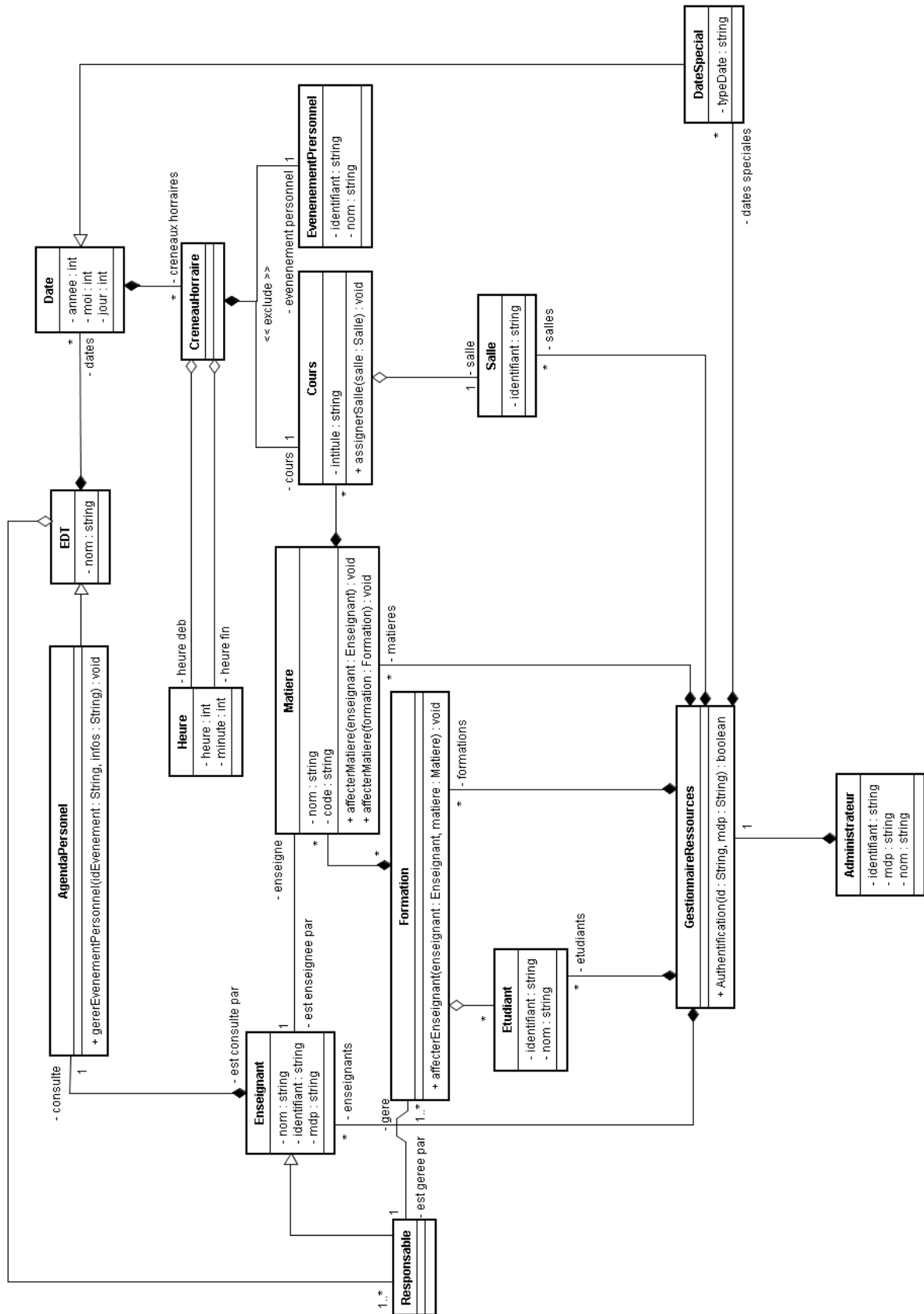
7- Diagramme de classes niveau analyse

7.1- Présentation

Ce diagramme permet de représenter la structure statique d'une application, à savoir les éléments (classes et types), la structure interne des éléments et leurs relations les uns par rapport aux autres. Le diagramme de classe que nous présentons ici est au niveau analyse, c'est à dire que chaque classe ne correspondra pas forcément à un objet par la suite. En effet, certaines classes pourront devenir des composants où d'autres structures plus ou moins complexes. Chaque classe représente ici un concept important qu'il est essentiel de modéliser pour ensuite pouvoir réaliser l'application sans utiliser de solution miracle. Il met en relation toutes les informations exposées dans les diagrammes précédents et présente un condensé de toutes les réflexions faites dans ce livrable.

7.2- Le diagramme

Représentation graphique :



Remarques :

- Un administrateur gère les ressources du système, à savoir création, modification et suppression des concepts suivants : salles, matières, enseignants, étudiants et dates spéciales grâce à la classe *GestionnaireRessources*. Chacun de ces concepts étant important, ils sont chacun représentés par une classe.
- Un responsable de formation est aussi un enseignant. Cette notion se traduit par la relation d'héritage entre les classes *Enseignant* et *Responsable*.
- Nous autorisons le fait qu'un responsable de formation puisse gérer une ou plusieurs formations, c'est pourquoi nous avons une multiplicité de 1..* entre la classe *Responsable* et la classe *Formation*.
- Pour chaque formation, le responsable choisit quel enseignant va enseigner quelle matière grâce à la méthode *affecterEnseignant* de la classe *Formation*.
- Chaque cours est affecté à une salle, ce concept est représenté par la relation d'aggrégation entre les classes *Cours* et *Salle*. Un cours est aussi placé dans l'emploi du temps (représenté par la classe *EDT*) à un créneau horaire (représenté par la classe *CreneauHoraire*) et à une date précise (représenté par la classe *Date*). Chaque cours dépend d'une matière.
- L'enseignant consulte son agenda personnel (classe *AgendaPersonnel*). Nous considérons que si l'enseignant se désinscrit, son agenda n'a plus lieu d'être, c'est pourquoi nous avons mis une relation de composition entre les classes *Enseignant* et *AgendaPersonnel*.
- L'agenda personnel d'un enseignant contient les cours, dont il est à la charge, et ses événements personnels (représenté par la classe *EvenementPersonnel*) qui ont lieu à une date précise à un créneau horaire précis.
- L'enseignant peut modifier un événement personnel, grâce à la méthode *gereEvenementPersonnel*, mais il ne peut pas modifier un cours.
- Un créneau horaire contient soit un cours, soit un événement personnel.
- L'emploi du temps ne contient que des cours, alors que l'agenda personnel peut contenir des cours et des événements personnels (modéliser par *exclude*).

8- Ce que nous avons fait

Cette première phase nous permet de poser les bases du système. C'est le passage obligé pour bien cerner les différentes forces en présence. Il permet notamment de bien définir le rôle de chacun des acteurs et ainsi de segmenter la vision du système. Nous estimons avoir effectué un travail de fond (au niveau de la réflexion) qui nous permettra d'avancer relativement sereinement pour la seconde phase : *spécification des composants*. La segmentation à laquelle nous faisons allusion nous permettra de mettre en parallèle et de découper en « briques » plus facilement le travail par la suite. Ce travail nous permet d'avoir une première vision abstraite du système, bien que l'ensemble du système soit lié, il y a des parties qui peuvent être traitées de manière indépendante.

9- Conclusion

Cette phase étant terminée, nous allons pouvoir passer à un niveau d'abstraction inférieur et commencer à penser à la structure de données au niveau composant. Cette première phase d'analyse nous permet d'entrevoir facilement comment modulariser l'application de rendre ses parties plus ou moins indépendante.

Gestion Emploi du temps

Livrable 2 – Spécification des composants

Matthieu Aubry
Jeremie Guidoux
Quentin Marguet
Mathieu Vénisse-Garson

Projet Multimodule
01/11/08
Encadrant : Gerson Sunyé

2008-2009 – Projet Multimodule -

Matthieu AUBRY : mas1.aubry@gmail.com

Jérémie GUIDOUX : darkjer1@gmail.com

Quentin MARGUET : zorbitos@gmail.com

Mathieu VENISSE-GARSON : mathieu.venisse@gmail.com



Sommaire

Spécification des composants.....	3
1- Présentation.....	3
1.1- Introduction.....	3
1.2- Les différents diagrammes.....	3
2- Le diagramme de composants	4
2.1- Présentation.....	4
2.2- Représentation graphique.....	5
2.3- Commentaires.....	5
3- Diagrammes de classes.....	6
3.1- Présentation.....	6
3.2- Diagramme de classes – Formations.....	6
3.3- Diagramme de classes – Gestionnaire ressources.....	8
3.4- Diagramme de classes – Agenda.....	10
4- Diagrammes de séquence.....	12
4.1- Présentation.....	12
4.2- Diagramme de séquence – Créer enseignant.....	13
4.3- Diagramme de séquence – Ajouter responsable.....	14
4.4- Diagramme de séquence – Créer un événement personnel.....	15
4.5- Diagramme de séquence – Assigner enseignant.....	16
4.6- Diagramme de séquence – Assigner cours (version 1).....	17
4.7- Diagramme de séquence – Assigner cours (version 2).....	18
5- Diagramme d'activité.....	19
6- Conclusion.....	21
7- Problèmes rencontrés.....	21

Spécification des composants

1- Présentation

1.1- Introduction

Dans la partie précédente, nous avons modéliser l'application à réaliser à un haut niveau d'abstraction. Le résultat principal de cette modélisation réside en la création du diagramme de classes niveau analyse (*cf* 2.7) qui peut-être considéré comme une finalité par rapport à tous les autres diagrammes (cas d'utilisation, séquences, ...). En effet il présente d'un point de vu structurel le programme et défini les principales notions à implémenter. Cependant, à ce niveau de l'analyse, le diagramme ne présente pas une solution suffisante pour nous permettre de concevoir l'application. Il ne présente qu'une structure abstraite du programme. Pour la rendre implémentable nous allons introduire, dans cette nouvelle partie, la notion de composants.

1.2- Les différents diagrammes

Dans cette partie nous allons une nouvelle fois utiliser le langage UML afin de continuer la modélisation de notre application. Cependant nous descendrons d'un niveau d'abstraction afin de spécifier plus en détails l'application. A la fin de cette partie nous aurons une vue de l'application se rapprochant plus de l'implémentation. Nous utiliserons les diagrammes suivant :

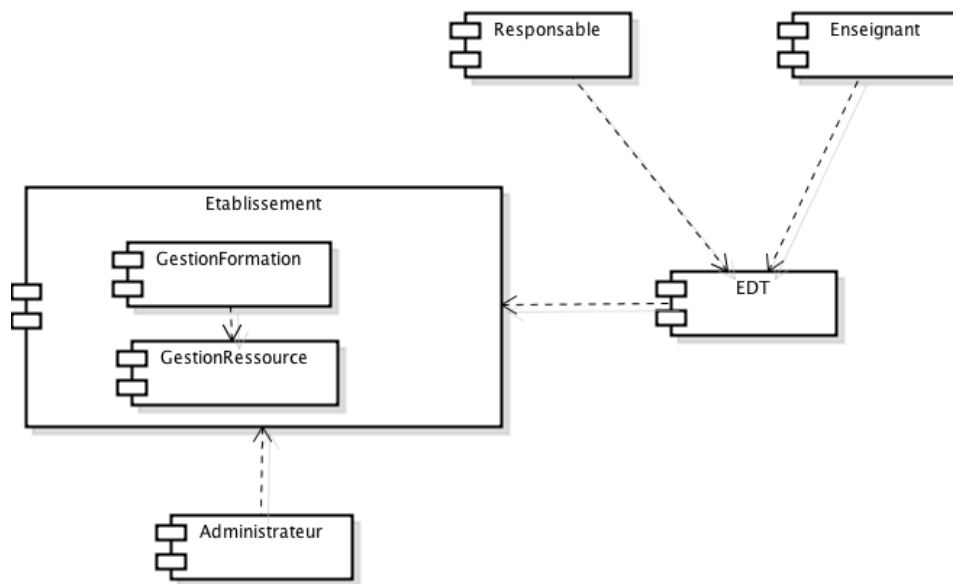
- Diagramme de composants qui permettront de diviser l'application en plusieurs sous parties indépendantes.
- Diagrammes de classes qui permettront de détailler chacun des composant définit dans le diagramme de composants.
- Diagrammes de séquences qui permettront de mettre en évidence les différentes méthodes relatives au classes de chacun des composants. De plus, il détaillerons le fonctionnement des composant d'un point de vue structurelle et temporelle et permettront d'explicitier les différents points important de notre modélisation.
- Diagramme d'activité mélangeant des noeuds d'actions et des noeuds objets. qui permet d'avoir une vision temporelle du système au niveau des objets.

2- Le diagramme de composants

2.1- Présentation

Ce diagramme présente les principaux éléments physiques et indépendants de l'application et comment ils interagissent entre eux. Il présente aussi les différentes interfaces des composants et donc les services offerts par ceux-ci. Afin de répondre au cahier des charges et de pouvoir par la suite générer automatiquement le code de l'application, nous avons utilisé le plug-in eclipse papyrus (<http://www.papyrusuml.org/>). Une fois que tous les diagrammes de classes et de composants seront réalisés et formellement définis, ce plug-in se chargera de générer automatiquement une bonne partie du code source de l'application.

2.2- Représentation graphique



2.3- Commentaires

Ce diagramme est réalisé à partir des informations du diagramme de classes niveau analyse. Nous avons définis cinq composants dont un composant composite :

- **Administrateur** : composant qui représente l'administrateur du système qui deviendra par la suite une interface graphique.
- **Responsable de formation** : composant qui représente un responsable de formation qui deviendra par la suite une interface graphique.
- **Enseignant** : composant qui représente un enseignant qui deviendra par la suite une interface graphique.
- **EDT** : composant qui représente l'emploi du temps et contient la base de données relative à cet emploi du temps ainsi que les classes permettant de la modifier.
- **Établissement** : composant composite contenant le gestionnaire de ressources et le gestionnaire de formation ainsi qu'une base de données qui sera utilisée uniquement par le composant *gestionRessources*. Elle contiendra toutes les informations concernant les ressources : enseignant, formation, étudiants, ...

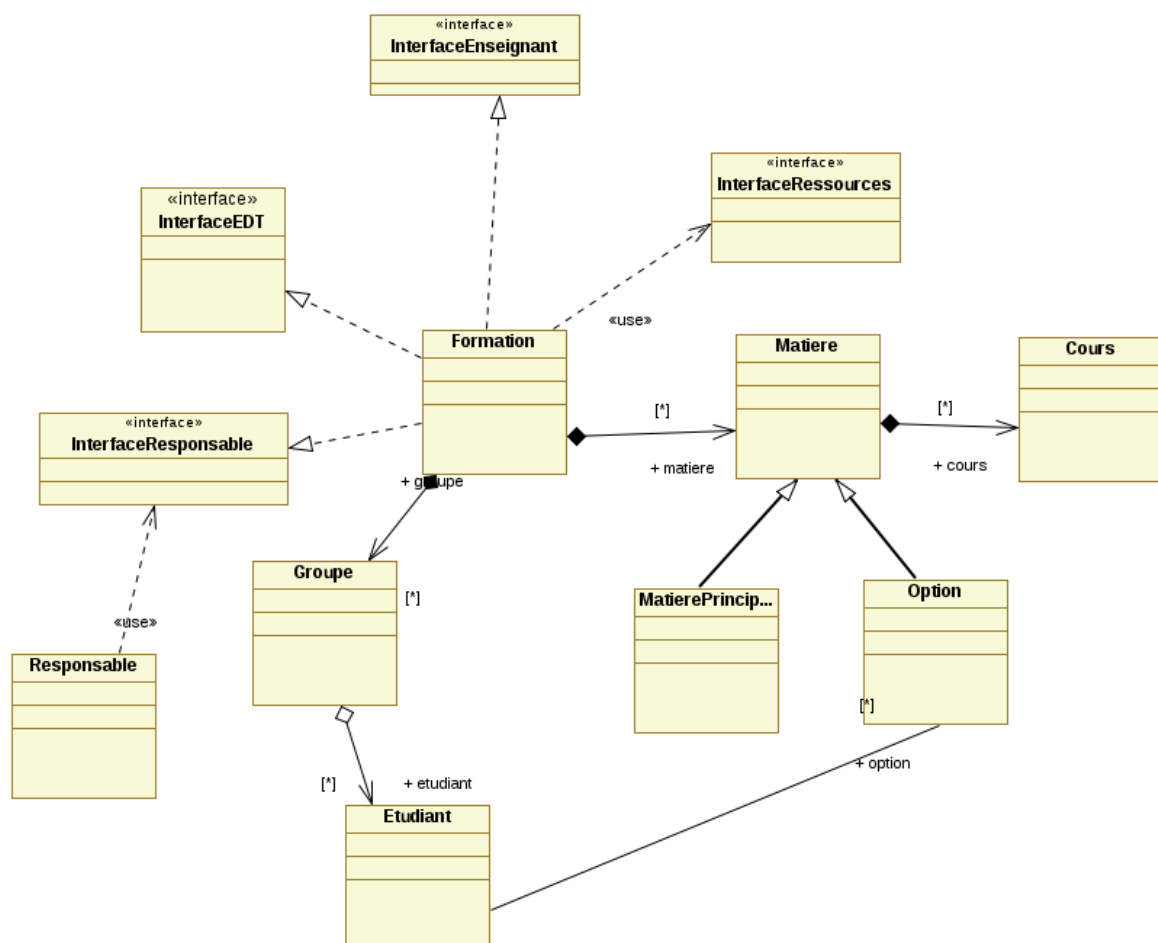
3- Diagrammes de classes

3.1- Présentation

Les diagrammes de classes que nous allons présenter dans cette partie vont permettre de détailler le contenu de chacun des composants définis dans la partie précédente. Ils présenteront d'un point de vue structurel comment les composants devront être implémenter par la suite en explicitant les différentes classes qui les composent. Bien entendu, ces différents diagrammes s'appuieront sur le diagramme de classes d'analyse présenter à la partie 2.7. Chaque diagramme de classe, définissant un composant, est donc indépendant des autres. Ils présentent les différents services offert par chacun des composant et le nombre d'interfaces dont ils disposent. De plus, ils présentent les méthodes principales nécessaires à la réalisation des composants.

3.2- Diagramme de classes – Formations

Représentation graphique :



Commentaires :

Ce diagramme permet de modéliser tout ce qui a trait à la formation. Si nous analysons l'ensemble, nous remarquons que cette modélisation pourrait être décrite comme une description de ce que représente le mot « formation ».

Une formation possède des matières qui vont donner lieu à produire des cours. La matière englobe deux sous-éléments qu'il faut distinguer. Il y a des matières principales, celles qui forment ce que nous appelons plus communément le « tronc commun » à laquelle s'ajoutent des options qui sont des spécificités. Le lien d'héritage permet de mettre en évidence cette spécification.

Une formation possède également des groupes d'étudiants. Ce que nous remarquons encore, c'est que l'objet « Etudiant » est relié à l'objet « Option » mais pas à l'objet « MatièrePrincipale ». Encore ici, nous souhaitons indiquer qu'une option est une forme de spécialisation. Un groupe a des matières principales et un étudiant a en plus des options.

Maintenant que nous avons spécifié le cœur du système, nous allons voir comment cette sous-partie du programme interagit avec les autres composants.

Les interactions avec le reste du programme sont faites par l'intermédiaire d'interfaces. Nous avons distingué 4 interfaces :

- InterfaceRessources
- InterfaceEnseignant
- InterfaceEDT
- InterfaceResponsable.

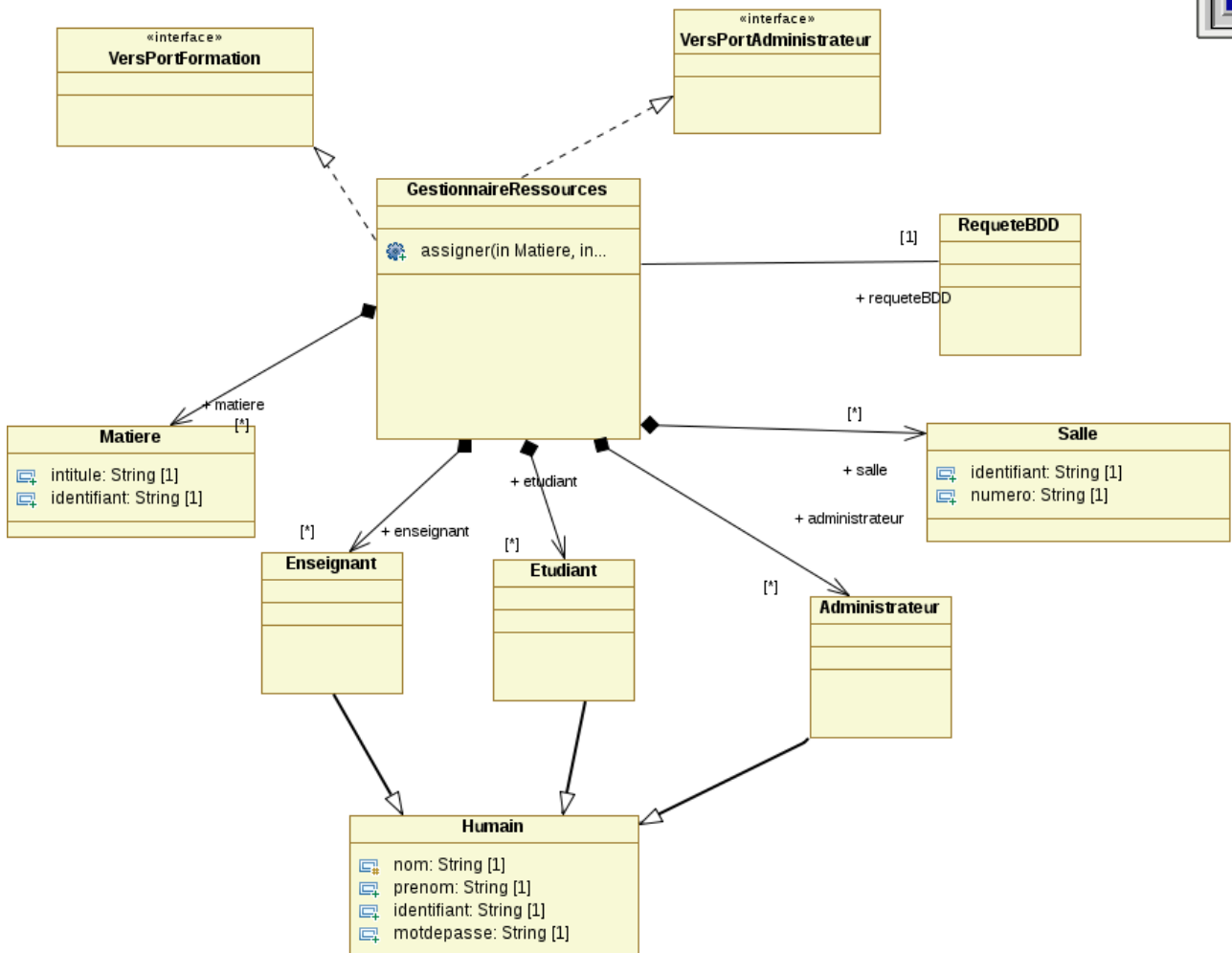
La première interface est la plus intéressante/utile/importante. La communication avec les ressources permet de procéder au stockage des objets dans la base de données. Sans l'utilisation de cette interface, il n'y aurait pas de sauvegarde des données.

La seconde et la dernière interface sont des connexions qui ne sont pas utiles pour ce sous-système. Elles vont utiliser ce sous-système afin de manipuler les formations (niveau assez haut -> proche de l'utilisateur).

La troisième interface va également comme précédemment utiliser ce sous-système mais à plus bas niveau.

3.3- Diagramme de classes – Gestionnaire ressources

Représentation graphique :



Commentaire :

Dans ce sous-système, nous souhaitons finalement montrer les objets qui représenteront les données du système EDT. Nous avons différentes données à stocker. Des matières, des salles et des humains (enseignants, étudiants et administrateur). La classe « Humain » ne va pas forcément rester par la suite. Elle permet de montrer que certaines ressources ont des attributs communs.

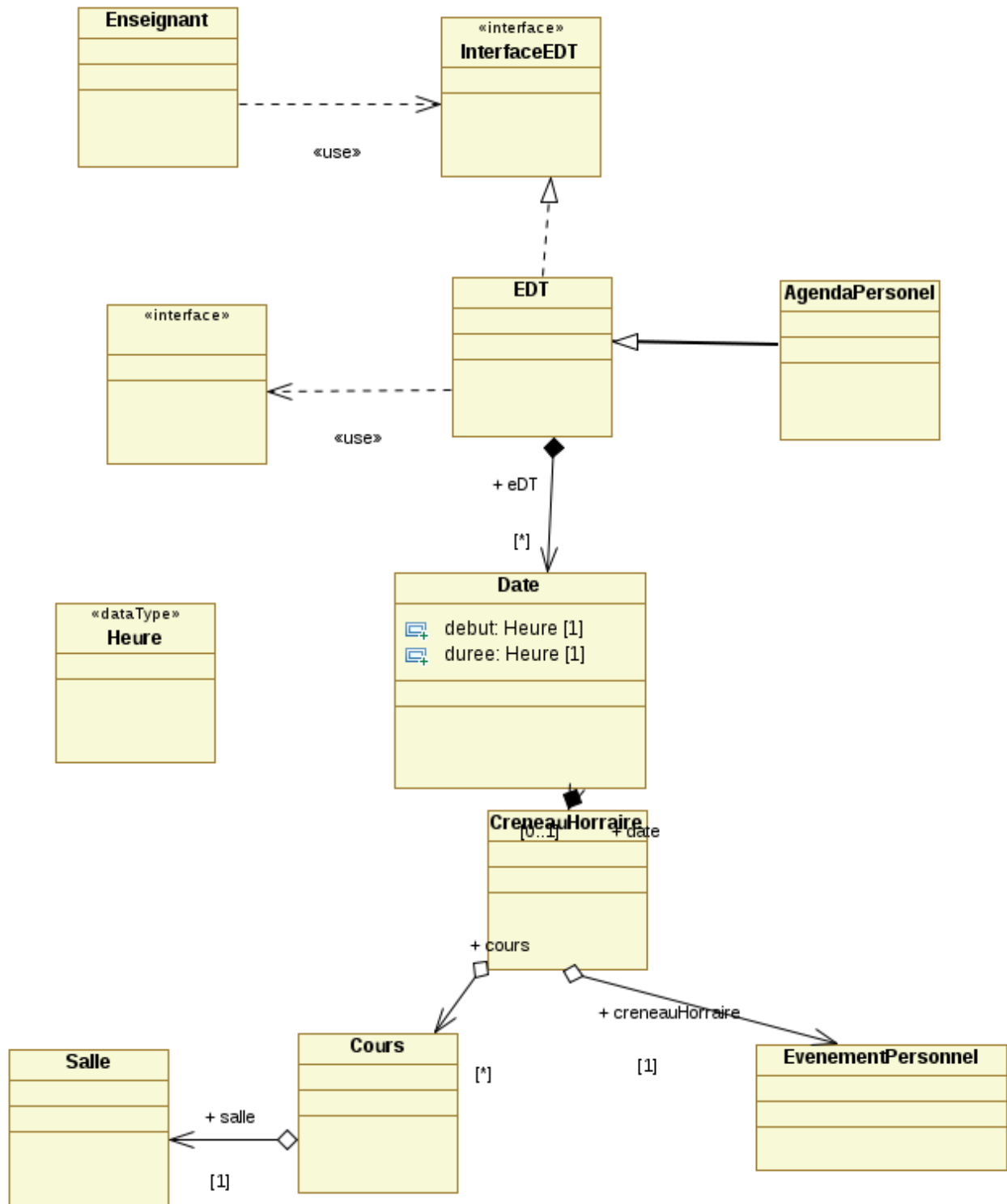
Nous avons également modéliser un lien vers une classe « RequeteBDD ». Cette classe va permettre à notre système de gérer toutes les requêtes d'écritures. Nous allons utiliser cette classe pour mettre à jour la base de données. Nous aurons préalablement vérifier que il n'y a pas de conflit via la classe « ErreurBDD » qui elle utilise un accès en lecture seulement.

Dans ce diagramme de classe, nous avons l'impression que les objets que nous avons modélisés ne sont pas vraiment utiles. En effet, lors de la conception, ils n'existeront à priori plus. Ils seront présent dans la base de données directement. Nous pourrions les réutiliser en les reconstruisant systématiquement via la base de données, mais cette technique est probablement plus lourde. Il nous faudra déterminer s'il vaut mieux transformer préalablement la base de données en plein d'objets ou s'il faut l'utiliser en faisant énormément de requête. En disant cela, nous sortons un peu de l'analyse mais en même temps, il est préférable de commencer à travailler sur le système le moins coûteux. Le système EDT pouvant être amené à manipuler un nombre conséquent de données.

Nous retrouvons des ports de communications. C'est port permettent de manipuler les ressources.

3.4- Diagramme de classes – Agenda

Représentation graphique :



Commentaire :

Cette partie montre la représentation objet des données au niveau de l'agenda. Le diagramme est incomplet (en raison de souci avec papyrus) mais il permet de voir la représentation en mémoire d'un événement. Un événement peut correspondre à un « EvenementPersonnel » qui est donc en rapport avec un enseignant. Cela peut également être un cours et donc cela correspond à la partie du responsable de formation. Ce n'est pas présent dans le diagramme mais il y aura d'autres types d'événements qui permettront par exemple de modéliser un jour férié, voir une période de vacances. Ce dernier point, nous amène à parler de l'optimisation. En effet, le diagramme actuel n'est pas une solution réellement applicable. Si nous appliquons directement cette logique, nous nous retrouverons avec une base de donnée surchargée car nous ne gérons pas les redondances d'informations (un même cours répété aura de multiples instances). Cependant, le fait de gérer la redondance est une manière de faire de l'optimisation, c'est pourquoi, étant encore dans une phase d'analyse, nous avons préféré ne pas encore nous intéresser à la manière dont seront stocké les données.

Un événement possède une date qui utilise un « Heure » permettant de modéliser l'heure de début de l'événement et la durée de celui-ci. L'emploi du temps possédera donc un ensemble de créneau horaire (ensemble d'événement finalement) qui sont délimité dans le temps.

Remarque :

L'interface qui n'a pas de nom correspond à celle qui sera utilisé par le responsable de formation.

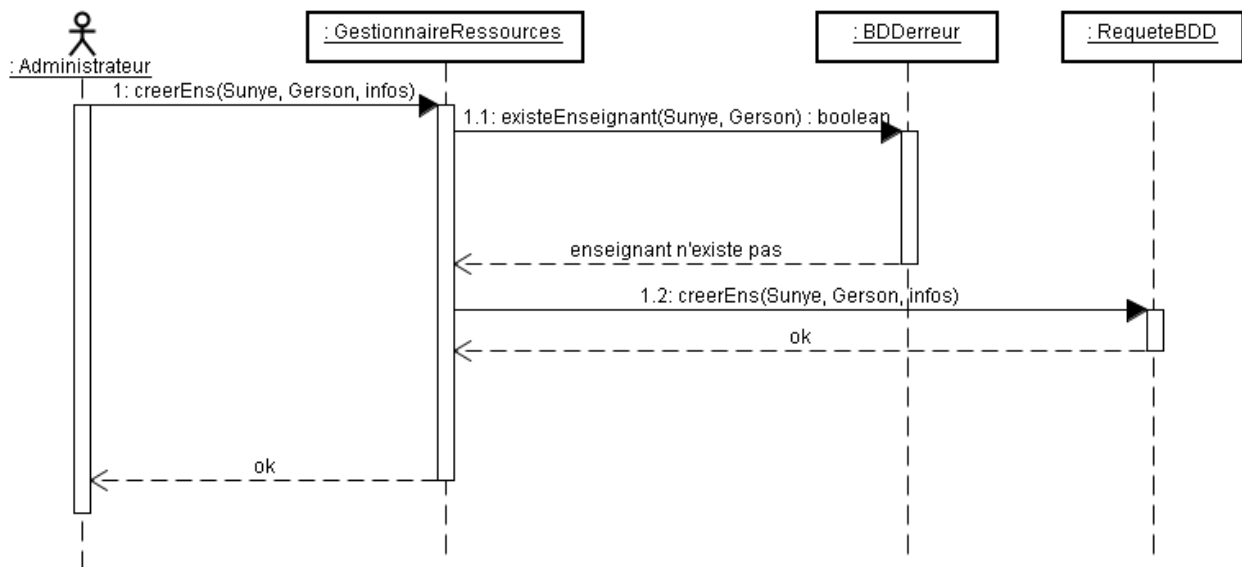
4- Diagrammes de séquence

4.1- Présentation

Les diagrammes de séquence que nous présentons ici permettent de mettre en évidence les principales méthodes des classes implémentant les composants. Ils montrent d'un point de vue structurelle et temporelle les différentes phases clés de notre modélisation. Ils sont donc essentielles dans la modélisation afin que par la suite, lors de l'implémentation, aucune solution miracle ne soit invoqué afin de résoudre un problème qui n'aurait pas été pensé en amont.

Nous tenons à préciser que pour réaliser ces diagrammes de séquence nous avons particulièrement fait attention à ne jamais « sortir » du composant et donc de respecter leurs indépendances. Nous avons donc, pour chaque cas, un diagrammes de séquences qui montre une évolution dans un composant. Si une communication doit s'effectuer entre deux composants, nous aurons alors deux diagrammes de séquence qui se complètent.

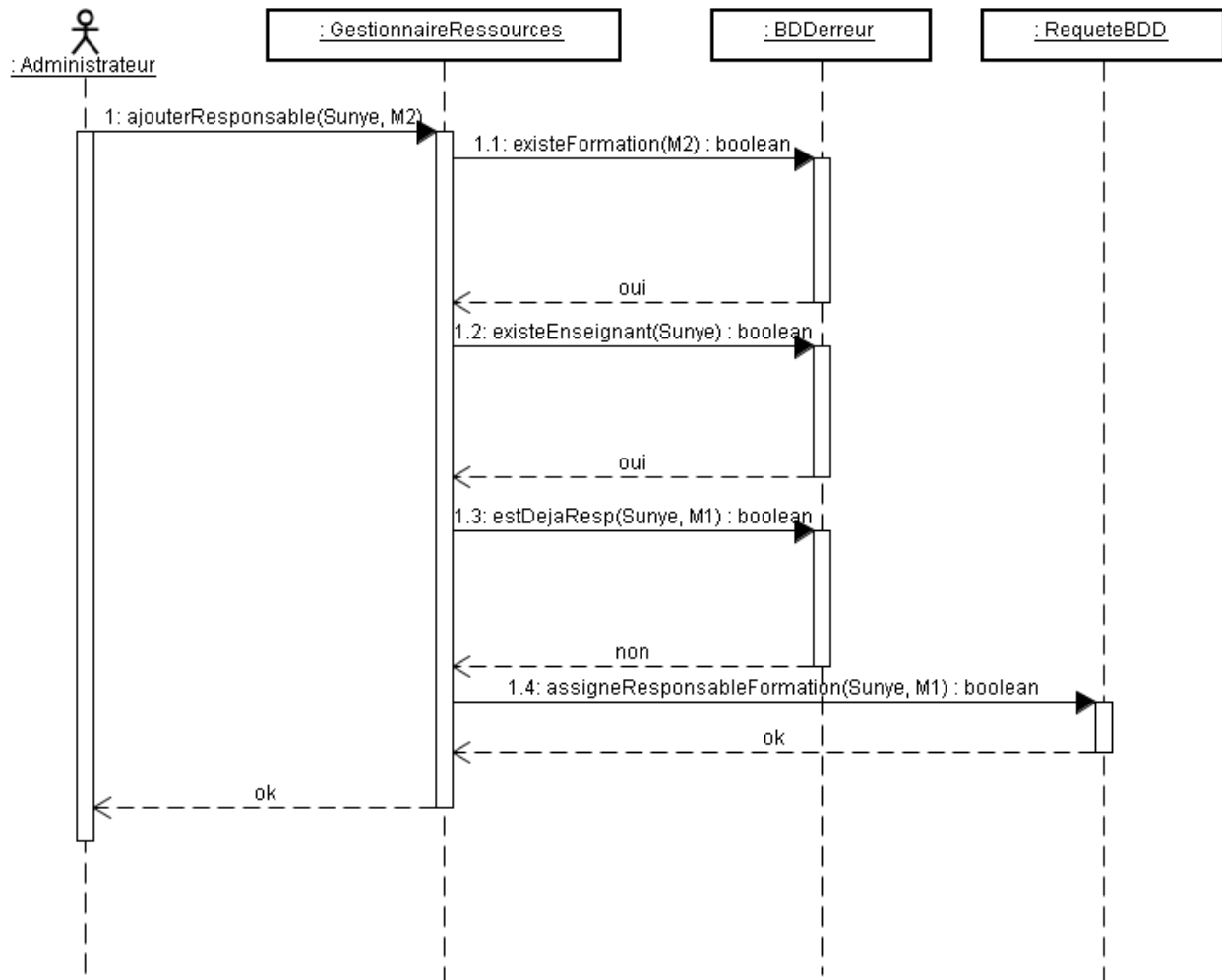
4.2- Diagramme de séquence – Créer enseignant



Commentaire :

Ce diagramme montre comment est créé une ressource, ici un enseignant. Avant de créer physiquement l'enseignant, nous vérifions s'il n'existe pas déjà. Comme nous l'explicitons dans le diagramme d'activité (cf 3.5), le contrôle pour gérer les erreurs de doublon s'effectue avant la mise à jour de la base de donnée. Nous avons donc un accès en lecture pour vérifier les problèmes puis un en écriture qui consiste simplement à mettre à jour la base de donnée.

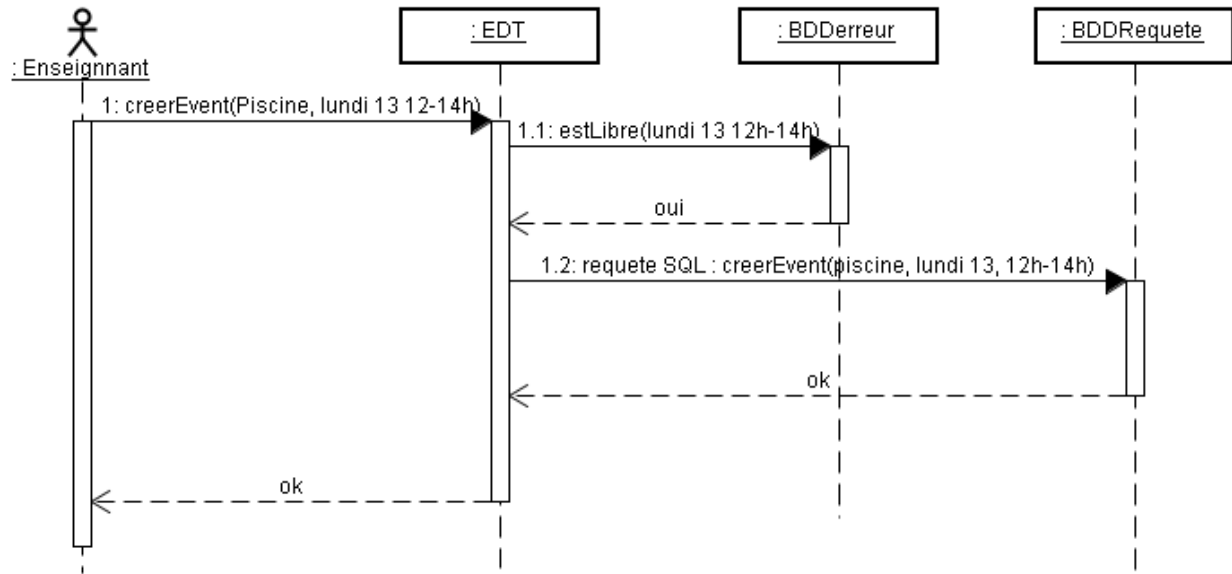
4.3- Diagramme de séquence – Ajouter responsable



Commentaire :

Ce diagramme est très simple à comprendre, il explique comment nous assignons un responsable de formation à une formation, cette opération se passe dans le composant *gestionRessources*. On vérifie tout d'abord que la formation existe bien, puis que l'enseignant est bien dans la base de données. Puis nous vérifions que cet étudiant n'est pas déjà responsable de cette formation, si tel est le cas nous pouvons faire une requête sur la base de données afin de mettre à jour celle-ci avec les nouvelles informations.

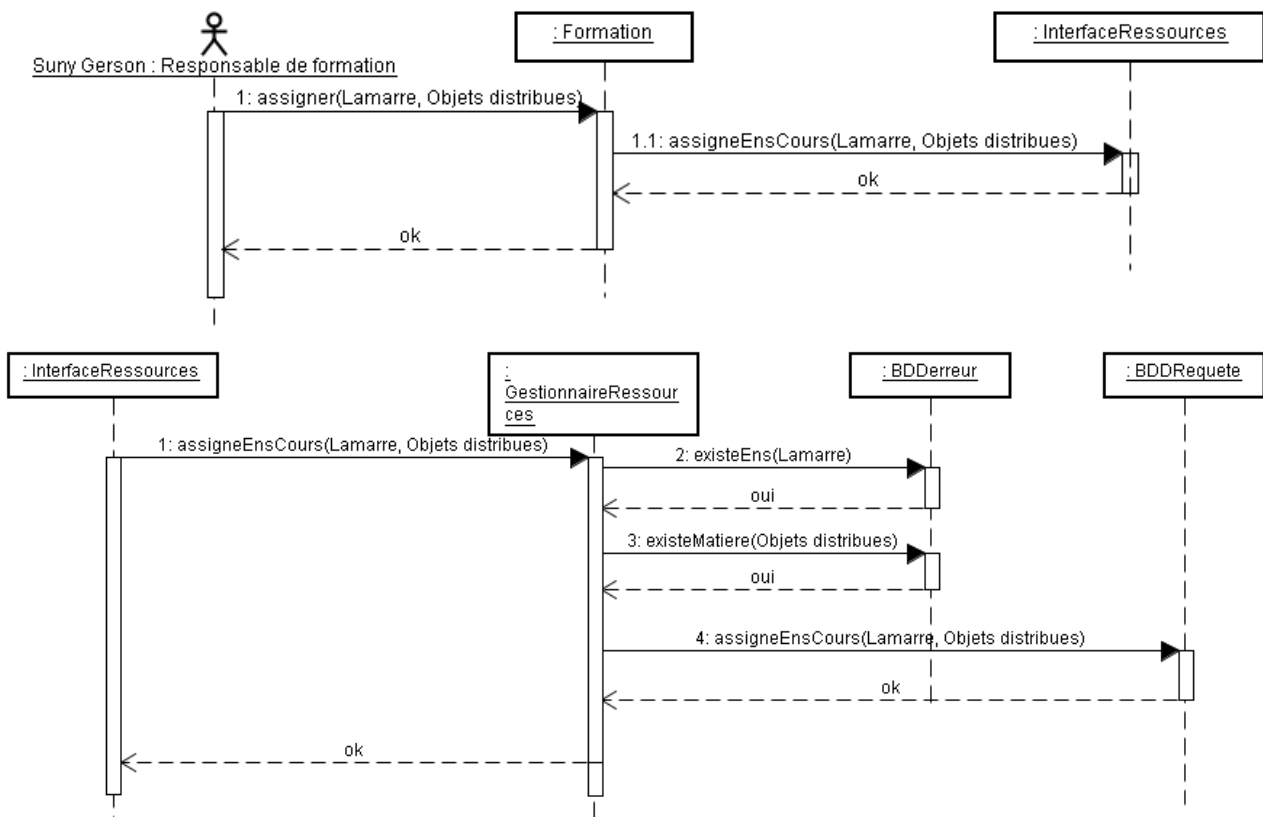
4.4- Diagramme de séquence – Créer un événement personnel



Commentaire :

Ce diagramme de séquence est en rapport avec les classes du composant EDT qui sera par la suite déployer sur la machine de l'enseignant et donc disposant de la base de données contenant les informations sur ces événements personnels. Lorsque l'enseignant souhaite créer un nouvel événement, on vérifie que le créneau auquel il veut l'affecter est libre et si tel est le cas, on met à jour la base de données personnelle (l'agenda personnel) de l'enseignant.

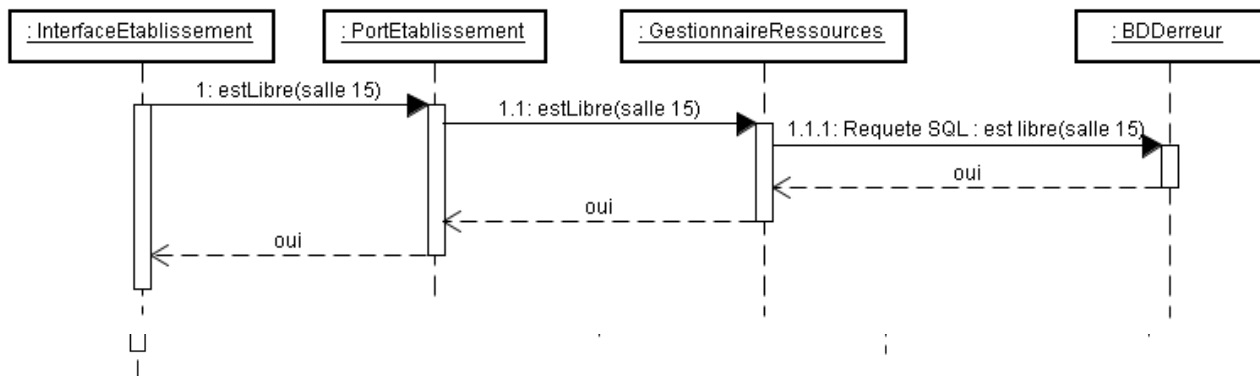
4.5- Diagramme de séquence – Assigner enseignant



Commentaire :

Ce scénario est un peu plus complexe que les précédents, il montre comment un responsable de formation assigne un enseignant à une matière. Le premier diagramme de séquence se déroule dans le composant *gestionFormation*, le deuxième se déroule dans le composant *gestionRessources*. Le responsable de formation demande au composant *gestionRessources* par l'intermédiaire de son interface d'assigner un enseignant à une matière. Tout le reste du traitement se passe ensuite dans le composant *gestionRessources*. On vérifie tout d'abord si l'enseignant existe bien dans la base de données. Si tel est le cas on vérifie que la matière existe elle aussi. Enfin on assigne l'enseignant à la matière en mettant à jour la base de données avec les nouvelles informations.

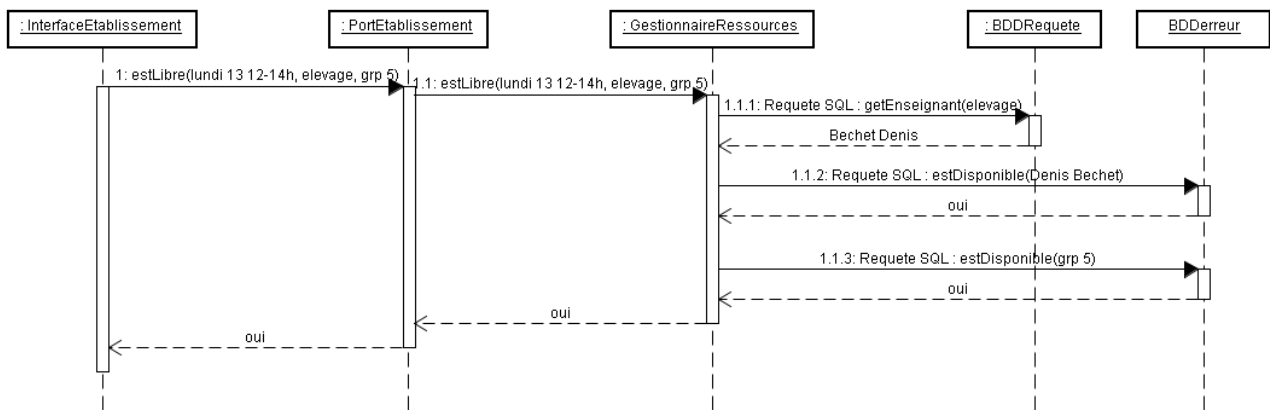
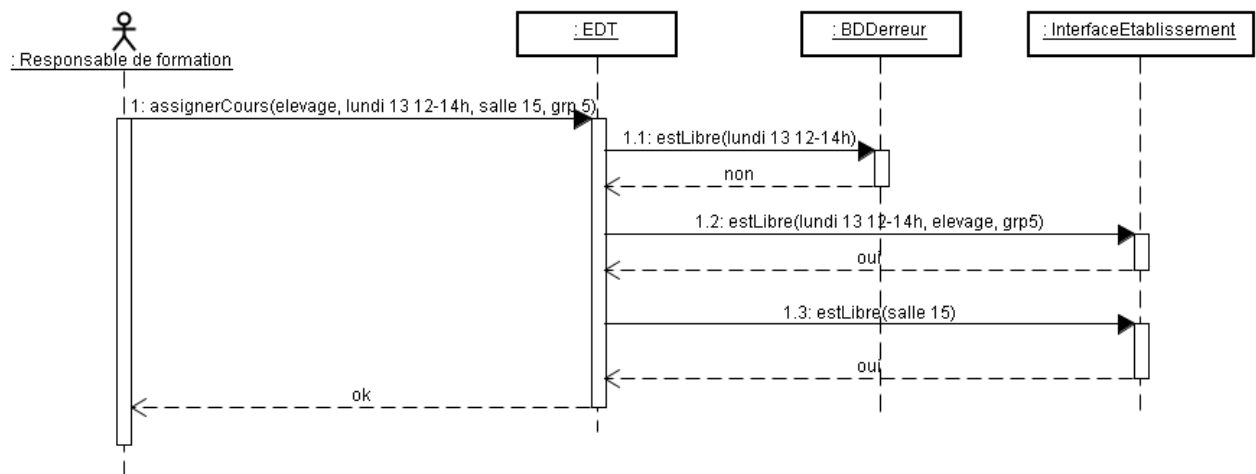
4.6- Diagramme de séquence – Assigner cours (version 1)



Commentaire :

Ce diagramme de séquence comporte des similitudes avec le diagrammes précédent. La première partie de ce diagramme se déroule dans le composant *gestionFormation*. Le responsable de formation souhaite assigner un cours à un créneau horaire. On commence par vérifier si le créneau est libre puis on demande au composant *gestionRessources* d'indiquer si la salle est bien libre par l'intermédiaire de l'interface *interfaceEtablissement*.

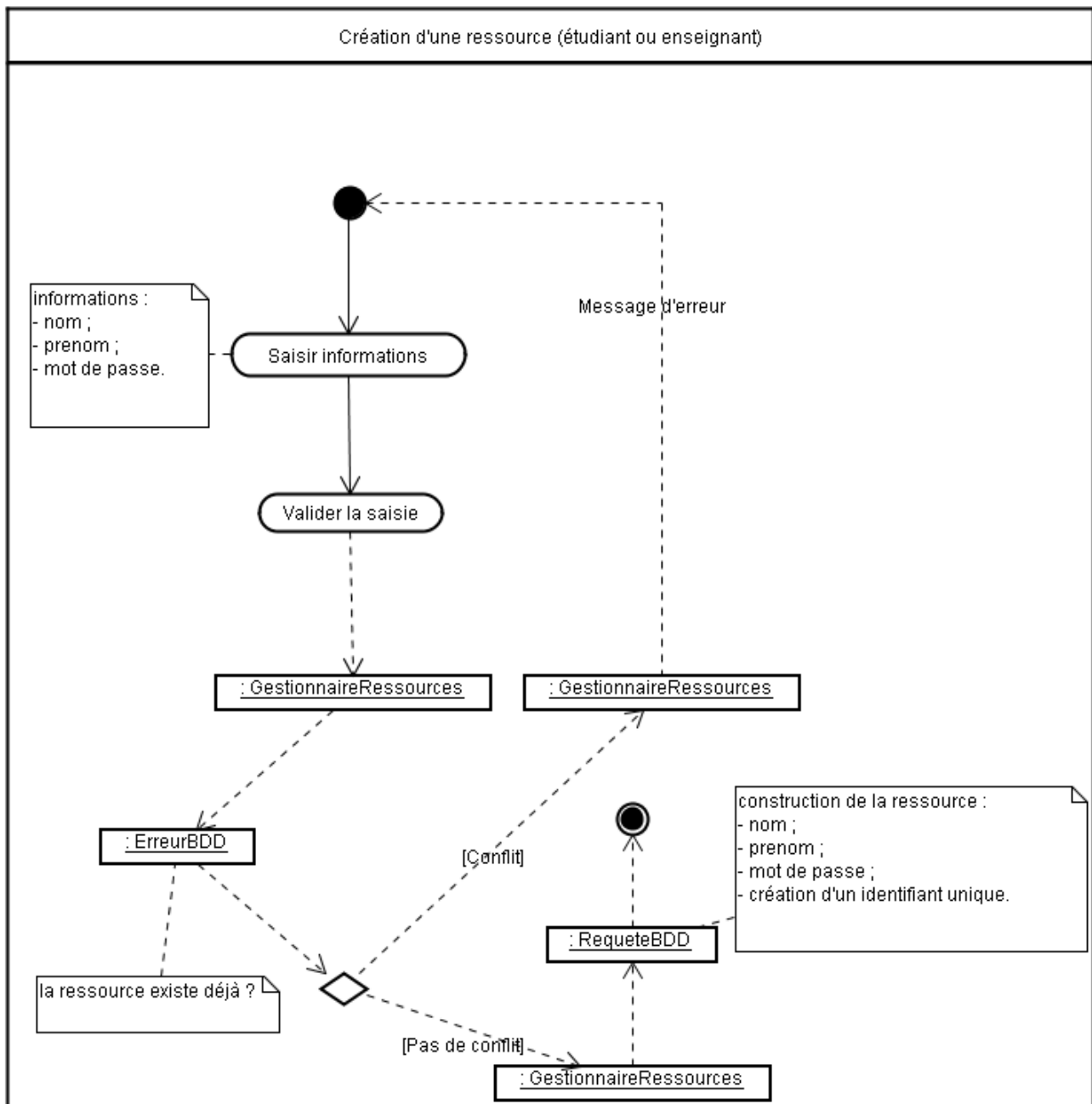
4.7- Diagramme de séquence – Assigner cours (version 2)



Commentaire :

Dans ce diagramme de séquence, le responsable de formation souhaite assigner un cours à un créneau horaire mais cette fois le créneau horaire n'est pas disponible, d'autres vérifications doivent alors avoir lieu. On commence donc par vérifier si ce créneau horaire est libre pour le groupe qui devra assister au cour. Cela entraîne deux vérifications qui ont lieu dans le composant *gestionRessources* : on vérifie que l'enseignant qui doit faire cours est bien disponible et que le groupe est bien disponible à ce moment là. La confirmation de ces disponibilités est renvoyée au composant *gestionFormation* qui vérifie ensuite si la salle est disponible (grâce au composant *gestionRessource*) comme précédemment.

5- Diagramme d'activité



Commentaire :

Ce diagramme d'activité mélange des noeuds d'actions et des noeuds objets. Cela permet d'avoir une vision temporelle du système au niveau des objets, et de leurs instances, et de garder à l'esprit le contexte de l'opération. Dans le cas traité, nous nous intéressons à la création de ressources et plus précisément à la création d'une ressource pour un étudiant ou un professeur. L'administrateur saisie les données de la nouvelle ressource qu'il va créer. Le système va ensuite voir s'il n'y pas un conflit avec un autre objet en regardant les ressources déjà présentes dans la base de données. S'il n'y a pas d'erreur, l'objet est construit, c'est-à-dire qu'il est ajouté à la base de donnée avec un identifiant unique. En cas d'erreur, l'administrateur est averti.

6- Conclusion

Dans cette partie nous avons défini l'application à un niveau d'abstraction plus bas que dans la partie *spécification des besoins* afin de nous rapprocher un peu plus de l'implémentation. Nous disposons maintenant des différentes composants de notre application. Dans la prochaine partie nous devrons voir l'application comme un système réparti et analyser les différentes technologies que nous utiliserons afin de faire communiquer les différents composants.

7- Problèmes rencontrés

Nous tenons à dire dans cette partie que nous avons rencontrés beaucoup de problèmes dans l'utilisation du plug-in eclipse *papyrus*. En effet, sur deux machines différentes (un mac et un pc) nous avons eu un bogue qui nous empêche de sauvegarder le projet papyrus. De plus certaines classes ne sont plus modifiables et le projet devient corrompu. Nous sommes alors obligés de repartir sur une précédente version du projet et nous tournons ainsi en rond. En conséquence, les diagrammes de classes présentés ici sont dans l'état d'avancement que nous avons pu atteindre avec *papyrus* mais ne sont pas en accord avec notre modélisation sur papier.

Gestion Emploi du temps

Livrable 3 – Architecture

Matthieu Aubry
Jeremie Guidoux
Quentin Marguet
Mathieu Vénisse-Garson

Projet Multimodule
01/11/08
Encadrant : Gerson Sunyé

2008-2009 – Projet Multimodule -

Matthieu AUBRY : mas1.aubry@gmail.com

Jérémie GUIDOUX : darkjer1@gmail.com

Quentin MARGUET : zorbitos@gmail.com

Mathieu VENISSE-GARSON : mathieu.venisse@gmail.com



UNIVERSITE DE NANTES

Remerciements

Bironneau Elie	Pour nous avoir aidé avec l'outil acceleo.
Thimel Arnaud	Pour son aide dans tous les domaines et son excellent TP sur les EJB 3.0.
Julien Courcelle Nicolas Payneau	Pour leur implication dans le projet de manière générale (ils sont le moteur du projet) et plus spécifiquement au niveau du serveur emploi du temps. Et également pour leur excellent travail de synthèse.

Sommaire

1 – Introduction.....	4
1.1 – Présentation.....	4
1.2 – Problématique.....	6
1.3 – Les parties de ce livrable.....	7
1.4 – Remarques.....	8
2 – L'architecture – généralité.....	9
2.1 – Présentation.....	9
2.2 – Les technologies.....	10
2.2.1 – Enseignant.....	10
2.2.2 – Responsable de formation.....	12
2.2.2 – Divers.....	13
3 – L'architecture – dans le détail.....	14
2.1 – Présentation.....	14
2.2 – Responsable de formation.....	15
2.2.1 – L'interface graphique.....	15
2.2.2 – La génération de code.....	15
2.2.3 – Les interfaces.....	16
2.3 – Enseignant.....	17
2.3.1 – Les stéréotypes et étiquettes.....	17
2.3.2 – La génération de code.....	17
2.3.3 – Résultat de la génération de code.....	24
2.3.4 – L'architecture du serveur.....	27
4 – Ce que nous avons fait.....	29
5 – Conclusion.....	29
 Annexe.....	 30
1 – Remarques préalables.....	30
2 – Emploi du temps - Base de donnée.....	31
2 – Emploi du temps - DTO.....	35
3 – Emploi du temps - EntityBean.....	36

Architecture

1 – Introduction

1.1 – Présentation

Dans le livrable précédant, nous avons défini nos composants. Nous devons dorénavant repartir de ses composants pour en extraire une architecture exploitable en vue de proposer un code utilisable. Il est à noter qu'à partir de ce livrable, nous allons concevoir les applications tout en tenant compte du fait que l'architecture est distribuée (comme définie dans le sujet de projet). Nous nous intéressons plus vraiment au projet dans son ensemble. Nous gardons à l'esprit cette vision globale du système mais en nous concentrant sur deux parties distinctes :

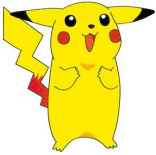
- la partie du responsable de formation ;
- la partie de l'enseignant (avec notamment le serveur).

Nous avons défini avec les autres groupes les interfaces permettant de communiquer avec le serveur d'emploi du temps via l'utilisation notamment des DTO. Ces interfaces vont nous permettre de développer, de concevoir notre architecture sans ce préoccuper du code métier présent dans le serveur d'emploi du temps.

Pour finir, nous avons décidé de modifier notre façon d'aborder le rapport. Nous souhaitons rendre cela plus attrayant à lire comme à concevoir. Nous nous sommes attribué des petits personnages qui interviendront dans le rapport. Cependant par souci d'anonymat, nous préférons garder leur identité secrète. Même si cela peut paraître un peu déplacé, nous estimons que cela n'entâche en rien notre volonté à vouloir fournir un travail de qualité.



Son avis est toujours longuement réfléchi. Il pose en général les fondements car son abstraction est sans limite.



Toujours prêt pour travailler, il ne rechigne pas à s'y mettre quelque soit l'heure.



Réputé pour être grognon, il est souvent là où on ne s'y attend pas.



D'humeur joyeuse, il saura égayer et éclairer certaines zones sombres.

1.2 – Problématique

Il nous est demandé de construire dans le détail l'architecture du système. A cela, s'ajoute la création de règles pour permettre la génération de code UML via acceleo, la représentation des bases de données et la description des stéréotypes et étiquettes. Nous présenterons également, les outils qui interviennent dans le système.

Cette partie a une importance cruciale. Nous devons dorénavant proposer une architecture définitive qui de ce fait devra être cohérente, correctement pensé, dont les outils seront fixés. L'analyse avait l'avantage de ne pas porter réellement à conséquence dans le sens ou elle fournit une base pour la conception mais ce n'est pas une étape qui impose de faire des choix réellement définitif. Cependant, la conception de l'architecture doit valider d'une certaine manière la cohérence de notre analyse. Ce livrable a donc un rôle particulièrement important car il doit selon nous :

- valider les livrables 1 et 2 donc l'analyse (indirectement) ;
- permettre la conception de l'interface graphique par la suite sans devoir revenir sur l'architecture.

1.3 – Les parties de ce livrable

Le sujet pour ce livrable est plus permissif que pour les autres. Nous avons plus de liberté quant au contenu. Nous avons donc choisi de présenter principalement tous les outils qui permettront la création de l'application. Ce livrable pourrait être décomposer en 3 parties :

- une présentation assez générale des sous-systèmes que nous allons étudier par la suite avec principalement la présentation des technologies qui seront utilisées ;
- une vision plus dans le détail de la partie enseignant et responsable de formation (bien que pour cette dernière, il n'y est pas beaucoup de contenu) avec notamment :
 - la description des interactions qu'il y aura avec le serveur emploi du temps ;
 - les fonctionnalités des interfaces graphiques ;
 - les stéréotypes et étiquettes ;
 - la génération de code avec acceleo ;
 - d'éventuels diagrammes de classe.

1.4 – Remarques

Les précédents livrables que nous avons fournis ont été modifiés (suite aux remarques). Nous avons subdivisé le livrable un et deux (qui étaient regroupés) en trois sous-livrables :

- livrable 0 qui présente le projet ;
- livrable 1 qui spécifie les besoins ;
- livrable 2 qui spécifie nos composants à haut niveau.

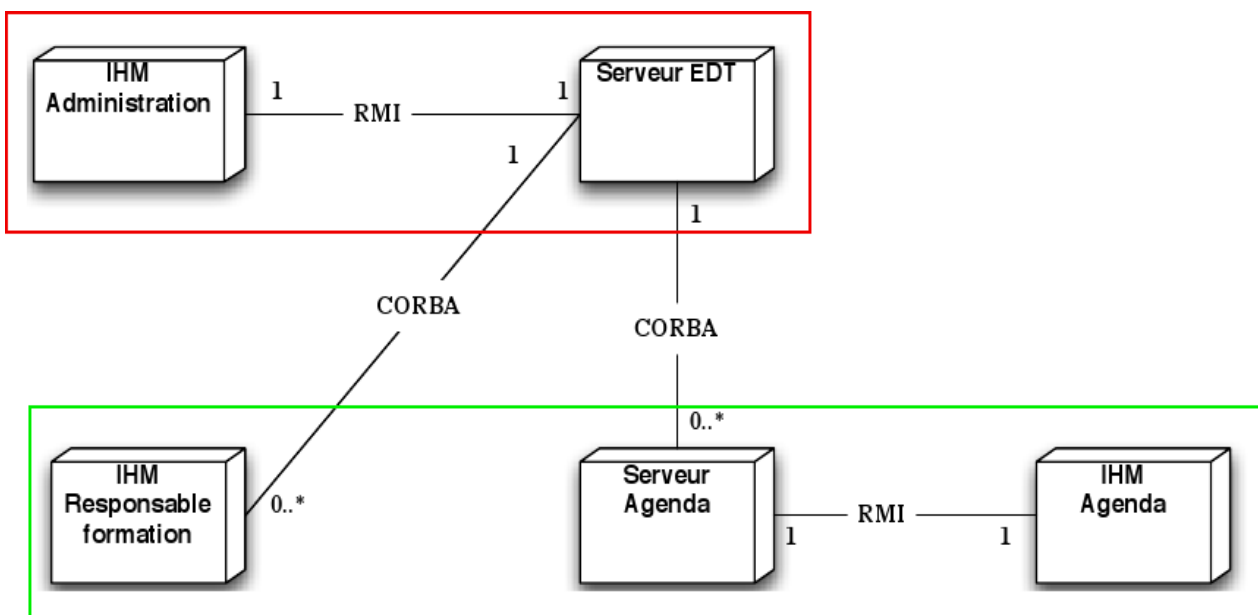
Nous avons rajouté un diagramme de classe dans le livrable 2 qui décrit notre composant « Agenda » (page 10 et 11). Nous aurions aimé modifier l'index du livrable un et deux pour qu'ils soient conçus de manière à coller au projet et non à une méthodologie UML (comme a fait le groupe Courcelle Payneau Maurinet Desbois). Cette modification nécessite cependant beaucoup de temps et nous ne l'avons donc pas faite pour nous concentrer sur la suite mais nous avons maintenant pleinement conscience de notre erreur. Pour ces mêmes raisons de temps, nous n'avons pas refait de séquence pour montrer comment nous gérons les jours fériés. Cependant, nous avons l'intention d'y revenir par la suite. Il est à noter que le livrable deux n'est plus vraiment exploitable, étant donné que les interfaces et l'architecture du serveur emploi du temps ont été faites avec les autres groupes.

Dans les annexes, nous avons mis les schémas qui sont en rapport avec le serveur d'emploi du temps. Il y a par exemple la base de donnée et le diagramme de classe qui présente les DTO.

2 – L'architecture – généralité

2.1 – Présentation

Pour commencer, nous allons nous mettre au niveau de l'application dans son ensemble. Voici le schéma du sujet modifié. Il présente l'architecture à haut niveau. Nous avons ajouté une information qui permet de savoir quelle partie va nous intéresser plus particulièrement.



La partie que nous allons approfondir

La partie dont les interfaces ont été définies
avec les autres groupes

Comme nous pouvons le constater, nous avons une grosse partie basée sur de la conception d'interface graphique. La partie qui s'apparente plus à de l'architecture objet se trouve au niveau du « Serveur Agenda ». La partie sur le responsable de formation est moins complexe au niveau architecture. Nous n'avons en effet qu'à utiliser les interfaces fournies par le « Serveur EDT » pour concevoir notre code métier et ensuite développer l'interface graphique.

2.2 – Les technologies

Il est temps maintenant de parler des technologies que nous avons choisi pour développer les deux applications « Enseignant et Responsable de formation ». Il y a aussi les technologies plus générales qui vont permettre par exemple de générer le code.

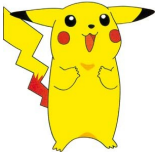
2.2.1 – Enseignant

La partie enseignant correspond aux composants « Serveur Agenda » et « IHM Agenda » du schéma. Cette partie utilise *Java RMI*, en conséquence, il apparaît plutôt judicieux d'utiliser le Java. Nous aurions pu utiliser une autre technologie, mais cela aurait compliqué inutilement l'application. Pour l'interface graphique, nous avons décidé d'utiliser Swing et éventuellement une surcouche d'*AWT* mais seulement si cela s'avérait nécessaire.



Pour la conception de l'interface graphique, nous utilisons l'ide eclipse et les outils présents sous netbeans pour créer rapidement le canevas de l'IHM (il n'y a plus d'outil graphique digne de ce nom sous eclipse).

Le « Serveur Agenda » est relié au « Serveur EDT » par le biais de la technologie CORBA. CORBA existe sous Java, nous n'avons donc pas non plus intérêt à utiliser une autre technologie. Le « Serveur Agenda » va contenir une base de donnée qui sera persistante. Nous pensions nous diriger vers la technologie *hybernate* mais ils nous a été recommandé d'utiliser les *EJB 3.0*.



Allez, nous sommes sympa, voici un petit schéma pour résumer tout cela.

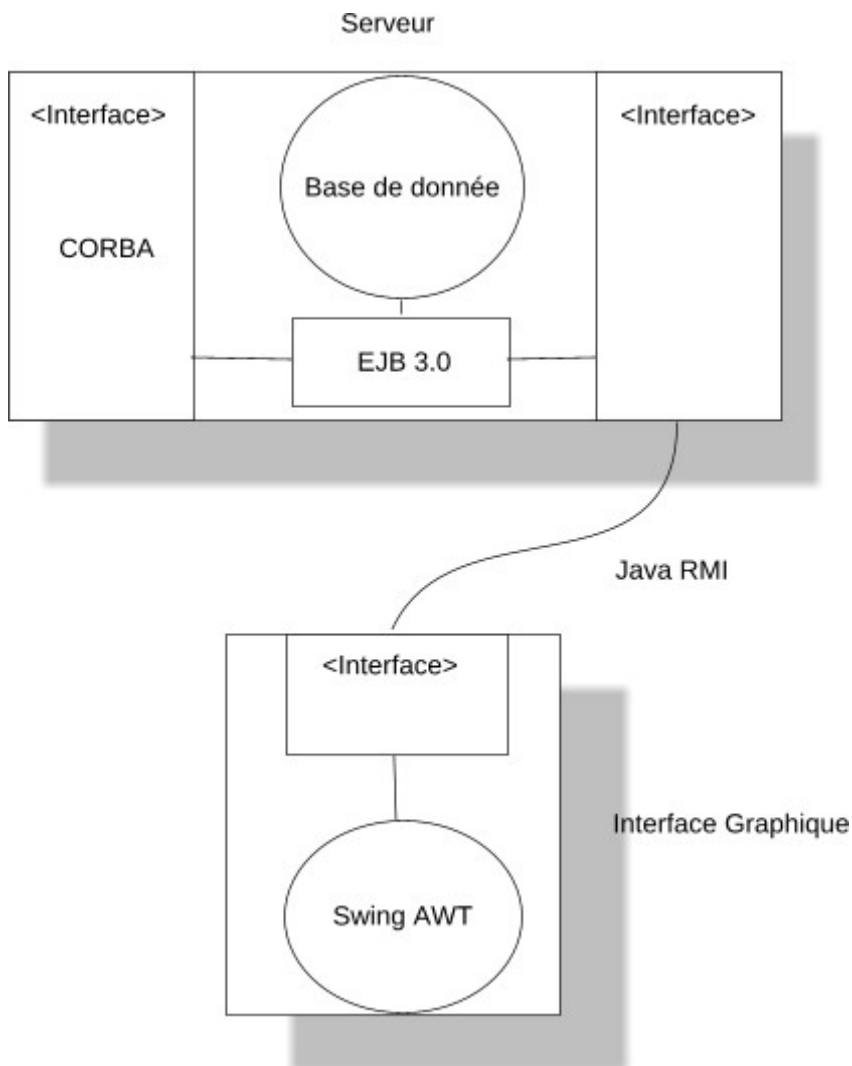


Schéma générale présentant les technologies de l'architecture au niveau de l'enseignant

Ce schéma met en évidence le fait qu'au niveau local (de l'interface graphique), nous utilisons les EJB pour accéder et manipuler la base de donnée qui est distante. Le code métier reste cependant du côté de l'interface graphique.

2.2.2 – Responsable de formation

Pour cette partie, nous n'avons pas le droit d'utiliser le Java. Nous avons donc regardé ce qui se faisait ailleurs. Nous avons cependant la contrainte que le langage choisi soit compatible avec CORBA. Nous avons étudié différentes possibilités.

Nous avons par exemple pensé à utiliser les technologies WEB (*php, ajax, xhtml, css...*) qui ont l'avantage de permettre un accès distant à l'aide d'un simple navigateur WEB. Pour l'interaction avec CORBA, nous aurions utilisé *Corbascript*. Cependant, les technologies WEB sont trop lourdes à développer. Les navigateurs respectent plus ou moins les standards ce qui oblige à faire des variations pour adapter le programme à chaque logiciel. Nous avons alors pensé à utiliser *GWT* mais cela reste du *Java*.

Nous avons ensuite pensé à utiliser le *C#* qui de par sa syntaxe proche du Java et ces outils de développement est très agréable mais nous trouvons que finalement, il est trop proche du Java.

Nous avons alors opté pour du *C++* avec du *Qt* pour l'interface graphique. Au niveau des avantages, le *C++* est un langage qui est plutôt rapide. Cependant, il reste difficile de développer en *C++* du fait qu'il n'y ai pas par exemple de *garbage collector*. *Qt* est un choix qui nous a semblé presque évident. Il y a une grosse communauté de développeur. Il est de plus en plus utilisé. Il possède une documentation très riche et ses possibilités sont impressionnantes. Il pêche probablement un peu par sa complexité (et sa lourdeur ?).



J'aurais préféré faire du python , on aurait pas tous ces problèmes avec les pointeurs.

Pour ce qui est du développement, nous utilisons plusieurs outils. QtDesigner nous aide à faire les canevas de l'interface graphique. Geany est un éditeur minimaliste qui nous permet juste d'avoir la coloration syntaxique. Nous utilisons également QtCreator qui n'est pas encore tout à fait au point mais qui a l'autocomplétion. Pour finir, l'assistant de Qt nous permet d'accéder à la documentation.

2.2.2 – Divers

Il y a des technologies qui sont utilisées par nos deux sous-systèmes. Nous pensons à la génération de code. Nous utilisons acceleo. Nous avons conçu un générateur pour le code Java qui bien que basique répond à nos besoins. Nous avons également créer un générateur de code pour les EJB. Il nous permet de générer directement l'ensemble des classes qui permettent de manipuler la base de donnée. Nous rentrerons dans le détail de la génération de code plus tard.



Nous avons décidé de troquer papyrus contre topcased. Les premiers diagrammes nous ont posés de sérieux problèmes (notamment pour les stéréotypes). Mais au moins, nous pouvions sauver à tout moment...

Pour ce qui est de la documentation, nous utilisons la Javadoc pour le Java et Dioxygen pour le C++.

3 – L'architecture – dans le détail

2.1 – Présentation

Il est temps de plonger dans le vif du sujet en regardant de près l'architecture que nous avons conçue. Comme nous avons identifié deux systèmes qui sont finalement plus ou moins indépendant, nous avons décidé de travailler en parallèle. Nous nous réunissons régulièrement tous ensemble afin de profiter des nouvelles aptitudes que nous acquérons au travers de la réalisation de ce projet.

Cette partie va finalement ressembler fortement à la précédente au niveau de la hiérarchie des chapitres. C'est-à-dire que nous allons d'abord observer chaque sous-système (responsable de formation et enseignant) séparément pour ensuite nous intéresser aux parties qui leurs sont communes.



Il faut bien avouer que la partie sur le responsable de formation est un peu vide...

2.2 – Responsable de formation

2.2.1 – *L'interface graphique*

Il est probablement encore un peu tôt pour s'intéresser aux fonctionnalités fournies par l'interface graphique cependant, nous avons déjà commencer à travailler dessus. Nous avons donc identifiés certaines fonctionnalités qui devraient se retrouver dans notre application finale :

- visionner les ressources (salles, enseignants, groupes...) ;
- visionner les événements (cours et événements personnels) ;
- créer/modifier/supprimer des cours complets ;
- gérer les conflits avec les événements personnels ;
- créer/modifier/supprimer des groupes ;

Nos premiers canevas d'interface graphique se sont inspiré de ce que fait « google ». Cependant, cette représentation très graphique, nous a semblé finalement peu appropriée pour gérer beaucoup d'événements simultanément. Le responsable de formation peut avoir envie de voir plusieurs formations en même temps. Il est plus que probable qu'il est plusieurs cours sur les mêmes créneaux, une représentation graphique ne permettrait alors pas de voir clairement les événements.

2.2.2 – *La génération de code*

Afin de gagner en temps de développement, nous avons conçu sous acceleo un générateur de code qui nous permet de transformer les interfaces fournies par le serveur emploi du temps en code respectant la syntaxe des IDL. Nous avons fait de même pour les DTO (une version Java et C++). La version Java étant utilisée pour l'enseignant. Nous avons décider ne pas expliciter le code acceleo correspondant. Nous trouvions plus intéressant de montrer le code qui permet la génération des « bean entity ».

2.2.3 – Les interfaces

Les interfaces fournies par le serveur emploi du temps ont des méthodes qui nous apparaissent comme suffisante pour construire l'interface graphique. Notre interface graphique manipulera donc des objets DTO.

2.3 – Enseignant

2.3.1 – Les stéréotypes et étiquettes

Afin de générer du code spécialisé pour les EJB 3.0. Il a été indispensable de créer des stéréotypes. Nous n'avons pas utilisé les étiquettes. Nous avons identifiés différents stéréotypes :

- le « BeanEntity » qui nous permet de savoir que la classe est une entity bean ;
- « PrimaryKey » et « ForeignKey » permettent de savoir si une variable est une clé primaire et/ou étrangère.

2.3.2 – La génération de code

Afin de gagner du temps et surtout pour apprendre à utiliser acceleo, nous avons conçu le code permettant de générer nos classes Java. Comme la partie qui nous intéresse principalement dans la génération se situe au niveau du serveur de l'enseignant. Nous nous sommes spécialisés dans la génération de code permettant de créer les classes pour les EJB 3.0. Nous allons présenter le code permettant la génération.



Nous avons eu des difficultés au niveau de l'indentation. Nous trouvions difficile d'avoir un code clair tout en tenant compte des espaces, tabulations et saut de ligne qui seront produit dans le code généré. Les deux étant intimement liés (le code à produire et le code produit). Il est donc nécessaire de reformater le code généré via eclipse (deux raccourcis). Nous avons appris par la suite que « Code Lutin » utilise la même technique pour les mêmes raisons.

L'entête du fichier acceleo

```
1 <%  
2 metamodel http://www.eclipse.org/uml2/2.0.0/UML  
3 import JavaTools  
4 %>
```

Nous importons une classe Java « JavaTools » que nous avons ajoutée (le code vient du forum d'acceleo) qui va nous permettre de disposer de deux méthodes « getStereotype » et « hasStereotype » qui vont nous servir par la suite.

La création d'une classe

```

1  <!-- Classe -->
2  <%script type="uml.Class" name="toClass" file="<
3  %parent().name.replaceAll("\\.", "/")%>/<%name%>.java" post="trim()" %>
4
5  package <%parent().name%>;
6
7  <%if (getStereotype("BeanEntity") != "") {%>
8      import org.hibernate.annotations.GenericGenerator;
9      import java.io.Serializable;
10     @Entity
11     <%if (isAbstract) {%>
12         abstract
13     <%}%>
14     <%visibility%> class <%name%> implements Serializable
15 {<%}else{%>
16     <%if (isAbstract) {%>
17         abstract
18     <%}%>
19     <%visibility%> class <%name%><%toGeneralisation%> {
20 <%}%>
21
22
23 <%self().push()%>
24 <%if ( (getAssociations() != null) && (getAllAttributes() != null) ) {%>
25     /***** VARIABLES *****/
26 <%}%>
27 <%for (getAssociations()) {%>
28     <%current().toAttributeAssociation()%>
29 <%}%>
30 <%for (getAllAttributes()) {%>
31     <%current().toAttribute.sep("\n")%>
32 <%}%>
33
34
35 /***** CONSTRUCTEUR *****/
36 <%createConstructor.sep("\n")%>
37
38
39 <%if ( (getAssociations() != null) && (getAllAttributes() != null) ) {%>
40     /***** SETTERS *****/
41 <%}%>
42 <%for (getAssociations()) {%>
43     <%current().toSettersAssociation()%>
44 <%}%>
45 <%for (getAllAttributes()) {%>
46     <%current().createSetters.sep("\n")%>
47 <%}%>
48
49
50 <%if ( (getAssociations() != null) && (getAllAttributes() != null) ) {%>
51     /***** GETTERS *****/
52 <%}%>
53 <%for (getAssociations()) {%>
54     <%current().toGettersAssociation()%>
55 <%}%>
56 <%for (getAllAttributes()) {%>
57     <%current().createGetters.sep("\n")%>
58 <%}%>
59

```

```

60
61 <%if ( (getAllOperations() != null) ) {%>
62 /***** METHODES *****/
63 <%}%>
64 <%for (getAllOperations()) {%>
65 <%current().toMethodes.sep("\n") %>
66 <%}%>
67 <%pop() %>
68 }

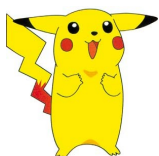
```

Généralité :

La création d'une classe consiste principalement à appeler des scripts auxiliaires qui vont permettre la génération des « getters », « setters »... A cela s'ajoute, l'ajout de propriété auxiliaire si nécessaire (héritage...).

Les spécificités :

- si la classe UML possède le stéréotype « BeanEntity », nous générons les imports et l'entête de la classe pour qu'il correspondent à une classe « Entity » comme défini dans les EJB 3.0 ;
- le « self().push() » permet de sauvegarder la classe qui est actuellement en cours de génération (cela nous sert par la suite pour les associations).
- les associations sont présentes au niveau des packages et non au niveau des attributs sous « topCased », ce qui implique de générer les attributs et méthodes au niveau de la classe.



Pour les interfaces, il n'y a pas de différence fondamentale avec la génération d'une classe (c'est à peu près la même chose en version allégée). Le code complet se trouve dans le répertoire code.

La création d'un attribut

```

1 <!-- Attribut association -->
2 <%script type="uml.Association" name="toAttributeAssociation"
3 post="trim()" %>
4 <%for (getAllAttributes()) {%>
5 <%if (peek().name != type.name) {%>
6 <%toParameters%>
7 <%}%>
8 <%}%>
9
10 <!--Attribut de la classe -->
11 <%script type="uml.Property" name="toAttribute" post="trim()" %>
12 <%if ((upper==1) || (upper>1)) {%>
13 <%createListAttribute%>
14 <%}%>
15 <%createAttribute%>
16 <%}%>
17
18 <!--Attribut basique -->
19 <%script type="uml.Property" name="createAttribute" post="trim()" %>
20 <%visibility%> <%isStaticToString()%> <%type.toTypeJava%> <%name%>;
21
22 <!-- Attribut de type liste -->
23 <%script type="uml.Property" name="createListAttribute" post="trim()" %>
24 <%visibility%> <%isStaticToString()%> List<<%type.toTypeJava%>> liste<
25 <%type.toTypeJava%>;

```

Généralité :

Nous avons identifiés deux types d'attributs bien distincts :

- les attributs de classe (obtenus directement) ;
 - les attributs d'association (obtenus en regardant les liens entre classes).
- Comme pour les classes, nous gérons les propriétés auxiliaires (visibilité, static...).

Les spécificités :

Pour les associations, nous parcourons les classes qui sont liés au niveau du package. Si la classe actuellement générée fait partie de l'association, nous créons un attribut en regardant la classe qui est liée.
A noter que « upper » permet de récupérer la cardinalité.

La création d'une méthode

```

1  <!-- Methode -->
2  <%script type="uml.Operation" name="toMethodes" post="trim()" %>
3  <%visibility%> <%type.toTypeJava%> <%name%> (<
4  %ownedParameter.toParameters.sep(",") %>) {}
5
6  <!--Parametres d'une methode-->
7  <%script type="uml.Parameter" name="toParameters" post="trim()" %>
8  <%if (direction=="in") {%>
9      <%type.toTypeJava%> <%name%>
10 <%}%>
11
12 <!--Constructeur de classe-->
13 <%script type="uml.Class" name="createConstructor" post="trim()" %>
14 public <%name.toUmlCase() %> (<%ownedAttribute.toParameters.sep(",") %>) {
15 <%ownedAttribute.toAffectation.sep("\n") %>
16 }
17
18 <!-- Setters des attributs -->
19 <%script type="uml.Property" name="createSetters" post="trim()" %>
20 <%isStaticToString() %>
21 <%if ((upper==1) || (upper>1)) {%>
22     public void setListe<%type.toTypeJava%> (List<%type.toTypeJava%>>
23 liste<%type.toTypeJava%>){
24     <%toAffectation%>
25 }
26 <%}else{%>
27     public void set<%name.charAt(0).toUmlCase() %><
28 %name.substring(1,name.length()) %> (<%type.toTypeJava%> <%name.toLlCase() %>)
29 {
30     <%toAffectation%>
31 }
32 <%}%>
33
34 <!-- Gestion des Setters des associations -->
35 <%script type="uml.Association" name="toSettersAssociation" post="trim()" %>
36 <%for (getAllAttributes()) {%>
37     <%if (peek().name != type.name) {%>
38         <%createSetters%>
39     <%}%>
40 <%}%>
41
42 <!-- Getters des attributs -->
43 <%script type="uml.Property" name="createGetters" post="trim()" %>
44 <%isStaticToString() %>
45 <%if ((upper==1) || (upper>1)) {%>

```

```

46      <%for (parent.getAppliedStereotypes()) {%>
47          <%toPropertyStereotype() %>
48      <%}%>
49      public List<<%type.toTypeJava%>> getList<%type.toTypeJava%>() {
50          return this.liste<%type.toTypeJava%>;
51      }
52  <%}else{%>
53      <%for (getAppliedStereotypes()) {%>
54          <%toPropertyStereotype() %>
55      <%}%>
56      <%if ((getAppliedStereotypes().nSize()==0) &&
57  parent.hasStereotype("BeanEntity")) {%>
58          @Basic
59      <%}%>
60      public <%type.toTypeJava%> get<%name.charAt(0).toUpperCase()%><
61  %name.substring(1,name.length()) %>() {
62          return this.<%name%>;
63      }
64  <%}%>
65
66  <!-- Gestion des Getters stéréotypés --%>
67  <%script type="uml.Stereotype" name="toPropertyStereotype" post="trim()" %>
68  <%if (name == "PrimaryKey") {%>
69      @id
70      @GeneratedValue(generator="system-uuid")
71      @GenericGenerator(name="system-uuid", strategy="uuid")
72  <%}else{%><%if (name == "BeanEntity") {%>
73      @OneToMany(mappedBy="<%parent.toLowerCase()%>", fetch=FetchType.EAGER)
74  <%}%><%}%>
75
76  <!-- Gestion des Getters des associations --%>
77  <%script type="uml.Association" name="toGettersAssociation" post="trim()" %>
78  <%if (getAllAttributes().nSize()==2) {%>
79      <%if (getAllAttributes().nGet(0).type.name!=peek.name) {%>
80          <%if ((getAllAttributes().nGet(0).upper==1) ||
81  (getAllAttributes().nGet(0).upper>1)) {%>
82              @OneToMany(mappedBy="<
83  %getAllAttributes().nGet(0).type.name%>", fetch=FetchType.EAGER)
84          <%}else{%>
85              @ManyToOne()
86          <%}%>
87          <%getAllAttributes().nGet(0).createGetters%>
88      <%}%>
89      <%if (getAllAttributes().nGet(1).type.name!=peek.name) {%>
90          <%if ((getAllAttributes().nGet(1).upper==1) ||
91  (getAllAttributes().nGet(1).upper>1)) {%>
92              @OneToMany(mappedBy="<
93  %getAllAttributes().nGet(0).type.name%>", fetch=FetchType.EAGER)
94          <%}else{%>
95              @ManyToOne()
96          <%}%>
97          <%getAllAttributes().nGet(1).createGetters%>
98      <%}%>
99  <%}%>

```

Généralité :

Pour les méthodes, nous avons identifiés et gérées les « getters », « setters », les constructeurs tout cela en tenant compte des paramètres d'entrées/sorties et des éventuels stéréotypes qui nous permettent de gérer les EJB 3.0.

Les spécificités :

La difficulté a été de gérer les EJB 3.0 au niveau des associations. Il faut par exemple tenir compte de la cardinalité des liens pour en déduire les annotations à appliquer aux « getters ». Le stéréotype « PrimaryKey » présents au niveau des attributs nous permet de gérer les clés primaires.

Les outils annexes

```
1 <!--Creation d'un heritage-->
2 <%script type="uml.Class" name="toGeneralisation" post="trim()" %>
3 <%if (general.name.equalsIgnoreCase("")) { %>
4     extends <%generalization.general.name%>
5 <%}%>
6
7 <!--Transforme le type du modele en type Java-->
8 <%script type="uml.Type" name="toTypeJava" post="trim()" %>
9 <%if (name=="Integer") { %>
10     int
11 <%}else{ %>
12     <%if (name=="Boolean") { %>
13         boolean
14     <%}else{ %>
15         <%name%>
16     <%}%>
17 <%}%>
18
19 <!--Ecrit "static" si l'attribut du modele est static-->
20 <%script type="uml.Property" name="isStaticToString" %>
21 <%if (isStatic()) { %>
22     static
23 <%}%>
24
25 <!--Choisi la bonne syntaxe et les bons types pour les parametres (entrees
26 et sortie)-->
27 <%script type="uml.Property" name="toParameters" post="trim()" %>
28 <%if (!isStatic()) { %>
29     <%if ((aggregation!="none") || (association!="")) { %>
30         <%if ((upper==1) || (upper>1)) { %>
31             List<<%type.toTypeJava%>> liste<%type.toTypeJava%>
32         <%}else{ %>
33             <%type.toTypeJava%> <%type.toTypeJava.toLlCase() %>
34         <%}%>
35     <%}else{ %>
36         <%type.toTypeJava%> <%name%>
37     <%}%>
38 <%}%>
39
40 <!--Realise la bonne syntaxe et les bons types pour les affectations-->
41 <%script type="uml.Property" name="toAffectation" post="trim()" %>
42 <%if (!isStatic()) { %>
43     <%if ((aggregation!="none") || (association!="")) { %>
44         <%if ((upper==1) || (upper>1)) { %>
45             this.liste<%type.toTypeJava%> = liste<%type.toTypeJava%>;
46         <%}else{ %>
47             this.<%type.toTypeJava.toLlCase() %> = <
48 %type.toTypeJava.toLlCase() %>;
49         <%}%>
50     <%}else{ %>
```

51	this.<%name%> = <%name%>;
52	<%}%>
53	<%}%>

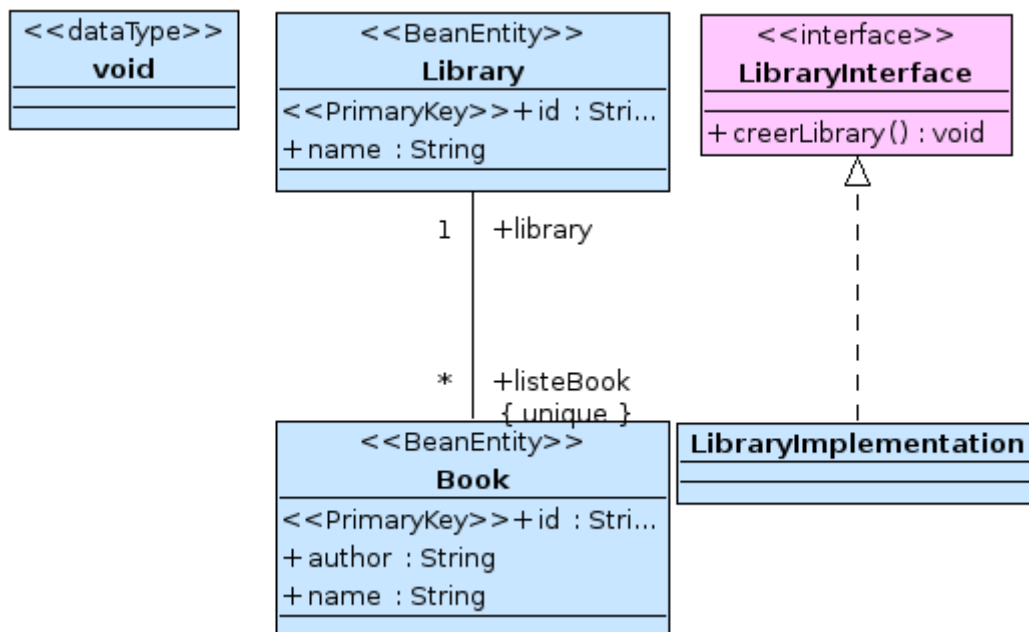
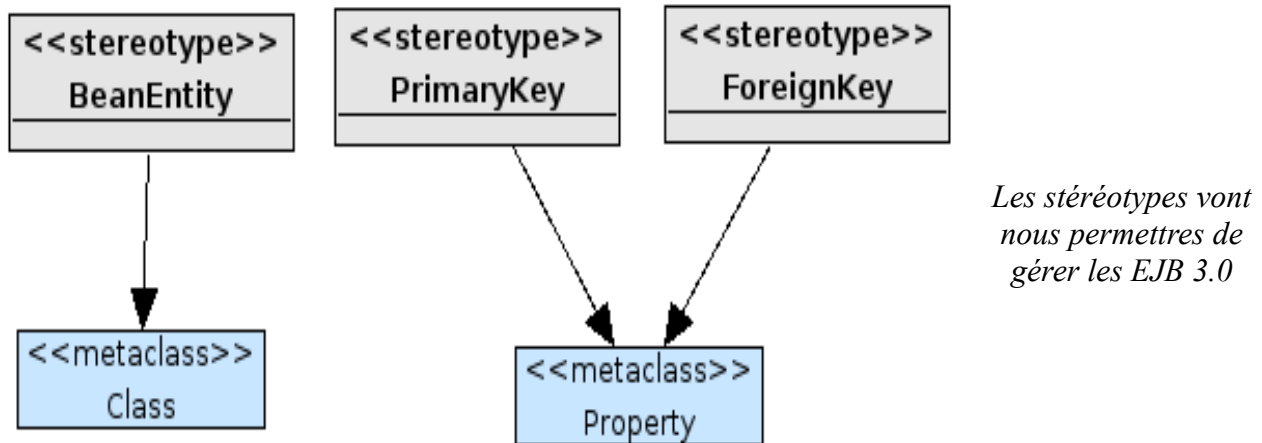
Généralité :

Ces scripts permettent de gérer certains aspects comme :

- l'héritage ;
- les affectations ;
- la conversion d'un type UML générique en type Java ;
- la staticité.

2.3.3 – Résultat de la génération de code

Afin de pouvoir déterminer la qualité de notre génération, nous avons repris le TP que nous avons fait pour apprendre les EJB 3.0. Nous avons tout d'abord modéliser sous « topCased » le diagramme qui permet la génération du code. Et avant tout, nous avons du créer nos stéréotypes.



Le diagramme de classe qui nous permet de tester la génération de code

Le résultat de la génération de code.

Library	
1	package <i>Exemple_1</i> ;
2	
3	import org.hibernate.annotations.GenericGenerator;
4	import java.io.Serializable;
5	
6	@Entity
7	public class Library implements Serializable {
8	
9	/***** VARIABLES *****/
10	List<Book> listeBook;
11	public String id;
12	public String name;
13	
14	/***** CONSTRUCTEUR *****/
15	public Library(String id, String name) {
16	this .id = id;
17	this .name = name;
18	}
19	
20	/***** SETTERS *****/
21	public void setListeBook(List<Book> listeBook) {
22	this .listeBook = listeBook;
23	}
24	
25	public void setId(String id) {
26	this .id = id;
27	}
28	
29	public void setName(String name) {
30	this .name = name;
31	}
32	
33	/***** GETTERS *****/
34	@OneToMany(mappedBy = "Library", fetch = FetchType.EAGER)
35	public List<Book> getListBook() {
36	return this .listeBook;
37	}
38	
39	@id
40	@GeneratedValue(generator = "system-uuid")
41	@GenericGenerator(name = "system-uuid", strategy = "uuid")
42	public String getId() {
43	return this .id;
44	}
45	
46	@Basic
47	public String getName() {
48	return this .name;
49	}
50	
51	}

Book	
1	package <i>Exemple_1</i> ;
2	
3	import org.hibernate.annotations.GenericGenerator;
4	import java.io.Serializable;
5	
6	@Entity
7	public class Book implements Serializable {
8	
9	/***** VARIABLES *****/
10	Library library;
11	public String id;
12	public String author;
13	public String name;
14	
15	/***** CONSTRUCTEUR *****/
16	public Book(String id, String author, String name) {
17	this .id = id;
18	this .author = author;
19	this .name = name;
20	}
21	
22	/***** SETTERS *****/
23	public void setLibrary(Library library) {
24	this .library = library;
25	}
26	
27	public void setId(String id) {
28	this .id = id;
29	}
30	
31	public void setAuthor(String author) {
32	this .author = author;
33	}
34	
35	public void setName(String name) {
36	this .name = name;
37	}
38	
39	/***** GETTERS *****/
40	@ManytoOne()
41	public Library getLibrary() {
42	return this .library;
43	}
44	
45	@id
46	@GeneratedValue(generator = "system-uuid")
47	@GenericGenerator(name = "system-uuid", strategy = "uuid")
48	public String getId() {
49	return this .id;
50	}
51	
52	@Basic
53	public String getAuthor() {
54	return this .author;
55	}
56	
57	@Basic
58	public String getName() {
59	return this .name;

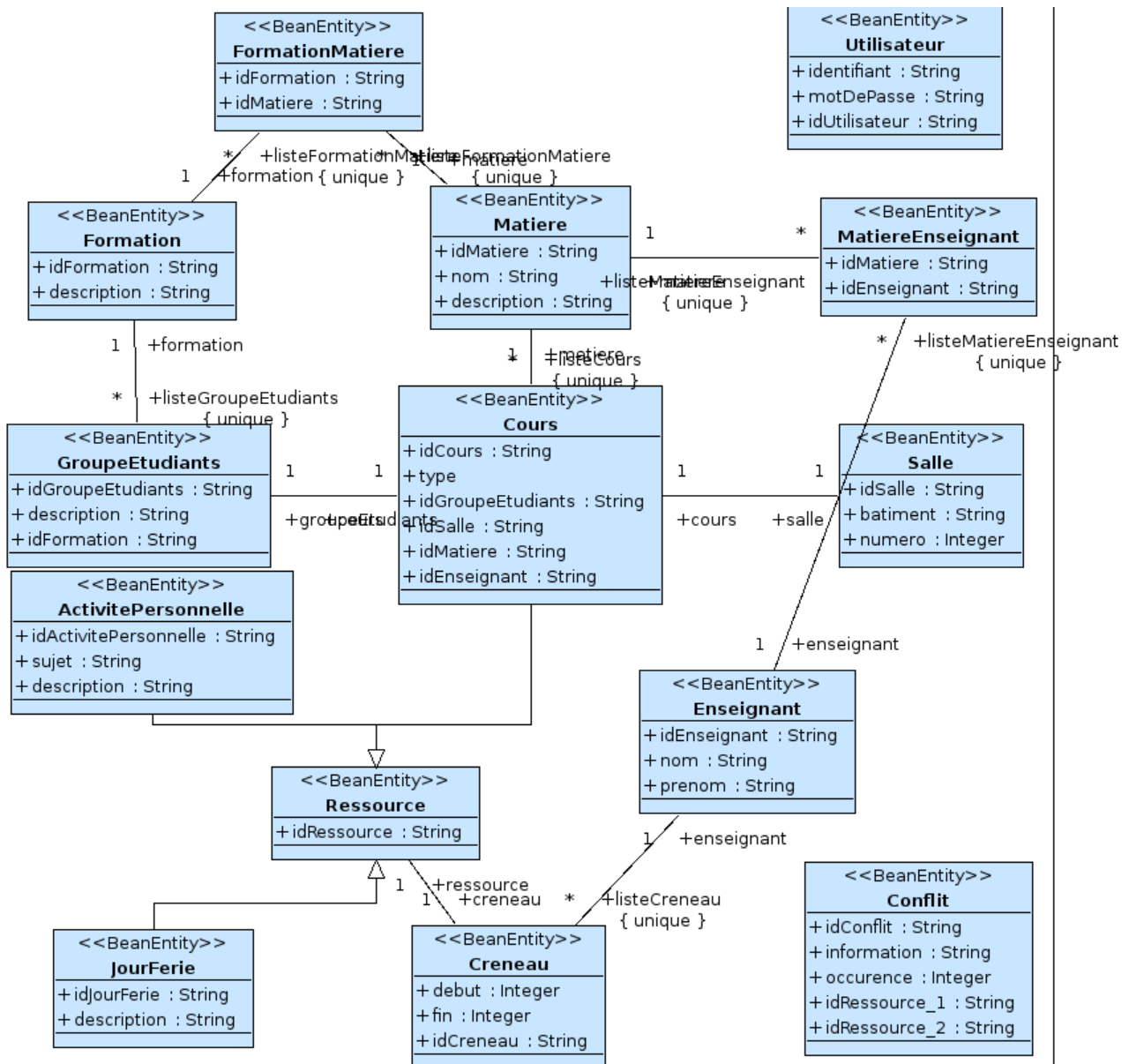
60	}
61	
62	}

Le résultat apparaît comme très bon. Cela dit, nous avons identifiés des points qui doivent être améliorés. Le code de la génération n'est donc pas totalement terminé.

2.3.4 – L'architecture du serveur

Afin de pouvoir faire fonctionner l'application de l'enseignant en mode déconnecté, nous avons estimé nécessaire de concevoir une base de donnée qui se situe au niveau du serveur de l'enseignant afin de stocker les données que l'enseignant rentre même lorsque le serveur emploi du temps n'est plus accessible. L'objectif est de resynchroniser les données lorsque la connexion entre le serveur enseignant et emploi du temps est de nouveau disponible. Au niveau de la synchronisation, nous n'avons pas encore totalement regardé dans le détail comment nous allions nous y prendre afin d'avoir des données qui soient cohérentes au niveau des deux serveurs.

Notre base de donnée est finalement une version allégée de la base de donnée qui est disponible côté serveur emploi du temps. Cependant, pour ce livrable, nous n'avons pas encore totalement épurée l'architecture. Les fonctionnalités présentes dans l'interface graphique pour l'utilisateur seront pour nous un moyen de savoir quelles sont les informations qu'il sera nécessaire de garder accessible au niveau du serveur local. Si par exemple, des informations sont régulièrement consultées, il sera probablement intéressant de les stocker dans le serveur enseignant afin de gagner en performance.



Base de donnée au niveau du serveur emploi du temps



Les conflits sont apparent dans ce diagramme de classe mais apparemment, ils ne sont plus gérés par l'enseignant (d'après la dernière analyse faite par le groupe d'Erwan).

Nous avons envisagé une autre technique qui aurait été intéressante à mettre en oeuvre. Du fait que les objets « DTO » soient serialisés, nous aurions pu les sauvegarder un par un sous forme de fichier pour la persistance avec éventuellement un cryptage pour la sécurité. Ces objets auraient ensuite été utilisable par l'interface graphique par l'intermédiaire de Java RMI. Il n'y aurait ainsi plus été nécessaire de reconcevoir une base de donnée. Cela évite notamment une duplication des objets qui est lourde.

4 – Ce que nous avons fait

Une grande partie du travail a consisté à préparer les outils qui vont nous permettre d'implémenter rapidement nos programmes. Nous avons donc :

- créé différents scripts acceleo (EJB, IDL, Java, C++) ;
- défini les technologies à utiliser ;
- préparé l'architecture du serveur Agenda (base de donnée, interfaces) ;

5 – Conclusion

Nous allons commencer par les points négatifs. Il n'y a pas vraiment eu de communication, d'explications sur les interfaces fournies par le serveur emploi du temps... Afin de rester cohérent avec les autres groupes, nous allons abandonner la base de donnée qui nous avions conçue pour le serveur agenda et prendre la version qui a été proposée. Tout cela pour dire que ce n'est pas évident de travailler sans vraiment savoir quelles sont les parties que nous devons vraiment produire. Ce projet perd de son intérêt car finalement tous les groupes vont se retrouver à produire plus ou moins la même chose.

Mise à part ses points négatifs, au final, nous sommes tout de même plutôt satisfait de ce que nous avons produit. Nous avons dorénavant une bonne vision du système dans son ensemble et aussi à plus bas niveau. Les outils que nous avons créé dans cette partie devrait nous permettre de gagner du temps pour la suite.

Annexe

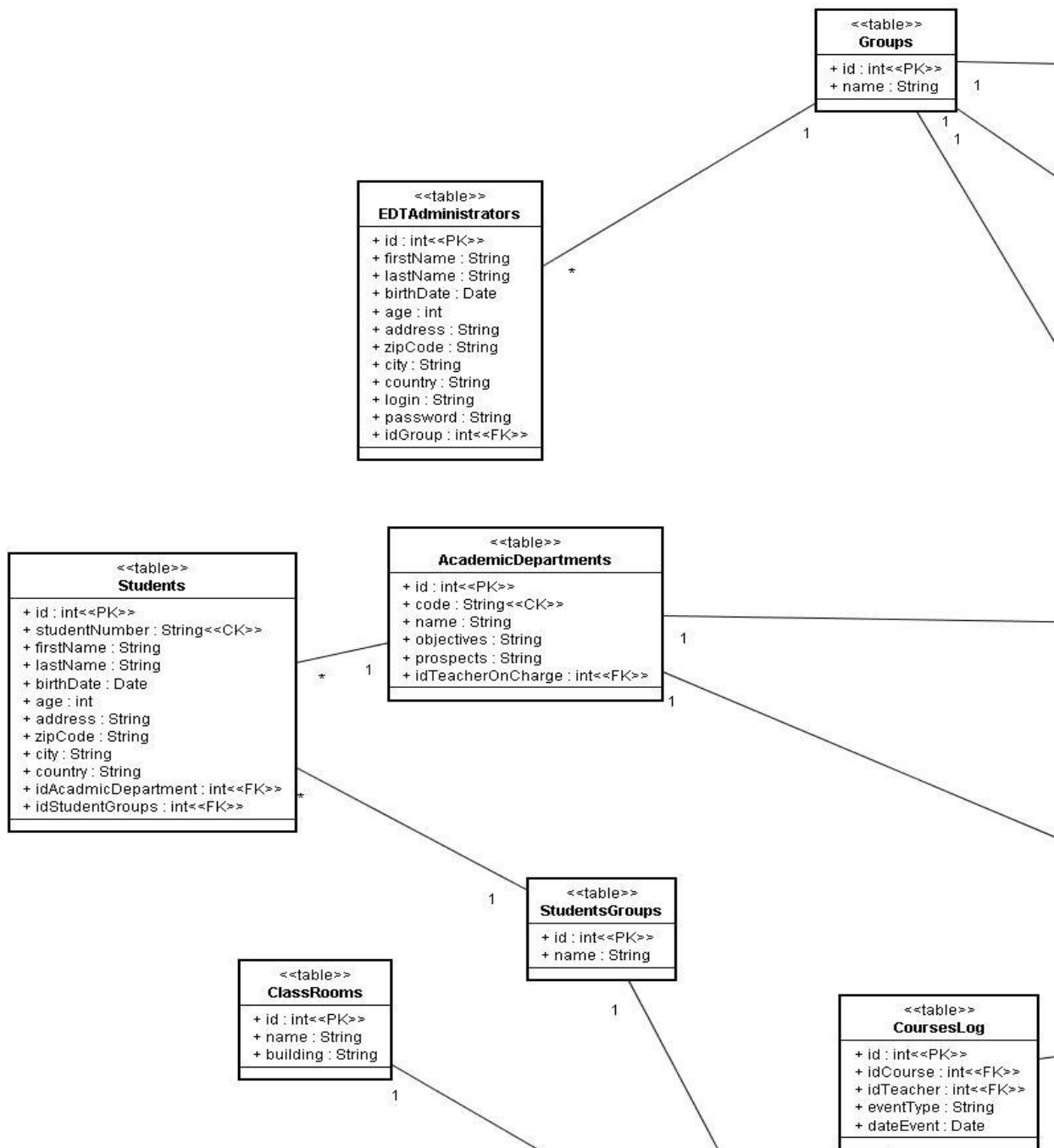
1 – Remarques préalables

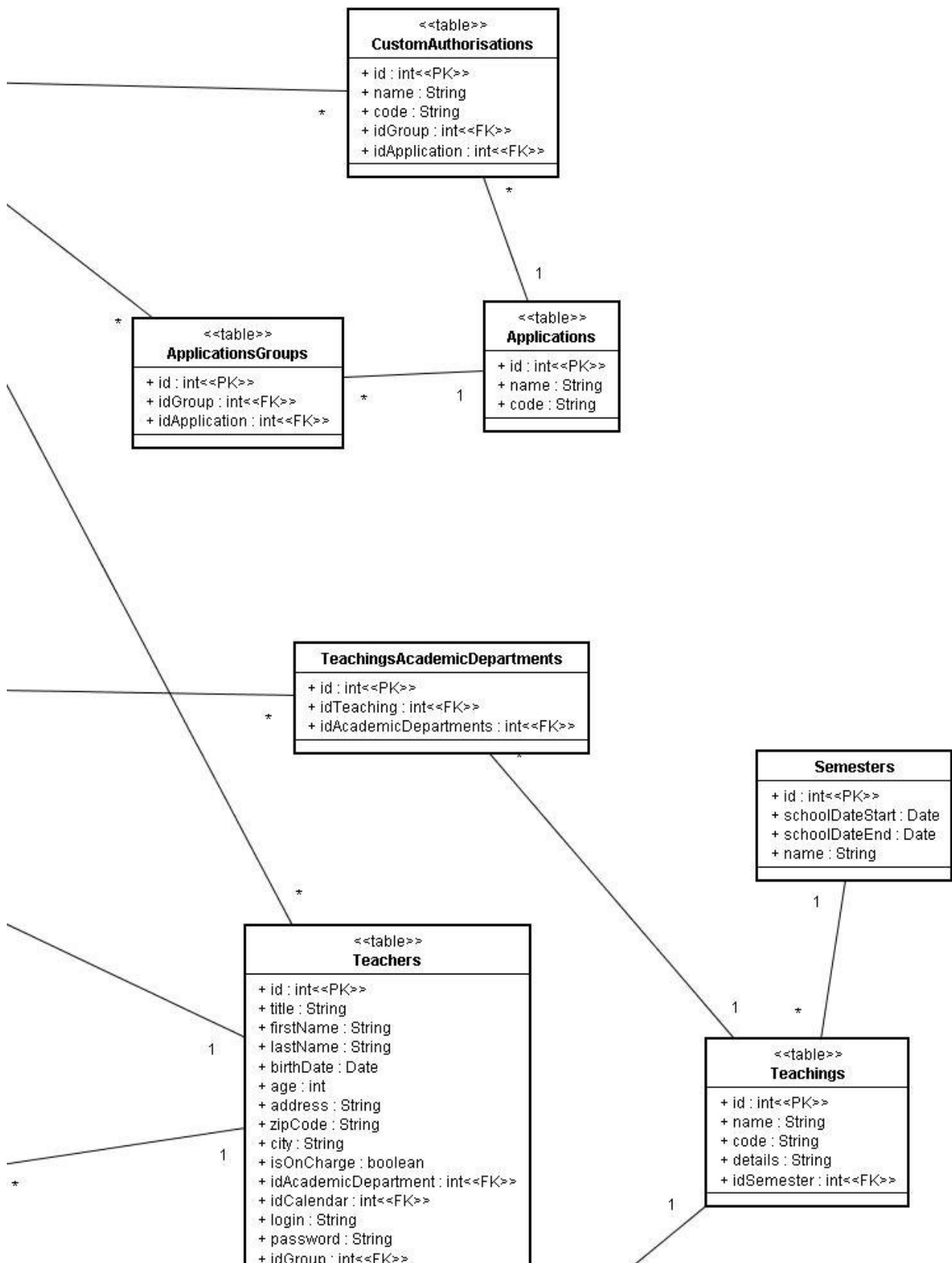
Vous pouvez retrouver l'ensemble des images et schémas ci-dessus et de manière générale présentée dans le rapport dans le dossier « images ». Pour des raisons de lisibilité, certains schémas ont été découpés.



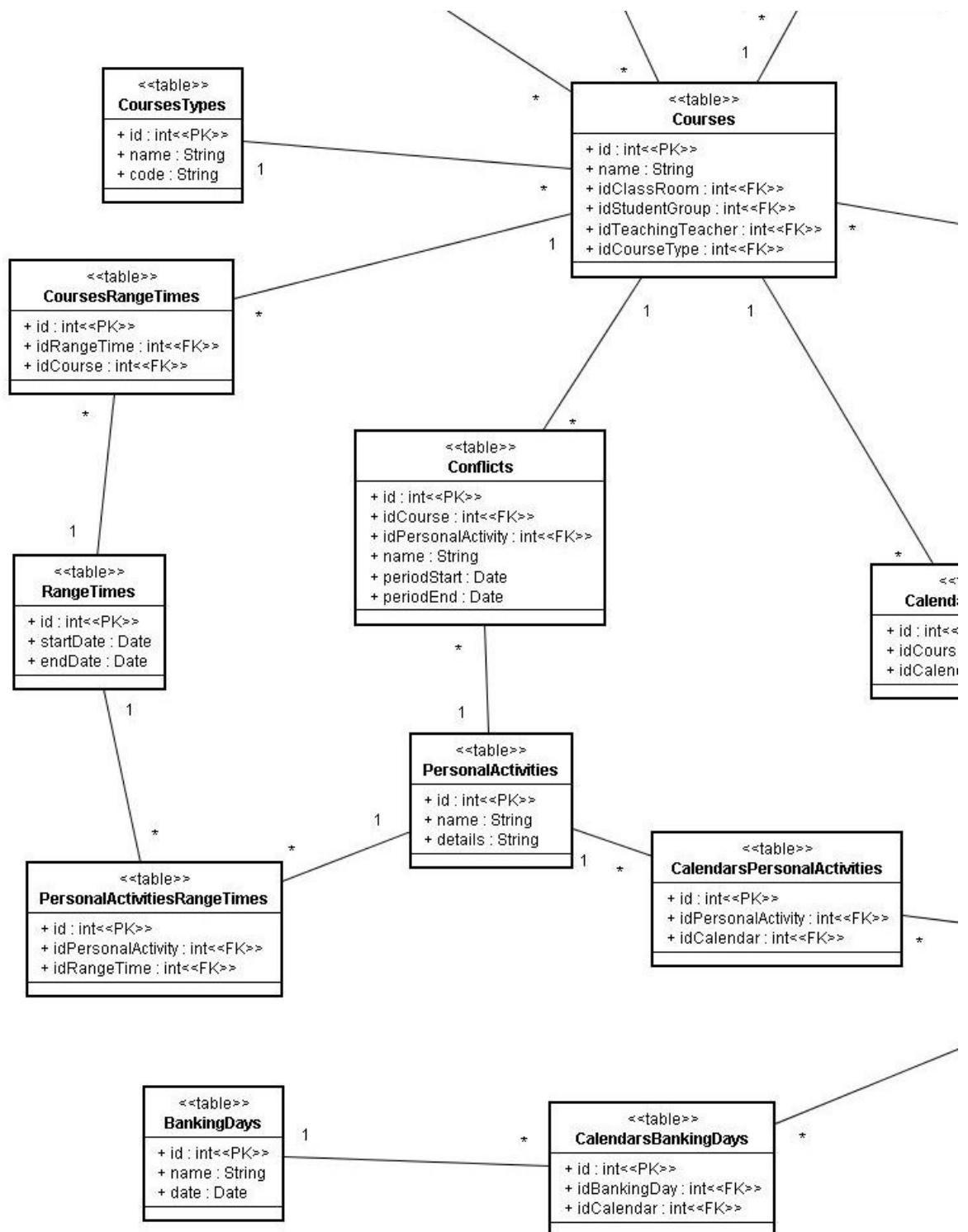
Le découpage des images a été la partie la plus intense du projet.

2 – Emploi du temps - Base de donnée

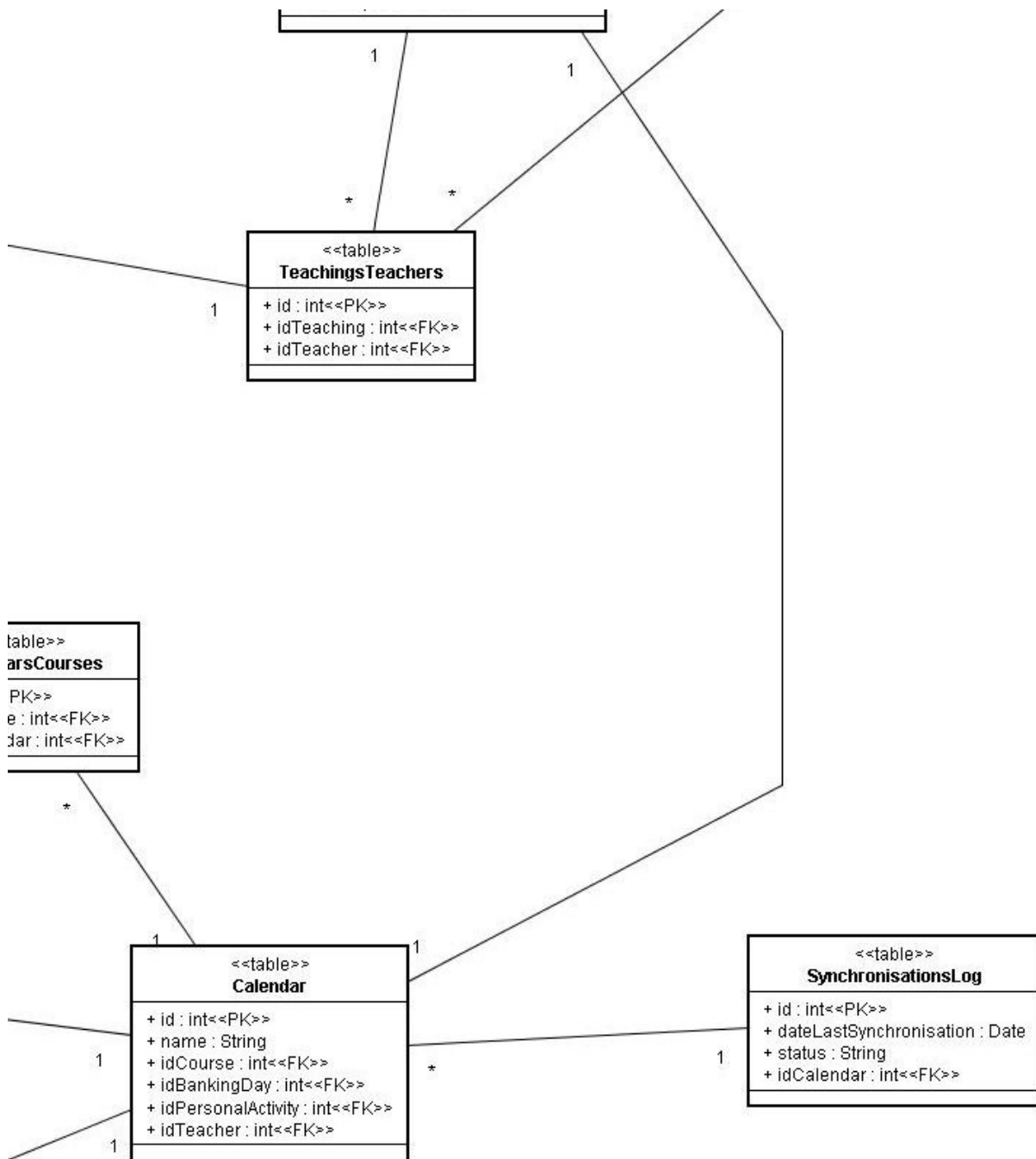
*Partie haut gauche*



Partie haut droite

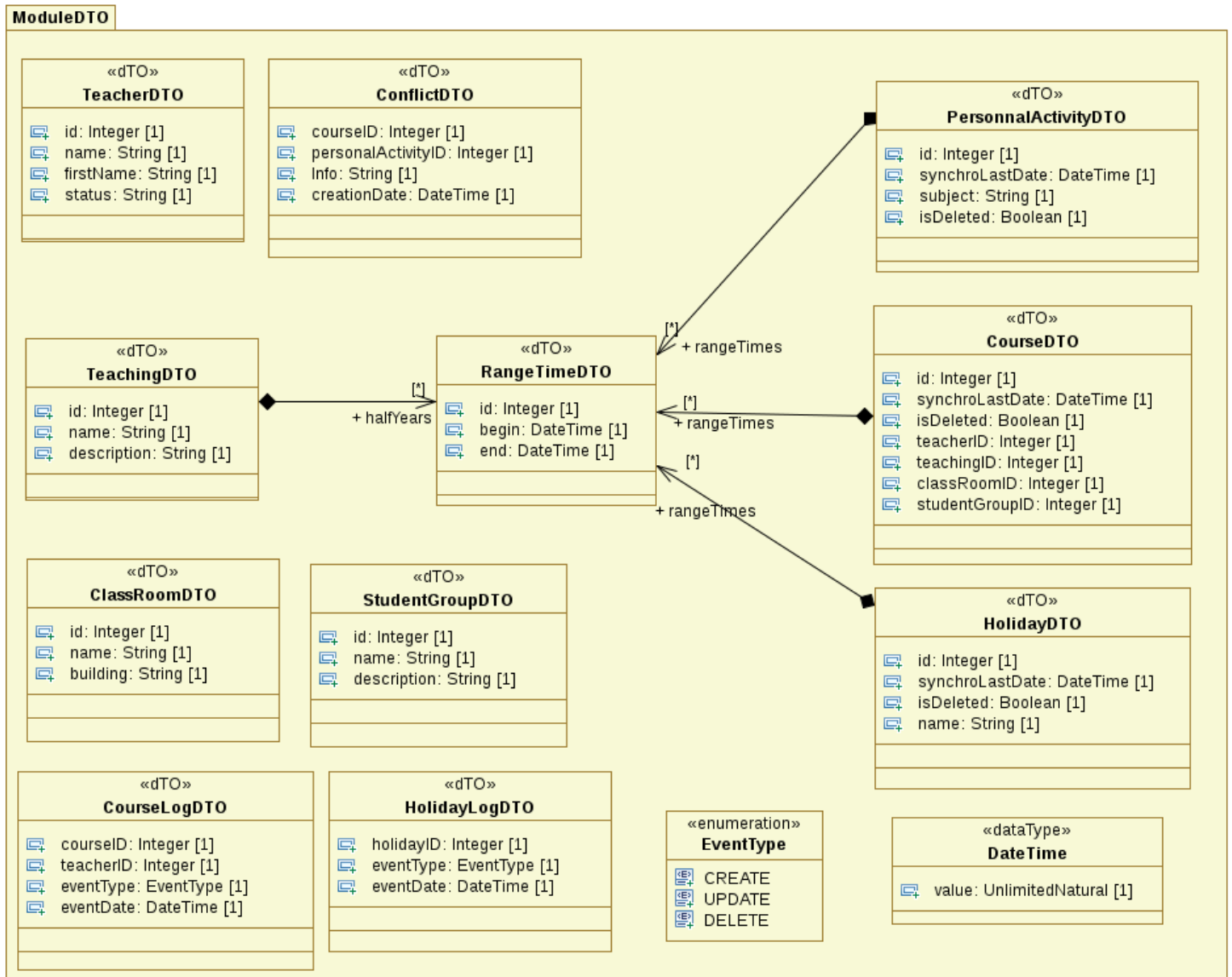


Partie bas gauche



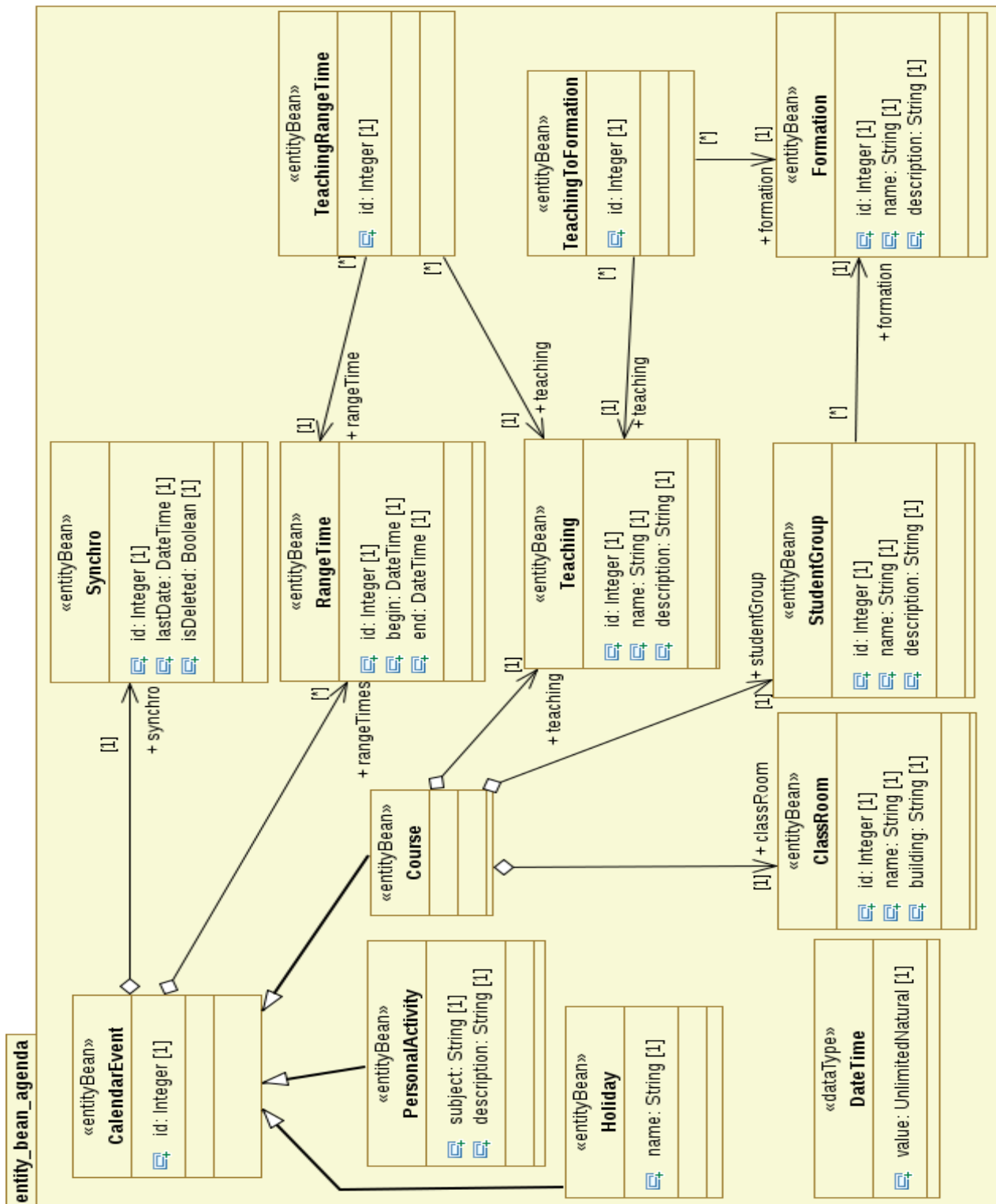
Partie bas droite

2 – Emploi du temps - DTO



Le schéma des DTO

3 – Emploi du temps - EntityBean



Les « EntityBean » côté Serveur Agenda (BD)

Gestion Emploi du temps

Livrable 4 – Conception détaillée

Matthieu Aubry
Jeremie Guidoux
Quentin Marguet
Mathieu Vénisse-Garson

Projet Multimodule
[21/12/08](#)
Encadrant : Gerson Sunyé

2008-2009 – Projet Multimodule -

Matthieu AUBRY : mas1.aubry@gmail.com

Jérémie GUIDOUX : darkjer1@gmail.com

Quentin MARGUET : zorbitos@gmail.com

Mathieu VENISSE-GARSON : mathieu.venisse@gmail.com



UNIVERSITE DE NANTES

Sommaire

1 – Introduction.....	3
1.1 – Présentation.....	3
1.2 – Problématique.....	3
1.3 – Remarques.....	3
2 – La conception – dans le détail.....	4
2.1 – Présentation.....	4
2.2 – La partie « Enseignant ».....	4
2.2.1 – Présentation.....	4
2.2.2 – La conception.....	4
a) La synchronisation.....	8
b) La création d'une activité personnelle.....	14
c) La connexion.....	15
d) La récupération des données pour l'IHM.....	16
2.3 – La partie « Responsable de formation ».....	17
2.2.1 – Présentation.....	17
2.2.2 – La structure.....	17
4 – Ce que nous avons fait.....	20
5 – Conclusion.....	20
Bibliographie.....	21
Annexe.....	21

Conception

1 – Introduction

1.1 – Présentation

Dans le livrable précédant, nous avons défini plus précisément l'architecture du programme avec notamment les différents composants. Nous avons également défini les outils, que nous allons utiliser pour procéder à l'implémentation de nos différents programmes. Nous rentrons dans une phase beaucoup plus bas niveau que précédemment. Cette partie doit permettre de procéder à l'implantation du code sans avoir de mauvaises surprises.

1.2 – Problématique

Nous devons dorénavant préciser le contenu concret de nos composants avec des explications précises sur l'ensemble des méthodes présentes. Nous allons ensuite proposer un ensemble de diagrammes statiques et dynamiques qui auront pour objectif de montrer que notre application est fonctionnelle, cohérente et correctement conçue.

1.3 – Remarques

Afin de bien rendre compte des difficultés présentes dans ce rapport, nous avons choisi de proposer beaucoup de petits diagrammes qui ont pour objectif de mettre en avant les tâches non-triviales.

2 – La conception – dans le détail

2.1 – Présentation

Nous avons deux sous-systèmes à implémenter. Les deux applications demeurent étroitement liées du fait qu'elles utilisent les mêmes interfaces en entrée.

2.2 – La partie « Enseignant »

2.2.1 – Présentation

En dehors de l'interface graphique, c'est la partie qui nécessite le plus de travail. Il y a en effet le mode « offline » à gérer et il faut donc faire une conception rigoureuse du serveur Agenda. Nous avons présenté dans le livrable précédant, notre diagramme de base de donnée. Comme nous l'avons signaler dans la conclusion, nous l'avons abandonné au profit de la modélisation de la base de donnée qui a été faite en commun.

2.2.2 – La conception

Nous devons récupérer les données présentes dans la base de donnée du serveur emploi du temps et à partir de ces données, nous devons repeupler la base de donnée locale. Pour ce faire, nous aurions pu probablement utiliser l'ingénierie des modèles et effectuer une transformation de modèle par ATL, cependant, comme nous ne maîtrisons pas encore totalement toutes les forces en présences (Corba, EJB 3.0...), nous avons procéder par une technique de transformation plus conventionnelle. C'est le code métier qui va se charger d'effectuer la transformation.

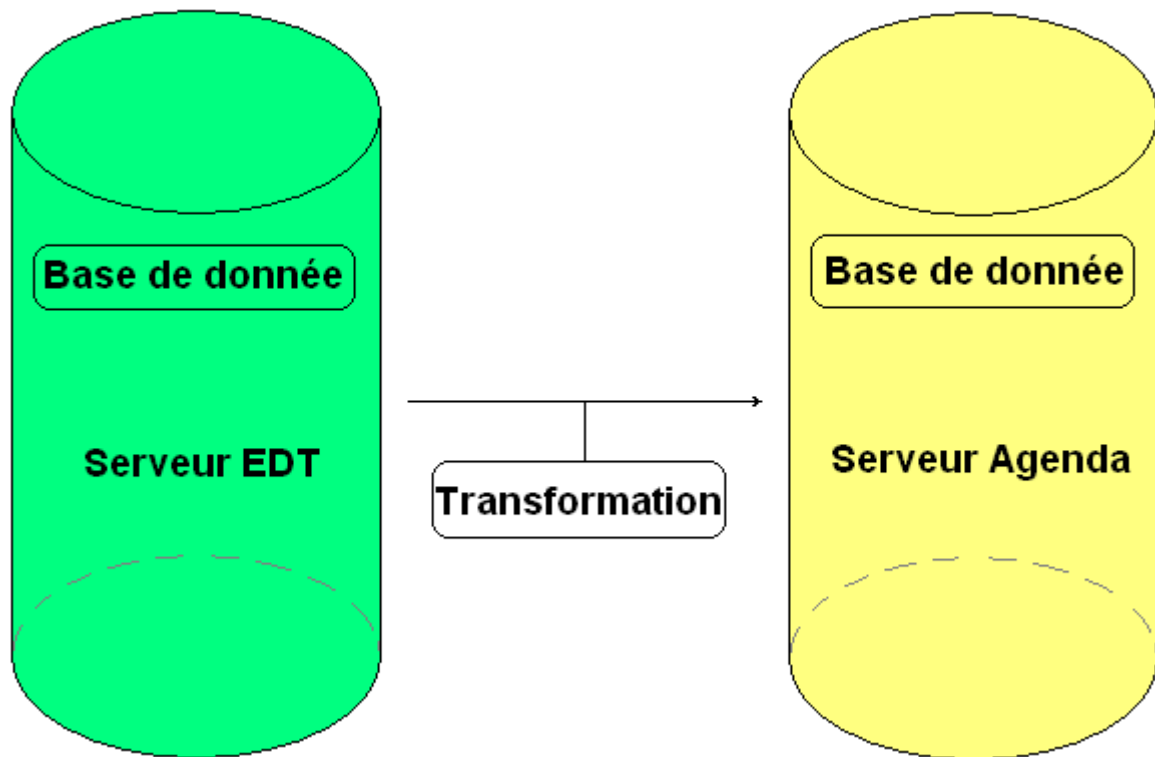


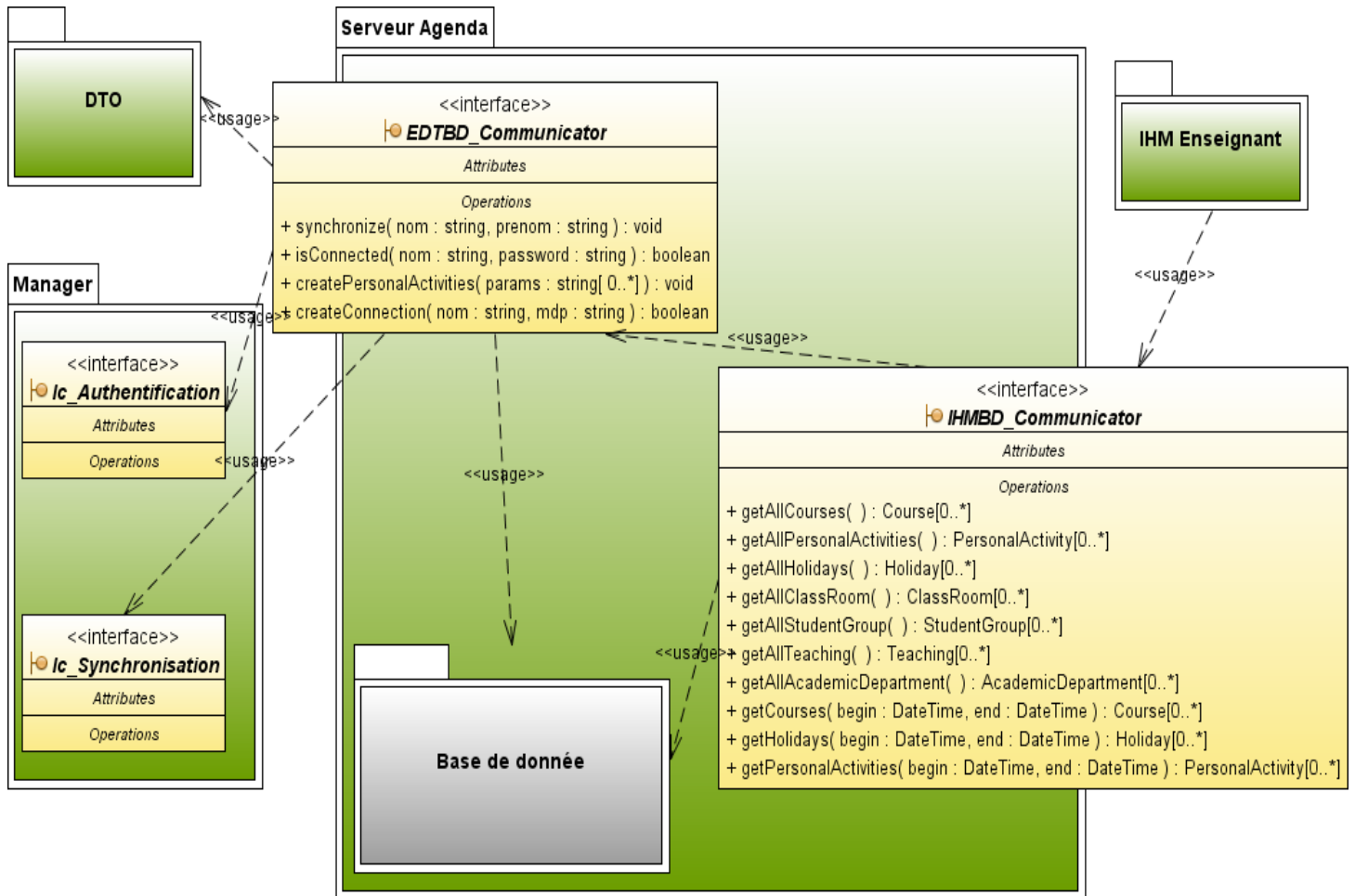
Schéma illustrant le principe de la transformation

Nous appelons cette opération une transformation car la base de donnée du Serveur Agenda est une version allégée de celle du Serveur EDT, ce n'est donc pas une simple copie. Pour ce faire, nous disposons des interfaces fournies par le serveur emploi du temps, les objets DTO et la représentation de notre base de donnée.



Le mode « offline » ajoute une complexité plus que significative à l'application de l'enseignant.

Voici le schéma qui fournit la base au fonctionnement de notre application.



Les interactions entre les interfaces et composants

Nous n'avons pas mis les méthodes que nous estimions triviales afin d'avoir un diagramme lisible. Nous allons tout d'abord expliquer ce diagramme de manière précise puis nous proposerons quelques diagrammes de séquence pour les opérations délicates.

L'interface « EDTBD_Communicator » permet principalement de récupérer les données du serveur EDT en utilisant les interfaces qui sont fournies. Ces données sont ensuite traitées puis enregistrées dans la base de donnée locale. L'interface « Ic_Synchronisation » nous fournit les méthodes qui vont nous permettre de nous connecter au serveur EDT et de gérer la connexion. Afin de cacher les interfaces du serveur EDT, nous redéfinissons ces méthodes dans notre interface « EDTBD_Communicator ». Une fois la connexion établie, la méthode « synchronise » peut-être appelée. Cette fonction va permettre la recopie de la base de donnée. Le code métier est donc assez important pour cette partie. Elle va utiliser les fonctionnalités fournies par l'interface

« Ic_Synchronization » avec notamment les méthodes qui permettent de récupérer les dernières données (cours, jours fériés...). La méthode « synchronise » prend deux chaînes de caractères en entrée. Ces chaînes de caractères vont permettre de se loguer en tenant compte de l'enseignant qui utilise le logiciel. Il y aura probablement des sous-méthodes à implémenter pour segmenter le code métier. Le rôle de cette fonctionnalité est bien définie ce qui facilitera son implémentation.

La méthode « createPersonalActivities » est la seule méthode qui peut écrire sur le serveur distant et sur la base de donnée locale. Elle va appeler la méthode permettant d'ajouter une activité personnelle présente dans l'interface « Ic_Synchronization ». Cette interface sera accessible par le composant IHM Enseignant.



Nous avons hésité à utiliser une interface intermédiaire pour bien montrer la séparation qu'il y a entre l' IHM et le serveur EDT, mais cela aurait juste alourdi notre composant.

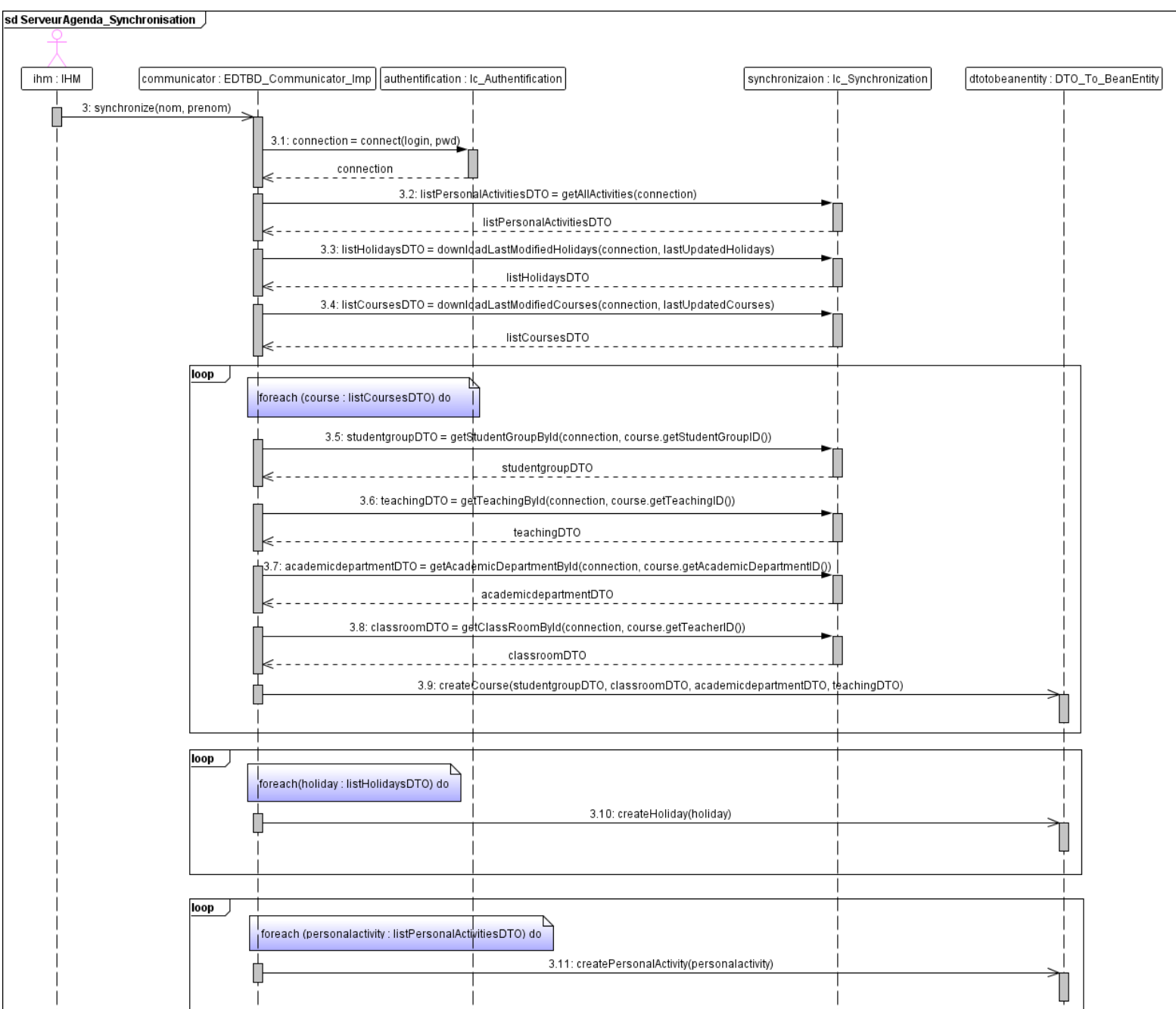
L'interface « IHMBD_Communicator » est utilisée par l'interface graphique pour récupérer les données présentes sur la base de donnée locale. Cela sera sensiblement les mêmes méthodes que celles qui se trouvent au niveau du serveur EDT sauf qu'ici, nous manipulons des « Bean Entity » et pas des « DTO ». Cette interface aura un accès en lecture seulement sur la base de donnée locale.

La base de donnée locale va posséder les « setters », « getters » pour chaque « Bean Entity » présent dans la base de donnée. Pour les opérations non-triviales, nous envisageons de créer une interface plus haut niveau permettant de faciliter, d'automatiser certaines opérations (création d'un cours, avec les heures, la matière...).

Nous allons voir les opérations qui nous semblaient délicates à traiter. Cela va nous permettre de voir un peu plus en détail les communications entre composants. Pour ce faire, nous avons choisi de proposer des diagrammes de séquences qui permettent d'illustrer simplement mais efficacement les interactions du système.

a) La synchronisation

C'est la partie la plus délicate au niveau du code métier, nous avons donc trouvé intéressant de proposer un diagramme de séquence afin de montrer de manière assez poussée le déroulement de la méthode permettant cette opération. Pour rappel, l'objectif est de récupérer les données du serveur EDT et de mettre à jour les données présentes sur la base de donnée locale.



La synchronisation – récupération des données

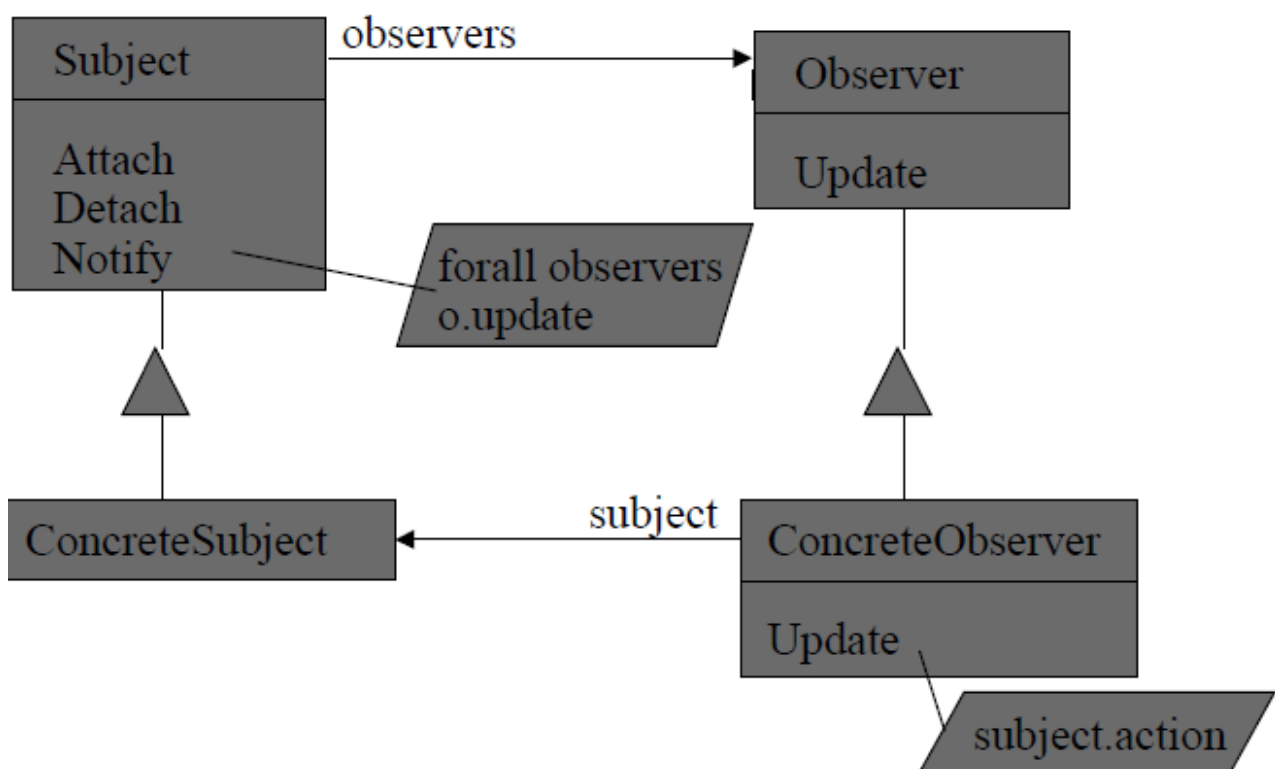
Comme le montre ce diagramme de séquence, nous récupérons les données présentes sur le serveur EDT. La méthode est asynchrone afin que l'IHM ne soit pas bloqué par l'exécution de cette opération. Nous aurions pu utiliser un thread au niveau de la conception de l'IHM mais ainsi le développeur de l'IHM n'a pas à se soucier du parallélisme.

Ce diagramme ne montre pas tout. En effet, afin de signaler que l'opération est terminée, nous avons pensé à utiliser le patron de conception « Observer » et cela n'est pas visible dans ce diagramme.



Le patron de conception « Observer » évite d'avoir un couplage trop important entre les différents composants.

La classe « EDTBD_Communicator_Impl » va se comporter comme une classe dite « Observable ». Elle va donc disposer d'une liste permettant de stocker tous les objets qui l'observent. Ainsi, elle pourra prévenir les classes qui sont enregistrées que l'opération de synchronisation est terminée.



La patron « Observer »

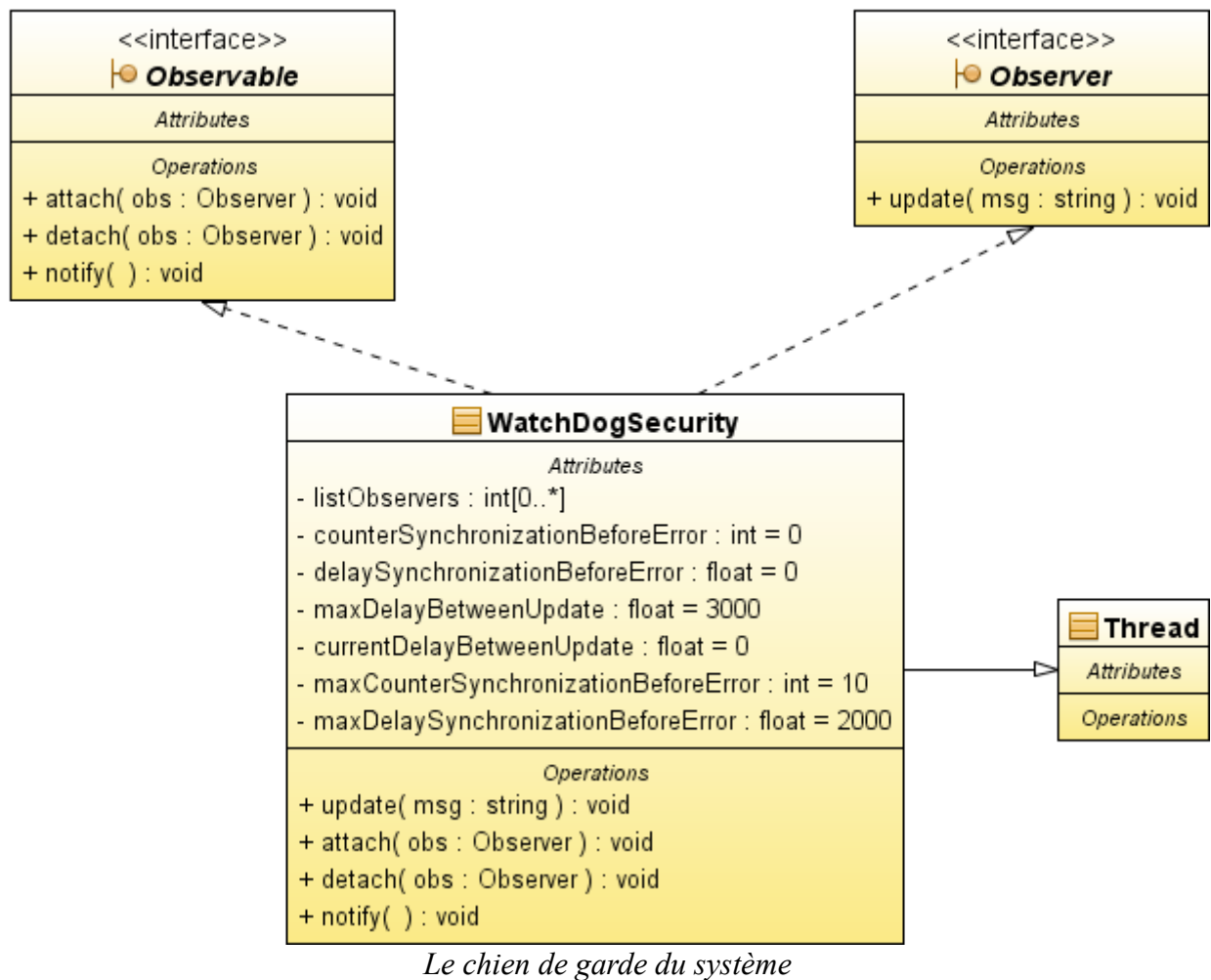
Au niveau de la sécurité, nous avons identifié un problème. En effet, si une personne mal intentionnée souhaite saturer le serveur EDT, elle peut obliger le système à faire synchronisation sur synchronisation. Cela peut entraîner une surcharge. Un autre problème qui peut-être très ennuyant, est un blocage durant la récupération des données. Il est nécessaire de tenir compte de ces éventuels problèmes.



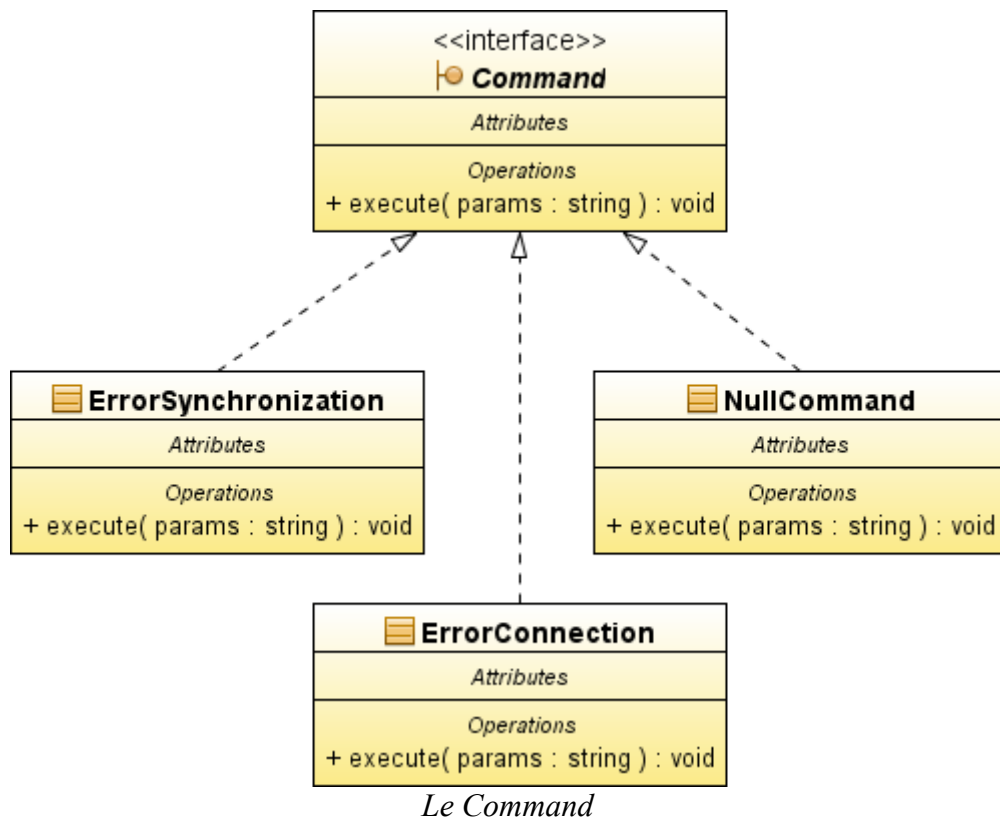
Le patron de conception « Watchdog » est un bon moyen d'observer le système pour éviter le dernier problème. Il peut également servir pour résoudre l'autre problème. En résumé, il nous est vraiment utile.

Pour essayer d'éviter ce genre de problème, nous pensons mettre en place un système de surveillance. Le « watchdog » nous semble vraiment être la solution à notre problème. Il sera un des observateurs du système. Mais son rôle sera d'assurer la sécurité et de contrôler le bon déroulement des opérations.

Entre chaque récupération de données dans la méthode de synchronisation, nous allons émettre un signal. Le « chien de garde » va ainsi pouvoir vérifier que le système répond en s'assurant que les étapes ne prennent pas trop de temps. En cas de problème, il générera lui-même un signal qui sera récupérable par l'IHM et permettra donc d'avertir l'utilisateur qu'il y a un problème. Le « chien de garde » sera donc à la fois « observateur » et « observable ». Pour résoudre le problème des synchronisations multiples, il va compter le nombre de synchronisations tout en tenant compte du temps. C'est-à-dire que si par exemple une personne effectue 20 synchronisations sur une période de 1 minute, il va bannir cette personne durant 10 minutes et lui envoyer un message pour l'informer.

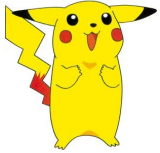


Comme nous pouvons le voir, le « watchdog » sera utilisé dans un thread séparé ce qui paraît logique vu qu'il doit tenir compte du temps. Comme il gère des erreurs de tout le système devant être traitée de manières différentes, il est possible que nous soyons amené à intégrer le patron de conception « Command ». La méthode « update » récupérera le type d'erreur et le traitement sera effectué par une classe spécifique. Nous attendons d'avoir un code fonctionnel pour gérer proprement toutes les erreurs. En attendant, voici les quelques sources d'erreurs que nous avons identifiés et que nous avons préparé à être traités par le patron « Command ».



Ce diagramme est juste là pour l'exemple. Il y aura au final beaucoup plus de classes car il y a beaucoup de cas d'erreurs possibles à traiter. Il est possible que le « WatchDog » devienne par extension la classe de communication privilégié entre l'IHM et le serveur Agenda. Au moment de l'implémentation, nous regarderons si il est nécessaire de redéfinir une autre classe communicante.

Revenons à l'étude du diagramme de séquence. Nous avons pu voir que nous récupérions les données sur le serveur EDT. Il est maintenant temps de s'intéresser à la mise à jour de la base de donnée locale.



C'est pas trop tôt. Le « watchdog », il commençait à me fatiguer.. On va enfin pouvoir regarder le détail de la mise à jour des données.

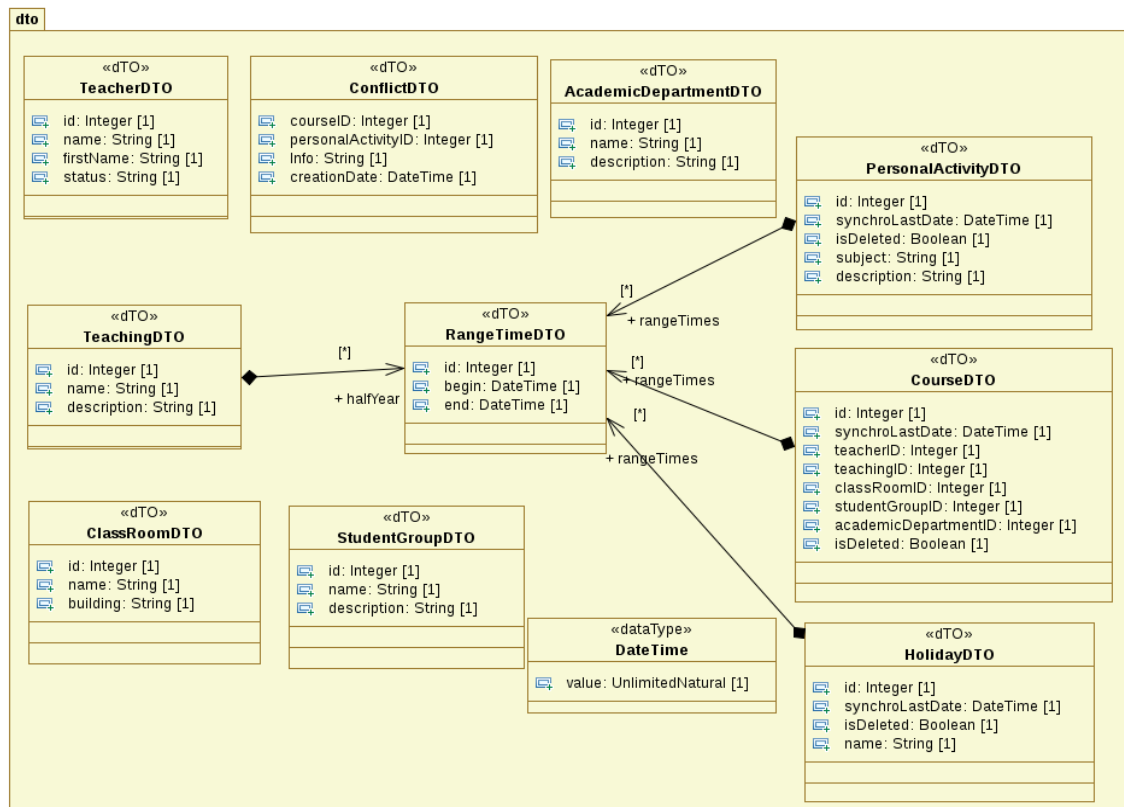
Avant de regarder le détail de la mise à jour. Il faut avoir à l'esprit qu'il y a deux manières d'aborder une mise à jour. Soit, nous utilisons une technique simple et efficace mais qui à le désavantage de faire perdre un peu en performance, soit nous faisons cela proprement. Le premier cas consiste à détruire toutes les données avant de les mettre à jour. En conséquence, le code métier est plus que simple mais c'est plus lent. L'autre technique consiste à ne mettre à jour que les données qui ont été modifiées. Le code métier est alors plus compliqué mais nous gagnons en performance puisqu'il ne faut pas tout reconstruire.



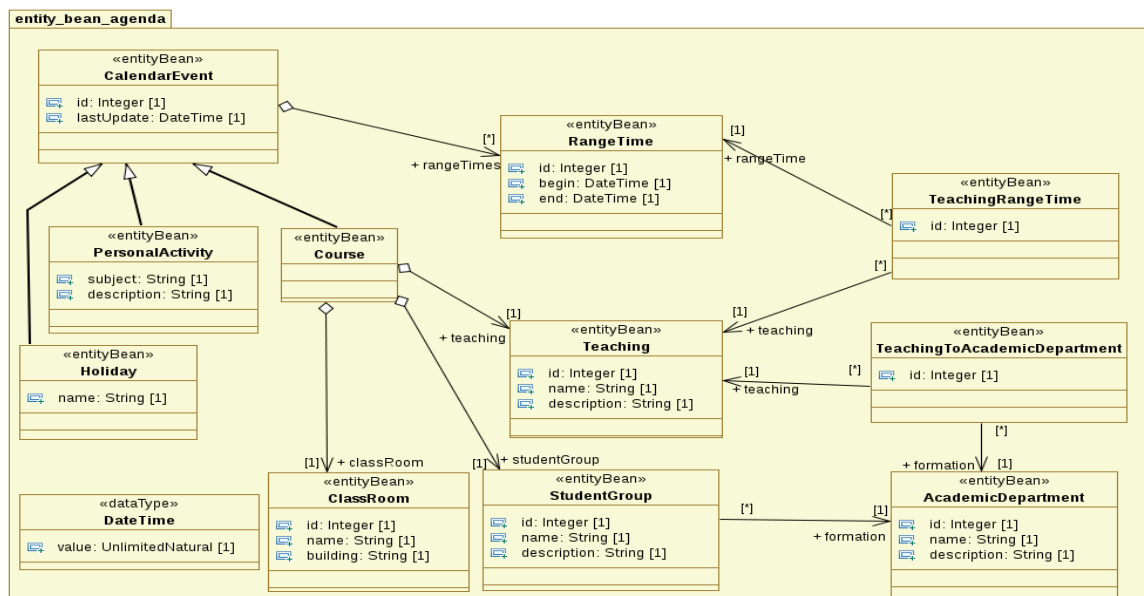
Comme nous ne sommes pas vraiment sérieux (plutôt flemmard me dit-on), nous avons choisi la méthode la plus mauvaise mais la plus facile...

En fait, la première méthode pourrait être plus mauvaise mais ce n'est pas le cas. Si nous regardons les interfaces fournies par le serveur EDT, nous remarquons que nous pouvons récupérer les dernières données modifiées mais cela, malheureusement pas pour tous les DTO. Déjà, l'intérêt de la mise à jour des dernières données modifiées en prend un coup. Un autre inconvénient de la méthode de mise à jour des données modifiées est qu'il faut parcourir les données de la base de donnée locale pour savoir quoi mettre à jour (suppression, modification ?), ce qui finalement ajoute du temps de traitement au niveau du serveur local. De plus, les identifiants doivent être les mêmes que pour le serveur EDT afin que les modifications se fassent correctement. Les données transmises dans le cas où il y a reconstruction complète seront légèrement plus importantes mais au moins, il n'y aura aucun souci au niveau des données présentes en locales. Elles seront systématiquement parfaitement cohérente avec le serveur EDT. Tout cela, nous a amené à choisir la première solution.

La mise à jour des données consiste donc à transformer un DTO en « BeanEntity ». Voici donc un schéma de la mission qui nous attend.



Transformation



Pour faire la transformation, nous avons défini une classe qui sera finalement un EJB Session qui va manipuler les DTO pour les transformer en « BeanEntity » et ainsi mettre à jour notre base de donnée. C'est la classe « DTO_To_BeanEntity » qui est présente dans le diagramme de séquence. Elle ne possède que 4 méthodes mais cela s'avère suffisant pour permettre à la base de donnée de se remplir.

DTO_To_BeanEntity
Attributes
Operations
+ createHoliday(params : HolidayDTO) : void + createPersonalActiviy(params : PersonalActivityDTO) : void + createCourse(params_1 : StudentGroupDTO, params_2 : ClassroomDTO, params_3 : AcademicDepartmentDTO, params_4 : TeachingDTO) : void + eraseAll() : void

La transformation des DTO en BeanEntity

Il n'y a pas grand chose à signaler pour cette classe. Ce que l'on remarque, c'est qu'elle a une méthode qui permet d'effacer totalement la base de donnée. Cela doit être fait avant son remplissage. La méthode pour créer les cours devra tenir compte de quelques subtilités. Il faudra éviter qu'elle réécrive systématiquement les « DTO » comme les « AcademicDepartmentDTO », les « TeachingDTO » et les « ClassroomDTO ».

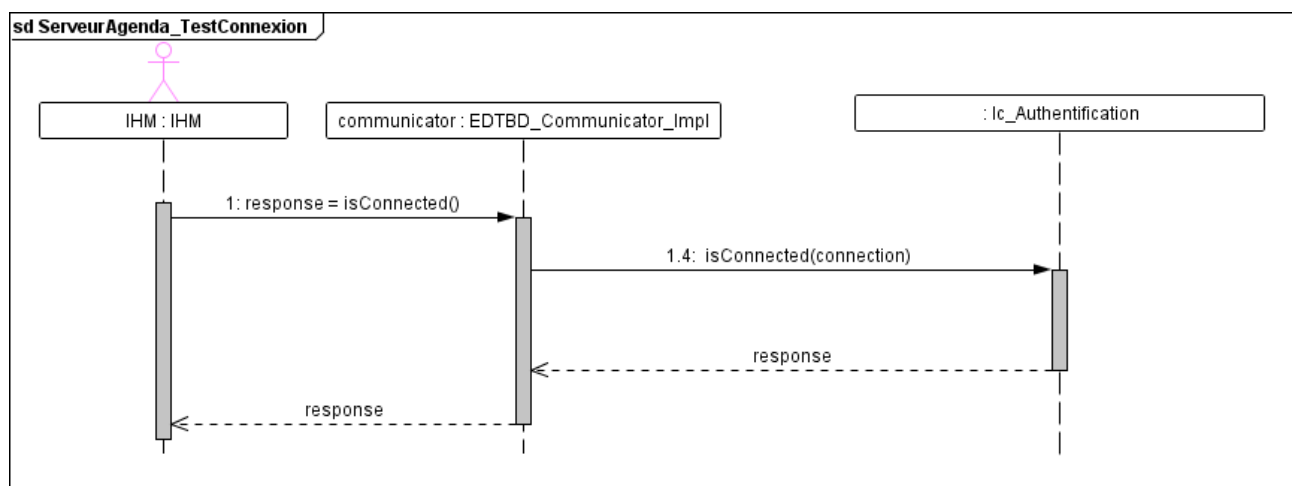
b) La création d'une activité personnelle

C'est la seule opération intrusive que l'enseignant peut effectuer. La méthode « createPersonalActivity » va envoyer au serveur EDT un « PersonalActivityDTO ». Nous avons donc un accès en écriture et c'est le seul pour la partie de l'Enseignant. Si la mise à jour sur le serveur est un succès, la méthode va ensuite mettre à jour la base de donnée locale. L'objet en paramètre de la méthode est une liste de « string ». Il aurait également été possible de mettre un DTO directement en entrée mais nous aurions augmenté le couplage avec l'application de l'interface graphique.

c) La connexion

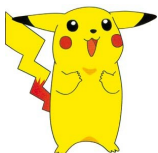
Quand nous parlons de la connexion, nous voulons parler de la méthode qui permettra de déterminer si le serveur EDT accepte la connexion du serveur Agenda. C'est donc une méthode très simple qui servira à l'IHM afin d'indiquer à l'utilisateur si tout est correctement configuré par exemple (pour accéder au serveur EDT).

Cela n'a pas vraiment d'utilité mais puisque nous l'avons fait autant le mettre, voici un diagramme de séquence qui illustre l'utilisation de la méthode « isConnected ».



Un test de connexion

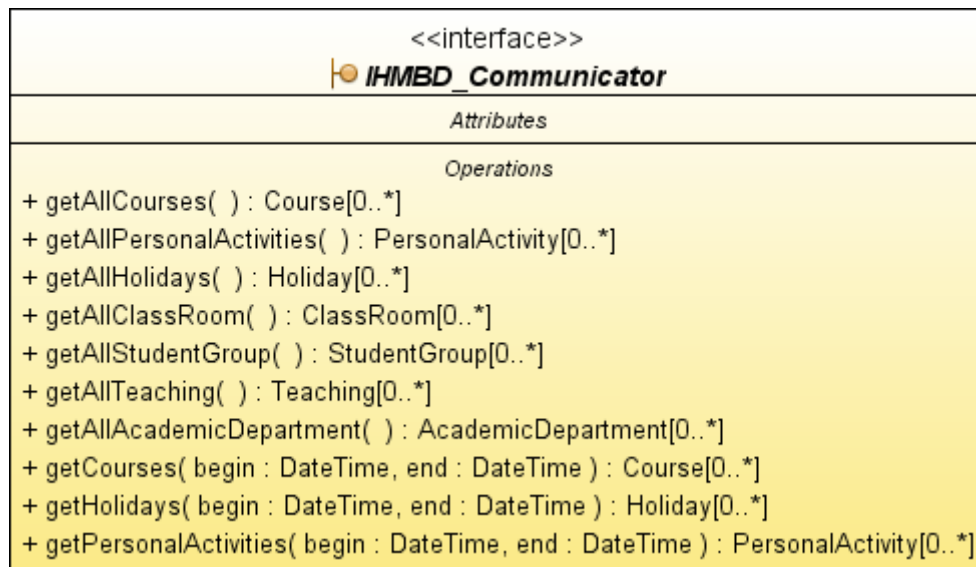
Ce diagramme montre que « isConnected » prend un paramètre au niveau de « Ic_Authentification ». Ce paramètre doit être initialisé avant d'utiliser cette méthode. « connection » est une variable de classe qui est initialisé par la méthode « createConnection ».



Si on commence à mettre des diagrammes comme ça, on va pas aller loin...

d) La récupération des données pour l'IHM

Il est temps maintenant de présenter l'interface qui va permettre à notre IHM d'afficher les données présentes sur la base de donnée. C'est l'interface « IHMBD_Communicator » qui va fournir l'ensemble des méthodes d'accès en lecture à la base de donnée locale. Elle ressemble un peu à ce que fournit l'interface du serveur EDT mais au lieu de manipuler des DTO, nous manipulons cette fois des « BeanEntity ».



L'interface de communication IHM / Serveur Agenda

Nous ne sommes pas encore tout à fait au point sur cette interface. Il est possible que nous ayons des besoins qui nous amène à ajouter des méthodes et des fonctionnalités. La seule particularité de cette interface est de proposer de récupérer des données selon une plage horaire (les trois dernières méthodes).

2.3 – La partie « Responsable de formation »

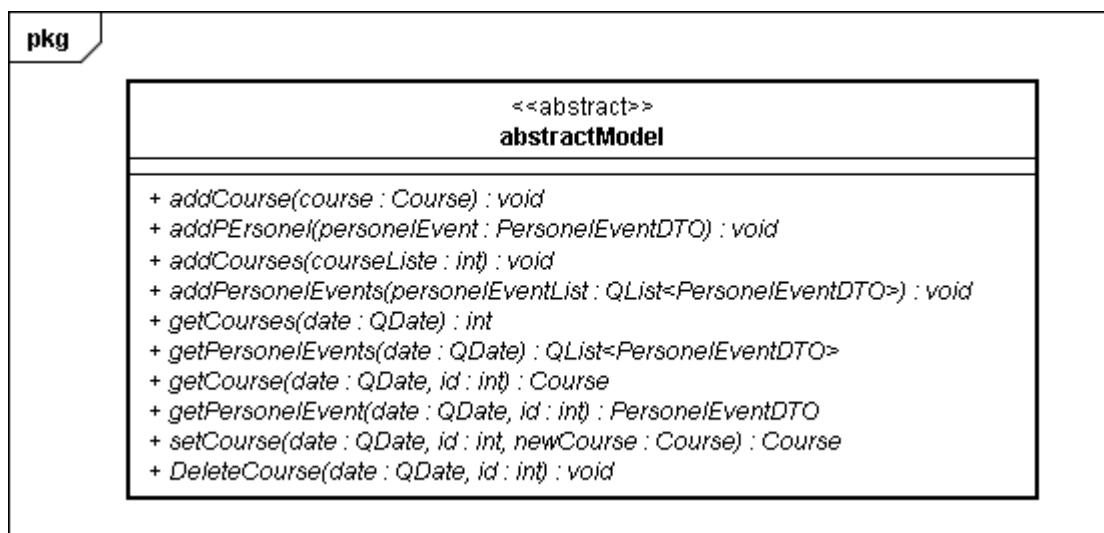
2.2.1 – Présentation

Il y a beaucoup d'échanges entre le serveur EDT et l'application locale dans cette partie. Mais à la différence de la section précédente, nous n'avons pas à stocker les données en locales. Nous pouvons donc directement nous servir des données du serveur EDT et donc de l'interface distante. Le travail à effectuer est donc moindre pour la partie conception. Mais l'interface graphique associée est clairement complexe à concevoir du fait qu'il y ait beaucoup de données liées à traiter. Afin de produire un contenu intéressant, nous allons aborder cette partie en regardant ce qui se passe du côté de la structure de l'IHM.

2.2.2 – La structure

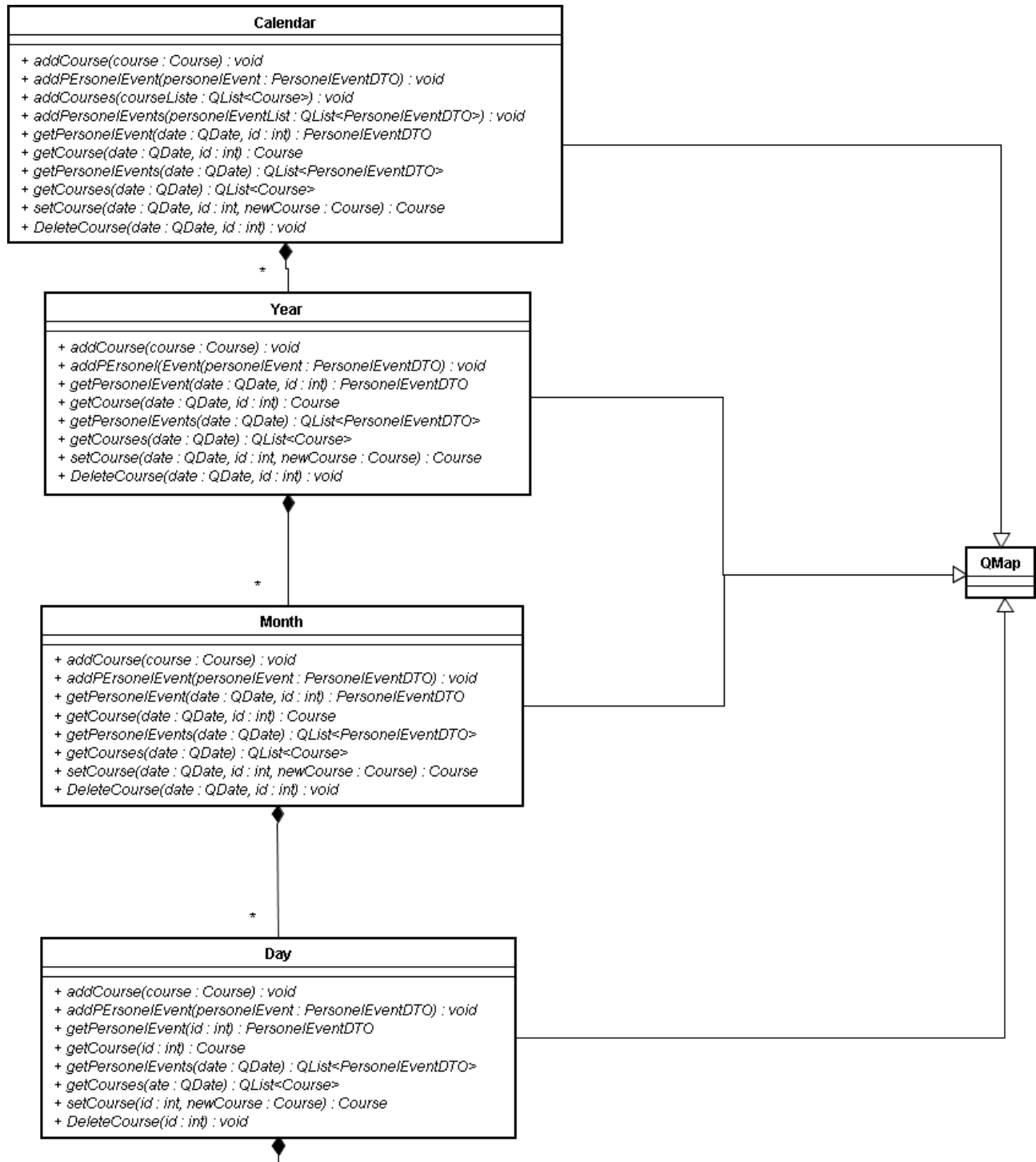
Notre IHM a besoin d'interagir directement avec les DTO fournis par l'interface du serveur EDT. C'est pour cela que dans les interfaces qui suivent, les DTO se mélangent à nos propres classes. Le couplage est donc très fort. Comme nous allons le voir, nous avons utilisé le patron de conception MVC pour diriger la conception de nos interfaces.

Voici la classe abstraite qui définit les méthodes de notre modèle.



Le modèle de donnée

Nous retrouvons ensuite notre structure qui représente les événements de manière temporelle. Notre calendrier est composé d'années qui contiennent des mois, eux-même contenant des jours. Le tout hérite des « hash_map ». Cela permet d'avoir un accès simplifié aux différents éléments. Cela dit, l'utilisation des « hash_map » imbriqué augmente considérablement l'accès aux données.

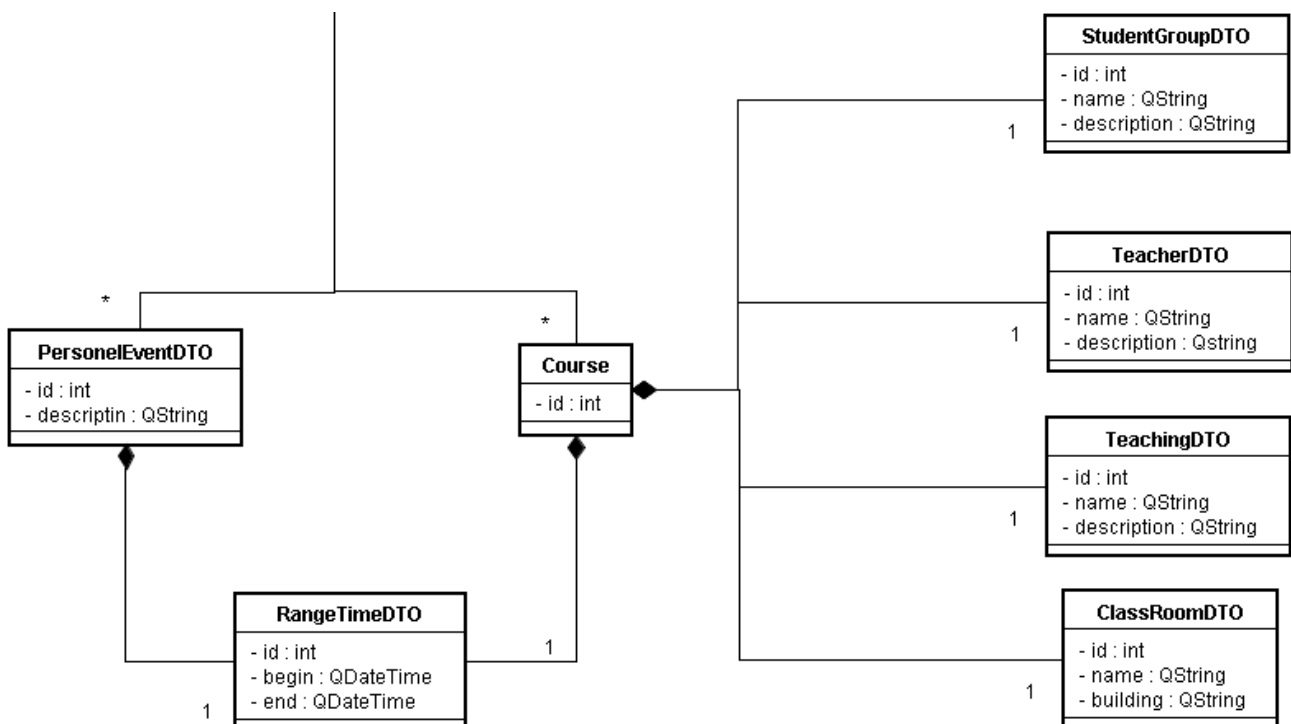


La structure temporelle côté IHM



Les mauvaises langues diront que nous avons choisi la facilité.

Le diagramme n'est pas fini. Un jour contient en effet soit un événement personnel, soit un cours, soit un jour férié (bien que ce dernier ne soit pas apparent). Nous retrouvons les liens vers les DTO qui montrent le fort couplage.



La structure bas niveau

4 – Ce que nous avons fait

Pour ce livrable, il a été nécessaire de vraiment approfondir ce que nous avons fait précédemment. Nous avons pu nous rendre compte qu'il y avait encore beaucoup de travail pour cette partie qui est maintenant heureusement accompli. Peut-être avons nous été toujours trop haut niveau lors des livrables précédents ce qui fait que nous n'avons pas forcément senti les problèmes d'implémentation venir. Enfin cela dit, nous gardons toujours une bonne partie de ce que nous avons pensé dès le premier livrable ce qui est plutôt encourageant.



Reprends toi ! Tu pars dans un hors-sujet total. Il faut parler de ce qu'on a fait !

Revenons à nos moutons, voici ce que nous avons fait dans ce livrable :

- développement des interfaces du serveur Agenda ;
- mise en place des outils pour gérer la sécurité ;
- mise en place du système de mise à jour de la base de donnée ;
- création de la structure MVC pour la partie « Responsable de Formation ».

Il ne nous reste plus qu'à produire le code métier puisqu'avec le code Accileo, nous pouvons générer les interfaces des serveurs distants et la base de donnée locale avec les EJB.

5 – Conclusion

Ce livrable nous a permis d'appliquer concrètement des patrons de conception que nous avons pu voir au cours de cette année. C'est un vrai plaisir de voir qu'ils répondent parfaitement à certain de nos besoins. Cela est une réelle satisfaction de se rendre compte que ce que nous avons produit, il y a de cela 3 mois va finalement pouvoir être fonctionnel sous peu. Il y aura toujours un petit pincement au cœur du fait que nous n'ayons pas pu concevoir l'application dans son ensemble. Mais ce projet reste pour nous vraiment une excellente expérience qui a permis d'appliquer une grande partie de ce que nous avons appris tout en faisant bouger nos neurones.

Bibliographie

Pour ce livrable, nous n'avons pas vraiment eu besoin d'aide extérieure. Nous avons seulement un corps étranger dans le rapport : le diagramme du patron de conception « Observer ».

[ENST Bretagne]
A. Beugnard 1999 Les design patterns

Annexe

Vous pouvez retrouver l'ensemble des images et schémas ci-dessus et de manière générale présentée dans le rapport dans le dossier « images ».



Au lieu de mettre des diagrammes illisibles à la fin ou découpés de manières inappropriées, on a fait le choix de fournir une archive avec les images de nos diagrammes. Il nous aura fallu 4 livrables pour changer cette mauvaise habitude...