



Université Catholique de Louvain  
Louvain School of Engineering  
Department of Computer Engineering

## AUTOMATISATION DE LA GESTION DES PROGRAMMES DE COURS

*Auteur*

Xavier CROCHET

*Promoteurs*

Kim MENS

Chantal PONCIN

*Lecteur*

Bernard LAMBEAU

Mémoire présenté dans le cadre  
du *Master 120 en Sciences*  
*Informatiques* option *sécurité et*  
*réseaux*

Louvain-la-Neuve  
Juin 2014

---

Un grand merci à Chantal  
Poncin et à Kim mens pour leur  
support, conseils et excellente  
disponibilité.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contexte . . . . .	1
1.2	Problème . . . . .	3
1.3	Motivation . . . . .	6
1.4	Objectifs . . . . .	9
<b>2</b>	<b>Énoncé du problème</b>	<b>13</b>
2.1	Programmes proposés . . . . .	13
2.2	Contraintes . . . . .	16
2.3	Conclusion . . . . .	17
<b>3</b>	<b>Présentation du système</b>	<b>19</b>
3.1	Introduction . . . . .	19
3.1.1	Exemple d'utilisation . . . . .	20
3.2	Technologies utilisées . . . . .	22
3.2.1	Introduction . . . . .	22
3.2.2	Ruby on Rails . . . . .	22
3.2.3	Base de données - PostgreSQL . . . . .	27
3.2.4	Éditeur de graphes - yEd . . . . .	27
3.3	Conception . . . . .	29
3.3.1	Introduction . . . . .	29

---

3.3.2	Gestion des données . . . . .	30
3.3.3	Contraintes . . . . .	33
3.3.4	Fonctionnalités de l'application - Commission de programme . . .	37
3.3.5	Fonctionnalités de l'application - Étudiant . . . . .	47
3.3.6	Conclusion . . . . .	54
<b>4</b>	<b>Développement du système</b>	<b>57</b>
4.1	Introduction . . . . .	57
4.2	Architecture . . . . .	58
4.2.1	Architecture globale . . . . .	58
4.2.2	Application Rails . . . . .	59
4.2.3	Architecture du vérificateur de contraintes . . . . .	65
4.2.4	Parser de graphes . . . . .	66
4.2.5	Parser de fichiers excel . . . . .	67
4.3	Implémentation . . . . .	68
4.3.1	Introduction . . . . .	68
4.3.2	Hébergement de l'application . . . . .	68
4.3.3	Gestion des utilisateurs . . . . .	68
4.3.4	Importation du graphe . . . . .	69
4.3.5	Gestion des contraintes . . . . .	73
4.3.6	Importation du formulaire Excel . . . . .	77
4.4	Qualité . . . . .	78
4.4.1	Maintenabilité . . . . .	78
4.4.2	Tests . . . . .	79
4.5	Conclusion . . . . .	79

---

<b>5</b>	<b>Validation</b>	<b>81</b>
5.1	Introduction . . . . .	81
5.2	Scénarios . . . . .	81
5.2.1	Situation initiale . . . . .	81
5.2.2	Catalogue utilisé . . . . .	82
5.2.3	Premier scénario - Année académique 2009-2010 . . . . .	83
5.2.4	Second scénario - Année académique 2010-2011 . . . . .	85
5.3	Conclusion . . . . .	86
<b>6</b>	<b>Travaux futurs</b>	<b>89</b>
6.1	Introduction . . . . .	89
6.2	Service mail . . . . .	89
6.3	Gestion des incohérences . . . . .	90
6.4	Mise à jour du graphe . . . . .	90
6.5	Intégrer le logiciel de graphe dans l'application . . . . .	90
6.6	Amélioration de l'interface . . . . .	91
6.7	Fonctionnalités diverses . . . . .	91
6.8	Conclusion . . . . .	91
<b>7</b>	<b>Conclusion</b>	<b>93</b>

<b>FWB</b>	<b>F</b> édération <b>W</b> allonie - <b>B</b> ruelles
<b>Commission INFO</b>	Commission de Programme du département d'informatique de l'école polytechnique de Louvain -là-Neuve
<b>Module</b>	Ensemble de cours, obligatoire ou non Ex : L'option intelligence artificielle dans le programme de master
<b>EPC</b>	<b>E</b> tudiant <b>P</b> rogramme <b>C</b> ours Outil de gestion du parcours étudiant de l'UCL
<b>Rails</b>	Ruby On Rails

# Chapitre 1

## Introduction

### 1.1 Contexte

Chaque année à l'Université catholique de Louvain, et dans toutes les autres universités, un étudiant est amené à devoir effectuer des choix au niveau du programme de cours qu'il va suivre. Même si ces choix demeurent plus restreints en bac qu'en master, le processus actuel représente une lourde tâche de travail pour le personnel en charge de la vérification de ces programmes et risque de s'amplifier encore plus avec le nouveau décret.

En bachelier, cela se limite au choix d'une mineure de manière générale. La tâche est déjà plus compliquée lorsqu'il faut traiter le programme d'un étudiant qui recommence son année.

En master par contre, l'ensemble des choix est plus vaste. Le catalogue des programmes est plus versatile (Master 120, Master 60) et plus modulable. En effet, un étudiant doit choisir une ou plusieurs options. De plus, chacune d'entre elles est composée de cours obligatoires et optionnels. Ensuite, pour les programmes étalés sur deux ans, chacun des cours peut être suivi durant la première ou la deuxième année. Enfin, il faut prendre en compte les diverses équivalences lorsqu'un étudiant de notre faculté part en Erasmus, ou lorsqu'un étudiant étranger vient étudier dans notre université. Le processus de validation est donc plus complexe et nécessite considérablement plus de temps pour chacune des deux parties, à savoir la commission des programmes en informatique de l'École Polytechnique de Louvain - (**la commission INFO**) - et ses étudiants.

En outre, la Fédération Wallonie - Bruxelles (*FWB*) a voté un décret qui a un impact

considérable sur comment, quand les étudiants peuvent et doivent créer leur programme et choisir leur cours. Le but de cette réforme est de proposer des programmes plus à la carte, afin que la structure des formations enseignées chez nous se calque sur celle des grandes universités anglo-saxonnes.

Cette réforme a pour conséquence de complexifier la gestion des programmes de master, mais surtout ceux de bachelier, offrant plus de liberté aux étudiants. Alors que le gros du travail se fait actuellement à l'aide d'un formulaire papier, il est urgent de passer à une version automatique pour simplifier la tâche au personnel et aux étudiants.



## 1.2 Problème

Le problème est, une fois découpé, relativement simple à comprendre. Nous avons deux acteurs; *la commission INFO* et les étudiants. La *commission INFO* propose un catalogue de cours aux étudiants et plusieurs types de contraintes s'exercent sur chacun des cours de ce catalogue. L'étudiant doit se construire un programme de cours à partir du catalogue et *la commission INFO* doit vérifier l'intégrité du programme de chaque étudiant. Il y a donc un processus de *négociation* entre les deux parties, durant lequel *la commission INFO* souligne les erreurs éventuelles pour être, par après, corrigées par l'étudiant jusqu'à la validation de son programme.

Cependant, il faut garder à l'esprit que l'état du catalogue, des contraintes ou même des cours peut évoluer à tout moment. Comme mentionné plus haut, la *FWB* peut émettre de nouvelles règles, de nouvelles lois dont l'impact peut bouleverser de façon relativement importante la structure et l'organisation des programmes. De plus, il est fréquent de voir certaines contraintes, comme les crédits d'un cours, le semestre ou même le cycle durant lequel il est dispensé, changer au cours des années.

L'encodage, l'interprétation, la connaissance et la vérification de l'ensemble de ces contraintes, qui ne sont parfois pas très explicites, sont à la charge de *la commission INFO*.

L'image 1.1 représente une vue de l'ensemble des cours, modules et contraintes du programme de **master** en informatique. Les nœuds représentent les différents cours, modules et programmes disponibles, tandis que les liens entre ces différents nœuds correspondent aux dépendances entre les cours. La complexité du graphe, malgré qu'il a été retravaillé et simplifié pour augmenter sa lisibilité, est assez évidente.

Présentement, le processus de construction du programme de cours pour un étudiant est assez rudimentaire. Ce processus est représenté sur le diagramme 1.2

*La commission INFO* commence par mettre à jour ses différents programmes de cours, puis crée le formulaire excel contenant le programme de cours et enfin le met en ligne sur le portail de la faculté (afin qu'il soit téléchargeable par les étudiants par après).

L'étudiant télécharge le formulaire contenant le programme de cours et le remplit.

Ici commence la phase de négociation entre *la commission INFO* et les étudiants. Les deux parties vont s'échanger un formulaire que l'étudiant va compléter et *la commission*

FIGURE 1.1 – Programme de Master

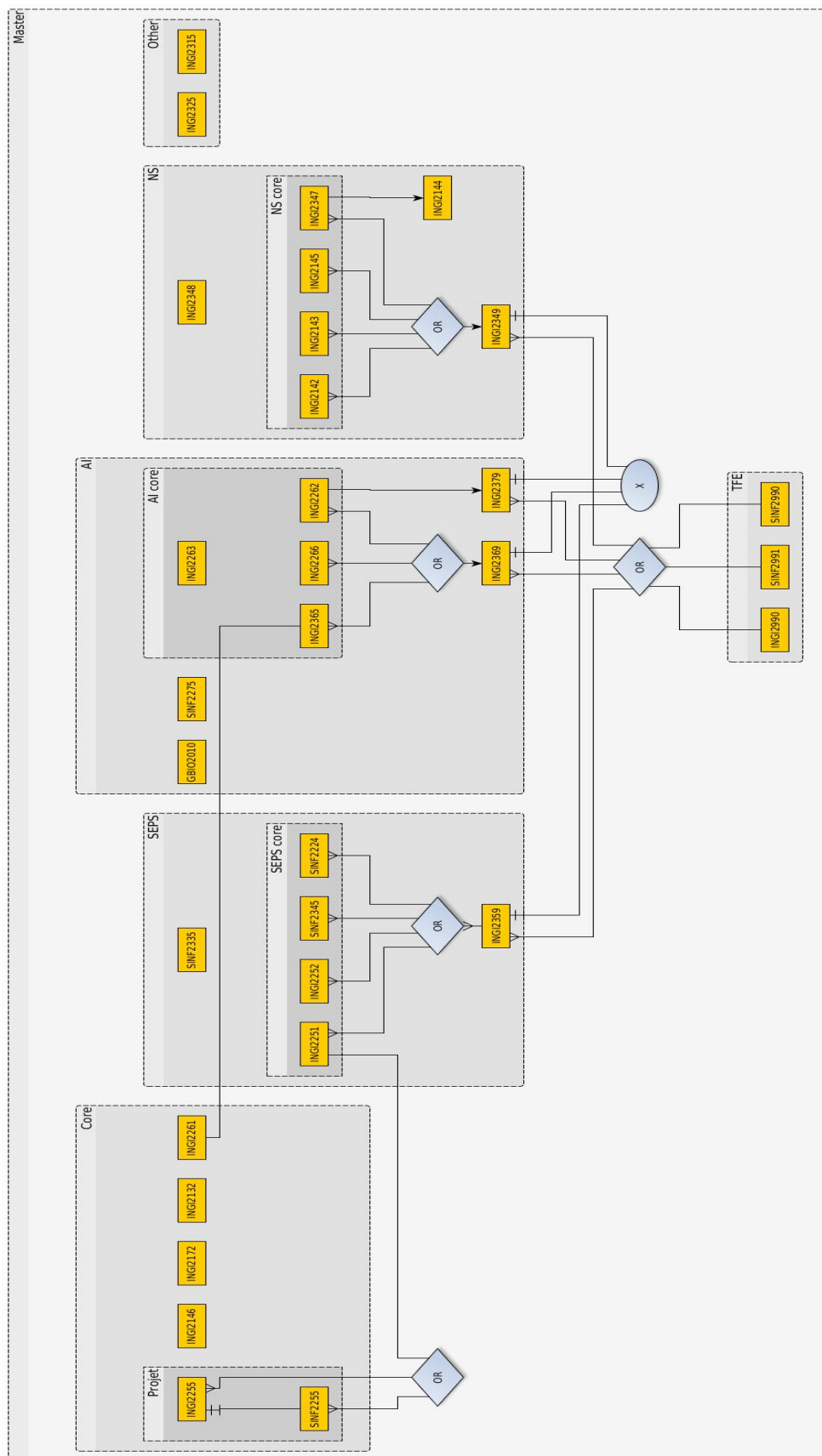
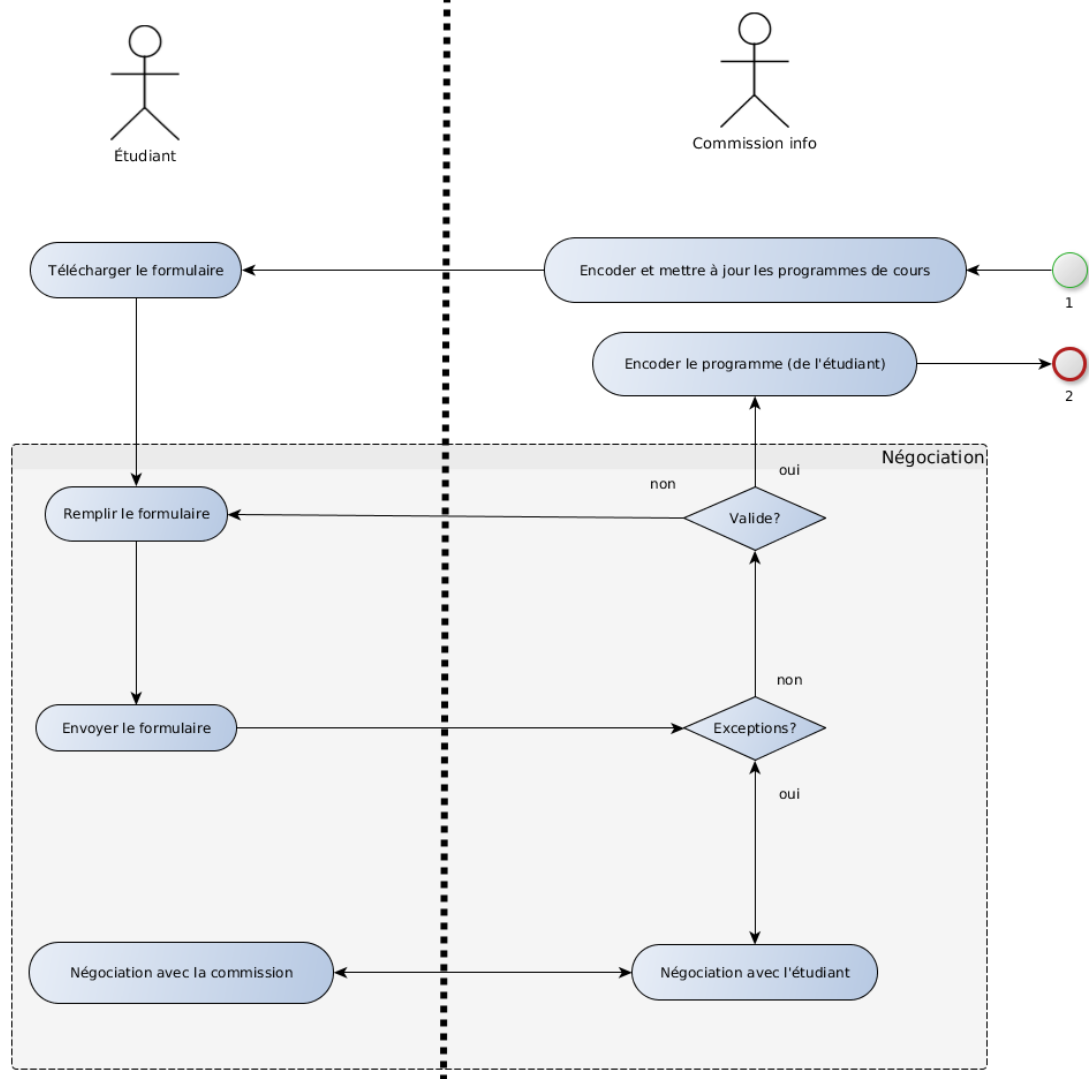


FIGURE 1.2 – Processus actuel



*INFO* vérifier. L'échange va se poursuivre jusqu'à ce que le programme soit valide.

L'étudiant complète d'abord son formulaire en tenant compte des spécifications du programme. S'il est en bac par exemple, il y a des cours obligatoires à prendre chaque année. S'il est en master, il y a une option à choisir, des cours obligatoires, etc. Il peut cependant exister des exceptions dans les différents programmes qui permettent d'enfreindre ces spécifications. Par exemple, un étudiant en provenance d'une autre université ou d'une autre faculté peut avoir déjà suivis l'équivalent d'un cours obligatoire dans son université ou sa faculté d'origine. Il va devoir rendre un formulaire qui ne respecte pas les contraintes initiales, justifier ces exceptions et négocier leur acceptation avec *la commission INFO*.

Il envoie son formulaire par mail à *la commission INFO*.

*La commission INFO* réceptionne le formulaire. Elle vérifie la validité du programme. Si l'étudiant revendique des exceptions dans son programme, elle vérifie si elle sont fondées ou non. Si le programme ou ces exceptions ne sont pas validées, *la commission INFO* peut demander de plus amples informations à l'étudiant ou négocier avec lui et, si nécessaire, il est demandé à l'étudiant de compléter son formulaire avec les informations qui manquent. Le processus de validation recommence ainsi à zéro. Si le programme est valide, il est encodé.

Le processus actuel, que ce soit en bac ou en master, se déroule essentiellement à l'aide d'une feuille de papier et les négociations par mail ou oralement. L'étudiant recherche les informations dont il a besoin sur le site de l'UCL, auprès de ses collègues ou encore sur les forums de cours, complète le formulaire et le dépose au secrétariat.

*La commission INFO* quant à elle, effectue une vérification à la main de ces formulaires, négocie oralement ou par mail avec les étudiants si besoin est. Informatiser ce processus à l'air du web 3.0 est une nécessité plus qu'absolue.

## 1.3 Motivation

Reprenons le diagramme 1.2 représentant le processus de création de programme tel qu'il se déroule actuellement.

Le mot d'ordre est **automatisation**.

Comme présenté dans la section précédente, certaines étapes sont sources de beaucoup de problèmes pour *la commission INFO*.

**Encodage/Mise à jour des programmes de cours** Cette étape se fait manuellement.

De plus, la *commission INFO* doit, une fois le programme encodé, compléter à la main le formulaire excel qui va être utilisé par les étudiants.

**Vérification des programmes des étudiants** Cette étape se fait manuellement.

**Négociation avec les étudiants** Cette étape se fait oralement ou par mails.

Avant toute chose, il y a beaucoup d'informations qui sont échangées entre les deux parties, que ce soit implicitement ou explicitement.

Les données implicites correspondent aux informations relatives aux étudiants, comme l'historique de leur parcours universitaire, qui pourrait par exemple justifier certaines des exceptions mentionnées précédemment. Ces données ne concernent pas directement les choix faits par les étudiants au niveau de leur programme de cours, mais sont plutôt des méta-informations qui complètent leur profil.

Les données explicites correspondent aux choix faits par les étudiants. Ces données concernent les

- les choix faits par les étudiants au niveau de leur programme de cours ;
- les justifications des étudiants à propos des exceptions de leur programme par rapports aux règles et structures fixées par le programme de cours suivi.

Il faut aussi garder une trace de ce que l'étudiant a suivi et réussi les années précédentes. En effet, il est nécessaire de savoir quel cours un étudiant a réussi l'année précédente pour attribuer les différentes dispenses lorsqu'il recommence son année par exemple.

Dès lors, la *commission INFO* doit pouvoir valider le programme d'un étudiant lorsque celui-ci l'a réussi.

Il est donc indispensable d'inclure dans la solution une base de données pour y stocker toutes ces informations, afin qu'elles soient à disposition des deux parties à tout moment.

Différents points du processus actuel 1.2 doivent être automatisés.

Le premier point identifié se situe au niveau du support utilisé par l'étudiant et *la commission INFO*, le formulaire excel, pour ajouter de l'information sur le programme de

cours. Tout d’abord, *la commission INFO* doit générer manuellement ce formulaire, en y incluant les cours et différentes options du programme de cours en question. Ajouter les différents blocs, cours et leur crédits respectifs est assez contraignant sur un simple tableur. De plus, beaucoup d’erreurs peuvent être laissées par l’étudiant lorsqu’il complète celui-ci. Certes, il y a certains moyen à disposition sous Excel (en utilisant des macros), pour vérifier certaines contraintes du programmes (comme le nombre de crédits d’un module par exemple) mais celles-ci peuvent être ignorées par l’étudiant, et doivent de toute façon être revérifiées par après par *la commission INFO*.

**Le premier point** de la solution proposée est donc **d’offrir une plate-forme permettant aux acteurs d’échanger ces différentes informations.**

Deuxièmement, l’étape de complétion du formulaire par l’étudiant doit être améliorée. Il n’est pas possible de mettre des informations concernant les contraintes autres que numéraires dans le formulaire excel utilisé pour le moment. La plate-forme doit donc inclure des vues représentant de façon claires et concises les différentes contraintes des programmes de cours. De plus, ces **contraintes doivent être aisément encodables, gérables et modifiables.**

Troisièmement, l’étape de vérification des contraintes, pour valider un programme d’étudiant est coûteuse en temps pour *la commission INFO*, alors que cela ne prendrait que quelques secondes pour un ordinateur. Un module pour vérifier ces contraintes doit être inclus dans la plate-forme

Quatrièmement, il faut s’attaquer au processus de **négociation** qui amène l’étudiant, en relation avec *la commission INFO*, à construire un programme valide. Tant que le programme de l’étudiant n’est pas valide, l’étudiant doit corriger son programme en tenant compte du feedback de *la commission INFO*. *La commission INFO* doit contacter l’étudiant en pointant les parties non correctes de son programme et l’étudiant doit à son tour comprendre les requêtes de *la commission INFO*, puis tenter de les résoudre ou de les corriger. Par mail ou par papier, cela peut être très long.

## 1.4 Objectifs

De manière générale, le but de cette application est d'automatiser la gestion des programmes de cours.

Le *pré-objectif* est d'être indépendant de la base de données EPC. Pour des raisons institutionnelles, l'équipe EPC est surchargée en plus d'être réticente à donner un accès aux données. Dans le futur, peut-être l'outil pourrait **échanger des données avec EPC ou même être intégré**, mais ce ne sera pas pour tout de suite. C'est pourquoi il faut pouvoir importer les données de façon efficace et intuitive.

La solution doit être maintenable et évolutive. En effet, la structure des programmes est en constante évolution. De plus, il est probable que la commission de programme découvre de nouveaux besoins qui devront être implémentés à l'avenir. Il est donc primordial de structurer l'application intelligemment pour que celle-ci soit modulaire et **qu'on ne doive pas repartir de zéro lors de développement ultérieurs**.

La *commission INFO* doit pouvoir apporter des catalogues de cours sur l'application. Un catalogue de cours est un ensemble de programmes de cours, contenant les différents modules, cours et dépendances. Un programme de cours est un cursus qu'il est possible de suivre dans la faculté, comme par exemple le programme de MASTER destinés aux SINFs (SINF2M), le programme de passerelle (SINF1PM) ou encore celui de BAC destinés aux ingénieurs civils (FSA1BA).

Les informations des programmes de cours doivent pouvoir être téléchargées depuis l'application ainsi qu'être mises à jour. En effet, les programmes de cours sont sujets régulièrement à des changements. Il est donc nécessaire que ces données ne soient pas ajoutées en "*en dur*" dans l'application. De plus, ces données doivent être visibles de manière synthétique par la commission (vue *admin*).

Il doit y avoir un historique des différentes versions des programmes de cours mis en ligne par la *commission INFO* tout au long des années académiques, pour gérer l'évolution de ceux-ci et pouvoir permettre aux étudiants (à qui cela est permis) de choisir dans leurs programmes des cours d'anciens catalogues, en cas de report de note par exemple.

Les étudiants doivent pouvoir construire leur programme, en choisissant les cours et les modules qu'ils vont suivre chaque année. L'application doit leur dire si leur programme est cohérent ou non.

Au niveau de ces contraintes, il doit y avoir une certaine souplesse. Il n'est pas possible d'avoir une vision *manichéenne* à ce niveau.

Il doit être possible aux étudiants d'attirer l'attention sur certaines parties de leur programme en y ajoutant un commentaire pour poser une question, ou pour justifier un choix.

En tant qu'étudiant il doit être possible de

- se créer un compte utilisateur avec son adresse mail UCL ;
- sélectionner la version du catalogue de cours avec laquelle il va travailler ;
- se créer un programme en choisissant un des programmes de cours disponibles à suivre ;
- choisir les différents modules de cours à suivre, option, tronc commun, ... ;
- avoir une vue sur les différentes années du programme qu'il suit (les deux années de MASTER par exemple) ;
- configurer son programme par année académique, en choisissant les cours que l'on va suivre durant les différents semestres ;
- voir les contraintes qui ne sont pas respectées, par exemple, le nombre de crédits manquants pour valider un module, ou encore les dépendances d'un cours ;
- pouvoir soumettre à la validation son programme, même s'il ne respecte pas toutes les contraintes ;
- pouvoir communiquer avec la commission info à travers l'application ; par exemple justifier une contrainte non respectée par écrit. ("J'ai déjà suivi un cours très semblable durant mon cursus dans la faculté X à l'université Y").

La *commission INFO* doit pouvoir :

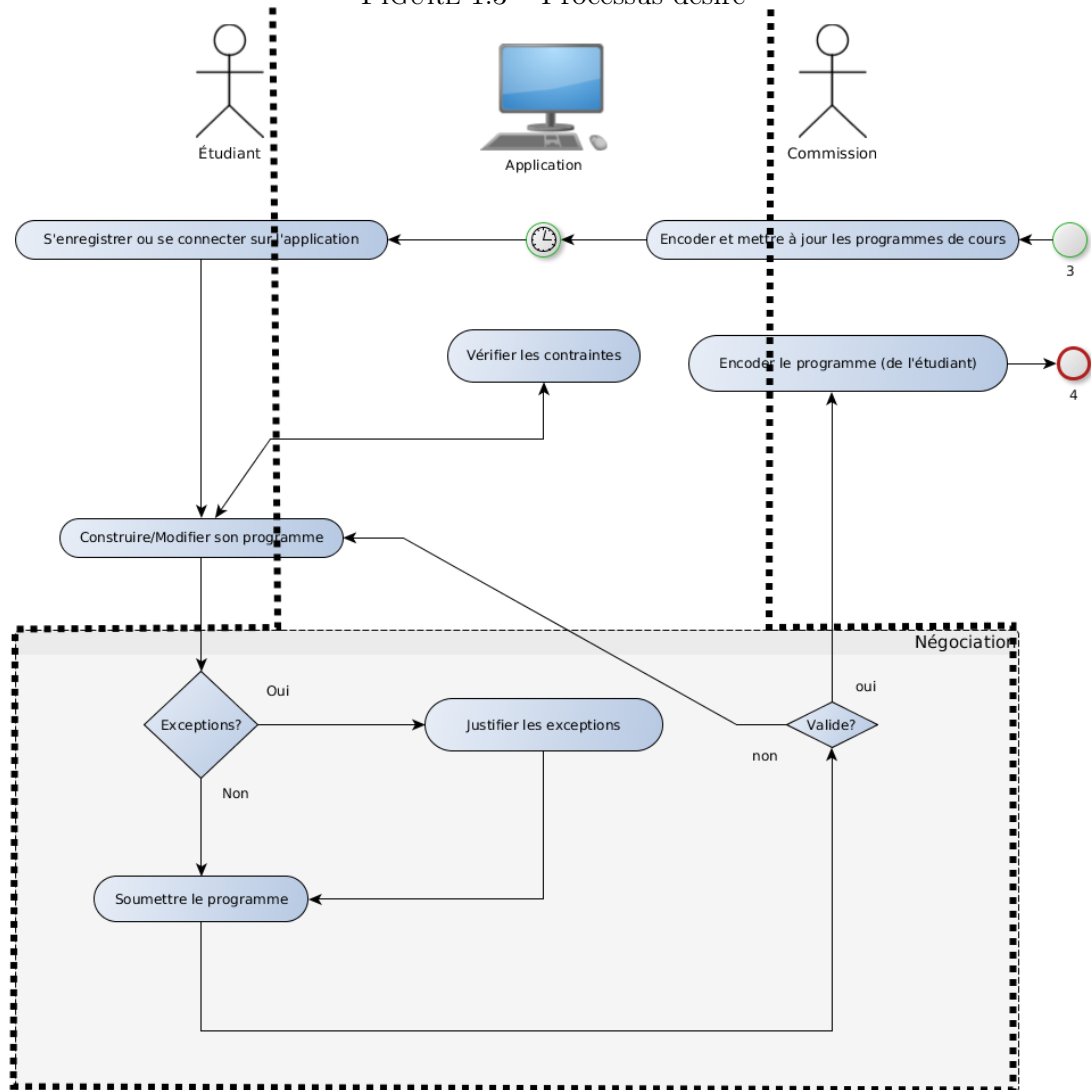
- vérifier, de la façon la plus automatisée possible, le programme d'un étudiant ;
- importer les différents programmes de cours dans l'application ;
- mettre à jour les données relatives à ces programmes ;
- être notifié lorsqu'un étudiant envoie son programme à la validation ;
- accéder aux programmes des étudiants ;
- communiquer avec les étudiants à travers l'application ;
- marquer les années précédentes des étudiants comme réussies ou ratées.

Le processus auquel nous désirons arriver est illustré sur l'image 1.3 ;

La *commission de programme* importe et met à jour les différents programmes à l'aide d'un fichier yEd ou Excel externe qui est ensuite importé dans l'application



FIGURE 1.3 – Processus désiré



L'étudiant se connecte à l'application, crée son programme de cours, et le configure pas à pas en réduisant au maximum le nombre de contraintes non vérifiées. Pour les contraintes non vérifiées restantes, il insère en commentaire les justifications à ces exceptions.

L'étudiant envoie son programme à la validation.

**Négociation** - La *commission de programme* récupère la requête de validation. L'application lui montre les contraintes qui ne sont pas respectées. La *commission de programme* regarde ensuite les justifications de l'étudiant. Si elles ne sont pas suffisantes, la *commission INFO* refuse la demande de validation et commente sa validation en expliquant pourquoi certaines justifications ne sont pas suffisantes.

Le processus de négociation (la boîte en gris intitulée *négociation* sur le schéma 1.3) se répète jusqu'à ce que le programme de l'étudiant soit valide. S'il n'y a pas d'exceptions, l'application peut immédiatement vérifier si le programme de l'étudiant est valide et le valider (si c'est le cas) ou demander de modifier son programme (s'il n'est pas valide).

## Chapitre 2

# Énoncé du problème

La gestion des programme de cours n'est pas une chose aisée. La situation est complexe essentiellement pour deux raisons.

1. La *commission INFO* s'occupe d'un nombre assez élevé de programmes ;
2. il existe des contraintes de différentes sortes qui restreignent les étudiants dans les choix qu'ils peuvent faire lorsqu'ils configurent leur programme de cour ;
3. les programmes évoluent souvent.

Ces trois points vont être présentés en détail dans les sections qui suivent.

### 2.1 Programmes proposés

La liste des programmes proposés par la *commission INFO* est la suivante :

**Bachelier en sciences informatiques - SINF1BA** [de Louvain(2013-2014a)] - C'est un programme de 180 crédits. Comme dans tout programme de bachelier à l'UCL, l'étudiant est amené à devoir choisir une mineure dans ce programme. Par exemple, une mineure intitulée *Approfondissement en sciences informatiques* est disponible pour les étudiants qui suivent ce programme. La durée normale de ce programme de bachelier est de trois ans.

**Bachelier en sciences de l'ingénieur, orientation ingénieur civil - FSA1BA (Majeure ou Mineure en informatique)** - [de Louvain(2013-2014b)]. Ici il n'est pas

question du programme FSA1BA dans son entièreté mais de la majeure ou mineure que l'étudiant en sciences de l'ingénieur est amené à choisir lorsqu'il suit ce programme.

**Master [120] en sciences informatiques - SIN2M** [de Louvain(2013-2014c)] -

Ce programme de 120 crédits est destiné aux étudiants en provenance du programme SIN1BA. Il comporte un module obligatoire (le tronc commun) et la possibilité de choisir un ou plusieurs modules optionnels (Génie logiciel, systèmes de programmation, intelligence artificielle, réseaux et sécurité). La charge du travail de fin d'étude est de 28 crédits. La durée normale de ce programme de master est de deux ans.

**Master [60] en sciences informatiques - SIN2M1** [de Louvain(2013-2014d)] -

Ce programme alternatif de 60 crédits est destiné aux étudiants en provenance du programme SIN1BA. Il comporte un module obligatoire (le tronc commun) et la possibilité de choisir quelques cours au choix mais pas d'options. La charge du travail de fin d'étude est plus petite que celle de son homologue SIN2M : 18 crédits. La durée normale de ce programme de master est d'un an.

**Master [120] : ingénieur civil en informatique - INFO2M** [de Louvain(2013-2014e)]

- Ce programme est destiné aux étudiants en provenance du programme FSA1BA ayant suivi soit la mineure soit la majeure en informatique. Comme dans le master SIN2M, il comporte un module obligatoire (le tronc commun) ainsi que la possibilité de choisir un ou plusieurs modules optionnels (Génie logiciel, systèmes de programmation, intelligence artificielle, réseaux et sécurité). La charge du travail de fin d'étude est de 28 crédits. La durée de ce programme de master est de deux ans.

**Année d'études préparatoire au master en sciences informatiques - SIN1PM**

[de Louvain(2013-2014f)] - Ce programme est destiné aux étudiants en provenance de Hautes écoles d'informatique. Il permet d'accéder aux programmes SIN2M et SIN2M1. C'est un programme *à la carte* qui dépend du *background* de l'étudiant (Dans la plupart des cas, c'est un programme standard qui est proposé). Les cours sont choisis parmi ceux proposés dans le programme SIN1BA. Ce programme affiche entre 46 et 60 crédits. La durée normale de ce programme est d'un an.

Outre ces programmes, la *commission INFO* doit s'occuper du cas des étudiants en programme d'échange. La principale difficulté, que cela soit un étudiant immigrant ou émigrant, est de trouver des équivalences entre les cours suivis dans l'université d'origine et ceux proposés à l'UCL ou l'inverse.

De plus, les intersections entre ces cours sont nombreuses. Beaucoup de cours sont disponibles pour une partie voir la totalité des programmes cités ci-dessus.

## 2.2 Contraintes

Les contraintes sont un point important du problème. En plus d'être nombreuses et diversifiées, elles requièrent beaucoup de travail au niveau de leur vérification du côté de *la commission INFO*. En outre, elles sont difficiles à exprimer (et vérifier) avec les moyens (formulaire excel, présentation en début d'année, portail du département) mis à la disposition de l'équipe. La conséquence directe de ceci est qu'il est difficile pour les étudiants de comprendre le pourquoi du comment de ces contraintes. Ils n'en tiennent donc pas compte à 100% lorsqu'ils construisent leur programme de cours.

Voici une liste qui en présente brièvement les différentes sortes que l'on peut rencontrer.

1. **Dépendances entre les différents cours** - Ces contraintes sont de deux types :

- (a) Les prérequis : Les prérequis d'un cours sont les cours qu'il faut avoir réussis (dans des années académiques antérieures) afin de pouvoir suivre ce cours
- (b) Les corequis : Les corequis d'un cours sont les cours qu'il faut avoir suivis **au plus tard** durant la même année académique que ce cours.

L'image 2.1 illustre ce type de contrainte. On peut voir que le cours *INGI2365* a pour prérequis le cours *SINF1121* et pour corequis le cours *INGI2261*. Il est donc nécessaire d'avoir **validé** *SINF1121* ainsi que de suivre (au plus tard) **durant la même période** le cours *INGI2261* (s'il n'a pas été suivi précédemment) pour avoir accès à *INGI2365*. Ces deux contraintes sont directionnelles ! le cours *SINF1121* n'a pas pour corequis le cours *INGI2365*, de même que le cours *INGI2261* n'a pas pour prérequis le cours *INGI2365*.

Notez que l'arrête qui relie le cours *SINF1121* au cours *INGI2365* est interprétée comme directionnelle dans le logiciel.

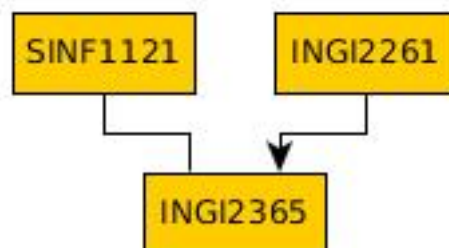


FIGURE 2.1 – Dépendances du cours INGI2261

2. **Contraintes induites par les programmes** - Ce sont les différents cours ou ensembles de cours qu'il est obligatoire de suivre avant de valider un programme. En master, il y a, par exemple, le module intitulé *Tronc Commun* qu'il est obligatoire de suivre, ainsi que le mémoire. Certains modules optionnels, comme les options de master, sont constitués de sous-modules dont il est obligatoire de suivre la totalité des cours qu'ils contiennent.
3. **Contraintes temporelles** - Ce sont les contraintes les plus basiques. Elles représentent la période de temps durant laquelle il est possible de suivre le cours en question. Initialement, elles sont exprimées en terme de semestre. Pour des raisons variables, comme un professeur qui part à la retraite, ou qui prend une année sabbatique, elles peuvent très bien s'exprimer en terme d'années académiques. Un autre exemple sont les cours bisannuels qui sont des cours se donnant une fois tous les deux ans.
4. **Contraintes sur les propriétés** - Ces contraintes portent sur les propriétés des cours, programmes ou modules. Principalement, elles portent sur les crédits minimum et maximum d'un programme ou d'un module.

## 2.3 Conclusion

L'objectif est de développer une application pour que la charge des différents problèmes présentés plus haut (la gestion des contraintes, le nombre des programmes proposés et leur complexité) soit à la charge d'une machine. L'idée est de pouvoir :

- importer et mettre à jour les données relatives aux catalogues de cours ;
- permettre aux étudiants de conserver un historique des cours et programmes qu'ils ont déjà suivis au cours des années précédentes (et des catalogues d'où proviennent ces cours) ;
- visualiser ces données, que l'on soit étudiant ou membre de la commission de programme ;
- effectuer une vérification immédiate de la validité des programmes de cours créés par les étudiants ;
- permettre à la commission de communiquer avec les étudiants via l'application, pour par exemple attirer l'attention de l'étudiant sur une partie de son programme

qui ne semble pas cohérente, ou dans l'autre sens, demander ou donner des explications à la commission sur certains points ;

- porter l'application aisément en production (via le "*cloud*" par exemple).

De manière plus générale, le but de ce mémoire est de développer une application conviviale, maintenable et évolutive afin qu'elle puisse être utilisée par les étudiants et la commission de programme.

Ce mémoire sera typiquement un projet que pourrait rencontrer un informaticien dans la vie active, où la *commission INFO* joue le rôle de client, et le promoteur celui de chef de projet.



## Chapitre 3

# Présentation du système

### 3.1 Introduction

Ce chapitre vise, à l'aide d'un exemple concret d'utilisation, à :

1. introduire les technologies utilisées pour développer l'application ;
2. expliquer les différentes fonctionnalités de l'application d'un point de vue utilisateur. Les utilisateurs étant de deux types, la section relative aux fonctionnalités sera divisées en deux parties. Dans un premier temps, nous expliqueront les fonctionnalités offertes à la commission de programme à savoir ;
  - l'import de données dans l'application (Programmes de cours, modules, cours) ;
  - la gestion des données (Accéder au différents cours, modules et programmes, mettre à jour leurs données) ;
  - la gestion des contraintes ;
  - comment vérifier les programmes de cours des étudiants.

Ensuite, nous présenterons ce qu'il est possible de faire en tant qu'étudiant avec l'application à savoir :

- se créer un compte utilisateur ;
- se connecter avec son compte utilisateur ;
- se créer (ou modifier) un programme de cours, en choisissant le programme à suivre ;
- configurer son programme de cours, en configurant les différentes années qui le composent et en choisissant les modules de cours à suivre ;
- vérifier la validité de son programme ;

- négocier avec la commission de programme la justification des exceptions.

Un des principaux objectifs de ce mémoire étant d’offrir une solution maintenable et évolutive, l’application est divisée en trois parties distinctes, indépendantes entre elles.

Ces trois parties sont les suivantes :

- l’import et la mise à jour des catalogues, de programmes, modules, cours et des différentes contraintes par la *commission de programme* ;
- la création du programme par l’étudiant, via l’interface utilisateur ;
- la vérification (par l’application) des différentes contraintes induites par le programme créé précédemment par l’étudiant.

### 3.1.1 Exemple d’utilisation

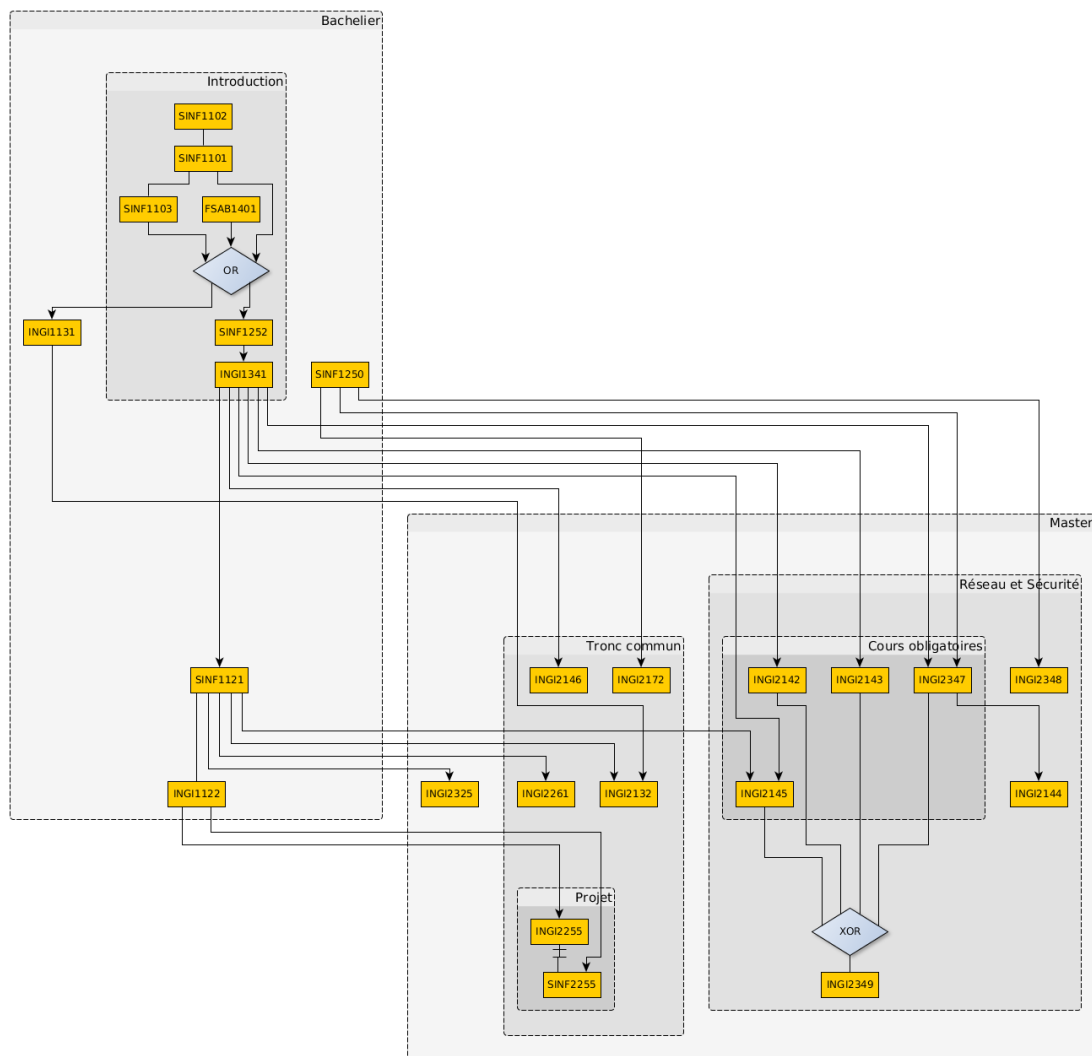
Tout au long de ce chapitre, l’exemple suivant va être utilisé pour illustrer le fonctionnement de l’application. Il s’agit d’un catalogue de cours fictif proposant deux programmes de cours. Ce programme fictif se rapproche très fort de la réalité. Certaines parties du catalogue (certains modules optionnels du programme de master, certains cours) ont été élaguées pour rendre l’exemple plus clair. L’exemple reste néanmoins représentatif de la réalité, il comporte en effet les différents types de contraintes que l’on peut rencontrer par exemple.

Ce catalogue est composé de deux programmes de cours.

Un programme de **Bachelier** (le grand rectangle à droite de la figure 3.1) composé d’un module obligatoire (Introduction, la petite boîte imbriquée s’appelant *Intro*)

Un programme de **Master** (le grand rectangle à gauche de la figure 3.1) composé d’un module optionnel (l’option réseaux et sécurité, la boîte imbriquée à gauche) et d’un module obligatoire (Le tronc commun, la boîte imbriquée à droite). De plus, certains cours de master dépendent de cours de bac.

FIGURE 3.1 – Exemple fictif de catalogue de cours



## 3.2 Technologies utilisées

### 3.2.1 Introduction

Le but poursuivi par cette section est de présenter les différents choix faits aux niveaux des technologies utilisées par l'application. Ces choix sont de deux types. Premièrement, les technologies utilisées pour construire l'application seront présentées, comme le framework ou la base de données qui est utilisée. Les technologies externes à l'application qui sont utilisées pour construire les différents curricula ou mettre à jour leur informations seront présentées par la suite.

On parle de technologie interne pour représenter celles qui sont utilisées pour développer les fonctionnalités de l'application. Les technologies externes sont quant à elles des applications déjà existantes qu'il faut utiliser **en dehors** de l'application et pour lesquelles il ne faut pas écrire de lignes de code.

### 3.2.2 Ruby on Rails

#### Introduction

Cette section a pour but de présenter la technologie principale utilisée pour développer l'application. Le but n'est pas de parcourir en détails le fonctionnement de Rails, mais bien d'en présenter les concepts clés. En effet, il semble important d'en comprendre les grandes lignes, car son architecture, aussi bien que ses principes ont influencé la structure de la solution.

#### Le choix d'un framework

A première vue, l'utilisation d'un framework n'est pas absolument nécessaire. Cependant, un framework apporte toute une collection d'outils qui aident à développer mieux et plus rapidement.

**Mieux** car il permet de développer une application qui est structurée, ce qui rend le code plus maintenable et évolutif.

**Plus rapide** car il permet de gagner du temps en réutilisant des modules génériques afin de se concentrer sur d'autres domaines. Avec un framework, on assemble des briques plutôt que de réinventer la roue.

Enfin, le dernier atout d'un framework se situe au niveau de l'intégration de nouveaux développeurs sur le projet. Dans le cadre de ce mémoire, il est clair que de nouvelles fonctionnalités devront être ajoutées dans le futur. De plus, les fonctionnalités existantes devront peut être modifiées ou améliorées. Il sera plus facile pour cette personne de se plonger dans du code qui n'est pas le sien, s'il a une structure propre aux standard web d'aujourd'hui.

**Il est donc fortement conseillé d'utiliser un framework web pour créer ce genre d'application.**

Il en existe une multitude aujourd'hui. Il y a tout d'abord les frameworks PHP comme CakePHP, DRUPAL ou Symfony (pour ne citer que les plus connus). Viennent ensuite Ruby on Rails, un framework en ruby et Django, un framework en Python. Cette liste n'est bien entendu pas exhaustive.

### **Le choix de Ruby on Rails**

L'intérêt réside dans le niveau de productivité et de maintenabilité accru que l'on obtient en travaillant avec le framework. Les design patterns sous-jacents, et la philosophie de Rails permettent de concentrer son travail sur les fonctionnalités de l'application plutôt que de passer son temps à écrire du code répétitif ou remplir des fichiers de configuration.

De plus, il existe une multitude de librairies tierces appelées *ruby-gem* qui réduisent encore le nombre de lignes de code à produire, en apportant des fonctionnalités à l'application. Le meilleur exemple est **devise**, une librairie qui permet d'ajouter la gestion de l'utilisateur (création, connexion, récupération de mot de passe, envoi de mails, etc ..) en quelques lignes en plus de gérer les sessions et les accès aux différentes actions et vues.

Rails pousse aux bonnes pratiques, c'est d'ailleurs cette philosophie qui m'a incité à développer la plupart des fonctionnalités, comme vous le verrez plus tard, dans des librairies externes à l'application.

En outre, Rails dispose d'une communauté très active et passionnée, qui teste, documente et améliore les fonctionnalités du framework.

Les limites du framework sont les suivantes

- Lorsque l'on débute, on est souvent tenté de charger les modèles en voulant suivre la philosophie *tiny view - skinny controller - fat model*<sup>1</sup> et l'on oublie souvent qu'il est possible de déléguer la plupart des fonctionnalités à des bibliothèques externes, qui sont plus faciles à développer - car créées en pure *ruby* - et plus facile à tester - car indépendantes de rails [Helmkamp(2012)].
- Un des principaux aprioris sur les frameworks et particulièrement Ruby on Rails, est qu'il sont lents (et peu efficace). En effet, ruby est un langage interprété . Ce type de langage tend à être plus lent que les langages compilés. Cependant, il faut garder à l'esprit qu'écrire du code, trouver et corriger des bugs ou encore ajouter des nouvelles fonctionnalités dans une application sont des tâches encore plus coûteuses en temps. Rails permet de réduire le temps consacré à ces tâches, ce qui est bien plus important lorsque l'on développe ou maintient une application, particulièrement dans le cadre d'un mémoire. Ainsi, à la place de configurer le mapping entre ces différentes ressources, on utilise une convention.

### Conclusion

Pour toutes les raisons présentées ci-dessus (bibliothèques externes disponibles, communauté, etc), le choix s'est naturellement porté vers Ruby on Rails. En outre, j'ai accumulé pas mal d'expérience en utilisant ce framework dans le cadre d'un stage et en réalisant des petits projets ces dernières années.

C'est un framework open-source, utilisé pour développer des applications web. Le développement se fait à travers le langage de programmation multi-paradigmes (Programmation fonctionnelle, orientée objet, ...) **ruby**. Il se base sur des puissants design patterns et principes qui vont être présentés en quelques lignes ci-dessous.

### DRY - Don't repeat yourself

Comme son nom l'indique, ce premier principe pousse à la réutilisation du code existant le plus souvent que possible, plutôt que d'avoir des bouts de code similaires un peu partout

---

1. Bonne pratique qui consiste à déléguer toute la logique aux modèles

dans l'application. L'idée de tendre vers une structure *Api*, où tout ce qui n'est pas nécessaire aux classes et méthodes externes est caché en interne. Le principal avantage se situe au niveau de la **maintenabilité**. On évite ainsi de devoir partir à la recherche des différents bouts de code dupliqués lorsque l'on veut modifier le comportement d'une méthode, d'une classe, ou même d'un module.

### CoC - Convention over Configuration

L'idée est de réduire au minimum les décisions à prendre avant de commencer à développer. Une convention importante en *Ruby on rails* se situe au niveau des noms des classes pour lesquelles il existe une table correspondante en base de données. Pour un modèle *Course* par exemple, la convention est d'avoir une table nommée *courses* en base de données. Cela permet d'éviter d'avoir à écrire du code supplémentaire pour spécifier à l'application quelle table correspond à quel objet.

Cela permet au développeur de se concentrer sur les parties non conventionnelles de l'application, comme l'architecture, plutôt que de perdre son temps à configurer les objets. L'avantage ici se situe plus au niveau de la **productivité**.

### MVC - Model-Vue-Contrôleur

Le framework s'appuie sur le pattern **MVC**. Destiné aux applications dites *interactives*, il divise l'application en trois parties ; le modèle, les vues et le contrôleur. Notez que *Ruby on Rails* ne respecte pas totalement MVC dans sa conception initiale. Cela se justifie par le fait que ce pattern n'est pas destiné à la base aux applications web, notamment car la vue est ici une page web. Le modèle ne peut donc pas lui envoyer tous les changements qui surviennent au niveau des données. C'est la vue, qui doit expressément faire les requêtes pour ces données, à travers le contrôleur.

MVC à la sauce Rails se présente comme suit. Nous avons

**le modèle** lié à une base de données, qui contient les données et l'état de l'application.

Il contient aussi tout les objets métiers<sup>2</sup>, qui détermine comment l'information est créée, mise à jour, et affichée ;

---

2. Objets qui font tout le travail.

**les vues** qui génèrent l'interface utilisateur et lui présentent les données. Ce composant est passif, il ne traite aucune information. *Vues* est au pluriel ici, car plusieurs vues peuvent avoir accès au même modèle ;

**le contrôleur** qui reçoit les événements du monde extérieur, interagit avec le modèle et choisit la vue à afficher à l'utilisateur. Par exemple, lorsque l'utilisateur veut éditer un commentaire dans un blog, le contrôleur va rendre la vue relative à l'édition de l'objet correspondant.

### Active Record

Ce pattern quant à lui stocke les données dans une base de données relationnelle. Il s'agit simplement de fournir une abstraction supplémentaire à la base de données et fournir des fonctions pour manipuler les données. Dans le cas de rails, il y a donc une couche ruby entre la base de données proprement dite et la logique dans notre modèle. Cela permet par exemple, d'être indépendant du système de base de données utilisée en dessous. Par exemple, *postgresql* est le système de gestion de base de données utilisé pour le moment (source : [Casts(2012)]). Si pour une raison X ou Y, il devient nécessaire de passer à *sqlite3*, il suffit de changer le fichier de configuration *config/database.yml* de

```
development:
  adapter: postgresql
  database: db/development
  pool: 5
  timeout: 5000
```

vers

```
development:
  adapter: sqlite3
  database: db/development
  pool: 5
  timeout: 5000
```

Et de recréer la base de données avec

```
rake db:create:all
rake db:migrate
```

Tout cela est fait sans devoir changer comment sont accédées les données dans les différents modèles.



### 3.2.3 Base de données - PostgreSQL

Rails supporte plusieurs systèmes de gestion de base de données : PostgreSQL, MySQL, SQLite. Le choix du système est cependant restreint à la plateforme utilisée pour héberger l'application (Heroku). En effet, il est nécessaire d'avoir une base de données PostgreSQL pour pouvoir héberger l'application sur Heroku.

### 3.2.4 Éditeur de graphes - yEd

Comme expliqué plus tard dans la section 3.3.2 détaillant comment sont importés les données, le choix s'est porté vers une importation en deux étapes des données dans l'application.

**La première étape** consiste à créer et importer le graphe de cours. Les données importées durant cette étapes sont les nom identifiant les différents objets du graphes (cours, programme, podule), la structure des différents programmes de cours (les cours et modules constituant les programmes de cours) et les dépendances entre les cours.<sup>3</sup>

**La deuxième étape** consiste à ajouter des nouvelles données des différents objets (ou les mettre à jour) à l'aide d'un formulaire excel. Cela permet d'ajouter des propriétés comme les crédits d'un cours ou les crédits minimum requis d'un programme de cours par exemple.<sup>4</sup>

Le but de cette section est d'expliquer les choix qui on été faits aux niveau des technologies utilisées pour s'occuper de **la première étape** (La construction d'un graphe).

Pour construire et importer le graphe de cours, plusieurs alternatives se sont présentées.

La première correspond à un logiciel intégré dans l'application, qui permet de construire explicitement un catalogue de cours sous forme de graphe, en proposant exclusivement de placer des objets cours, modules, ou programmes sur le graphe et en n'offrant que des arrêtes de type corequis ou prérequis. Cette application communiquerait directement avec les modèles et permettrait de générer directement les objets (cours, modules, ...) désirés. Cependant il n'existe pas d'applications réalisant ce genre de graphe pour le moment. Il faudrait donc développer un outil, intégré dans l'application, fournissant ces fonctionnalité. Cependant, cela sort du cadre de ce mémoire, faute de temps.

---

3. Se référer à la section 3.3.3 (relative aux dépendances) pour plus de détails.

4. Se référer à la section 3.3.3 pour plus de détails.

La deuxième alternative serait d'intégrer un outil générant des graphes plus standard dans l'application. YWorks offre une solution, yFiles, qui permet la création et l'édition de graphes en HTML5 et en javascript. Ce logiciel est cependant très chère.

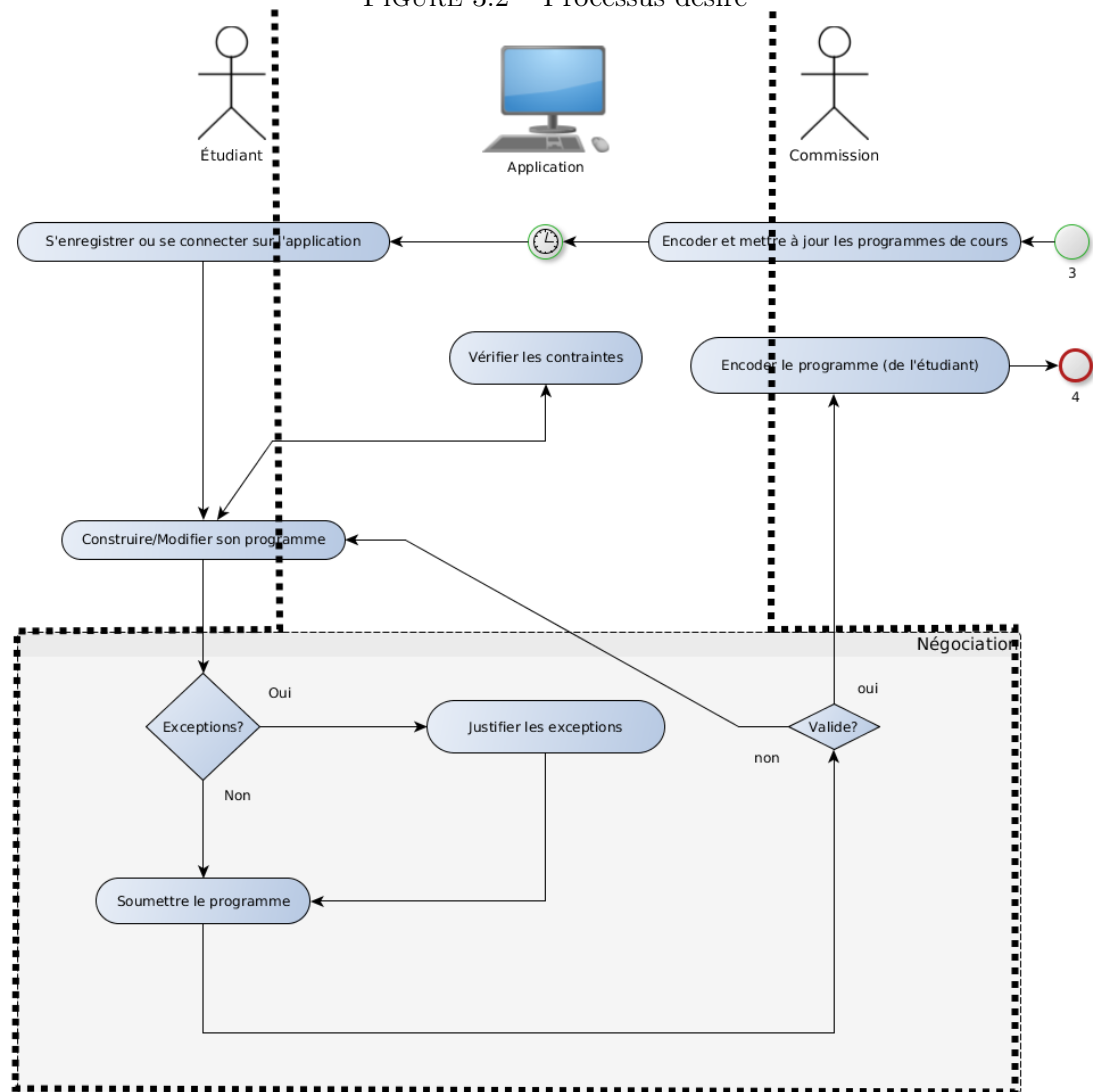
La troisième et dernière alternative serait d'utiliser un logiciel externe à l'application pour générer ces graphes. Ce logiciel, en plus d'être gratuit, doit être multi-plateformes (Mac os x, Linux & Windows) et capable d'exporter dans un format relativement facile à parser. La liste de ce genre de logiciel est assez longue (Dia, Yed, OmniGraffle, Graphviz)

- Dia - C'est un logiciel assez léger qui est capable d'exporter en Xml, un format standard pour représenter des données. Il devient cependant très ennuyeux à utiliser lorsque l'on manipule des graphes de taille importante. C'est cependant un éditeur graphique très générique qui n'est pas seulement destiné à la création de graphes.
- OmniGraffle - Ce logiciel n'est disponible que sur Mac Os X malheureusement.
- Yed - Ce logiciel est assez complet. Il est cross-plateforme et à l'avantage de contenir des algorithmes qui permettent de restructurer automatiquement les graphes. Il permet aussi d'exporter en deux formats de types xml (Graphml & XGml)
- (...)

Étant donné la contrainte de temps imposée par le cadre du mémoire, le choix s'est porté vers un logiciel déjà existant. Il a été préférable de choisir une solution externe pour éviter de surcharger l'application avec de lourds modules graphiques. De plus, intégrer ce genre de logiciels dans l'application ne changeait rien au fait qu'il fasse parser le fichier exporté par l'outil pour importer les données dans l'application.

YEd a été choisi pour toutes ces raisons.

FIGURE 3.2 – Processus désiré



### 3.3 Conception

#### 3.3.1 Introduction

L'application est la plateforme qui tient le rôle d'intermédiaire entre la commission et les étudiants. Elle est représentée par l'entité *Application* sur le diagramme 3.2. Il y a trois modules attachés à cette entité.

**Le module d'import de données** - Il permet à la commission d'importer les données liées aux curricula à l'aide de plusieurs supports (qui seront expliqués en détail plus loin dans le chapitre). Ce module permet tout d'abord d'ajouter considérablement plus d'informations dans le support confié à l'étudiant pour qu'il construise son programme.

Ensuite, il permet à la commission de mettre à jour facilement ces informations.

**Le module de gestion des contraintes** - Il vérifie la validité des programmes créés par les étudiants. L'intérêt de ce module est de réduire considérablement le temps que doit consacrer la commission à la correction des programmes d'étudiants. Premièrement, il contraint les étudiants à justifier les contraintes qui ne sont pas respectées avant d'envoyer leur programme à la validation s'il n'est pas correct. Il fournit aux étudiants un compte-rendu en temps réel de l'état de leurs programmes, en leur pointant les parties qui ne respectent pas les contraintes, quelles contraintes ne sont pas respectées et ce qu'il faut changer dans leur programme pour y remédier.

**La base de données** - Elle stocke les informations liées aux curricula enregistrés précédemment par la commission, mais aussi les programmes des étudiants. La commission pourra avoir accès aux anciens programmes de cours et à tous les programmes des étudiants. Ces derniers quant à eux, pourront accéder aux différents programmes qu'ils ont déjà suivis et avoir une vision claire de ce qu'il leur reste à valider pour obtenir leur diplôme.

Les fonctionnalités de l'application, telles qu'elles apparaissent sur la figure 1.3 vont être expliquées dans les sous-sections qui suivent.

## Lexique

Avant de poursuivre l'explication des fonctionnalités de l'application, cette sous-section va expliquer les différents objets qui sont utilisés tout au long de ce chapitre.

**Catalogue** Cet objet est la représentation d'un graphe généré avec yEd une fois importé en base de données. Un catalogue est constitué de plusieurs programmes, modules et cours et est identifié par l'année académique pour laquelle il est destiné.

**Programme** C'est la représentation d'un programme de cours (SINF2M, INFO2M, ...).

**Module** C'est un ensemble de cours qui peut être obligatoire (Le tronc commun du programme SINF2M) ou optionnel (les options du programme SINF2M).

### 3.3.2 Gestion des données

Cette section explique comment sont importées les données relatives aux programmes, leurs cours, leurs informations et leurs différentes contraintes.

Les différentes informations contenues dans chacun des programmes proposés sont les suivantes. Nous avons :

- plusieurs programmes de cours (SINF1BA, FSA1BA, SINF1PM SINF2M SINF2M1 et INFO2M) ;
- chacun de ces programmes contiennent des modules et des cours, avec des informations relatives aux cours et modules obligatoires ;
- chacun de ces cours peut avoir des dépendances<sup>5</sup> ;
- chaque cours, module et programme contient des propriétés. (Le nombre de crédits d'un cours, le semestre durant lequel il est dispensé, l'obligation de suivre un module, le nombre de crédits minium et maximum d'un programme ou module).

Il n'est pas viable pour la commission de programme d'ajouter ces informations une à une à l'aide de formulaires permettant d'ajouter et de modifier ces différents objets. Cette approche malgré qu'elle soit facile à mettre en place, contraint la commission de programme à de longues et fastidieuses séances d'encodage de programmes. Qui plus est, chacun des programmes de cours proposés a la forme d'un graphe. La solution la plus logique est donc d'utiliser un éditeur graphique pour construire les différents programme de cours.

La démarche qui a amené à utiliser le logiciel yEd, pour importer la structure des programmes de cours, est expliquée en détail dans la section 3.2.4. Cette solution a deux conséquences ;

1. Le logiciel étant externe à l'application, il est nécessaire d'exporter le graphe vers un fichier. Ensuite, il faut parser ce fichier pour en extraire les informations.
2. On est restreint dans les données que l'on peut ajouter dans un graphe. Ces informations se limitent aux différents labels des nœuds et à leurs dépendances. Une solution est requise pour ajouter les données manquantes.

Pour le **premier point**, il a été décidé d'exporter le graphe dans le format *GraphML* (une extension de XML destinée aux graphes). Les informations contenues dans ce fichier sont extraites par l'application à la création du catalogue. Plusieurs formats étaient disponibles pour exporter le graphe créé avec yEd, les raisons qui ont mené à utiliser le format GraphML sont expliquées en détail dans la section 4.3.4.

---

5. Se référer à la section 2.2 et la section 3.3.3 pour plus de détails sur ces contraintes

Les détails de l'implémentation concernant le module qui s'occupe d'extraire les informations contenues dans les graphes sont disponibles dans la section 4.3.4.

**Le deuxième point** justifie le fait que l'importation des données se fait en deux étapes (la première étant l'import du fichier de graphe). Les données manquantes après l'import du graphe sont les suivantes :

- les propriétés d'un cours ; son sigle (INGI1101 par exemple), le semestre durant lequel il est dispensé, le nombre de crédits, le fait qu'il soit obligatoire ;
- les propriétés d'un module ; son nom, le nombre de crédits minimum et maximum requis pour le valider ;
- les propriétés d'un programme de cours ; son nom, le nombre de crédits minimum et maximum requis pour le valider.

La liste de ces propriétés n'est pas exhaustive. Il se peut qu'à l'avenir, le besoin se fasse sentir d'ajouter des informations comme le nom d'un professeur, l'URL du cours, ou toute autre information nécessaire pour des contraintes qui n'ont pas été prévue lors de la conception de l'application. C'est pourquoi en plus de pouvoir d'ajouter et mettre à jour ces informations, il doit être possible d'ajouter des nouvelles propriétés.

Excel a été choisi pour ajouter toutes ces informations car :

- il est disponible sur toutes les plateformes (Mac OS, Windows, Linux) ;
- il est facile d'utilisation ;
- il permet de gérer des grandes quantités d'information de façon structurée ;
- le format vers lequel il exporte est très facile à parser (XLS)

### Conclusion

L'import de toutes les données se fait en deux étapes. La première consiste à importer le graphe de cours, contenant le nom de chacun des cours, modules et programmes, en plus de leur structure (les cours et modules inclus dans chaque module et programme) et des dépendances entre les cours. Ces informations ne sont pas modifiables une fois le catalogue de cours créé. En cas d'erreur, il n'est pas possible (pour le moment) de mettre à jour les données en téléchargeant une nouvelle version du graphe de cours. Il faut donc supprimer le catalogue de cours puis le recréer avec la nouvelle version du graphe.

Les explications relatives aux conventions à utiliser lorsque l'on crée un graphe avec yEd sont disponibles dans le manuel en annexe.

La deuxième étape quant à elle, consiste à importer le reste des données dans l'application, via un formulaire excel. Cette étape est répétable, à tout moment les données peuvent être mises à jour sans mettre en péril le fonctionnement de l'application. De plus, il n'y a pas de limites aux informations que l'on peut ajouter de cette manière dans le sens où l'on pourrait rajouter tout et n'importe quoi comme informations (pour peu que l'on vérifie les conventions d'import de formulaire excel) sans faire crasher l'application.

Cependant, on pourrait, en procédant de la sorte ajouter des incohérences, surtout si l'on modifie les informations relatives aux contraintes. Par exemple, si on ajoute un minimum de crédits à un module qui est plus grand que le maximum de crédits du programme dans lequel il se trouve, il serait impossible pour un étudiant de créer un programme valide.

Les explications relatives aux conventions à utiliser lorsque l'on complète le formulaire Excel sont disponibles dans le manuel en annexe.

Par souci de modularité, ces deux étapes sont implémentées dans des modules externes à l'application, comme expliqué dans le chapitre relatif au développement du système 4.1. L'application est relativement indépendante de ces modules, dans le sens où il serait relativement aisé d'ajouter une méthode encore plus pratique pour ajouter toutes ces informations, sans devoir modifier tout le reste de l'application. Par exemple, on pourrait ajouter manuellement sans passer par ces deux étapes ; il suffirait d'implémenter les vues (formulaires de création) correspondantes dans l'application.

### 3.3.3 Contraintes

Comme présenté dans la section précédente 2.2, les contraintes sont de plusieurs types. Le but de cette section est de présenter les choix que cette catégorisation impose de faire au niveau de la conception des fonctionnalités et d'expliquer plus en détails la logique intrinsèque des plus compliquées d'entre elles.

Comme expliqué dans les sections qui suivent, la location des informations relatives aux contraintes (les crédits et les dépendances d'un cours par exemple) n'est pas la même pour toutes les contraintes. Qui plus est, chaque contrainte n'est pas vérifiée de la même

façon. Une dépendance par exemple implique d'aller chercher si un cours est présent dans un programme, alors que le minimum de crédits requis d'un programme implique de compter les crédits de chacun ces cours. Enfin, chaque contrainte ne renvoie pas les même informations lorsqu'elle n'est pas valide (il faut renvoyer le cours le type de dépendance et le cours lié à la dépendance lorsqu'elle n'est pas valide par exemple).

Dès lors, pour ne pas surcharger les modèles de l'application et la rendre plus flexible (et ainsi conserver un faible couplage et une haute cohésion), la vérification de ces contraintes a été déléguée à un module externe. On évite, ainsi, de se retrouver avec des modèles dont la taille se chiffre en milliers de lignes de code qui sont très difficile à maintenir et faire évoluer.

### Dépendances

Comme expliqué précédemment les dépendances peuvent être des prérequis ou des corequis. En plus de cela, ces contraintes peuvent être :

**binaires** ; elles concernent deux cours ; un cours *source* et un cours *destination* ; le sens de la contrainte étant celui de la flèche (le cours source est le prérequis du cours destination par exemple) ;

**n-aire** ; elles sont composées de plusieurs cours *sources* et de plusieurs cours *destinations*.

Pour chaque ensemble n-aire de dépendances, il existe une condition qui s'applique sur chacune des dépendances qui le constitue. Cette condition est soit une disjonction (OR), soit une disjonction exclusive (XOR). L'effet de la condition est la suivante. Il n'y a pas de représentation explicite pour les conjonctions de dépendances car elle peuvent être représentées simplement par des relations binaires.

- Une contrainte disjonctive ne sera valide que s'il existe au moins une des sous-contraintes qui est vraie.
- Une contrainte disjonctive exclusive ne sera valide que s'il n'existe qu'une et une seule des sous-contraintes qui est vraie.

Il y a un exemple de chaque cas qui se trouve dans l'exemple fictif 3.1. La contrainte disjonctive se trouve dans le programme de Bachelier. La contrainte disjonctive exclusive quant à elle se trouve dans l'option Réseau et Sécurité du programme de Master.

Sur l'image 3.3 apparait en détail la contrainte disjonctive ;



FIGURE 3.3 – Contrainte n-aire disjonctive

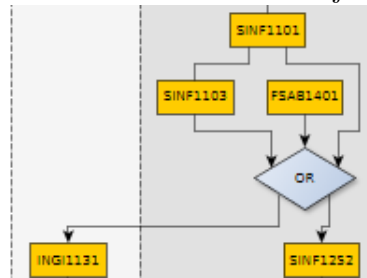
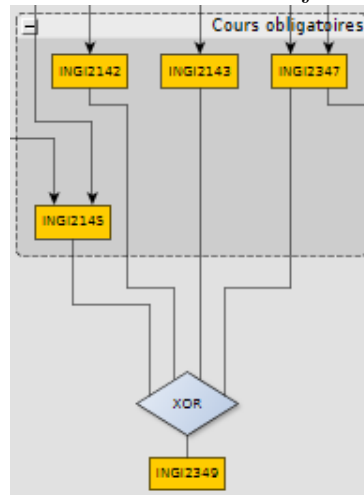


FIGURE 3.4 – Contrainte n-aire disjonctive exclusive



- *SINF1252* et *INGI1131* sont les cours *destinations*
- *SINF1101*, *SINF1103* et *FSAB1401* sont les cours *sources*

Ces contraintes étant des prérequis, il est donc nécessaire d'avoir suivi et réussi *SINF1103* **OU** (OR) *SINF1101* **OU**(OR) *FSAB1401* pour pouvoir suivre *SINF1252* **OU**(OR) *INGI1131*

Sur l'image 3.4 apparaît en détail une contrainte disjonctive exclusive ;

- *INGI2349* est le cours *destination* ;
- *INGI2145*, *INGI2142*, *INGI2143* et *INGI2347* sont les cours *sources*.

Ces contraintes étant des corequis, il est donc nécessaire d'avoir suivi au plus tard la même année un et un seul des cours *sources* pour pouvoir suivre le cours *INGI2349*. Si l'on suit deux, trois ou la totalité des cours *sources*, la contrainte en sera pas valide !

Notez que les dépendances sont importées à l'aide du logiciel yEd 3.2.4. Il n'est donc pas possible de les modifier, pour le moment, une fois le graphe importé dans l'application. Dans l'état actuel des choses, si l'on désire modifier la structure d'un catalogue de cours

(çàd sont graphe), il faut recréer le catalogue de cours avec le graphe modifié. Nous perdons cependant toute les informations additionnelles ajoutées via l'import de formulaires Excel. La mise à jour de la structure d'un catalogue de cours est une des améliorations à apporter à l'application, comme présenté dans le chapitre 6 relatif aux travaux futurs.

### Contraintes s'exerçant sur les propriétés des cours, modules et programmes

Ces contraintes regroupent plusieurs catégories de contraintes au sens où elles ont été présentées dans le chapitre 2.2. On dit que ces contraintes portent sur des propriétés car leur validité dépend de l'information contenue dans celles-ci. Pour reprendre l'exemple du minimum de crédits requis pour valider un programme de cours. La validité de cette contrainte dépend (en plus des cours et modules qui composent ce programme) de deux choses :

1. la valeur de cette propriété MIN, contenue dans la propriété du même nom de l'objet Programme ;
2. la valeur de la propriété CREDITS qui compose chacun de ses cours

Par exemple, si le module *Réseau et Sécurité* de l'exemple 3.1 comporte la propriété MIN, 15, la méthode qui vérifie les contraintes sur les propriétés de type MIN va aller compter un à un les crédits de chacun des cours qui compose le module, ainsi que chacun des cours qui composent chacun de ses sous-modules (le module intitulé *Cours obligatoires* ici).

Le module de contraintes n'a pas la charge de vérifier tous les types de contraintes. Dans le cas de la contrainte temporelle sur les semestres (qui implique qu'un cours doit être suivi durant le semestre au cours duquel il est dispensé), la vérification est implicite. En effet, la configuration de chaque année se faisant par semestre, il n'est pas possible pour un étudiant de choisir un cours, pour un semestre donné, qui n'est pas dispensé pendant ce semestre.

Les données relatives à ces propriétés sont importées et mises à jour par le module qui s'occupe d'importer les formulaires Excel. Elle peuvent donc être, contrairement aux dépendances, modifiées quand on le souhaite.

### 3.3.4 Fonctionnalités de l'application - Commission de programme

#### Introduction

Une fois connecté à l'aide du compte admin, nous arrivons à la page illustrée sur la figure suivante 3.5. Quatre menus sont accessibles depuis la barre de navigation (en haut dans la page d'accueil) :

**Catalogue** Ce menu offre l'accès à la gestion des catalogues. C'est ici que les catalogues sont importés (via le graph yEd) et mis à jour (via le formulaire Excel). Ce menu permet aussi de gérer les versions des catalogues, pour permettre de mentionner quel est le catalogue principal (celui qui sera utilisé par défaut par les étudiants), quels sont les anciens catalogues (Pour permettre aux étudiants d'avoir accès aux anciens programmes de cours) et quels sont les futurs catalogues qui, toujours en construction, ne sont pas accessibles aux étudiants.

**Demandes de validation** Ce menu offre l'accès aux requêtes de validation envoyées par les étudiants. La commission de programme aura accès ici au programme d'étudiant lié à la demande de validation, à l'état de celui-ci (est il valide ?) et à ses justifications en cas de contraintes non vérifiées dans son programme. C'est ici que la commission de programme accepte ou refuse les programmes d'étudiants.

**Gérer les années** Ce menu offre la possibilité à la commission de marquer les années comme réussies ou ratées (en choisissant les cours réussis).

**Discussions** Ce menu permet d'accéder aux différentes discussions qui apparaissent lors du processus de négociation entre la commission de programme et les étudiants, lorsque ces derniers sont amenés à justifier les exceptions éventuelles qui surviennent dans leur programme de cours.

#### Encoder et mettre à jour les programmes de cours

L'ajout de nouveaux programmes de cours se fait en deux étapes dans l'application (comme expliqué dans la section 3.2.4). La première étape correspond à l'import du graphe créé avec yEd. Cette étape importe dans la base de données les différents programmes de cours, leurs modules, leurs cours et les dépendances entre ces cours.

Cependant, les informations ajoutées par l'intermédiaire de cette étape ne sont pas suffisantes. Il manque toutes les propriétés des différents objets qui, en plus d'être des

FIGURE 3.5 – Page d’accueil



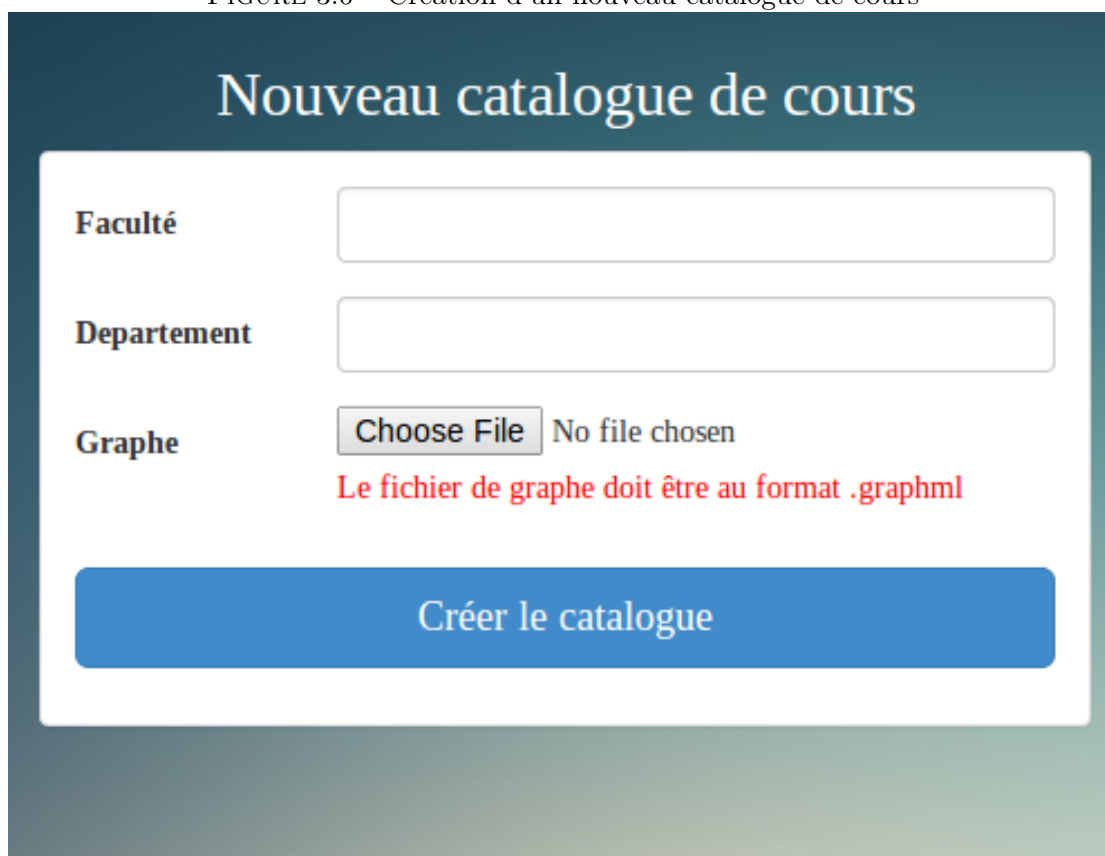
compléments d’information, servent pour certaines contraintes comme expliqué dans la section précédente 3.3.3. C’est pourquoi il existe une deuxième étape qui permet d’importer un formulaire Excel contenant les informations complémentaires des différents cours, modules et programmes (le nombre de crédits minium et maximum d’un programme, le nombre de crédits d’un cours, le semestre durant lequel il est dispensé).

Reprenons l’exemple 3.1. Les différents programmes de cours sont importés dans l’application par l’intermédiaire d’un module important les graphes yEd. Un catalogue de cours représente l’ensemble des programmes de cours présent sur le graphe 3.1. Le formulaire suivant 3.6 permet de créer ces catalogues.

L’utilisateur peut choisir l’année académique et le nom qui vont identifier le catalogue de cours. Cette différenciation est importante car l’application gère plusieurs version de catalogues. Ces versions sont de trois types :

1. la version principale ; cette version correspond au catalogue qui sera utilisé par défaut par les étudiants. Il ne peut avoir qu’un seul catalogue principal dans l’application. Lorsqu’un étudiant crée un compte utilisateur, il suit par défaut le catalogue principal. Un utilisateur peut cependant choisir un autre catalogue parmi les version anciennes.
2. les anciennes versions ; ces versions correspondent aux anciens catalogues principaux ; à chaque fois qu’un catalogue est élu “principal”, la version du catalogue principal devient ancienne ; il n’y a pas de limites sur le nombre de catalogues

FIGURE 3.6 – Création d'un nouveau catalogue de cours



**Nouveau catalogue de cours**

**Faculté**

**Departement**

**Graphe**  No file chosen

**Le fichier de graphe doit être au format .graphml**

**Créer le catalogue**

anciens ;

3. les version futures ; ces versions correspondent aux nouveaux catalogues créés par la commission de programmes, qui ne sont pas encore disponibles aux étudiants ; il n'y a pas de limites sur le nombre de catalogues futurs et ils ne sont pas accessibles aux étudiants.

La version d'un catalogue de cours évolue comme suit. À sa création il a la version *future*. Ensuite il passe à la version *principale* lorsque la commission le décide. Enfin, il passe à la version *ancienne* lorsqu'un autre catalogue est *élu principal*.

L'ajout, la modification et la récupération des informations relatives aux cours, modules et programmes se fait par l'intermédiaire d'un module d'import de fichiers Excel. Les consignes pour présenter les données sont expliquées en détails dans le manuel présent en annexe. Pour plus de facilité, il est possible de télécharger directement un template de ce formulaire depuis l'application. La structure reconnue par le module d'import est présente dans ce template, ainsi que le nom des différents cours, modules et programmes présents dans la base de données. De plus, il y a aussi, pour chaque type d'objet (cours,

modules et programmes) des exemples de propriété.

Reprenons l'exemple fictif illustré sur l'image 3.1. Lorsque l'on crée un catalogue avec ce graphe, et que l'on télécharge juste après le formulaire Excel, on obtient les informations suivantes.

- Sur la page des programmes (Figure 3.7), les propriétés relatives aux nombres minimum et maximum (MIN et MAX) de crédits d'un programme sont proposées.
- Sur la page des modules (Figure 3.9), les propriétés relatives aux nombres minimum et maximum (MIN et MAX) de crédits d'un module sont proposées. De plus, on peut aussi spécifier si le module est obligatoire ou non.
- Sur la page des cours (Figure 3.8), les propriétés relatives au semestre durant lequel le cours est dispensé sont notamment proposée, ainsi que la propriété *obligatoire* d'un cours.

La liste de ces propriétés n'est pas exhaustive. En effet il suffit, pour rajouter une nouvelle propriété, de simplement l'ajouter dans le formulaire. Le module d'import de données se chargera de créer les propriétés correspondantes si elles ne sont pas vide dans le formulaire.

Pour afficher les nouvelles propriétés, il suffit de les afficher dans la vue correspondante (se rendre dans la page `cours#show` et récupérer l'information de la nouvelle propriété par exemple)

Pour ajouter une contrainte sur une nouvelle propriété il faut ;

- ajouter la classe correspondante dans le module de contraintes ;
- créer l'objet de cette classe dans le modèle correspondant ;
- traiter les informations (de la nouvelle contrainte) renvoyées par le module de contraintes ;

Les détails d'implémentation des nouvelles contraintes sont expliquées dans la section 4.3.5.

Toutes ces fonctionnalités sont accessibles depuis la page telle qu'elle est illustrée sur l'image 3.10. Le menu déroulant du bas permet d'accéder aux différents programmes, modules et cours du catalogue (le tout, sans devoir recharger la page). En haut à gauche apparaissent les trois propriétés identifiant notre catalogue, à savoir son nom, son année académique et sa version.

FIGURE 3.7 – La page relative aux programme d'un template de formulaire

	A	B	C	D
1	NAME	MIN	MAX	CREDITS
2	MASTER			
3	BACHELIER			
4				

En naviguant dans les différents sous-menus du catalogue (cours, programmes, modules), vous pouvez accéder aux informations complètes concernant ces objets. Il est possible par exemple d'accéder aux détails des contraintes d'un cours. Il est notamment possible de créer des programmes de cours *customisés* à partir des informations présentes dans le catalogue (cours, modules). On pourrait donc par exemple créer un programme *Erasmus* ou *Mercator* avec les modules et cours disponibles, pour proposer aux étudiants étrangers un programme de cours adapté à leur profil.

### Gérer les années des étudiants

Bien qu'elle ne soit pas mentionnée sur le diagramme 1.3, la fonctionnalité qui permet à la commission de programme de marquer les années des étudiants comme réussie, ou comme ratée (en sélectionnant les cours réussis) est relativement importante pour le bon fonctionnement du module qui s'occupe de vérifier les contraintes.

En effet, pour vérifier la validité des contraintes de type *dépendance* (se référer à la section 3.3.3 pour plus de détails), il est nécessaire de différencier les années réussies des années ratées, et de différencier, dans ces années ratées, les cours crédités des cours ratés. Ainsi, le module qui vérifie les contraintes ne prendra pas en compte un cours qui est présent dans une année mais qui n'a pas été crédité lorsqu'il vérifiera certaines contraintes. Par exemple, lorsque l'on vérifie si le nombre de crédits minimum d'un module est atteint, ou si le prérequis d'un cours est valide, il ne faut pas prendre en compte les cours qui n'ont pas été crédités.

Pour les mêmes raisons, il est primordial de garder une trace des années ratées de l'étudiant, pour savoir quels cours l'étudiant a validé durant cette année qu'il n'a pas réussie.

On peut donc gérer sur la page 3.11 les années des étudiants. Marquer une année comme

FIGURE 3.8 – La page relative aux cours d'un template de formulaire

	A	B	C	D	
1	SIGLE	CREDITS	SEMESTRE	OBLIGATOIRE	
2	INGI2325		NONE	NON	
3	INGI2255		NONE	NON	
4	SINF2255		NONE	NON	
5	INGI2132		NONE	NON	
6	INGI2172		NONE	NON	
7	INGI2261		NONE	NON	
8	INGI2146		NONE	NON	
9	INGI2142		NONE	NON	
10	INGI2143		NONE	NON	
11	INGI2145		NONE	NON	
12	INGI2347		NONE	NON	
13	INGI2144		NONE	NON	
14	INGI2348		NONE	NON	
15	INGI2349		NONE	NON	
16	SINF1250		NONE	NON	
17	SINF1121		NONE	NON	
18	INGI1122		NONE	NON	
19	INGI1131		NONE	NON	
20	SINF1102		NONE	NON	
21	SINF1103		NONE	NON	
22	FSAB1401		NONE	NON	
23	SINF1101		NONE	NON	
24	INGI1341		NONE	NON	
25	SINF1252		NONE	NON	
26					
27					
28					

FIGURE 3.9 – La page relative aux modules d'un template de formulaire

	A	B	C	D	E
1	NAME	CREDITS	MIN	MAX	OBLIGATOIRE
2	TRONC COMMUN				
3	RéSEAU ET SéCURITé				
4	INTRODUCTION				
5					
6					



FIGURE 3.10 – Un catalogue de cours après sa création

Nom: **Test informatique** Année: **2014 - 2015** Version: **Future**

1 - Télécharger le fichier excel  
2 - Compléter puis sélectionner le fichier excel  
3 - Mettre à jour le fichier excel

1 - Télécharger le fichier excel

2 - Fichier excel  No file chosen  
Le fichier excel doit être au format .xls

3 - Mettre à jour les informations

Utiliser comme catalogue principal

Menu

Programmes	3
Modules	9
Cours	55

réussie ou ratée empêchera à l'avenir l'étudiant de modifier ou de supprimer son année dans la page de gestion de son programme. Pour marquer son année comme ratée, il suffit de sélectionner les cours ont été crédité (comme on peut le voir sur la figure 3.12)

FIGURE 3.11 – La page de gestion des années

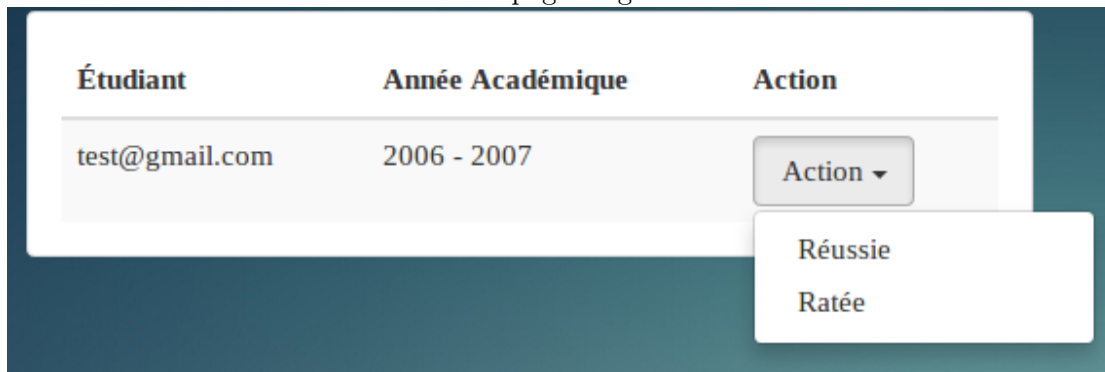


FIGURE 3.12 – Gérer une année ratée



### Gérer les demandes de validations

Cette fonctionnalité s'occupe de gérer les étapes *Négociation avec l'étudiant* et *Valider le programme de l'étudiant* du processus représentée sur le diagramme 1.3.

Lorsqu'un étudiant pense que son programme est correct, il envoie une demande de validation à la commission de programme. Dans le meilleur des mondes, le programme de l'étudiant respecte toutes les contraintes imposées par le programme qu'il suit. Cependant, pour une raison X ou Y, il arrive qu'un étudiant pense avoir une bonne raison pour enfreindre une ou plusieurs contraintes.

Prenons l'exemple d'un étudiant en provenance d'une autre université qui vient suivre un

programme de master à l’UCL. En regardant attentivement l’exemple fictif 3.1, on s’aperçoit que le cours *INGI2315* du programme de master a une dépendance (*SINF1140*) dans le programme de bachelier. Lorsque l’étudiant construit son programme de MASTER, il va se trouver avec des contraintes non respectées qu’il ne sera pas possible pour lui de corriger.

Comme expliqué dans la section relative aux fonctionnalités offertes à l’étudiant qui suit 3.3.5, l’application permet à l’étudiant, sous certaines conditions (remplir une justification s’il subsiste des contraintes non vérifiées par exemple), de soumettre un programme non valide à la validation. Lorsque notre étudiant soumettra son programme, il remplira un formulaire de justification avant d’envoyer sa demande de validation.

La page qui donne accès à la commission de programme aux demandes de validations est la suivante 3.13. Sur cette page, la commission de programme peut accéder aux programmes de l’étudiant (Menu Programme) et accéder à un menu qui permet de voir les contraintes non-vérifiées du programme et leur justification (Menu Exceptions). Ce dernier menu (Justification) permet aussi d’envoyer des message à l’étudiant pour demander des informations supplémentaires.

La page 3.14 donne accès à la justification du programme d’un étudiant. Dans cet exemple il manque, dans le programme de l’étudiant, plusieurs cours obligatoires. De plus, il a dépassé le maximum autorisé de son programme ainsi que des deux modules qui le compose. Sur cette page, la commission de programme à accès aux détails de chacune des contraintes non respectées du programme de l’étudiant. À droite de chaque contrainte non respectée se trouve la justification de l’étudiant. Comme affiché sur la page 3.15, la commission a aussi la possibilité d’envoyer un message à l’étudiant pour lui demander des informations supplémentaires.

Une fois les exceptions vérifiées, la commission n’a plus qu’à valider la demander ou bien la refuser.

FIGURE 3.13 – Page de gestion des demandes de validation

Demandes de validation				
Étudiant	Programme	Justification	Valider	Refuser
test@gmail.com				

FIGURE 3.14 – Les exceptions d’un programme et leurs justifications

test@gmail.com

#	Description	Justification
1	Le maximum requis 50 pour le Programme PROGRAM est dépassé. Crédits actuels: 80	AUCUNE
2	Le maximum requis 20 pour le Module MODULE 1 est dépassé. Crédits actuels: 40	AUCUNE
3	Les cours obligatoires du module MODULE 1 ne sont pas présents	
	Cours concernés	Présent?
	SINF2275	OUI
	INGI2263	NON
	INGI2379	OUI
	SINF1211	OUI
	INGE1322	OUI
	LSMF2013	NON
4	Le maximum requis 20 pour le Module MODULE 2 est dépassé. Crédits actuels: 25	AUCUNE

FIGURE 3.15 – Demander des informations supplémentaires à un étudiant

Hello world!

envoyé par xacrochet@gmail.com

Message

Envoyer

FIGURE 3.16 – Page d’accueil des étudiants



### 3.3.5 Fonctionnalités de l'application - Étudiant

#### Introduction

Une fois connecté à l'application, l'étudiant arrive sur la page illustrée sur la figure 3.16. Deux menus sont accessibles dans la barre de menu en haut ;

**Mes programmes de cours** Ce menu permet d'accéder à la création du ou des programmes suivis par l'étudiant. C'est par ici que le module de contraintes est appelé pour vérifier la validité des programmes de cours.

**Programmes disponibles** Ce menu permet d'accéder en détail aux programmes proposés. Il est aussi possible de changer la version du catalogue utilisé via ce menu. (Les différentes versions et leur utilité est expliquée dans la section 3.3.4)

Les sous-sections suivantes expliqueront les différentes fonctionnalités offertes aux étudiants.

#### Accéder au programmes de cours disponibles

Lorsqu'un étudiant désire créer un programme de cours, il doit tout d'abord choisir le programme qu'il va suivre. Si nous reprenons l'exemple fictif 3.1, il y a deux programmes proposés dans ce catalogue ; celui de BACHELIER et celui de MASTER. Comme expliqué dans la section 3.3.4, plusieurs versions de catalogues de cours sont disponibles dans l'application.

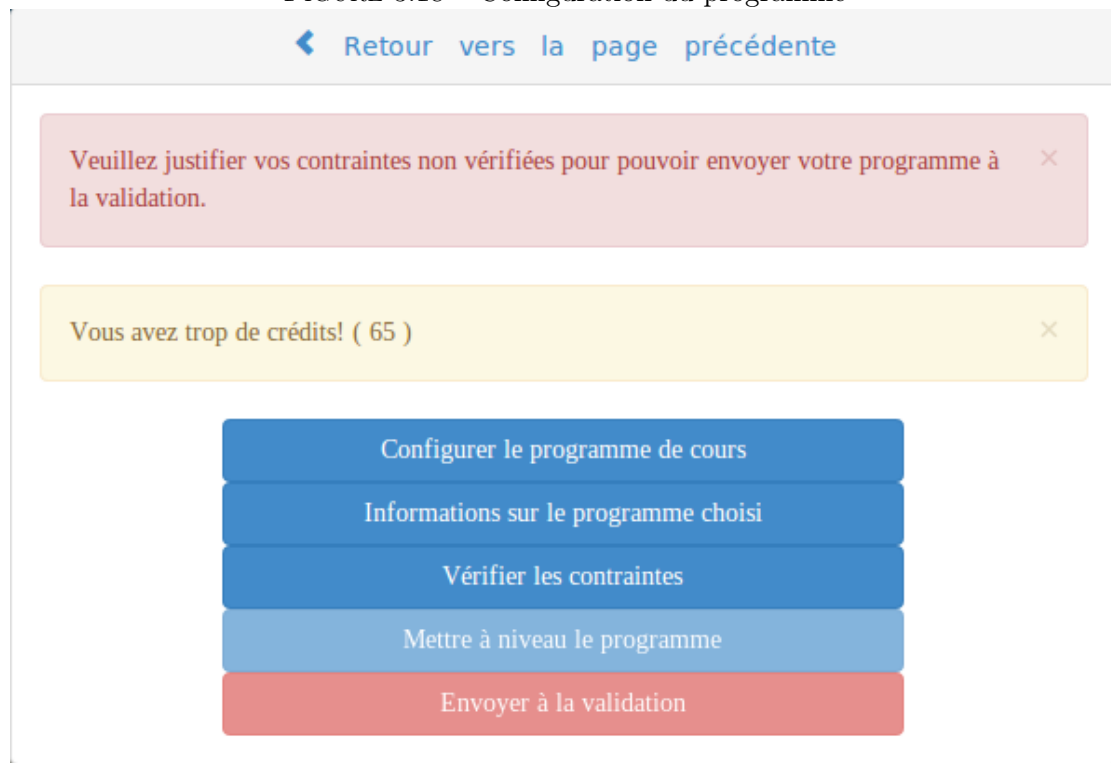
FIGURE 3.17 – Programmes disponibles



Pour le moment, il n'est pas possible de rendre un cours disponible dans plusieurs programmes lorsque l'on crée le graphe avec yEd. Cependant, il arrive souvent que cette situation se produise, notamment pour les programmes SINF2M et INFO2M qui ont de nombreux cours et modules en commun. C'est pourquoi la commission, en attendant qu'une solution plus efficace soit implémentée, doit créer deux catalogues différents. Un étudiant doit donc avoir la possibilité de choisir le catalogue qu'il va suivre en fonction du type de programme dans lequel il se trouve (ingénieur civil ou sciences informatiques).

On peut voir sur l'image 3.17 les programmes disponibles dans le catalogue de l'exemple fictif.

FIGURE 3.18 – Configuration du programme



### Configuration d'un programme d'étudiant

Un étudiant a deux choses à faire pour configurer son programme ; configurer les différentes années qui le constituent et choisir les différents modules. La page illustrée sur l'image 3.18 affiche les différentes fonctionnalités à la disposition de l'étudiant.

Notez que le bouton *Envoyer à la validation* est désactivé pour le moment. En effet, il n'est possible d'envoyer son programme à la validation que lorsque certaines conditions sont remplies, pour éviter que la commission de programme reçoivent des demandes de validations pas assez complètes. Les conditions requises pour pouvoir envoyer son programme à la validation seront expliquées en détail dans la section 3.13.

L'idée, pour construire son programme est de configurer chacune des années qui le composent et de choisir les modules correspondant, pour éliminer une à une les contraintes qui ne sont pas respectées. Si nous reprenons le diagramme 1.3, l'étape qui correspond à cette section est celle qui s'intitule *Construire/Modifier son programme*.

La page 3.18 permet d'accéder aux différents menus qui permettent de configurer un programme d'étudiant à savoir ;

1. configurer le programme

2. accéder aux informations du programme suivi par l'étudiant
3. vérifier les contraintes et justifier celles qui ne sont pas respectées
4. mettre à niveau le programme si une nouvelle version est émise par la commission de programme
5. envoyer le programme à la validation

La page permettant à l'étudiant de configurer une année de son programme est illustrée sur l'image 3.19. Les cours sont affichés en fonction de leur semestre et de s'il sont obligatoires ou non. Un cours sera obligatoire s'il a été marqué comme tel via le formulaire Excel, ou si son module parent l'est aussi.

Notez que si la commission n'a pas complété les informations après avoir importé le graphe, les champs *SEMESTRE* de chacun des cours ne seront pas initialisés. Aucun de ces cours ne sera affiché dans cette vue, cela, afin de permettre la vérification automatique des contraintes temporelles et éviter ainsi qu'un étudiant choisisse un cours qui n'appartient pas au bon semestre. On évite ainsi de surcharger la page de vérification des contraintes (qui est déjà fort chargée).

Tant que ces années ne sont pas marquées comme réussies ou ratées par la commission (via la fonctionnalité expliquée dans la section 3.3.4), elles sont modifiables et même supprimables. Par contre, une fois ces années marquées, il ne sera plus possible de les modifier.

De la même façon, il est possible de sélectionner les modules qui constituent le programme.

Lorsqu'un étudiant construit un programme, il configure essentiellement les cours de l'année qu'il va suivre et fait des prévisions sur les cours qu'il pourrait suivre l'année prochaine pour qu'il ne se trouve pas dans une situation où, en troisième année de bachelier par exemple, il ne puisse pas valider son programme.

Lorsque l'étudiant reprend son programme, l'année après sa validation par la commission, il doit recommencer ce processus. Il est cependant fort probable qu'une nouvelle version du programme de cours soit émise par la commission. C'est pourquoi il est possible via le bouton *Mettre à niveau le programme* de mettre à jour celui-ci avec la nouvelle version disponible. Cette fonction va mettre à jour toutes les instances des cours constituant les années qui n'ont pas encore été créditées (ou partiellement créditées dans le cas d'une



FIGURE 3.19 – Configuration des années

**Année**

2013 - 2014 ▼

Premier Quadri

Second Quadri

Obligatoires

☒ Sinf2275

Optionnels

☒ Ingi2261  
☒ Inge1322  
☒ Ingi2262  
☐ Ingi2146

Modifier l'année

année ratée) avec celles de la nouvelle version du programme de cours (et toutes leurs contraintes par la même occasion). Les années (totalement ou partiellement) créditées sont automatiquement détectées par le module de contraintes. Celui-ci s'occupe d'aller chercher les contraintes correspondantes.

Cependant, lorsque certains cours changent de sigle, il ne sera pas possible pour cette fonctionnalité de les récupérer. Si ce cours qui change de nom est le prérequis d'un autre cours, cette situation peut devenir problématique pour l'étudiant. En effet, l'application ne sera pas capable de retrouver l'ancienne version du cours ; l'étudiant aura donc une contrainte qu'il sera impossible de corriger. Pour gérer cette situation, il est permis à l'étudiant de soumettre des programmes qui ne vérifient pas certaines contraintes, du moment qu'ils les justifient et que la commission de programme accepte cette justification. Il suffira à l'étudiant d'expliquer, dans le champ prévu à cet effet qu'il a déjà suivi l'ancienne version du cours.

### Vérification des contraintes

Le module de gestion des contraintes renvoi beaucoup d'informations à propos des choses manquantes dans le programme. La page illustrée sur la figure 3.20 filtre ces informations et affiche essentiellement les dépendances manquantes.

Toutes les contraintes ne sont pas affichées sur cette page. La liste complète des contraintes gérées par le module qui gère les contraintes est la suivante :

- les différentes dépendances entre les cours, telles qu'elles sont présentées sur la figure 3.20 ;
- les contraintes relatives au nombre de crédits (crédits totaux, crédits minimum, crédits maximum) d'un ensemble de cours (programme ou module) ; en plus d'être affichées lorsqu'elles ne sont pas respectées, un menu récapitulatif (montrant la progression en terme de crédits du programme) est disponible en haut de la page 3.20 ;
- les contraintes relatives au champ *OBLIGATOIRE* d'un cours ou d'un module.

Comme expliqué dans la section 3.3.5, les contraintes temporelles sont gérées directement lorsqu'un cours est proposé à l'étudiant. En effet, une année se configure par semestre et, lorsque l'on configure le premier semestre par exemple, les cours du second semestre ne sont pas affichés. e

FIGURE 3.20 – Gestion des contraintes

[Retour vers la page précédente](#)

Crédits: 65

Minimum: 20

Maximum: 50

Vous avez dépassé la limite de crédit autorisée pour ce programme!

×

65 / 20 crédits

Vous avez 4 contrainte(s) non respectée(s)

Cours obligatoires manquants ( 1 )

Les cours suivants du module obligatoire MODULE 1 n'ont pas été sélectionnés

Cours manquant	JUSTIFICATION
INGI2263	<div></div>
LSMF2013	

Maximum de crédits dépassés ( 3 )

Compléter la justification

FIGURE 3.21 – Statut du programme d'un étudiant

[Retour vers la page précédente](#)

Crédits: 65

Minimum: 20

Maximum: 50

Vous avez dépassé la limite de crédit autorisée pour ce programme!

×

65 / 20 crédits

C'est sur cette page 3.20 que l'étudiant justifie **chacune des contraintes que son programme ne respecte pas**. Il n'est d'ailleurs pas possible d'envoyer le programme à la validation tant que chacune de ces exceptions n'a pas été justifiée.

### Négociation - Envoyer son programme à la validation

Pour filtrer au maximum les demandes de validation et éviter ainsi de surcharger la commission de programme de requêtes inutiles, il n'est possible d'envoyer son programme à la validation que si certaines conditions sont respectées :

**avoir assez de crédits** le programme de l'étudiant doit respecter le minimum de crédits requis par le programme qu'il suit (dans le cas où le programme ne propose pas suffisamment de cours, il suffit d'avoir autant de crédits que le programme en propose)

**avoir accédé au menu de gestion des contraintes** à chaque fois que l'étudiant modifie son programme, il lui est demandé d'avoir visité au moins une fois la page relative à la vérification des contraintes pour pouvoir soumettre son programme ;

**ne pas avoir de dépendances non respectées ou avoir rempli le formulaire de justification** si le programme de l'étudiant comporte des contraintes non respectées, il lui est demandé de remplir un formulaire de justification dans le quel il doit justifier chacune des exceptions (contraintes non respectées) ;

**ne pas avoir une requête en cours** si une requête a déjà été envoyée pour le programme, il n'est pas possible d'en envoyer une nouvelle tant que la précédente n'a pas été refusée ou acceptée par la commission de programme. Pour ne pas bloquer l'étudiant et demander à la commission de programme de faire plusieurs fois le travail de vérification, l'étudiant peut à tout moment annuler sa demande de validation, modifier son programme puis la renvoyer.

### 3.3.6 Conclusion

L'ensemble des fonctionnalités proposés aux deux types d'utilisateurs (étudiant et commission de programme) vient d'être présenté tout au long de cette section. Bien que cette section puisse servir de manuel, une version plus détaillée est présente en annexe expliquant plus en détail la démarche complète à suivre pour gérer les catalogues de cours et les programmes d'étudiant.

Notez que l'interface de l'application a été conçue pour être utilisée aussi bien sur un ordinateur que sur un smartphone.

Cependant, il se peut que l'interface et certaines des fonctionnalités évoluent sensiblement suite au feedback récupéré après la remise des scénarios de validation.



## Chapitre 4

# Développement du système

### 4.1 Introduction

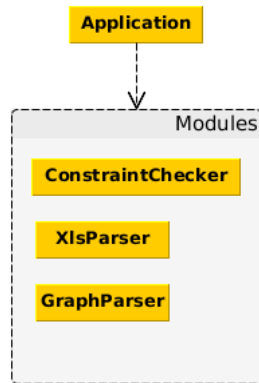
Le but poursuivi par ce chapitre est d'expliquer les choix qui ont influencé le développement de l'application et de chacun de ses modules externes (ConstrantChecker, XlsParser et GraphParser).

La structure du chapitre sera la suivante.

La première section présentera l'architecture de l'application et de ses différents modules. On évitera ici de parler de l'architecture MVC car très peu de choix ont été faits à ce niveau (peu de libertés sont laissées par le framework au final). L'application gère et échange (avec ses modules) une grande quantité de données. Leur modélisation a un impact critique sur les fonctionnalités de l'application (et leur implémentation). C'est pourquoi, l'accent sera mis ici sur la modélisation des classes de l'application.

La deuxième section présentera les choix faits au niveau de l'implémentation des différentes fonctionnalités. Il sera expliqué par la même occasion comment ces fonctionnalités ont été implémentées.

FIGURE 4.1 – Architecture globale



## 4.2 Architecture

### 4.2.1 Architecture globale

Ce modèle représente l'architecture de l'application et de l'ensemble de ses modules. L'objet application représente la partie *Rails* de l'application. Cette partie comporte essentiellement les différents modèles, leurs vues et leurs contrôleurs (en plus de la base de données). L'application utilise trois modules développés indépendamment.

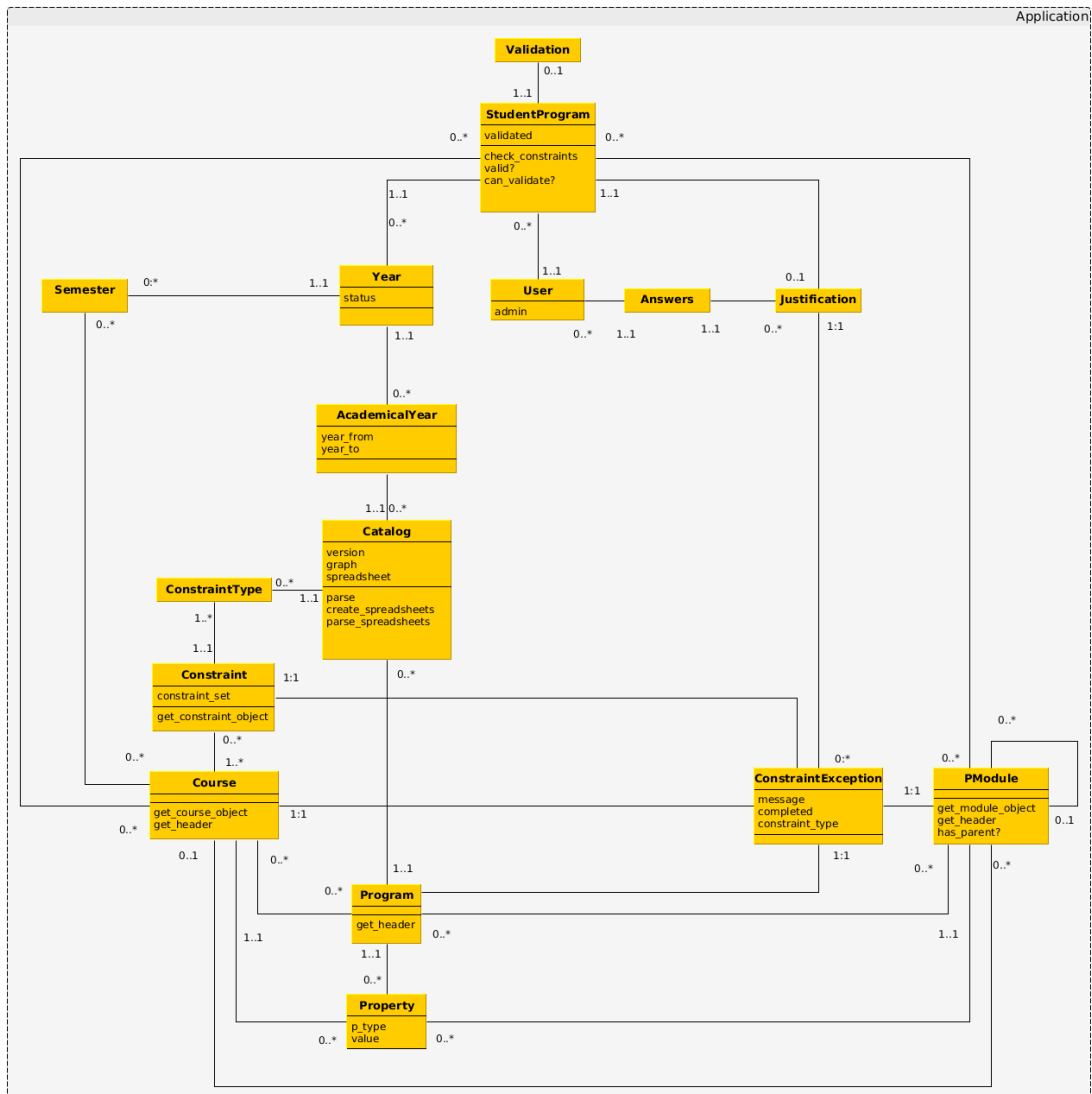
1. **GraphParser** - le module s'occupant d'extraire l'information contenue dans les fichiers *Graphml* générés par yEd,
2. **ConstraintCheker** - le module s'occupant de vérifier les contraintes des programmes d'étudiants
3. **XlsParser** - le module occupant d'exporter les données relatives aux programmes de cours vers un formulaire Excel et d'extraire les informations contenues dans les formulaires Excel que la commission importe via l'application.

Les informations manipulées par ces trois modules sont présentées en détail dans la section relative à la gestion des données du chapitre précédent 3.3.2.

L'architecture de chacun de ces modules, en plus de celle de l'application, sera expliquée dans les sous-sections qui suivent. Notez que la notation UML sera utilisée pour présenter les différents diagrammes de classe.



FIGURE 4.2 – Architecture de l'application



## 4.2.2 Application Rails

Il y a plusieurs types d'associations en Ruby on Rails ;

- l'association de cardinalité (0..1, 1..1) est représentée par la double relation A has one B et B belongs to A ; l'id de l'objet A étant stocké dans la table de l'objet B ;
- l'association de cardinalité (0..\*, 1..\*) est représentée par la double relation A has many B et B belongs to A ; l'id de l'objet A étant stocké dans chacun des objets B qui est en relation avec A ;
- l'association de cardinalité (0..\*, 0..\*) est représentée par la double relation has and belongs to many entre A et B ; les ids de chacun des objets sont stockés dans une table intermédiaire.

### User

Cette classe représente les utilisateurs. L'attribut **admin** sert à différencier les deux acteurs de l'application, à savoir la *Commission de programme* et les étudiants. La section 4.3.3 explique en détail comment sont gérés les accès de ces deux types d'utilisateurs.

### Property

Un objet *Property* est composé d'un type et d'une valeur. Chacun des objets *Program*, *PModule* et *Course* peu en avoir zéro ou plusieurs. Par exemple, pour représenter le sigle d'un cours, une propriété de type *SIGLE* et de valeur *SINF1101* sera ajoutée au cours correspondant. Il a été choisi d'opter pour cette solution, plutôt que d'ajouter des champs arbitraire (Sigle, crédits, ...) à chacun des objets car on ne sait pas à l'avance quelles seront leur propriété. En effet, elle sont déterminées par les informations mises dans le fichier excel qui est importé régulièrement dans l'application.

### PModule

C'est un ensemble de cours. Un *PModule* peut avoir plusieurs *PModule*. Ce comportement est justifié par le fait qu'un module de cours peut comporté un sous module qui comporte une série de cours obligatoires (L'option réseau et l'ensemble de ses cours obligatoires par exemple).

### Program

Il représente un programme de cours. (Le programme de master par exemple) C'est un ensemble de cours et de modules divers. On peut créer des programmes via l'outil de graphes yEd, mais il est possible dans l'application de créer des Programmes *à la carte* en choisissant les modules et cours qui le compose. C'est pourquoi il y a une relation *many to many* entre *PModule* et *Program* et une autre entre *Program* et *Course*. En effet, chaque Programme peut avoir un ou plusieurs cours, et chaque cours peut appartenir à un ou plusieurs programmes (Le même comportement est observé pour les modules). Il n'est donc pas possible de représenter cette relation avec une relation *has many* classique.

## Catalog

Un catalogue est la représentation d'un graphe généré avec le logiciel yEd en base de données. Il est composé de plusieurs *Program*, *PModule* et *Course*. Il contient aussi les informations à propos du fichier de graphe et du formulaire excel (nom, date, type).

L'attribut *version* est un entier qui représente la version du catalogue de cours. Il peut prendre trois valeurs différentes :

- version *future*, représentée par l'entier 0 ;
- version *principale*, représentée par l'entier 1 ;
- version *ancienne*, représentée par l'entier 2.

La version du catalogue est utilisée pour identifier les catalogues qui sont utilisables par les étudiants (version principale et ancienne) de ceux qui sont encore à l'état de *draft*. Les anciens catalogues contiennent les informations (propriétés et contraintes) liées aux années réussies ou ratées. Les contraintes et propriétés évoluant chaque année, il est nécessaire de garder la trace des anciennes pour que les programmes des étudiants restent valides au fur et à mesure que les catalogues évoluent.

Les trois méthodes principales de cette classe sont :

**import\_graph** qui appelle le module *GraphParser*

**import\_catalog\_data** qui extrait les informations du formulaire excel (et upload la nouvelle version du formulaire sur le cloud).

**export\_catalog\_data** qui crée le formulaire excel (et l'upload sur le cloud).

Les détails d'implémentation de ces méthodes seront expliquées dans la section relative au module avec lequel elles interagissent.

## StudentProgram

C'est le programme que se crée l'étudiant lorsqu'il utilise l'application. Un *StudentProgram* est une instantiation d'un des *Program* disponible dans le *Catalog* utilisé (d'où la relation *many to many*). De plus, un étudiant doit choisir les modules qu'il va suivre. Ce comportement est expliqué par la relation *many to many* entre les deux modèles.

Pour configurer son programme année par année, l'étudiant va se créer une ou plusieurs années (*Year*)

L'attribut *validated* représente le fait qu'un programme d'étudiant aie été validé ou non par la commission.

La méthode *can\_validate?* vérifie que certaines conditions sont remplies pour pouvoir envoyer un programme d'étudiant à la validation (Se référer à la section 3.3.5 pour plus de détail sur cette fonctionnalité).

La méthode *check\_constraints* gère l'interaction avec le module *ConstraintsChecker*.

### **Year**

Une année est composé de plusieurs semestres (deux idéalement). L'attribut *status* représente les trois état dans le quel peut se trouver une année (réussie, ratée ou en cours). Un semestre est représenté par l'objet *Semester*. Le choix de chacun des cours du semestre est représenté par une association *has\_and\_belongs\_to\_many* qui existe entre les deux objets.

Les programmes étant validés par année en s'appuyant sur l'ensemble du programme de l'étudiant, ce modèle est d'autant plus important. En effet, en aucun cas, le programme n'est validé dans son ensemble.

### **Header**

Chacun des modèles *Course*, *Pmodule* et *Program* contient une méthode *get\_header* qui renvoie une suggestion de noms de propriétés utilisées à titre indicatif avec le module *XlsParser* (voir 4.2.5) pour créer les formulaires Excel. Nous avons le header suivant : {"SIGLE", "CREDITS", "SEMESTRE", "OBLIGATOIRE"} pour le modèle *Course* par exemple.

Si à l'avenir, on découvre une autre propriétés qu'il est important d'avoir dans pour un cours, un programme ou un module, une bonne pratique serait d'ajouter le nom de cette propriété dans ce header !

## Méthodes `get_objet`

Ces méthodes s'occupent de créer les objets qui sont utilisés par le module *ConstraintsChecker*. Pour un objet du modèle *Cours* par exemple, il va créer un objet de la classe *Cours* (spécifique au module *ConstraintsChecker* et des objets pour représenter ces différentes contraintes (dépendances, etc). Ces objets (qui sont spécifiques au module) sont ensuite manipulés par le module *ConstraintsChecker*.

## Constraints

Cette classe représente les dépendances entre les cours. Le type de la dépendance est représenté par le modèle *ConstraintType*. L'attribut *set\_type* quant à lui représente le type de l'ensemble des contraintes. Comme expliqué dans la section 3.3.3, une dépendance peut être une relation binaire entre un cours et sa dépendance. Elle peut être aussi une relation n-aire entre plusieurs cours et leurs dépendances.

Il existe deux associations (bien qu'elles ne soient représentées que par une seule, par souci de clarté, sur le diagramme de classes 4.2) qui relient les objets *constraints* aux objets *courses*. Le modèle *Constraint* ne représente que les contraintes de type dépendance, comme expliqué dans la section 3.3.3. Ce type de contrainte ayant un cours source et un cours destination, il est donc nécessaire d'avoir deux relations. La relation entre la destination et la contrainte est représentée par une association *has\_many* entre le cours et la contrainte. La relation entre la ou les sources et la contrainte, quant à elle, est représentée par une association *has\_and\_belongs\_to\_many*.

Ce comportement est justifié par le fait que l'on accède aux contraintes depuis le cours *destination*, c'est à dire que l'on va chercher les dépendances d'un cours, et non chercher la relation inverse, à savoir quels sont les cours pour lesquels un cours joue le rôle de dépendance.

## AcademicYear

Ce modèle représente une année académique, c'est à cheval sur deux années civiles. Elle est utilisée dans deux situations :

1. identifier les objets *Year*

2. identifier les objets *Catalogue* ; en effet, chaque années lorsque l'on modifie les programmes de cours d'un catalogue, il faut importer le nouveau graphe. On peut ainsi identifier l'année académique à la quelle le catalogue est lié et savoir où aller chercher la nouvelle version du programme que l'étudiant suit lorsqu'une nouvelle année académique commence.
3. gérer, du coté du module *ConstraintsChecker* les prérequis et les corequis d'un cours. En effet, il est nécessaire de savoir quand a été suivi une dépendance pour savoir si la contrainte est valide ou non.

### ConstraintException

Lorsque le module *ConstraintsChecker* a terminer la vérification des contraintes du programme d'un étudiant, il renvoie les contraintes qui ne sont pas valides. Lors de la phase de négociation entre la commission de programme et l'étudiant, ce dernier doit justifier chacune des contraintes qui ne sont pas valides dans son programme.

L'utilité de ce modèle est de faire le lien entre la contrainte non vérifiée, le programme de l'étudiant et le message qui justifie cette exception dans son programme.

Un objet *ConstraintsChecker* peut être en relation avec ;

- un objet Cours, P\_Module ou un Program lorsqu'il concerne une contrainte, portant sur les propriétés, qui n'est pas valide ;
- un objet Constraint lorsqu'il concerne une dépendance (prérequis par exemple) qui n'est pas valide.

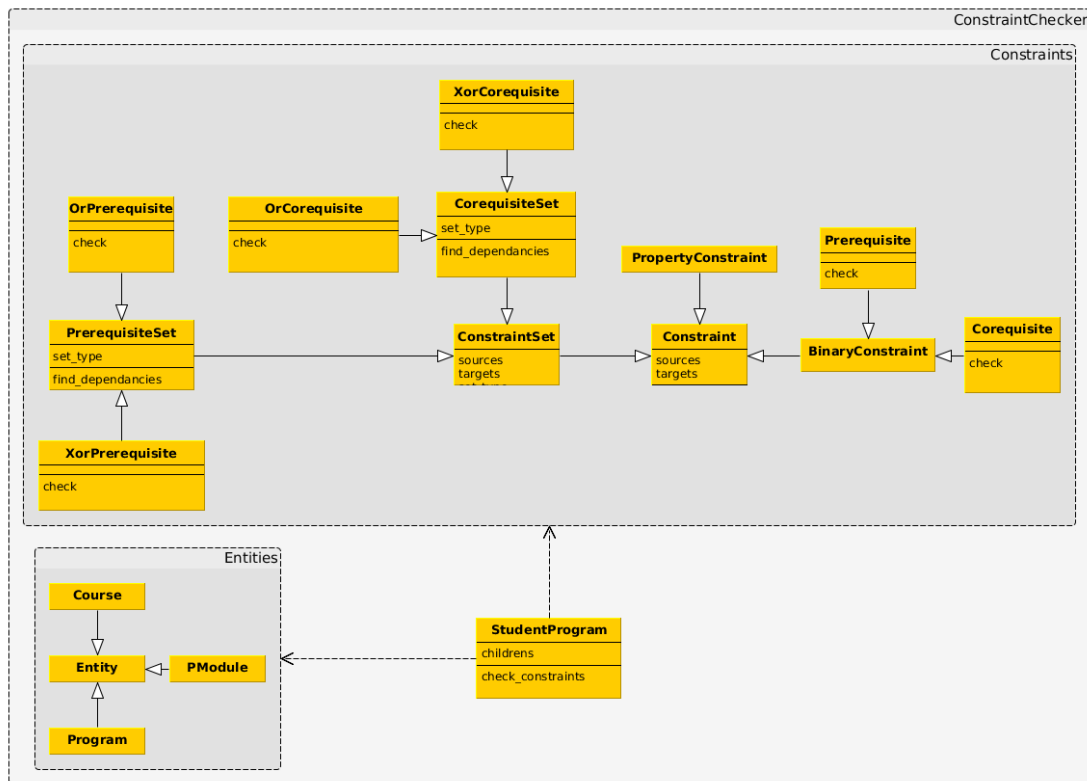
De plus il contient ;

- un attribut *constraint\_type* pour différencier les types de contraintes (prérequis disjonctif(OR), minimum de crédits requis, etc) ;
- un message qui stocke la justification de l'étudiant.

Toute ces exceptions sont stockées dans une instance du modèle *Justification* qui va servir de *médium* entre les deux parties durant la phase de négociation.

### 4.2.3 Architecture du vérificateur de contraintes

FIGURE 4.3 – Vérificateur de contraintes



L'architecture de ce module est composé de deux parties :

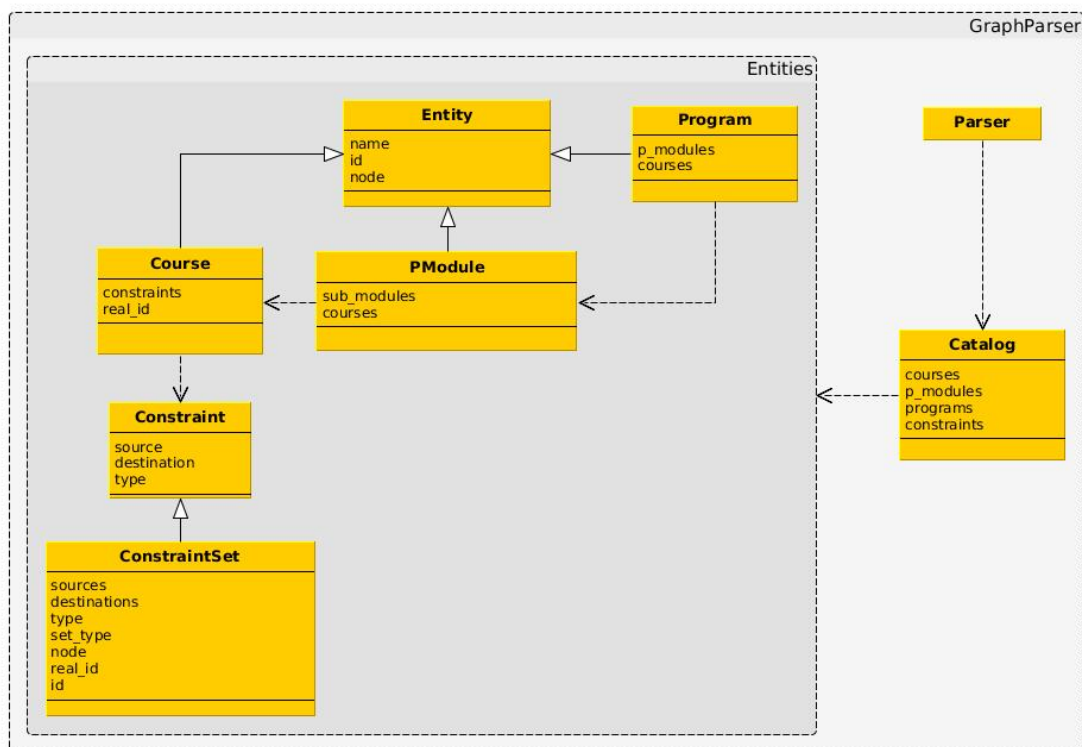
1. Les différents types de contraintes ;
  - Corequisite (classe parente : BinaryConstraint) ;
  - Prerequisite (classe parente : BinaryConstraint) ;
  - OrCorequisite (classe parente : ConstraintSet) ;
  - OrPrerequisite (classe parente : ConstraintSet) ;
  - XorCorequisite (classe parente : ConstraintSet) ;
  - XorPrerequisite (classe parente : ConstraintSet) ;
  - Min (classe parente : PropertyConstraint) ;
  - Max (classe parente : PropertyConstraint) ;
  - Mandatory (classe parente : PropertyConstraint).
2. Les différents types d'entités (Cours, modules, programme et student\_program).

Le lien avec l'application se situe au niveau de la classe *StudentProgram*. En effet, chacune des différentes entrées des objets (de la base de données) concernées (cours, p\_modules, constraints) est traduite en un objet entité.

L'idée ici est d'utiliser au plus l'héritage pour éviter d'avoir des duplications de code dans les classes. Par exemple, un objet *Course* peut avoir beaucoup de contraintes mais chacune d'entre elles peut être de n'importe quel type. Cet objet n'a pas besoin de savoir le type de ses contraintes. Tout ce qu'il sait, c'est qu'il doit appeler leur méthode *check* pour tester si les contraintes sont vérifiées.

#### 4.2.4 Parser de graphes

FIGURE 4.4 – Architecture du parser de graphe



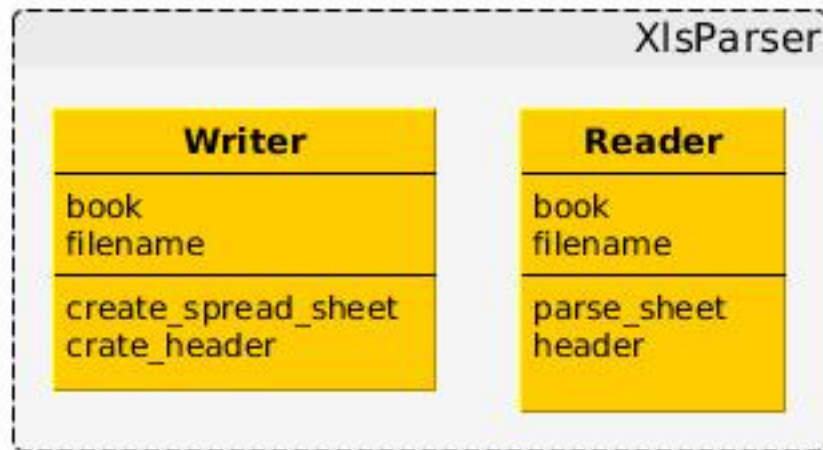
Tout comme dans le vérificateur de contraintes 4.2.3, on utilise un objet ruby qui étend la classe *Entity* pour chaque objet de la base de données (courses, p\_module, program). Une fois le graphe parsé, les informations contenues dans ces objets ruby sont ajoutées dans la base de données.

De nouveau, l'héritage occupe une page prépondérante ici, pour diminuer le couplage, augmenter la cohésion et éviter autant que possible la duplication de code [Miller(2008)].



#### 4.2.5 Parser de fichiers excel

FIGURE 4.5 – Architecture du parser de fichiers excel



Ce module est relativement simple ; il est composé de deux classes, un *Writer* qui prend en input un tableau de données et un *Reader* dont l'output est aussi un tableau de données.

## 4.3 Implémentation

### 4.3.1 Introduction

Le but de ce chapitre est de présenter les grandes lignes des algorithmes utilisés dans chaque partie de l'application.

### 4.3.2 Hébergement de l'application

L'application est hébergée sur *Heroku*. On évite ainsi de devoir s'occuper de la configuration et de la maintenance du serveur qui doit héberger l'application. Cela impose cependant quelques restrictions ;

1. On est obligé d'utiliser postgresql comme système de base de données.
2. Le répertoire de l'application est en lecture seule. On ne peut donc pas stocker le fichier de graphe et le formulaire Excel dedans. Il est donc nécessaire d'utiliser un service de *cloud storage* externe à l'application. Amazon S3 a été utilisé pour palier à ce problème. Pour rendre le téléchargement des fichiers vers ce service plus aisé, la gem *Paperclip* a été utilisé. Les détails de configuration de ces différents services sont expliqués en annexes.

### 4.3.3 Gestion des utilisateurs

Les utilisateurs sont gérés à l'aide de deux gems.

Devise est utilisé pour tout ce qui concerne la gestion des comptes (Création, modification, suppression), la gestion des sessions (Login/logout) et surtout la création de la table users et des différents attributs requis.

CanCan est utilisé pour tout ce qui concerne les permissions des utilisateurs, à savoir à quels modèles un utilisateur a accès, et quelles actions il peut effectuer sur ces modèles (Read, Create, Destroy, ...)

Pour gérer les deux types d'utilisateurs (Commission de programme et Étudiants), trois choix s'offrent à nous :

1. générer avec Devise deux tables séparées ;

2. utiliser la Single Table Inheritance [Wang(2013)]. On crée un modèle *user*, puis on crée deux modèles spécifiques (*student* et *admin*) qui héritent de ce premier modèle ;
3. générer un seul modèle *user* et y ajouter un attribut *admin* pour identifier le rôle de l'utilisateur.

La troisième solution a été choisie. Elle permet d'éviter la redondance induite par la première solution et est plus simple à implémenter et à maintenir que la deuxième solution. En effet nos deux types d'utilisateurs ne diffèrent que par leur rôle.

#### 4.3.4 Importation du graphe

##### Introduction

Une fois le graphe créé à l'aide de yEd, plusieurs choix s'offrent à nous pour exporter nos données. Les formats (non binaires) dans lesquels nous pouvons exporter les informations contenues dans notre graphe sont les suivants :

1. GraphML, un format de fichier basé sur XML pour les graphes ;
2. XGML, une alternative au format GraphML, mise en place par yWorks, la société qui développe le logiciel yEd ;
3. TGF (Trivial Graph Format) un format de fichier texte relativement simple pour décrire des graphiques.

TGF est de la forme

```
1 SINF1101
2 FSAB1401
3 SINF1103
4 SINF1102
5 SINF1140
6 INGI1123
7 INGI1101
8 FSAB1402
9 NS //(Option network & security du programme Master)
```

et contient trop peu d'informations sur le graphe, comme les appartenances des cours aux différents modules et programmes. C'est pourquoi une solution basée sur XML a été choisie.

Un fichier GraphML contient les informations suivantes pour un noeud de type COURS.

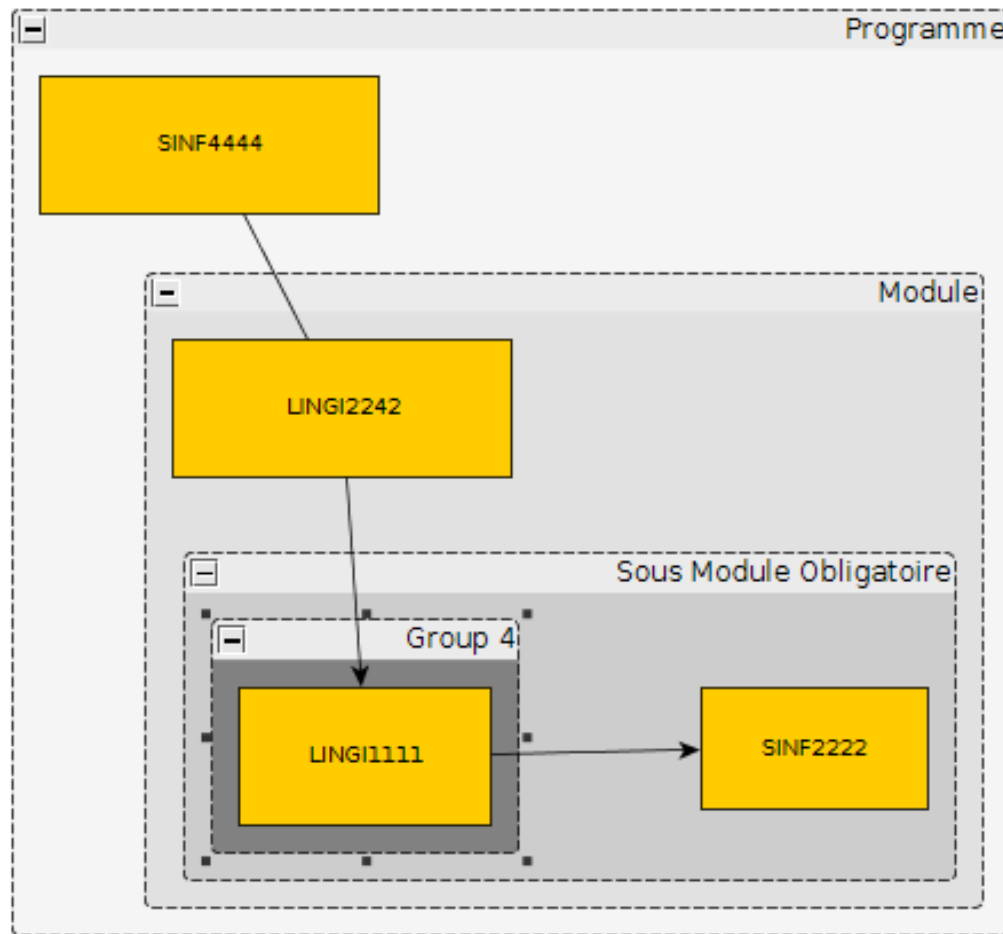
```
<node id="n1::n3" yfiles.foldertype="group">
  <data key="d5"/>
    <data key="d6">
      (...)
    <node id="n1::n3::n2">
      <data key="d5"/>
      <data key="d6">
        <y:ShapeNode>
          <y:Geometry height="30.0" width="68.0" x="1136.0" y="1541.453125"/>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content" fontFamily="
Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" height="17.96875" modelName="internal" modelPosition="c"
textColor="#000000" visible="true" width="61.57421875" x="3.212890625" y
="6.015625">SINF2335</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
  </data>
  (...)
</node>
```

En comparaison, la version *XGML* du même nœud.

```
<section name="node">
  <attribute key="id" type="int">69</attribute>
  <attribute key="label" type="String">SINF2335</attribute>
  <section name="graphics">
    <attribute key="x" type="double">1170.0</attribute>
    <attribute key="y" type="double">1556.453125</attribute>
    <attribute key="w" type="double">68.0</attribute>
    <attribute key="h" type="double">30.0</attribute>
    <attribute key="type" type="String">rectangle</attribute>
    <attribute key="fill" type="String">#FFCC00</attribute>
    <attribute key="outline" type="String">#000000</attribute>
  </section>
  <section name="LabelGraphics">
    <attribute key="text" type="String">SINF2335</attribute>
    <attribute key="fontSize" type="int">12</attribute>
    <attribute key="fontName" type="String">Dialog</attribute>
    <attribute key="anchor" type="String">c</attribute>
  </section>
  <attribute key="gid" type="int">61</attribute>
</section>
```

La principale différence entre le format XGML et GraphML se situe au niveau de la structure des informations. XGML structure toute les données de façon linéaire, en ne

FIGURE 4.6 – Exemple de graphe hiérarchique



respectant pas la hiérarchie des différents nœuds et boîtes.

Pour le graph 4.6 par exemple, la structure d'un fichier XGML sera de la sorte :

```

Node Programme
Node SINF4444
Node MODULE
Node LINGI2242
Node SOUS MODULE OBLIGATOIRE
(...)
  
```

Lorsque l'on parsera le fichier, on devra :

- Parcourir le fichier, extraire les informations (nom, parent) de chaque cours, module et programme ;
- Parcourir la liste des cours, modules et programmes pour ajouter les références vers leurs parents et leurs enfants.

Avec GraphML par contre, la structure du fichier sera la suivante ;

```
Node Programme
Childs : [
  Node SINF4444
  Node MODULE
  Childs : [
    Node LINGI2242
    Node SOUS MODULE OBLIGATOIRE
    Childs : [
      ]
    ]
  ]
]
(...)
```

Lorsqu'on parse le fichier ; on devra simplement extraire les informations comme précédemment. On saura par contre au moment où l'on parse un objet à quel parent il appartient. Il n'est donc pas nécessaire de retraiter tous les éléments pour compléter les informations à propos de leur parent et de leurs enfants. Ceci est la principale raison pourquoi *Graphml* est le format supporté par l'application.

### Parsing

Le but de ce module est de fournir une abstraction supplémentaire à la gem *Nokogiri*<sup>1</sup> pour extraire les informations contenues dans le fichier *GraphML* de yEd. Ce module va, à partir des informations contenues dans le graphe, créer un objet du modèle Catalogue. Les informations contenues dans le graphe généré avec *yEd* sont les suivantes :

- les Programmes de cours (Bachelier, Masters) ;
- les Modules et leur nom ;
- les Sous-modules et leur nom ;
- les cours et leur sigle ;
- les contraintes hiérarchiques (Qui contient quoi) ;
- les dépendances entre les cours (Corequis et Prérequis).

Ce module est appelé par le modèle *Catalogue* de l'application à sa création. Le fichier graphe est d'abord envoyé sur le *cloud amazon*, puis parsé par le module *GraphParser*. Une fois le parsing terminé, les différents objets (Cours, Modules et Programmes) sont récupérés par le modèle *Catalogue*, puis traités par chacun des modèles concernés, avant d'être enregistrés en base de données.

---

1. Librairie ruby permettant de parser des fichiers XML

Tous ces éléments sont représentés par des nœuds dans le fichier GraphML. Seul les dépendances entre les cours sont représentées par des arrêtes dans le graphes, nommées *Edge* dans le fichier. Les nœuds sont stockés en premier dans le fichier de graphe, suivis par toutes les arrêtes.

Le module ajoute deux fonctionnalités à *Nokogiri*.

1. Il parse un fichier GraphML et extrait les métadonnées de ses nœuds (type, enfants, parent) et arrêtes (source, destination, type de contrainte, type de l'ensemble de contraintes).
2. Il renvoie des objets cours, modules et programmes ainsi que leurs différentes dépendances.

L'algorithme de parsing est illustré sur l'image 4.7. Les grandes lignes de cet algorithme sont les suivantes :

- les noeuds sont soit des noeuds groupes, soit des noeuds isolés ;
- les noeuds correspondant au programmes ont comme parent le noeud racine (les modules ont comme parent un programme)
- les contraintes n-aires sont représentées par des noeuds (pas par des arrêtes) ;
- chaque fois que l'on parse un noeud, on extrait son nom et son id de et on l'ajoute dans l'objet tout juste créé correspondant (cours, programme, module, contrainte n-aire) ;
- lorsque l'on parse une arrête (qui correspond à une contrainte binaire), on extrait son id source et destination pour identifier sur quels objets elle s'applique.

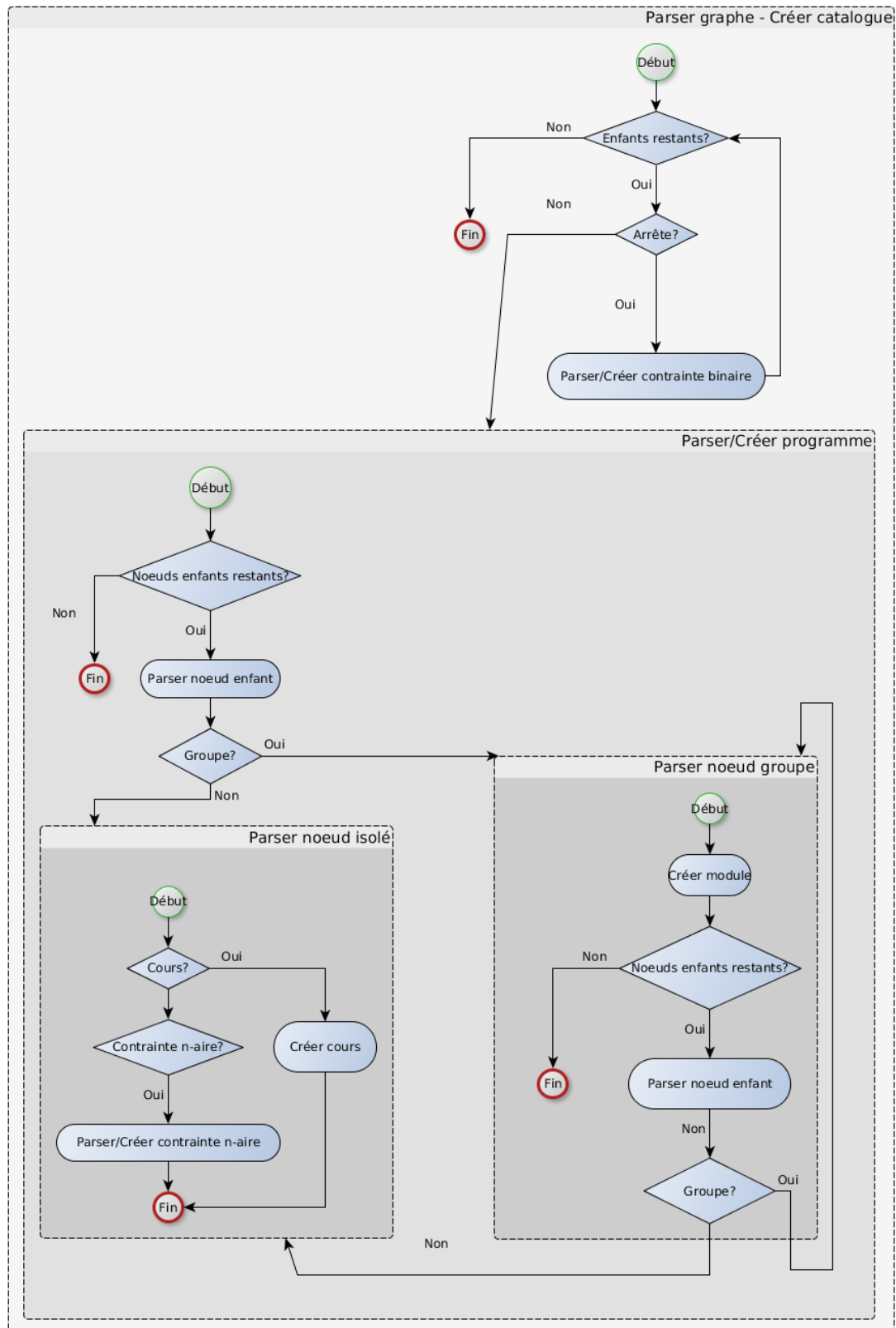
Une fois le parsing terminé, toutes les informations contenues dans chacun des **objets** cours, modules et programmes et contraintes sont ajoutées dans la table correspondante en base de donnée.

#### 4.3.5 Gestion des contraintes

Les contraintes sont vérifiées au niveau du modèle *StudentProgram*, le modèle qui contient les informations à propos des programmes de cours des étudiants (C'est ce modèle qui appelle le module *ConstraintsChecker*).

Le module *ConstraintsChecker* est divisé en deux parties. D'un coté nous avons les contraintes. Tout les types de contraintes héritent de la super classe *Contrainte* qui,

FIGURE 4.7 – Algorithme de parsing





en plus de constructeur ne contient qu'une seule méthode **Check**. De l'autre côté, nous avons les entités qui représentent les modèles *Course*, *Module* et *Program*

Les grands principes du vérificateur de contraintes sont illustrés sur le diagramme 4.8 à savoir ;

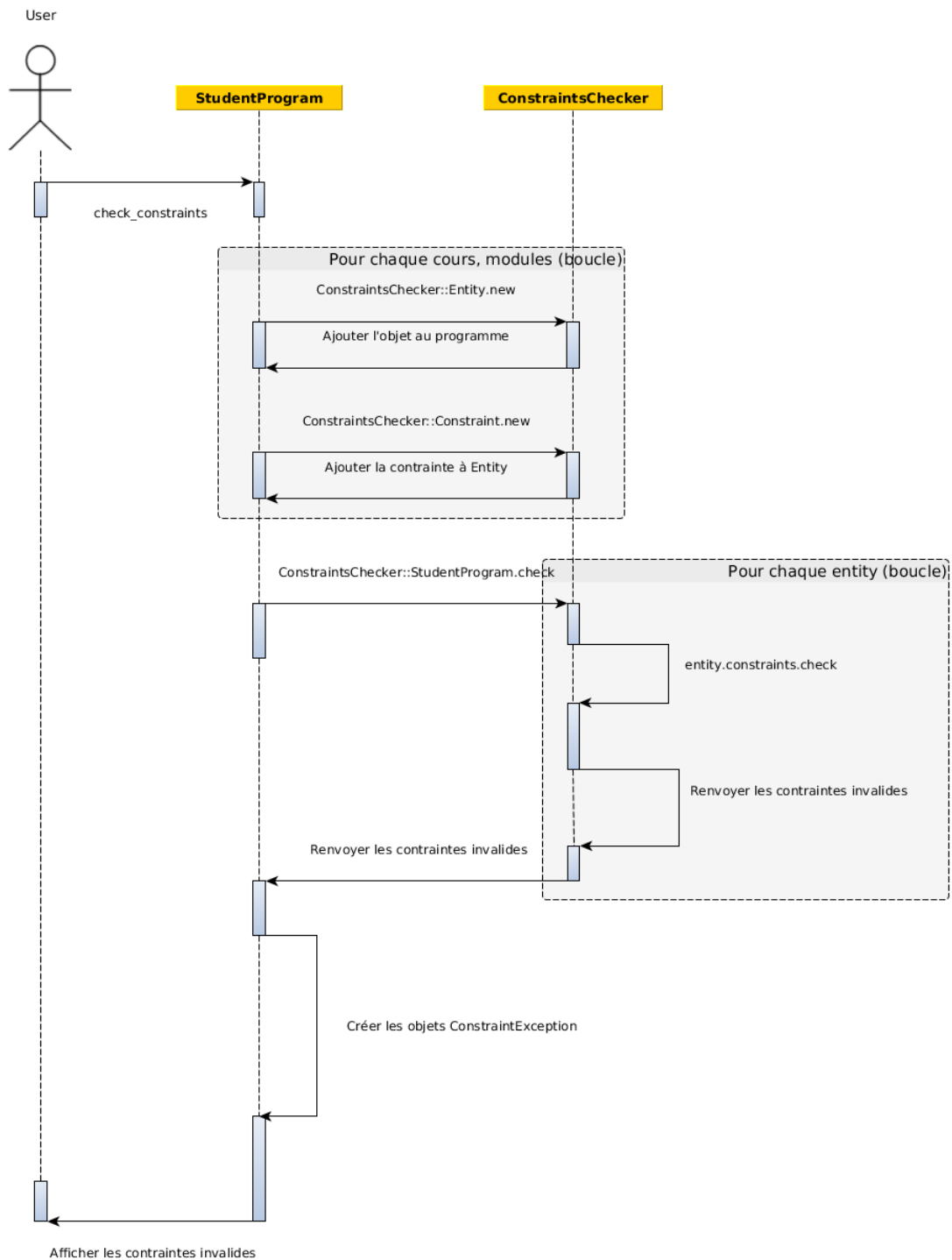
1. lorsque l'étudiant veut vérifier les contraintes de son programme la méthode *check\_constraints* est appelée ; cette méthode crée, pour chacun des éléments qui constitue son programme (cours, modules) un objet *ConstraintChecker.Entity* qui va contenir toutes les informations nécessaires à la vérification des contraintes ; la structure qui contient tout ces objets et un arbre, dont la racine est l'objet *ConstraintsChecker.Entity StudentProgram*
2. chaque modèle de la base de donnée (*Course*, *PModule*) contient une méthode *get\_(...)\_object* qui va créer l'objet *Entity* correspondant ;
3. cette méthode, s'occupe aussi d'ajouter à l'objet *Entity* les contraintes qui le caractérisent ;
4. une fois tout les objets créés, la méthode *check* de l'objet *ConstraintChecker.Entity.StudentProgram* est appelée ; cette méthode appelle la méthode *check* de chacun de ses enfants et récupère les contraintes invalides ;
5. une fois la vérification terminée, un objet *ConstraintException* est créé pour chaque contrainte non vérifiée ; on pourra ainsi avoir accès aux informations de chacune de ces contraintes dans la vue à laquelle l'étudiant a accès pour vérifier ses contraintes, et surtout attacher une justification à cette *exception* pour négocier la validation du programme avec la commission de programme.

### Ajout d'un nouveau type de contraintes

Pour ajouter un nouveau type de contraintes, il faut procéder comme suit :

1. si le type de la contrainte ne rentre pas dans la catégorisation des contraintes déjà existantes (*BinaryConstraint*, *PropertyConstraint*, *NaryConstraint*), il faut créer une nouvelle classe. Sinon, il suffit d'étendre la classe existante ;
2. implémenter la méthode *check* de cette contrainte avec le comportement désiré ;

FIGURE 4.8 – Vérification des contraintes



3. dans la méthode *get\_object* du modèle concerné par la contrainte, créer et ajouter l'objet contrainte et ajouter les informations nécessaires dans l'objet créé par le modèle ; par exemple, si je rajoute une contrainte sur les crédits, il faut passer le paramètre *credits : value* au constructeur de l'objet *Entity : :Course*.

#### 4.3.6 Importation du formulaire Excel

Le module est composé de deux parties :

1. un *Reader* qui propose une fonction pour récupérer sous forme de tableau de *Hash* les informations d'une page Excel, en lui fournissant **le nom de la page**, ainsi que **la propriété qui est utilisée pour identifier l'objet** (Le sigle pour les cours par exemple) ;
2. un *Writer* qui propose une fonction pour écrire des données dans une page d'un document Excel.

L'intérêt de fournir une abstraction supplémentaire se situe sur la structure des documents échangés avec l'utilisateur. En effet, chaque document comporte plusieurs pages. Chacune d'entre elles contient des informations sur un des objets (Course, Sub-Module, Modules ou Program). Ces informations sont représentées par le Modèle *Property* en base de données. Il est donc nécessaire d'avoir la première ligne de chacune de ces pages réservée pour y mettre le header afin de savoir pour chaque ligne à quel type de propriétés l'information appartient.

Pour les cours par exemple, ce header est de la forme :

Sigle	Crédits	...	...
-------	---------	-----	-----

Ici, il n'a pas été nécessaire d'utiliser une abstraction *Entity*, contrairement aux autres modules (GraphParser, ConstraintsChecker), pour représenter les données. En effet, nous ne manipulons que des tableaux de données, et surtout nous n'avons pas à nous occuper des inclusions entre les différents objets, ce module traitant exclusivement leur propriétés. C'est pourquoi ce module et l'application s'échangent des *Hash*.

Le *Writer* est appelé lorsque l'utilisateur télécharge un template de formulaire Excel après avoir créé le catalogue.

Le *Reader* est appelé à chaque fois que l'utilisateur met à jour les données d'un catalogue de cours via le formulaire Excel.

Notez que ce fichier est stocké, tout comme celui contenant le graphe, sur le cloud *Amazon*.

## 4.4 Qualité

Dans cette section, il sera question de la qualité du code qui a été produit pour développer l'application.

### 4.4.1 Maintenabilité

Tout au long du développement, les bonnes pratiques de *Ruby on rails* ont été suivies à la lettre à savoir :

- éviter d'avoir de la logique dans les vues et les alléger au plus possible ;
- alléger au plus possible les contrôleurs ;
- déléguer au plus possible le travail aux modèles.

Cependant, avec cette pratique, on se retrouve très vite avec de larges modèles. Avoir des larges classes est un comportement que l'on veut éviter, afin de renforcer la cohésion de celles-ci et diminuer leur couplage. C'est pourquoi certaines fonctionnalités ont été implémentées dans des modules externes (*ConstraintsChecker*, *GraphParser*, *XlsParser*).

À l'avenir, si l'on souhaite changer la façon dont on importe la structure d'un catalogue de cours par exemple, il suffira juste de changer l'appel au module (méthode *import\_graph*) dans le modèle *Catalog*. Plus simplement, si l'on souhaite importer des informations supplémentaires lorsque l'on importe ce graphe, il suffit de se rendre dans le module correspondant et non partir à la recherche du bout de code dans un modèle dont la taille se chiffrerait en milliers de lignes de code.

Au niveau de l'optimisation, la gem *bullet* a été utilisée pour détecter et corriger les endroits où apparaissaient des  $N+1$  queries.

Les  $N+1$  queries sont un ensemble de requêtes qui pourraient être réduites à une seule. Les accès à la base de données étant coûteuses en ressources, il est primordial de les

réduire au minimum afin de garantir une expérience fluide lors de l'utilisation de l'application.

Lorsque l'on récupère un module et ses cours par exemple, il est considérablement plus rapide de récupérer toutes les données à l'aide d'une requête (à l'aide de la méthode *includes*, plutôt que de récupérer le module, puis d'itérer sur celui-ci et de récupérer les cours un à un.

#### 4.4.2 Tests

La plupart des tests bas niveaux ont été réalisés dans les modules (*ConstraintsChecker*, *XlsParser*, *GraphParser*) à l'aide de RSpec. RSpec est un outil de testing orienté *Behaviour-Driven Development*. Il a été utilisé en définissant les exigences d'une fonctionnalité, avant de commencer le développement de celle-ci.

Pour les tests hauts niveaux des fonctionnalités, des scénarios ont été créés et testés en temps réel. Chacun de ces scénarios, ainsi que leur résultat est décrit dans le chapitre Validation 5

### 4.5 Conclusion

Ce chapitre clos l'explication et la description des fonctionnalités de la solution. Le chapitre qui suit analysera la qualité des fonctionnalités de l'application, à l'aide de scénarios, afin d'identifier ses limites et de pouvoir proposer, dans le dernier chapitre, des pistes de travaux futurs pour la solution.



## Chapitre 5

# Validation

### 5.1 Introduction

Le but poursuivi par ce chapitre est de tester les fonctionnalités de l'application à travers des scénarios spécifiques mettant en exergue certaines fonctionnalités importantes de l'application, comme la création d'un programme de master par un étudiant ayant un programme de bachelier complété et validé par la commission ou encore la transition de l'année de master 1 vers celle de master 2 avec une mise à jour de programme.

Les scénarios seront tout d'abord décrits, puis exécutés pas à pas et leurs résultats présentés en détails.

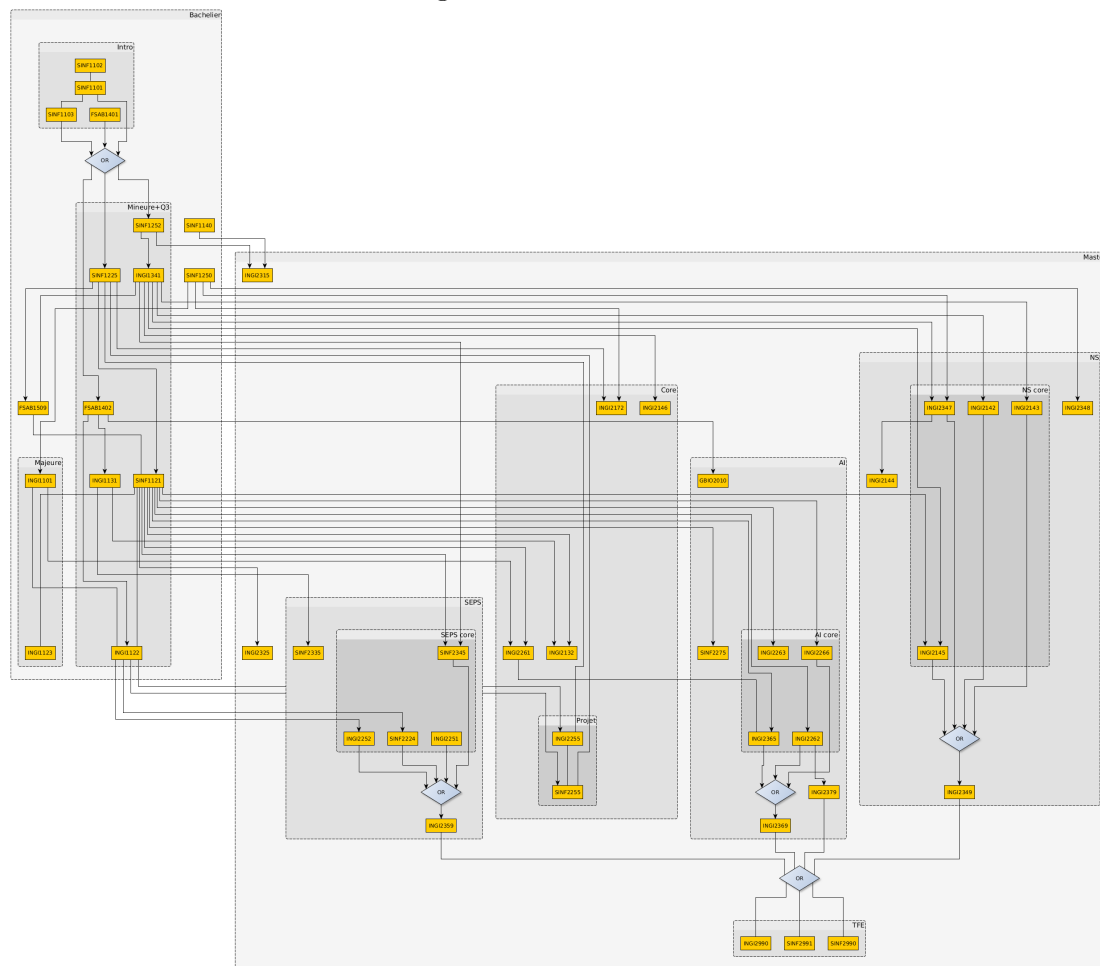
### 5.2 Scénarios

#### 5.2.1 Situation initiale

La situation initiale des deux scénarios qui suivent est la suivante :

L'étudiant aura créé et fait validé par la commission un programme de bachelier standard en sciences informatiques. Cette situation initiale est nécessaire car le programme de master qui est utilisé pour les deux scénarios suivant comporte des dépendances dans le programme de bachelier

FIGURE 5.1 – Programme situation initiale - 2013-2014



### 5.2.2 Catalogue utilisé

Le graphe utilisé pour générer le catalogue de la situation initiale sera le suivant 5.1. Notez que le programme de mineur proposé aux ingénieurs a été retiré volontairement pour rendre l'exemple plus lisible.

Après avoir importé le graphe, et ajouté les informations suivante dans le catalogue (via le formulaire Excel) :

- tous les cours sont à 5 crédits, sauf les 3 mémoires (SINF2990 et INGI2990 à 28 crédits, SINF2991 à 18 crédits) ;
- tous les cours sont alternés entre le premier et le second semestre, sauf les 3 mémoires qui sont marqué pour le second semestre ;
- on marque le module core (Master) et le module intro (Bachelier) comme obligatoire ;
- on ajoute un minimum de crédits à 20 et un maximum à 30 pour les trois options



du programme de master ;

- on ajoute un minimum de crédits à 40 et un maximum à 80 pour le programme de bachelier ;
- on ajoute un minimum de crédits à 120 et un maximum à 150 pour le programme de master.

Après avoir ajouté les informations, on marque le catalogue de cours comme principal, afin qu'il soit accessible aux étudiants.

### Configuration du programme de bachelier - Année 2006 à 2009

Lorsque l'on configure le programme de bachelier, on remarque que le programme tel qu'il est conçu dans le graphe n'est pas *validable* sur trois ans. En effet, il existe des **chaines de prérequis** dont la taille (en nombre de prérequis) est plus grande que trois, il faut donc étaler un programme sur plus de trois ans pour qu'il respecte les contraintes telles qu'elles sont présentées dans ce graphe, Une première amélioration à apporter à l'application est de détecter ces inconsistances lors de l'import du graphe dans l'application

Le programme ainsi configuré est validé par la commission. L'étudiant peut donc dès lors créer son programme de Master.

### 5.2.3 Premier scénario - Année académique 2009-2010

L'étudiant va configurer son programme de master en :

- choisissant les cours qu'il va suivre durant l'année académique 2009-2010 ;
- configurant l'année académique 2010-2011 pour s'assurer qu'il est possible de valider le programme dans lequel s'engage l'étudiant ; cette année sera modifiable par la suite ;
- choisissant les modules optionnels qu'il compte suivre dans son programme.

Le programme de master comporte un module obligatoire (Core) et plusieurs modules optionnels (Génie logiciel, Intelligence artificiel et Réseau). L'étudiant décide de suivre l'option **Génie Logiciel** (SEPS sur le graphe). Ce module optionnel comporte un sous-module obligatoire (SEPS core)

Les cours suivis durant la première année (2009-2010) sont les suivants :

Cours	Crédits	Semestre
INGI2315	5	1
INGI2255	5	1
INGI2172	5	1
INGI2146	5	1
SINF2345	5	1
SINF2335	5	1
INGI2325	5	2
INGI2132	5	2
INGI2261	5	2
INGI2263	5	2
SINF2275	5	2

Les cours que l'étudiant prévoit de suivre durant la seconde année (2010-2011) sont les suivants :

Cours	Crédits	Semestre
SINF2255	5	1
INGI2262	5	1
GBIO2010	5	1
INGI2251	5	1
SINF2224	5	1
SINF2290	28	2
INGI2379	5	2
INGI2252	5	2
INGI2359	5	2

L'étudiant décide de suivre l'option *Software Engineering and Programming Systems (SEPS)*

Le programme de l'étudiant comporte une contrainte non respectée. En effet, il a décidé, durant la deuxième année, de suivre le cours INGI2379 (au second semestre) et son prérequis INGI2262 (au premier semestre) durant la même année, ce qui enfreint la règle du prérequis. Cependant, il estime que, vu qu'il suit le cours INGI2262 au premier semestre et le cours INGI2379 au deuxième semestre cela ne pose pas de problème.

Il remplit donc la case justification de cette contrainte non respectée en expliquant la situation, avant d'envoyer sa demande de validation.

L'étudiant va ensuite négocier la validation de son programme avec la commission de programme, puis celle-ci va valider son programme après quelques discussions.

La commission de programme reçoit la demande de validation, regarde son programme et voit que :

- le programme de master comporte assez de crédits (123) ;
- l'ensemble des modules obligatoires est présent ;
- une option est présente ;
- un prérequis est enfreint.

La commission de programme regarde la justification de l'étudiant (il est obligé d'en remplir une si son programme ne respecte pas toutes les contraintes). Elle regarde la justification, et refuse la demande de validation en demandant à l'étudiant de choisir un autre cours (ou bien de suivre le prérequis durant l'année précédente).

Ici, une première amélioration pourrait être apportée à l'application. Il faudrait permettre à la commission de marquer chaque justification de contrainte non respectée comme acceptée ou refusée. L'étudiant pourrait ainsi voir plus clairement les parties qui ne sont pas valides dans sa demande de validation.

L'étudiant retourne sur son programme, voit qu'il a un nouveau message en attente. Il choisit un autre cours (LINGI2143) et renvoie sa demande de validation qui est ensuite validée par la commission.

Ici, une seconde amélioration pourrait être apportée à l'application. L'étudiant n'est pas clairement averti que sa demande a été refusée. Tout ce qu'il voit, c'est que son programme n'est toujours pas validé et qu'il a la possibilité de renvoyer une demande de validation.

### 5.2.4 Second scénario - Année académique 2010-2011

L'année académique suivante, l'étudiant a réussi son année. La commission marque son année comme réussie. Cependant, la commission de programme ajoute une nouvelle version du catalogue de cours qui modifie le programme de master.

L'application n'a aucun problème pour récupérer les contraintes des anciens programmes de cours. Par contre, si l'on renomme des cours, elle ne pourra pas récupérer les anciennes instances des cours (la recherche se faisant sur le nom du cours). De plus, si ces cours ont déjà été suivis par des étudiants, et qu'ils jouent le rôle de prérequis ou de corequis pour des cours présents dans des années qui ne sont pas encore validées, les étudiants concernés auront à justifier ces dépendances manquantes dans leur programme.

C'est pourquoi, la modification se portera essentiellement sur le nom de cours qui sont des dépendances pour des cours choisis dans la seconde année académique de l'étudiant.

On déclenchera ainsi un processus de négociation dans lequel l'étudiant expliquera qu'il a déjà suivi les anciennes versions des cours. De plus, l'application, pour le moment ne gère pas la récupération des anciens modules choisis. L'étudiant doit à nouveau choisir le module correspondant.

La modification apportée au catalogue sont les suivantes :

- renommage du cours *INGI2252* en *INGI2252BIS* ;
- renommage du cours *SINF2224* en *SINF2224BIS* ;
- renommage du cours *INGI2251* en *INGI2251BIS* ;
- renommage du cours *SINF2345* en *SINF2345BIS*.

La commission crée un nouveau catalogue de cours comportant ces modifications, avec le même nom que le précédent, mais en changeant l'année académique qui l'identifie.

Lorsqu'une version plus récente du catalogue de cours utilisé par l'étudiant est importé dans l'application, il est proposé à celui-ci de mettre à jour son programme. L'étudiant va donc, au début de l'année académique 2010-2011 mettre à jour son programme de cours, avant de configurer sa dernière année et de ré-envoyer son programme à la validation.

L'étudiant est invité, lorsqu'il se rend sur la page de configuration de son programme de cours, à migrer le programme de cours qu'il suit vers sa nouvelle version.

L'étudiant se retrouve donc avec des dépendances manquantes liées au cours qui ont changé de nom.

L'étudiant n'a donc qu'à expliquer qu'il a bien suivi ces cours. La commission peut quant à elle facilement vérifier de son côté que les cours ont bien été suivis lors des années précédentes. Elle n'a plus ensuite qu'à valider le programme de l'étudiant.

### 5.3 Conclusion

Les scénarios présentés sont assez complets que pour illustrer la plupart des fonctionnalités de l'application. Il ont permis d'identifier certaines fonctionnalités qui manquent encore à l'application à savoir :

- afficher plus clairement à l'étudiant que sa demande de validation a été refusée ;

- permettre à la commission de programme d'accepter ou de refuser chaque message de justification de l'étudiant pour qu'il puisse savoir quelles justifications ne sont pas suffisantes.

Le chapitre suivant présentera des pistes de travaux futurs.



## Chapitre 6

# Travaux futurs

### 6.1 Introduction

Bien que l'application, dans l'état où elle se trouve, soit assez conséquente en terme de fonctionnalités, un certains nombre d'améliorations pourrait être apportées.

### 6.2 Service mail

Pour le moment, les communications entre la commission de programme et les étudiants se font en interne via l'application. Ajouter un service mail à l'application permettrait de gérer de façon plus souple la couche utilisateur, en permettant notamment de se connecter directement à l'application avec son compte mail UCL.

De plus, l'utilisateur pourrait être notifier des changements par mail. Par exemple, la commission pourrait être tenue au courant par mail des nouvelles demandes de validations de la part des étudiants. Les étudiant quand à eux pourrait être alerté lorsque leur demande de validation a été acceptée ou refusée, ou si la commission a demandé des informations supplémentaire pour une justification quelconque.

Un service mail n'est pas très complexe à mettre en place, surtout avec Rails. En effet, il existe une librairie externe (gem) appelée *mailjet* qui permet en quelques lignes de configuration de communiquer avec le service d'envoi de mail du même nom.

## 6.3 Gestion des incohérences

L'application, pour le moment n'envoie que très peu de feedback à la commission de programme lorsque celle-ci importe un graphe ou un formulaire Excel dans l'application. Or, si on ne fait pas attention, il est facile d'ajouter des incohérences dans les données. Par exemple si on ajoute une chaîne de prérequis dont la taille (le nombre d'arêtes) est plus grande que la durée normale (le nombre d'années académiques) d'un programme, on se retrouve avec un programme qui n'est pas réalisable.

Avec le formulaire Excel, des incohérences peuvent apparaître si l'on se trompe dans les contraintes minimum et maximum de crédits. Rien qu'en ne mettant pas assez de cours que pour valider la contrainte minimum, ou encore en mettant des modules dans un programme, dont la somme des contraintes minimum dépasse la contrainte maximum de ce programme

C'est pourquoi, il faudrait ajouter des vérifications à l'import, qui empêchent l'ajout de ce genre d'incohérences, détecteraient leur position et les notifieraient à l'utilisateur

## 6.4 Mise à jour du graphe

Une fois le graphe importé dans l'application, il n'est plus possible de modifier ce graphe, pour modifier la structure du catalogue. La seule façon, pour changer la structure du catalogue (les différentes dépendances, à quel module appartient quel cours, ...) est de recréer un nouveau catalogue avec les modifications.

Une amélioration serait de rendre possible ces mises à jour.

## 6.5 Intégrer le logiciel de graphe dans l'application

Comme expliqué dans la section précédente, il n'est pas possible pour le moment de mettre à jour le graphe à l'origine d'un catalogue de cours dans l'application. De plus, comme expliqué dans la section 3.3.2, l'utilisation de yEd nous force à importer nos données en deux étapes (import de graphe et import de formulaire Excel).



Une fonctionnalité intéressante que l'on pourrait ajouter à l'application serait d'intégrer un outil permettant de construire des graphes directement dans l'application, nous évitant ainsi les deux étapes de parsing. Qui plus est, ce logiciel pourrait être totalement adapté à notre besoin, à savoir construire un graphe de cours.

## 6.6 Amélioration de l'interface

L'interface de l'application reste encore perfectible. La liste (non exhaustive) suivante énumère les améliorations qui pourrait être apportées à l'interface utilisateur :

- dans la vue qui permet aux étudiants de vérifier la validité de leur programme, il faudrait pouvoir rajouter les cours manquants directement depuis cette interface, à l'aide d'une requête javascript par exemple ;
- lorsque la demande de validation d'un étudiant a été refusée, cette information pourrait être affichée de façon plus intuitive ;
- pouvoir, lorsque la commission de programme gère les justifications d'un étudiant, marquer directement celles qui sont acceptées, et celles qui sont refusées.
- (...)

## 6.7 Fonctionnalités diverses

Empêcher l'étudiant de pouvoir modifier son programme de cours après une certaine date.

## 6.8 Conclusion

Malgré que la liste des améliorations soit assez longue, l'application propose une fonctionnalité pour chacun des objectifs qui ont été mis en avant dans les sections 1.3 et 1.4.

Un des objectifs principaux de cette application étant de fournir une solution flexible et maintenable, la liste ci-dessus pourrait fournir des pistes d'extensions de cette application à un jobiste ou à un mémorant l'année prochaine, l'architecture de la solution étant assez souple pour permettre ce genre de modifications.



## Chapitre 7

# Conclusion

Une solution, sous forme de fonctionnalité, a été apportée pour chacun des objectifs mis en avant dans les prémices de ce mémoire. Cependant, il faut garder à l'esprit que la solution présentée dans ce mémoire est le résultat d'une première itération de développement. Il serait insensé de penser que l'application et chacune de ses fonctionnalités se trouve dans un état de plénitude, la section 6 en témoigne.

Néanmoins, l'objectif principal de ce mémoire, à savoir - *proposer une solution qui automatise la gestion des programmes de cours afin de soulager les commissions de programmes de la charge de travail que constitue la création, maintenance et vérification des programmes de cours* - a été rempli avec succès.



# Bibliographie

- [de Louvain(2013-2014a)] Université Catholique de Louvain. Bachelier en sciences informatiques - sinf1ba, 2013-2014a. URL <http://www.uclouvain.be/prog-2013-sinf1ba>.
- [de Louvain(2013-2014b)] Université Catholique de Louvain. Bachelier en sciences de l'ingénieur, orientation ingénieur civil - fsa1ba (majeure et mineure info), 2013-2014b. URL <http://www.uclouvain.be/prog-2013-fsa1ba>.
- [de Louvain(2013-2014c)] Université Catholique de Louvain. Master [120] en sciences informatiques - sinf2m, 2013-2014c. URL <http://www.uclouvain.be/prog-2013-sinf2m>.
- [de Louvain(2013-2014d)] Université Catholique de Louvain. Master [60] en sciences informatiques - sinf2m, 2013-2014d. URL <http://www.uclouvain.be/prog-2013-sinf2m1>.
- [de Louvain(2013-2014e)] Université Catholique de Louvain. Master [120] : ingénieur civil en informatique - info2m, 2013-2014e. URL <http://www.uclouvain.be/prog-2013-info2m>.
- [de Louvain(2013-2014f)] Université Catholique de Louvain. Année d'études préparatoire au master en sciences informatiques - sinf1pm, 2013-2014f. URL <http://www.uclouvain.be/prog-2013-sinf1pm>.
- [Helmkamp(2012)] Bryan Helmkamp. 7 ways to decompose fat activerecord models, 2012. URL <http://blog.codeclimate.com/blog/2012/10/17/7-ways-to-decompose-fat-activerecord-models/>.
- [Casts(2012)] Rails Casts. Migrating to postgresql, 2012. URL <http://railscasts.com/episodes/342-migrating-to-postgresql>.

- [Miller(2008)] Jeremy Miller. Cohésion et couplage, 2008. URL <http://msdn.microsoft.com/fr-fr/magazine/cc947917.aspx>.
- [Wang(2013)] Eugene Wang. How (and when) to use single table inheritance in rails, 2013. URL <http://eewang.github.io/blog/2013/03/12/how-and-when-to-use-single-table-inheritance-in-rails/>.