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

| | |
|---|---|
| **FWB** | **F**édération **W**allonie - **B**ruxelles |
| **Commission INFO** | Commission de Programme du département d'informatique de l'école polytechnique de Louvain -là-Neuve |
| **Module** | Ensemble de cours, obligatoire ou non<br>Ex : L'option intelligence artificielle dans le programme de master |
| **EPC** | (...) |
| **Rails** | Ruby On Rails |

# Chapitre 1

# Introduction

## 1.1 Contexte

Chaque année à l'Université catholique de Louvain, et dans toute 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 restreint 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 de programme du département d'informatique de l'École Polytechnique de Louvain - (**La commission INFO**) - et ses étudiants.

En outre, la fédération Wallonie - Bruxelles ($FWB$) a réalisé un décret qui a un impacte

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 entres 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 mets 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

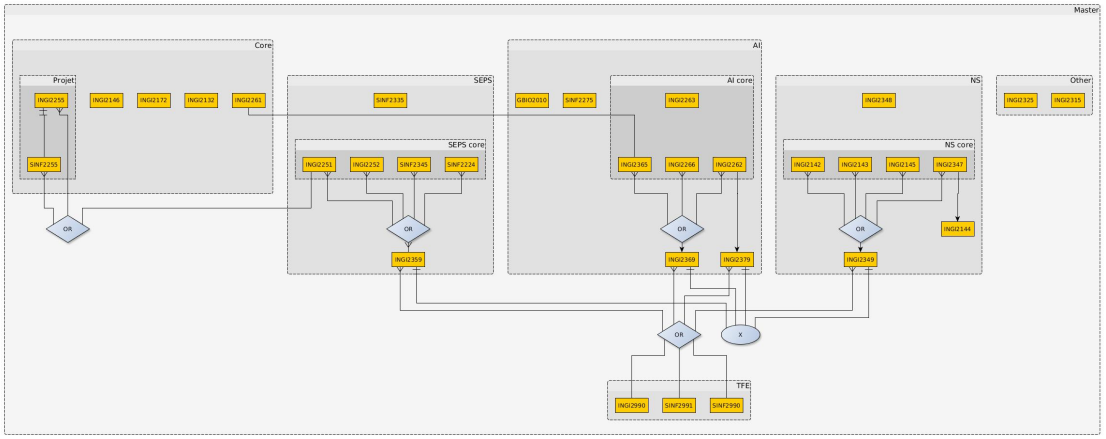
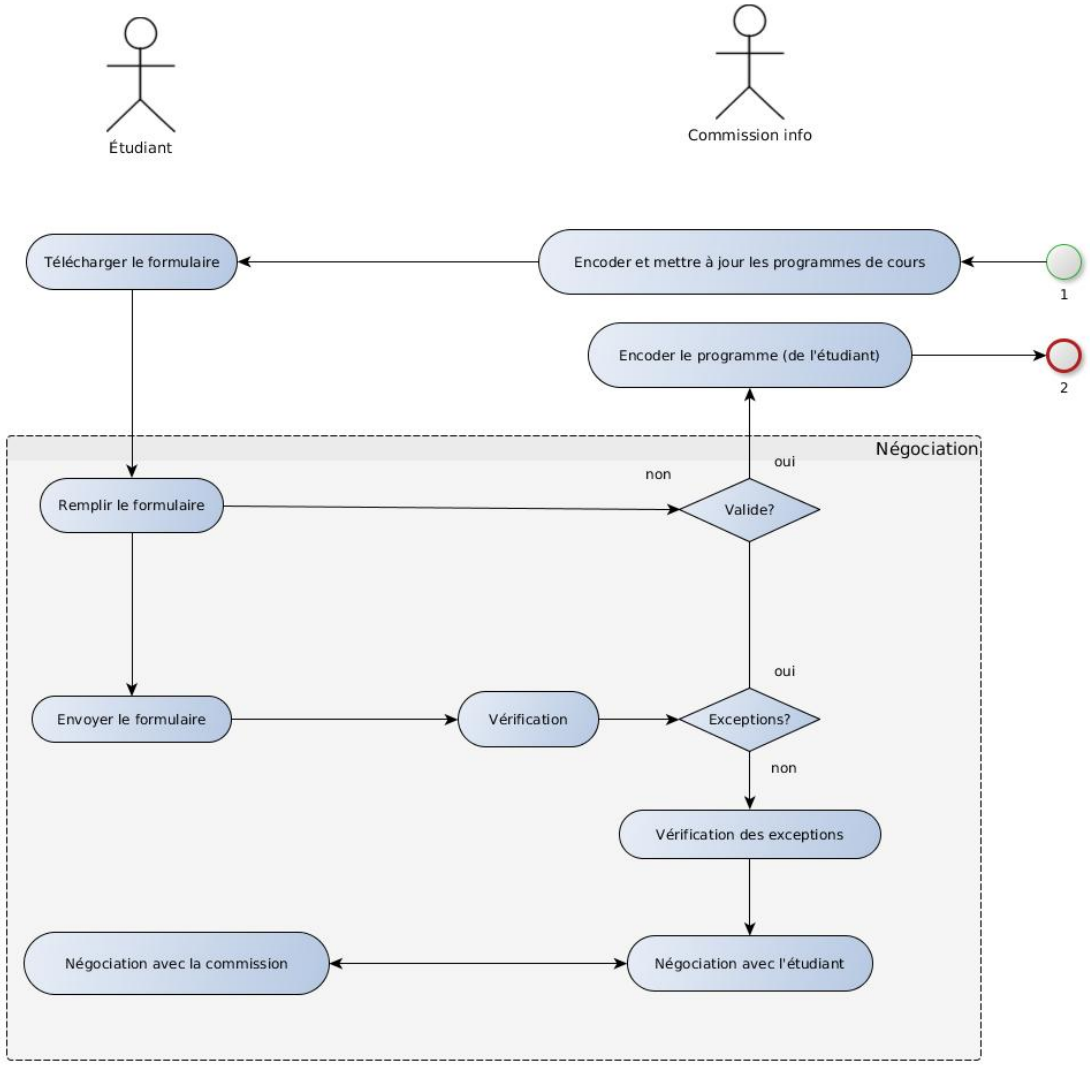



FIGURE 1.2 – Processus actuel



*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 obligatoire 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 suivit 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 d'amples informations à l'étudiant ou négocier avec lui et, si nécessaire, il est demandé à l'étudiant de compléter sont formulaire avec les informations qu'il manque. 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 nécessite 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 profile.

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

- les choix faits par les étudiants au niveau de leur programmes 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 suivit.

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 à 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 êtes 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 ceux-ci peuvent être ignorés 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 autre 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 en-codables, 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 au données. Qui plus est, il n'ont pas le temps de changer leur processus et ils s'intéressent peu, par manque de ressources, à la facilité d'utilisation. Mieux vaux éviter l'utilisation direct de EPC pour le moment, jusqu'à ce que notre outil aie un certain succès. Après quoi, peut être l'outil pourrait échanger des données avec EPC ou même être intégré, mais ce ne sera pas pour immédiatement. 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.

Les 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 aux 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 programmes. 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

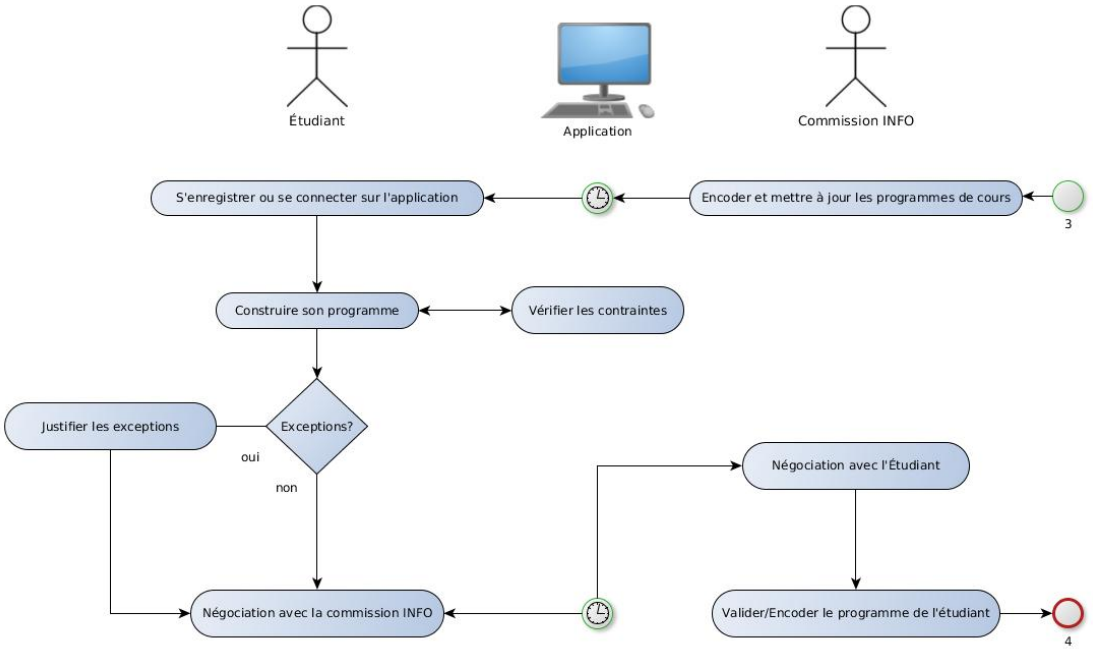
- se créer un compte utilisateur avec son adresse mail UCL ;
- sélectionner la version du catalogue de cours avec laquelle il va travailler ;
- se créer un programme en choisissant un des programmes de cours disponibles à suivre ;
- choisir les différents modules de cours à suivre, option, tronc commun, ... ;
- 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 suivit 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 ;

FIGURE 1.3 – Processus désiré




La *commission INFO* importe et mets à jour les différents programmes de cours directement dans l'application.

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 INFO* récupère la requête de validation. L'application lui montre les contraintes qui ne sont pas respectée. La *commission INFO* regarde ensuite les justifications de l'étudiant. Si elle ne sont pas suffisante, 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 se répète jusqu'à ce que le programme de l'étudiant soit 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 dans le département d'informatique est la suivante :

**Bachelier en sciences informatiques - SINF1BA** [5] - C'est un programme de 180 crédits. Comme dans tout programme de bac, 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)** - [4]. 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 se programme

**Master [120] en sciences informatiques - SINF2M** [7] - Ce programme de 120 crédits est destiné aux étudiants en provenance du programme *SINF1BA*. Il comporte un module obligatoire (le tronc commun) et la possibilité de choisir une 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 - SINF2M1** [8] - Ce programme alternatif de 60 crédits est destiné aux étudiants en provenance du programme *SINF1BA*. 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 *SINF2M* : 15 crédits. La durée normale de ce programme de master est d'un an.

**Master [120] : ingénieur civil en informatique - INFO2M** [6] - Ce programme est destiné aux étudiants en provenance du programme *FSA1BA* ayant suivit soit la mineure soit la majeure en informatique. Comme dans le master *SINF2M*, 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 - SINF1PM** [3] - Ce programme est destiné aux étudiants en provenance de Hautes écoles d'informatique. Il permet d'accéder aux programmes *SINF2M* et *SINF2M1*. 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 choisi parmi ceux proposé dans le programme *SINF1BA*. 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 suivit dans université d'origine et ceux proposé 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 coté 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 réussi (dans des années académiques antérieure) afin de pouvoir suivre ce cours
   (b) Les corequis : Les corequis d'un cours sont les cours qu'il faut avoir suivit **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 suivit 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.

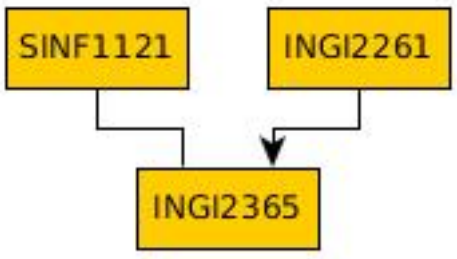



FIGURE 2.1 – Dépendances du cours INGI2261

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 exprimés en terme de semestre. Pour des raisons variables, comme un professeur qui part à la retraite, ou qui prend une année sabbatique, elle 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 tout 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.

### 2.2.1 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 suivi 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 INFO 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 leur 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 un programme de cours, en choisissant le programme à suivre ;
   - configurer son programme de cours, en configurant les différentes années qui le compose et en choisissant les modules de cours à suivre,
   - vérifier la validité de son programme

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 ses programmes, modules, cours et des différentes contraintes par la *commission INFO* ;
- 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 un catalogue de cours fictif proposant deux programmes de cours. Ce programme est fictif mais ce rapproche très fort de la réalité. Certaines partie du catalogue (Certains modules optionnels du programme de master, certains cours) on été élaguée pour rendre l'exemple plus clair. L'exemple n'en reste pas 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.

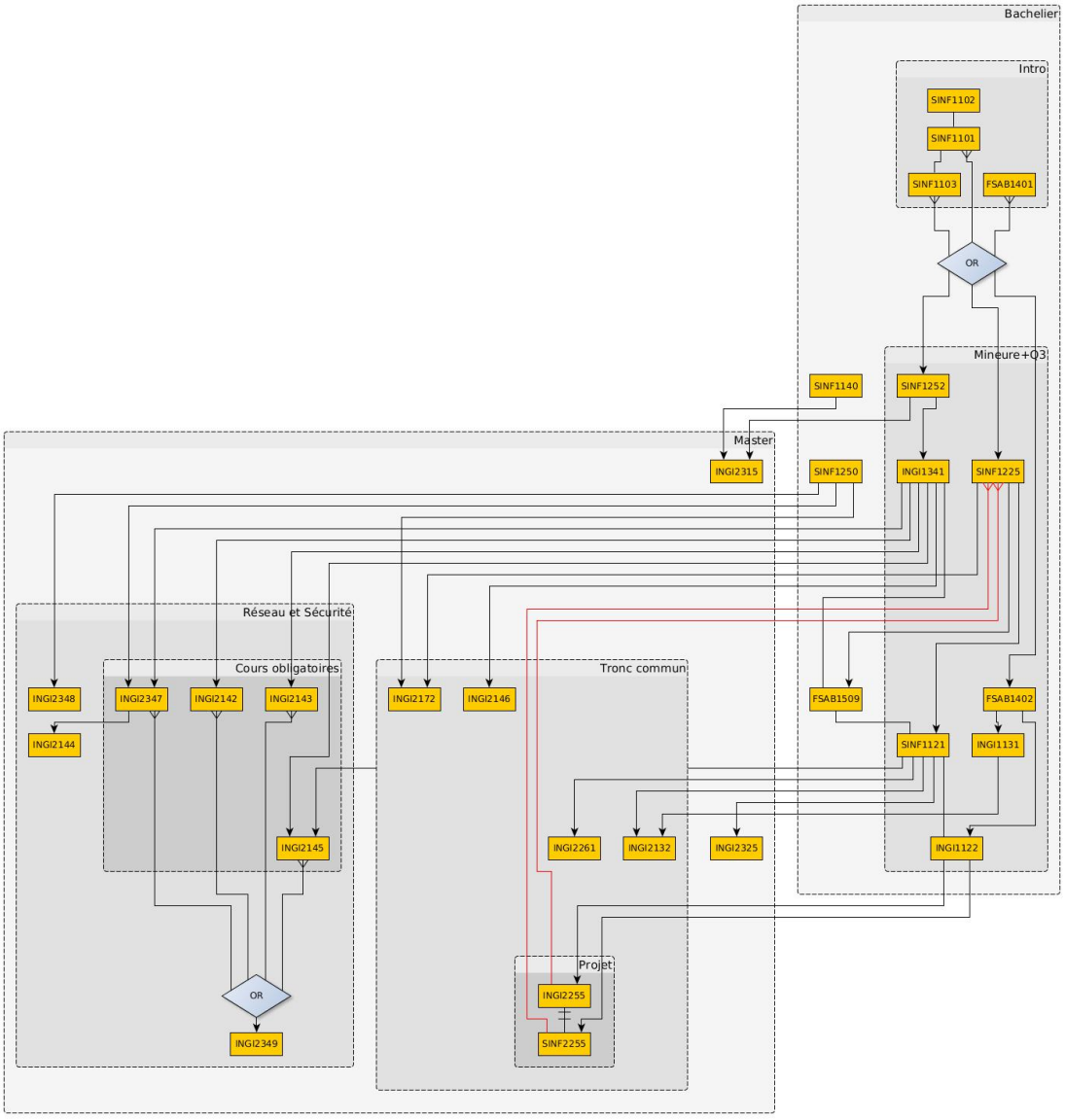
Un programme de **Bachelier** (le grand rectangle à droite de la figure 3.1) composé d'un module obligatoire (Introduction, la petite boite imbriquée s'appelant *Intro*)

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 boite imbriquée à gauche) et d'un module obligatoire (Le tronc commun, la boite imbriquée à droite). De plus, certains cours de master dépendent de cours de bac.

### 3.1.2

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 1.3. Il y a trois modules attachés à cette entité.

FIGURE 3.1 – Exemple fictif de catalogue de cours



**Le module d'import de donnée** - Il permet à la commission d'enregistrer 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 sont 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 empêche les étudiants d'envoyer leur programme à la validation s'il n'est pas correcte. il fourni aux étudiants un compte-rendu en temps réel de l'état de leurs programmes, en leur pointant les parties qui ne respecte pas les contraintes, quelles contraintes ne sont pas respectées et ce qu'il faut changer dans leur programme pour y remédier.

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

## 3.2 Technologies utilisées

### 3.2.1 Introduction

Le but poursuivit 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 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 codes.

### 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 me semble important d'en comprendre les grande 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 standart 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). Vient ensuite Ruby on Rails, un framework en ruby et Django, un framework en Python.

### 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 pattern 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 codes 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, envois 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.

### Limites

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*[1] et l'on oublie souvent qu'il est possible de déléguer la plupart des fonctionnalités à des librairies externes, qui sont plus faciles à développer - car crées en pure *ruby* - et plus facile à tester - car indépendantes de rails [10].
- (...)

### Conclusion

Le choix s'est naturellement porté Ruby on Rails. 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 object, ...) **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 bout de codes similaire un peu partout 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

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

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-Controlleur

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 toute la *business logic*, qui détermine comment l'information est créée, mise à jour, et affichée ;

**les vues** qui génèrent l'interface utilisateur et lui présente 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. 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
```

(source : [2]

Tout cela est fait sans devoir changer la façon dont j'accède aux 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 l'outil utilisé pour héberger l'application. En effet, il est nécessaire d'avoir une base de données PostgreSQL pour pouvoir héberger l'application sur la plateforme.

### 3.2.4 Éditeur de graphes - yEd

Comme expliqué plus tard dans la section gestion des données, le choix s'est porté vers une importation en deux parties des données dans l'application. Pour ce faire, il est

nécessaire de construire un graphe de cours pour représenter le catalogue contenant les différents programmes de cours proposés par la *commission INFO*.

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, et cela sort du cadre de ce mémoire

La deuxième alternative serait d'intégrer un outil générant des graphes plus standard dans l'application. YEd offre une solution très chère à ce sujet. Il y a aussi une librairie en Javascript assez souple (d3js ). Ces solutions implique quoiqu'il arrive de devoir parser les données par ces outils.

La troisième et dernière alternative serait d'utiliser un logiciel externe à l'application pour générer ces graphes. Ce logiciel doit être entre autre 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, Graphiz)

- 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.
- 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 imposé par le cadre du mémoire, le choix c'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 fallait parser le fichier exporté par l'outil pour importer les données dans l'application.

YEd a été choisi pour toutes ces raisons.

## 3.3 Conception

### 3.3.1 Contraintes

Les contraintes comme présentées dans les sous-sections qui suivent, sont de plusieurs type. Cette catégorisation est nécessaire pour plusieurs raisons.

Premièrement, elles ne sont pas représentées de la même façon de la même façon. En effet, certaines sont représentées par un object en base de données (Les dépendances entre les cours). D'autres, n'ont pas de représentation à proprement dite, dans le sens où elles n'ont pas un object appelé contrainte pour les représenter, car elles portent sur des champs de certains objets.

Deuxièmement, leur données ne sont pas importées de la même façon.

Enfin, elle ne sont pas vérifiées de la même manière. Ces différences, comme vous le verrez dans la partie Implémentation on justifié le choix d'implémenter cette gestion des contraintes dans un module externe à *Ruby On Rails*, pour permettre s'abstraire ces contraintes en un et unique object Contrainte, qui lui même connaît

- ses données. Par exemple, une contrainte spécifiant le minimum de crédits pour un module connaît l'objet sur lequel cette contrainte s'applique et la valeur de celle ci.
- la logique intrinsèque de cette contrainte. Toujours pour ce minimum de crédits requis, la méthode de vérification propre à ce type de contrainte sait qu'il faut calculer la somme des crédits des cours de l'objet Module sur lequel elle s'applique.

#### Dépendances

TODO AJOUTER IMAGES Ces contraintes portent sur les cours. Elle sont de deux types :

1. Les prérequis : exprimant les cours qu'il est nécessaire d'avoir suivit pour pouvoir suivre le cours
2. Les corequis : exprimant les cours qu'il est nécessaire de suivre durant la même année académique que le cours sur lequel la contrainte s'applique.

De plus, il existe aussi des disjonctions de contraintes, exprimant qu'il faut

- avoir suivit un des cours pour pouvoir suivre le cours en question (Conjonction de prérequis)
- suivre un des cours de la liste durant la même année académique que le cours en question (Conjonction de corequis)

Ces contraintes sont importées dans l'application via l'éditeur de graph yEd. Elle sont représentées par les arrêtes entre les différents nœuds dans le graphe.

#### Contraintes sur les propriétés

TODO : AJOUTER IMAGES Ces contraintes portent sur les programmes, les modules, les sous-modules et les cours. Elle représentent n'importe qu'elle type de contrainte qui pourrait exister sur les champs *Property* de ces objets. Un exemple serait les 180 crédits nécessaires à la validation d'un programme de **BAC**

Les données relatives à ces contraintes n'ont pas de représentation en tant que telles dans la base de données, c'est à dire qu'il n'y a pas d'objet *PropertyConstraint* qui les représente. La principale raison qui justifie ce choix est la suivante :

Ces contraintes portant sur les propriétés des objets, elles sont importées via le formulaire excel dans l'application. Or, il n'y a pas une contrainte par propriété, on ne peut donc pas créer une contrainte par propriété, cela n'aurait pas de sens. (On a par exemple, pour un cours, son sigle, son nom complet, le professeur, l'url du cours sur le portail de l'ucl, ses crédits, etc).

On pourrait créer une nouvelle page par objet dans le formulaire excel, pour représenter seulement les contraintes de l'objet en question. Cependant, il faudrait ajouter beaucoup de logique dans les modèles pour exprimer que tel propriété est une contrainte. Tout dépendrait des données que l'ont met dans le formulaire. Si la personne qui utilise l'application est fatiguée et se trompe dans le nom de la page, ou du nom de l'objet, ou encore du nom de la contrainte, la création de celle-ci ne fonctionnera pas. Pire, elle pourrait ajouter des contraintes correctes dans la forme, mais qui s'appliquent à d'autres objets. Il est très difficile de vérifier ce genre d'erreurs.

Ce genre de solution n'était pas du tout souple et source de beaucoup d'erreurs, il a été préférable de partir dans une autre direction.

1. Laisser ces contraintes exprimées implicitement dans la base de données (= Pas d'objet *PropertyConstraint*).

2. Délayer la logique de ces contrainte au module du même nom, en demandant au module de passés les informations nécéssaires à ces contraintes au module.

#### Contraintes temporelles

TODO

### 3.3.2 Gestion des Données

#### Introduction

Cette section répond à la question de comment sont importées les données relatives aux programmes, leur cours, leurs informations et leur différentes contraintes.

Les différentes informations contenue dans le catalogue de cours du département sont les suivantes. Nous avons

- Plusieurs programmes de cours (BAC SINF et INFO, MASTER SINF et INFO, Passerelle SINF et toute les options qu'elles contiennent)
- 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. (Voir section contrainte pour plus de détails)

Il faut donc trouver un moyen de télécharger ces informations dans l'application. La solution la plus naïve serait de fournir des formulaires pour chaque entité (Cours, modules, ...) permettant d'ajouter une à une toute les informations nécessaire. Si l'on veut modifier les informations d'une entité, *"il suffirait"* de naviguer dans les différents menu et de sélectionner le menu d'édition correspondant. Cependant cette solution n'atteint pas l'objectif fixé dans la solution, à savoir fournir un support pour enregistrer efficacement (et donc en peu de temps) les informations relatives aux curricula. Ajouter une à une toute les dépendances de cours peut être très ennuyant.

Une première amélioration que l'on peut ajouter à cette solution, est d'utiliser des formulaires excel pour ajouter ces informations. En créant une page par entité comme présenté sur l'image 3.2, on pourrait ajouter toute les informations liés au modules, aux cours plus efficacement, l'import de fichier excel étant une chose assez aisée.

FIGURE 3.2 – Feuille Excel pour importer les données relatives aux modules

| NAME | MIN | MAX | PARENT |
|---|---|---|---|
| SEPS | 5 | 15 | MASTER |
| TFE | 5 | 15 | MASTER |
| AI | 5 | 15 | MASTER |
| CORE | 5 | 15 | MASTER |
| NS | 5 | 15 | MASTER |
| OTHER | 5 | 15 | MASTER |
| MAJEURE | 5 | 15 | BAC |
| MINEURE+Q3 | 5 | 15 | BAC |
| OTHER | 5 | 15 | BAC |
| INTRO | 5 | 15 | BAC |
| APPROFONDISSEMENT | 5 | 15 | BAC |

Bien que cela permette d'ajouter des informations de façon plus efficaces, il subsiste plusieurs problèmes

- C'est toujours aussi éprouvant d'ajouter les dépendances. Il toujours les ajouter une par une, bien que cela soit sur une seule page.
- On ne peut pas donner au formulaire (et le programme qui l'importe) le pouvoir de créer les entités car :
  - En cas de faute de frappe, il faudra de nouveau naviguer dans l'application pour supprimer les erreurs.
  - Il est très difficile et peu efficace de gérer les inclusions des cours(L'appartenance d'une entité à une autre, un cours à un module par example). Cela implique des recherches sur le nom des entités. De nouveau, en cas de faute de frappe, il faudra aller corriger les erreurs dans l'application

Le catalogue de cours étant un graphe (Dont les nœuds sont des entités et les dépendances des arrêtes), une autre solution serait d'utiliser un logiciel qui permet d'en dessiner. Un graphe étant visuellement plus parlant qu'un tableur, le nombre d'erreurs serait donc moindre.

Le principal problème de cette solution réside dans les informations que l'on peut mettre dans ce graphe. Certes, on pourrait *bricoler* avec le logiciel pour ajouter des méta-données aux objets que l'on dessine, mais cela pourrait de nouveaux devenir très embêtant à utiliser et surtout relativement compliqué à importer.

La solution utilisée dans l'application est un compromis entre la solution *"graphe"* et celle *"excel"*. La gestion des données est subdivisées en deux processus :

- L'import de la structure du catalogue via un logiciel qui permet de dessiner des graphes
- L'ajout d'informations supplémentaires (Nom, crédits, ...) via un import de formulaire excel

**Pourquoi avoir subdiviser l'import des données en deux parties distinctes ?**

Pour rendre les choses plus aisées au personnel qui va encoder le programme, nous avons décidé d'utiliser yEd, un outil relativement haut niveau qui permet de générer des graphes. En peu de temps, il est possible de construire l'ensemble du programme de cours à l'aide de cet éditeur disponible sur Windows, Mac Os et Linux tout en sortant un diagramme clair et concis.

**Pourquoi** ? Car le programme de cours est un graphe, dont chaque nœud correspond à un cours et chaque *edge*, à une dépendance.

La solution idéale serait d'avoir un logiciel qui, en plus d'être intégré à l'application serait totalement adapté à notre besoin, à savoir **dessiner un graphe de cours**. Cela serait un dessinateur de programme de cours à part entière, proposant des nœuds intitulés cours, des arrêtes pour exprimer les contraintes, une façon de regrouper ces nœuds en module, en plus d'une autre pour y ajouter des informations relatives aux crédits, au contraintes des modules, etc. Cependant, cela dépasse malheureusement le cadre de mon mémoire. Libre à un étudiant, féru de développement web, de s'y attaquer dans les années à venir.

**Limites de la démarche**

Pour revenir à la façon dont nous importons les données, la principale limite d'un outil de la sorte est que nous sommes limités dans l'information que nous pouvons mettre dans ce graphe. Certes, il serait possible de sélectionner les différents nœuds et modules un à un et d'y ajouter l'information nécessaire, mais cette solution n'est pas efficace. Ils existe des solutions plus efficaces pour gérer des données à grande échelle : Excel.

La seconde limite, est qu'il faut se mettre d'accord sur l'utilisation de ce programme tiers afin de savoir *quoi* parser. Tout cela sera détaillé dans un manuel disponible dans

les annexes

L'import des données se fait donc en deux temps. Le graphe, qui contient les informations sur la structure du catalogue de cours (Nom des différentes entités et des dépendances), est d'abord parsé par l'application pour en extraire les informations. En suite, les informations plus spécifiques du catalogue de cours (les propriétés diverses des entités ; nom détaillé, url, date, informations sur les crédits) doivent être fournies via un formulaire Excel qui, à sont, tour doit être télécharger vers l'application.

### Construction et Import de la structure du catalogue

L'idée est de construire le graphe de cours en utilisant une application externe. Les exigences pour ce logiciel sont les suivantes :

- Avec ce logiciel, il doit être possible de grouper les différents nœuds pour représenter les curricula et leur différents modules.
- Le graphe étant relativement complexe, il est nécessaire d'avoir un outil qui arrive à construire une disposition correcte
- Il doit être possible d'exporter ces informations dans un format standard et aisé à parser.
- Ce logiciel doit être disponible sur Mac Os, Linux et Windows.

Plusieurs candidats on été retenus ; yEd, Dia et Graphiz. Tout les trois sont disponibles aussi bien sur Windows et Mac os que sur Linux et permettent d'exporter dans un format standard : le xml, mais un seul d'entre-eux permet de restructurer dynamiquement la structure du graphe : yEd.

C'est pourquoi notre choix c'est porté sur ce logiciel.

L'idée, pour construire le catalogue de cours, est (Dans yEd)

- D'utiliser les nœuds pour ajouter des cours, et de les labelliser avec leur sigle.
- D'utiliser les différents types d'arêtes pour représenter les différents types de dépendances
- De mettre les nœuds dans des groupes (labellisés avec leur nom) et les groupes dans des autres groupes pour représenter les différents modules, sous modules et programmes

Après, on demande au programme de calculer un layout hiérarchique. Sur l'image suivante, vous pouvez voir une partie de ce que génère yEd. (Le programme entier est

disponibles dans les annexes)

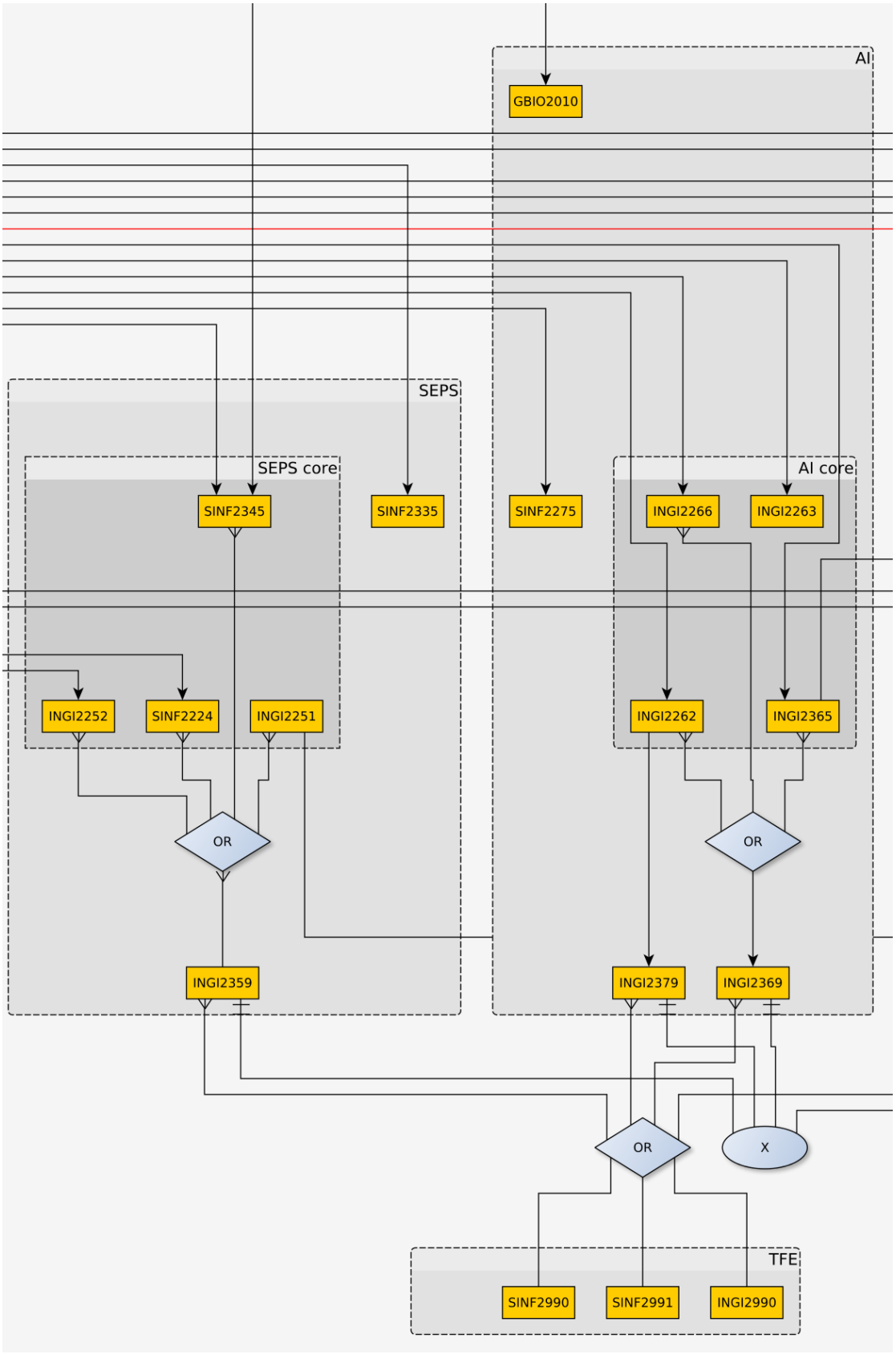



FIGURE 3.3 – Exemple de graph généré par Yed

### Construction et Import du formulaire Excel

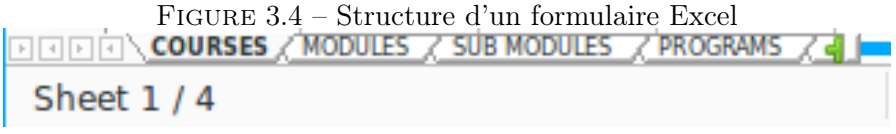
Comme expliqué précédemment, le graph à lui seul n'est pas suffisant pour ajouter toute les informations nécessaire à l'application. Le formulaire Excel est utilisé pour ajouter les informations relatives au propriétés des programmes, modules et cours des différents curricula.

Ces propriétés contiennent des informations simples sur les différents objets du catalogues comme le nom complet des différents cours, modules et programmes, les professeurs, les liens vers pages des cours. Elle contiennent aussi les informations relatives aux contraintes sur les propriétés, comme le nombre de crédits.

Il n'y a aucune restrictions sur les informations qui peuvent être ajoutées ici.

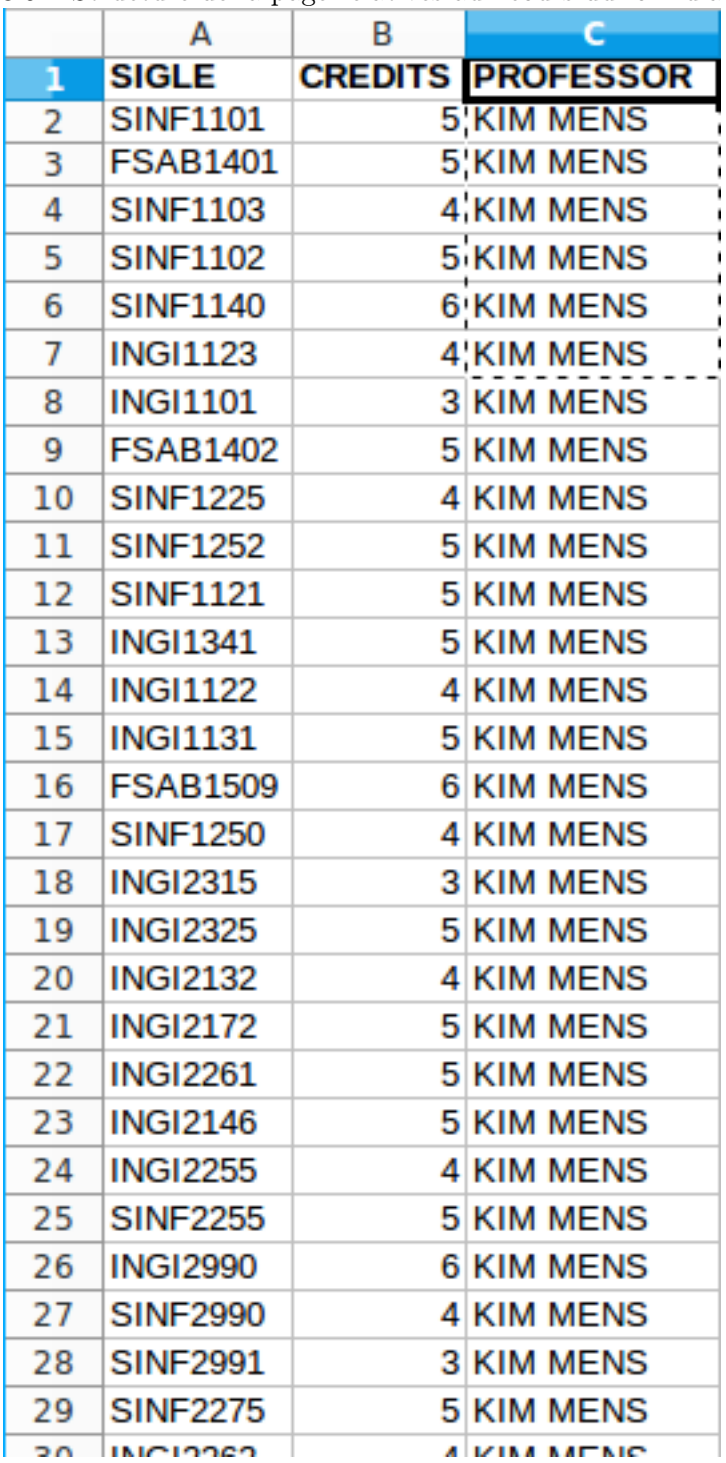
La structure du document est la suivante 3.4. Tout d'abord, il y a une page par objet (Cours, Programme, Module, Sous-module)

FIGURE 3.4 – Structure d'un formulaire Excel




Chacune des pages est structurée comme illustré sur l'image 3.5.

FIGURE 3.5 – Structure de la page relatives aux cours du formulaire Excel



| | A | B | C |
|---|---|---|---|
| 1 | **SIGLE** | **CREDITS** | **PROFESSOR** |
| 2 | SINF1101 | 5 | KIM MENS |
| 3 | FSAB1401 | 5 | KIM MENS |
| 4 | SINF1103 | 4 | KIM MENS |
| 5 | SINF1102 | 5 | KIM MENS |
| 6 | SINF1140 | 6 | KIM MENS |
| 7 | INGI1123 | 4 | KIM MENS |
| 8 | INGI1101 | 3 | KIM MENS |
| 9 | FSAB1402 | 5 | KIM MENS |
| 10 | SINF1225 | 4 | KIM MENS |
| 11 | SINF1252 | 5 | KIM MENS |
| 12 | SINF1121 | 5 | KIM MENS |
| 13 | INGI1341 | 5 | KIM MENS |
| 14 | INGI1122 | 4 | KIM MENS |
| 15 | INGI1131 | 5 | KIM MENS |
| 16 | FSAB1509 | 6 | KIM MENS |
| 17 | SINF1250 | 4 | KIM MENS |
| 18 | INGI2315 | 3 | KIM MENS |
| 19 | INGI2325 | 5 | KIM MENS |
| 20 | INGI2132 | 4 | KIM MENS |
| 21 | INGI2172 | 5 | KIM MENS |
| 22 | INGI2261 | 5 | KIM MENS |
| 23 | INGI2146 | 5 | KIM MENS |
| 24 | INGI2255 | 4 | KIM MENS |
| 25 | SINF2255 | 5 | KIM MENS |
| 26 | INGI2990 | 6 | KIM MENS |
| 27 | SINF2990 | 4 | KIM MENS |
| 28 | SINF2991 | 3 | KIM MENS |
| 29 | SINF2275 | 5 | KIM MENS |

La ligne 1, comprenant des cellules écrites en **gras** représente le header de la page. Chacune des cellules représente le nom de la propriété en question. Une propriété est un couple (Type, Value) où *Type* correspond au nom de la propriété, et *Value* à sa valeur. La première colomne représente la propriété qui **identifie** l'objet en question. Lorsque la page sera importé, une nouvelle propriété sera créée pour l'objet identifié par l'élément

de la première colonne. La valeur de cette propriété sera la cellule traitée, et le type de la propriété le nom de la colonne.

Par exemple, pour la ligne 4 de la page 3.5 Deux propriétés seront crées pour le cours intitulé *SINF1103*

1. La propriété ayant pour type **CREDITS** avec comme valeur **5**.

2. La propriété ayant pour type **PROFESSOR** avec comme valeur **KIM MENS**.

Pour plus de facilité, il est possible de télécharger un *template* de se formulaire depuis l'application, contenant toute les informations qui existe dans la base de données.

Par example, si l'on décide de télécharger ce template juste après avoir importer le graphe de cours, on aura les informations suivantes 3.6 pour les cours.

FIGURE 3.6 – Formulaire excel téléchargé juste après importation du graphe

| SIGLE |
| --- |
| SINF1101 |
| FSAB1401 |
| SINF1103 |
| SINF1102 |
| SINF1140 |
| INGI1123 |
| INGI1101 |
| FSAB1402 |
| SINF1225 |
| SINF1252 |
| SINF1121 |
| INGI1341 |
| INGI1122 |
| INGI1131 |
| FSAB1509 |
| SINF1250 |
| INGI2315 |
| INGI2325 |
| INGI2132 |
| INGI2172 |
| INGI2261 |
| INGI2146 |
| INGI2255 |
| SINF2255 |
| INGI2990 |
| SINF2990 |
| SINF2991 |
| SINF2275 |
| INGI2262 |
| INGI2263 |
| INGI2266 |
| INGI2365 |
| GBIO2010 |
| INGI2369 |
| INGI2379 |
| INGI2144 |
| INGI2142 |
| INGI2143 |
| INGI2145 |
| INGI2347 |
| INGI2348 |
| INGI2349 |
| INGI2359 |
| INGI2251 |
| INGI2252 |
| SINF2345 |
| SINF2224 |
| SINF2335 |
| CESS |

Il suffira après à la commission de programme de compléter ce fichier avec les informations souhaitées et de le télécharger vers l'application.

# Chapitre 4

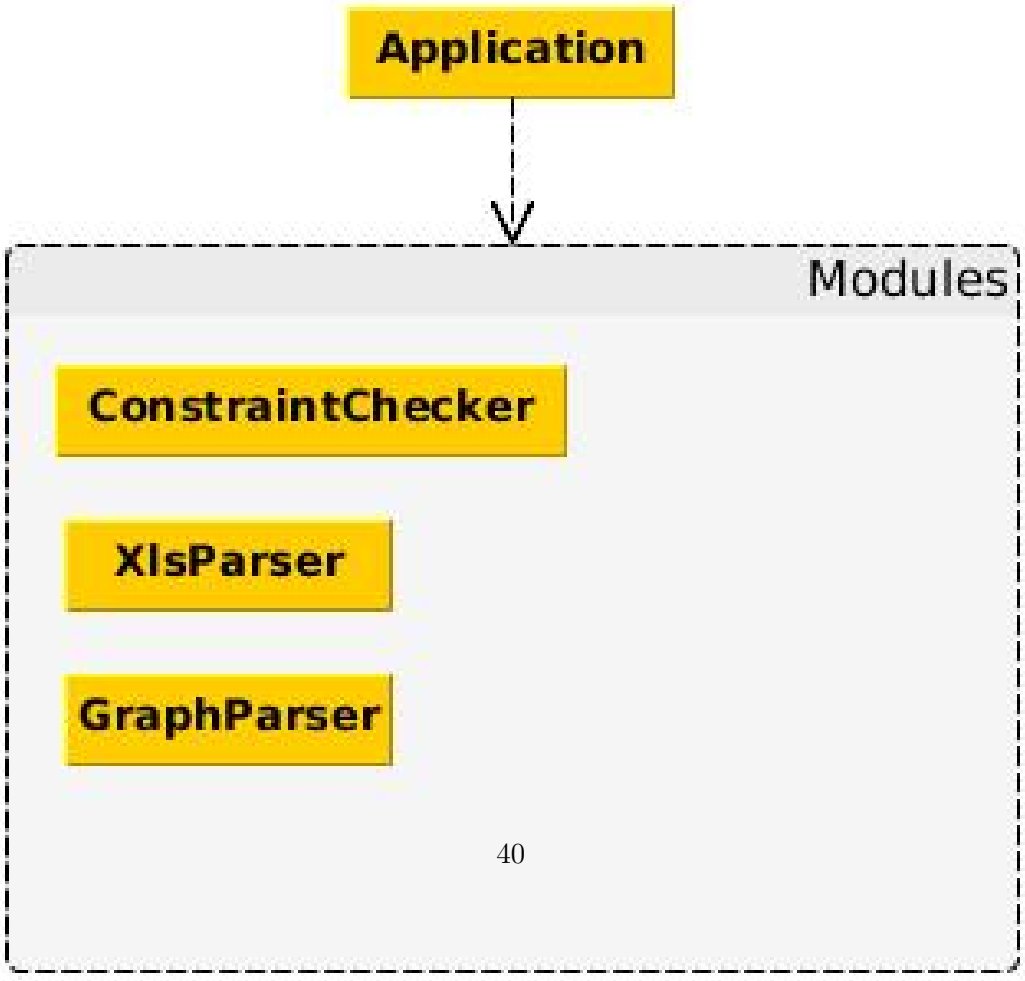
# Développement du système

## 4.1 Architecture

### 4.1.1 Modèle de données

#### Architecture globale

FIGURE 4.1 – 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. L'application utilise trois modules développés indépendamment.

1. **GraphParser** - le module s'occupant de parser les fichier .Graphml générer par yEd,
2. **ConstraintCheker** - le module s'occupant de vérifier les contraintes des programmes d'étudiants
3. **XlsParser** - le module occupant d'écrire et de lire dans des fichier excels.

L'architecture de chacun de ces modules, en plus de celle de l'application sera expliqué dans les sous-sections qui suivent.

Les conventions utilisées pour réaliser chacun des modèles sont les suivantes ;

FIGURE 4.2 – Généralisation

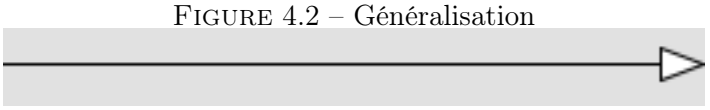


FIGURE 4.3 – Dépendance

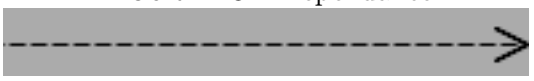


FIGURE 4.4 – Association "has many"

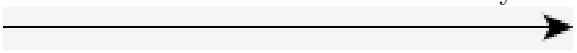
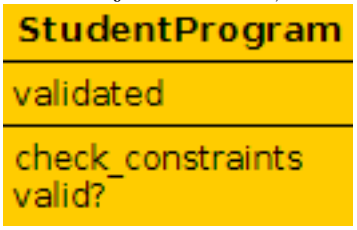


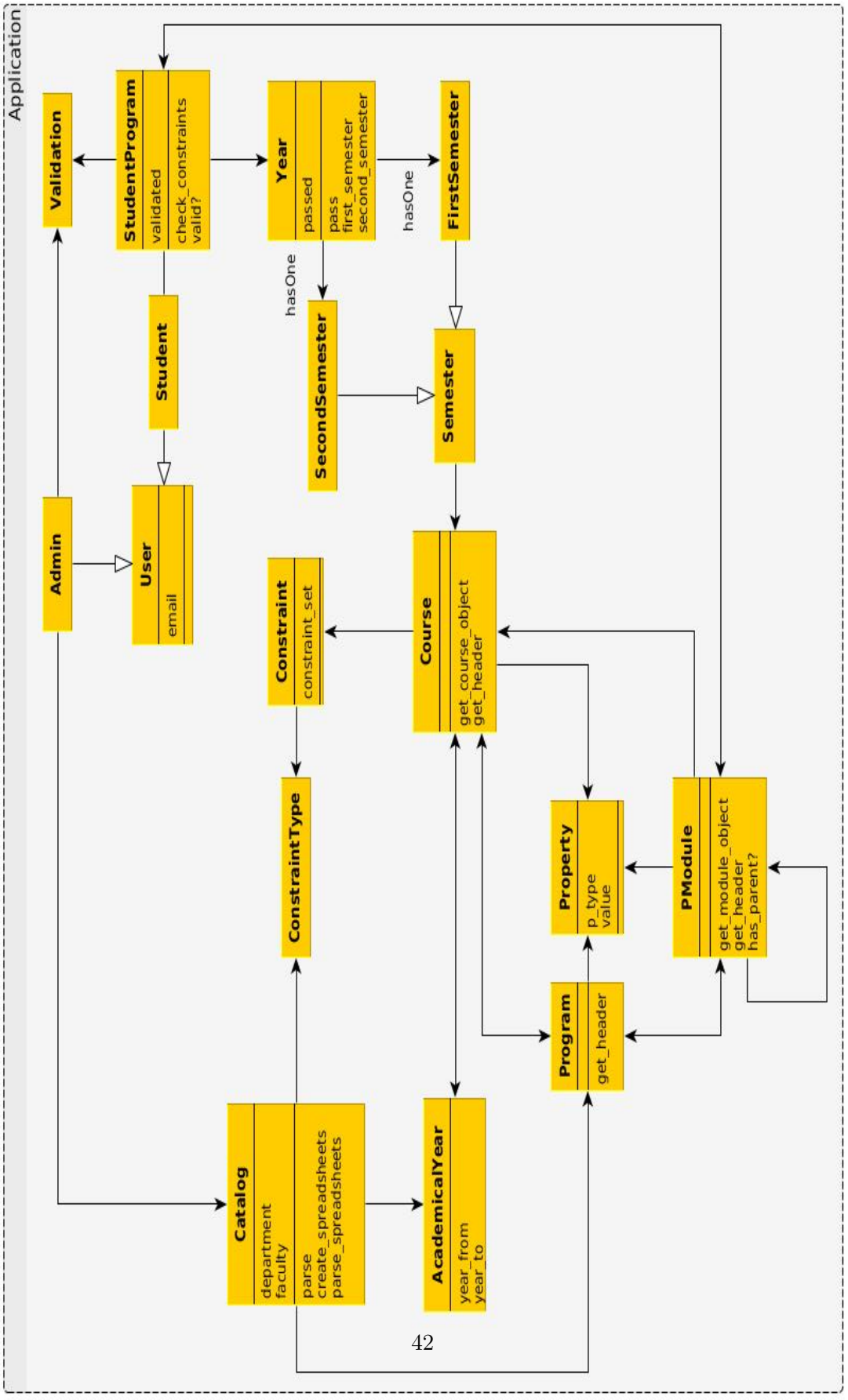
FIGURE 4.5 – Association "many to many"

FIGURE 4.6 – Représentationd d'un objet. En haut ; les attributs, en bas ; les méthodes.



**Application Rails**

FIGURE 4.7 – Architecture de l'application



**Student - Admin** - Ces objets représentent les deux acteurs de l'application, à savoir la *Commission INFO* (Admin) et les étudiants (Student).

**Property** - Un objet *Property* est composé d'un type et d'une valeur. Chacun des objets *Program*, *PModule* et *Course* peut en avoir 0 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, qui implique d'avoir l'identifiant d'un des deux objets dans l'autre. [9]. **Catalog** - Un catalogue est composé de plusieurs *Program*, *PModule* et *Course*.

**StudentProgram** - C'est le programme que se crée l'étudiant lorsqu'il utilise l'application. Un *StudentProgram* est une instanciation 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 année (*Year*)

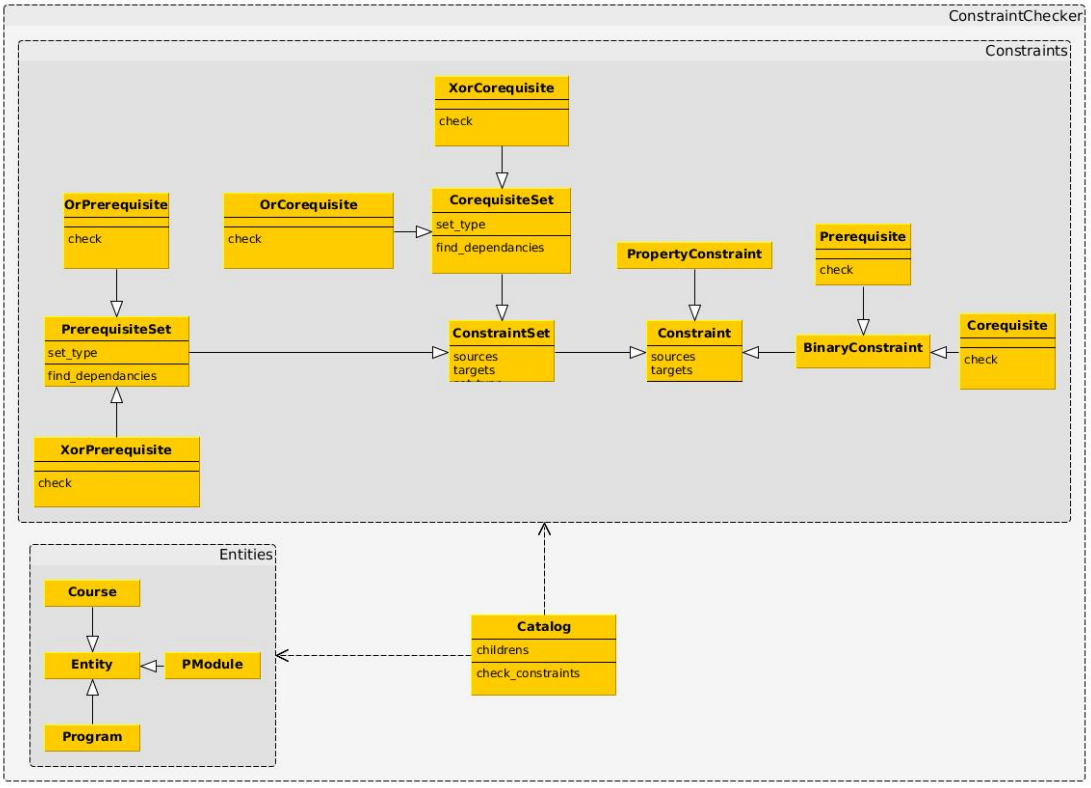
**Year** - Une année est composé de deux semestres. Un semestre est représenté par l'objet *Semester*. Le choix de chacun des cours du semestre est représenté par l'association *many to many* qui existe entre les deux objets. Pour représenter le premier et le second

semestre, un modèle *FirstSemester* et un modèle *SecondSemester* ont été créés, tout deux étendant le modèle *Semester* en utilisant la *Single Table Inheritance* de *Rails* [12]. Notez que la relation entre ces deux types de *Semester* et leur *StudentProgram* est une *has one* (la cardinalité est donc (1, 1) ici)

**Header** - Chacun des modèles *Course*, *Pmodule* et *Program* contient une méthode get_header qui renvoie une suggestion de propriétés utilisées à titre indicatif avec le module *XlsParser* (voir 4.1.1) pour créer les formulaires excel. Nous avons le header suivant : {"SIGLE", "CREDITS", "SEMESTRE", "OBLIGATOIRE"} pour le modèle *Course* par exemple.

### Architecture du vérificateur de contraintes

FIGURE 4.8 – Vérificateur de contraintes




L'architecture de ce module est composé en deux parties ;

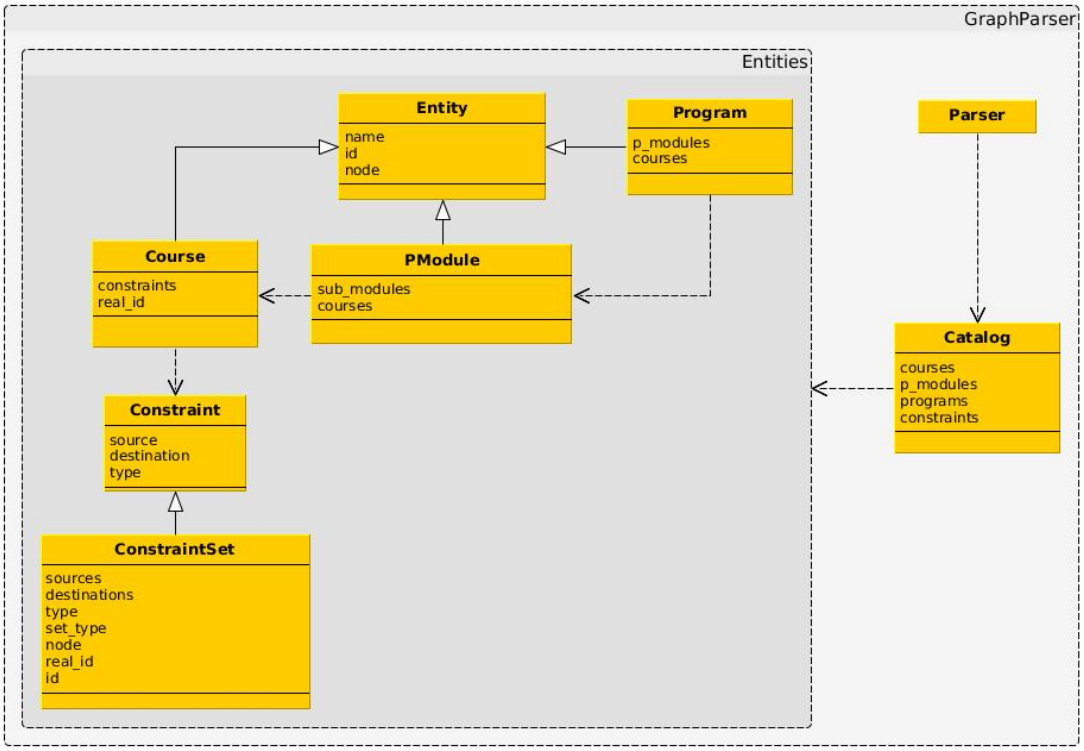
1. Les différents types de contraintes (Contraintes binaires, ensemble de contraintes ...)
2. Les différents types d'entités (Cours, modules, ...)

Le lien avec l'application se situe au niveau de la classe *Catalog*. En effet, chacune des différentes entrées des tables concernées (courses, p_modules, constraints) sont traduites 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.

### Parser de graphes

FIGURE 4.9 – Architecture du parser de graphe

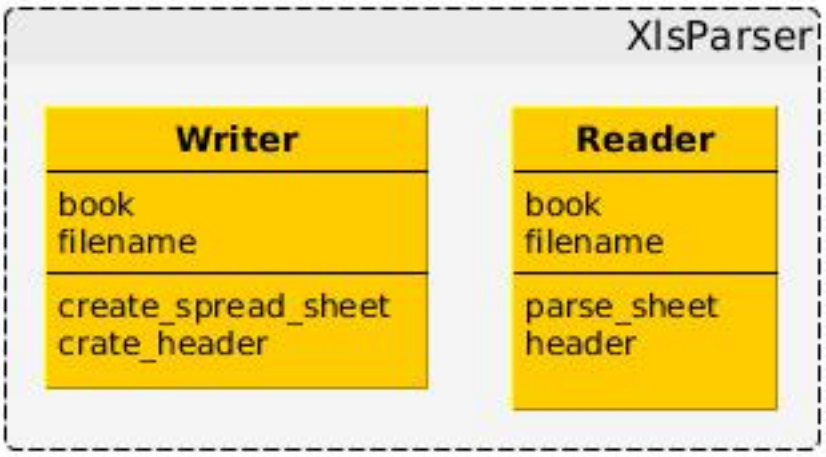



Tout comme dans le vérificateur de contraintes 4.1.1, le parser travail avec des objets *entités* à la différence que c'est lui qui les fournit à l'application (et pas l'inverse)

De nouveau, l'héritage tient une page prépondérante ici, pour diminuer le couplage, augmenter la cohésion et éviter autant que possible la duplication de code [11].

**Parser de fichiers excel**

FIGURE 4.10 – Architecture du parser de fichiers excel




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

## 4.2 Implémentation

### 4.2.1 Hébergement de l'application

L'application est hébergée sur *Heroku*. Cela impose cependant quelques restrictions ;

1. On 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 à 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.

### 4.2.2 Gestion des utilisateurs

Les utilisateur 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 utilité pour tout ce qui concerne les permissions des utilisateurs, à savoir à quel modèles un utilisateur à accès, et quels actions il peut effectuer sur ces modèles (Read, Create, Destroy, ...)

Pour gérer les deux types d'utilisateur (Commission INFO et Étudiants, trois choix s'offrent à nous ;

1. générer avec devise deux tables séparées
2. utiliser la Single Table Inheritance [12]. On crée un modèle user, puis on crée deux modèles spécifiques (student et admin) qui hérite 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'utilisateur ne diffèrent que par leur rôle.

La table générée par devise est la suivante ;

```
create_table "users", force: true do |t|
  t.string   "email",                  default: "",    null: false
  t.string   "encrypted_password",     default: "",    null: false
  t.string   "reset_password_token"
  t.datetime "reset_password_sent_at"
  t.datetime "remember_created_at"
  t.integer  "sign_in_count",          default: 0,     null: false
  t.datetime "current_sign_in_at"
  t.datetime "last_sign_in_at"
  t.string   "current_sign_in_ip"
  t.string   "last_sign_in_ip"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.boolean  "admin",                  default: false
end
```

Pour vérifier si un utilisateur est connecté dans les vues, il suffit d'appeler le helper suivant

```
user_signed_in?
```

Cela nous permet par exemple de cacher à l'utilisateur les menus permettant accéder aux différentes vues s'il n'est pas connecter

Pour vérifier si l'utilisateur à le rôle admin, il suffit de vérifier l'attribut dans la vue

```
current_user.admin?
```

current_user représentant l'utilisateur qui est connecté pour le moment.

Cependant, cela n'est pas suffisant. En effet, cela n'empêche pas l'utilisateur d'accéder aux différentes vues en entrant l'url dans la barre de navigation. C'est pourquoi il est nécessaire de dire à chaque contrôleur qu'il faut vérifier qu'un utilisateur est connecté avant d'afficher les vues. C'est fort heureusement très simple à faire avec Devise. Il suffit d'ajouter la ligne

```
before_action :authenticate_user!
```

dans chaque contrôleur où il est nécessaire que l'utilisateur soit connecté pour accéder aux vues.

*CanCan* intervient pour gérer les accès autorisés aux deux rôles de notre application (Utilisateur normal et admin). Il suffit simplement de créer un modèle *Ability* dans lequel on décrit ce à quoi chaque rôle à accès. C'est de nouveau très simple ;

```
if user.admin?
    can :manage, :all
else
    can :manage, [StudentProgram, Year, Semester]
    can :create, Validation
end
```

Il suffit de définir, en fonction du rôle de l'utilisateur, les modèles auxquels il a accès et ce qu'il peut faire. L'utilisateur *normal* par exemple, n'a accès qu'aux modèles *StudentProgram, Year, Semester*. S'il tente d'accéder aux vues des modèles auxquels il n'a pas accès, il sera redirigé vers la page d'accueil.

Notez qu'il est possible de générer les vues qui permettent à l'utilisateur de s'enregistrer, de se connecter et de gérer les informations relatives à son compte utilisateur. Ces vues ont cependant été modifiées pour que leur style s'adapte à celui de l'application.

Enfin, il n'est pas possible, pour des questions de sécurité évidentes, de se créer un compte *admin* via les formulaires d'enregistrement disponibles dans l'application. Il faut tout d'abord se créer un compte utilisateur dans l'application, et ensuite modifier l'attribut *admin* directement dans la console.

### 4.2.3 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 suivantes.

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.

Voici ce à quoi ressemble les informations d'un fichier GraphML 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>
  (...)
</node>
```

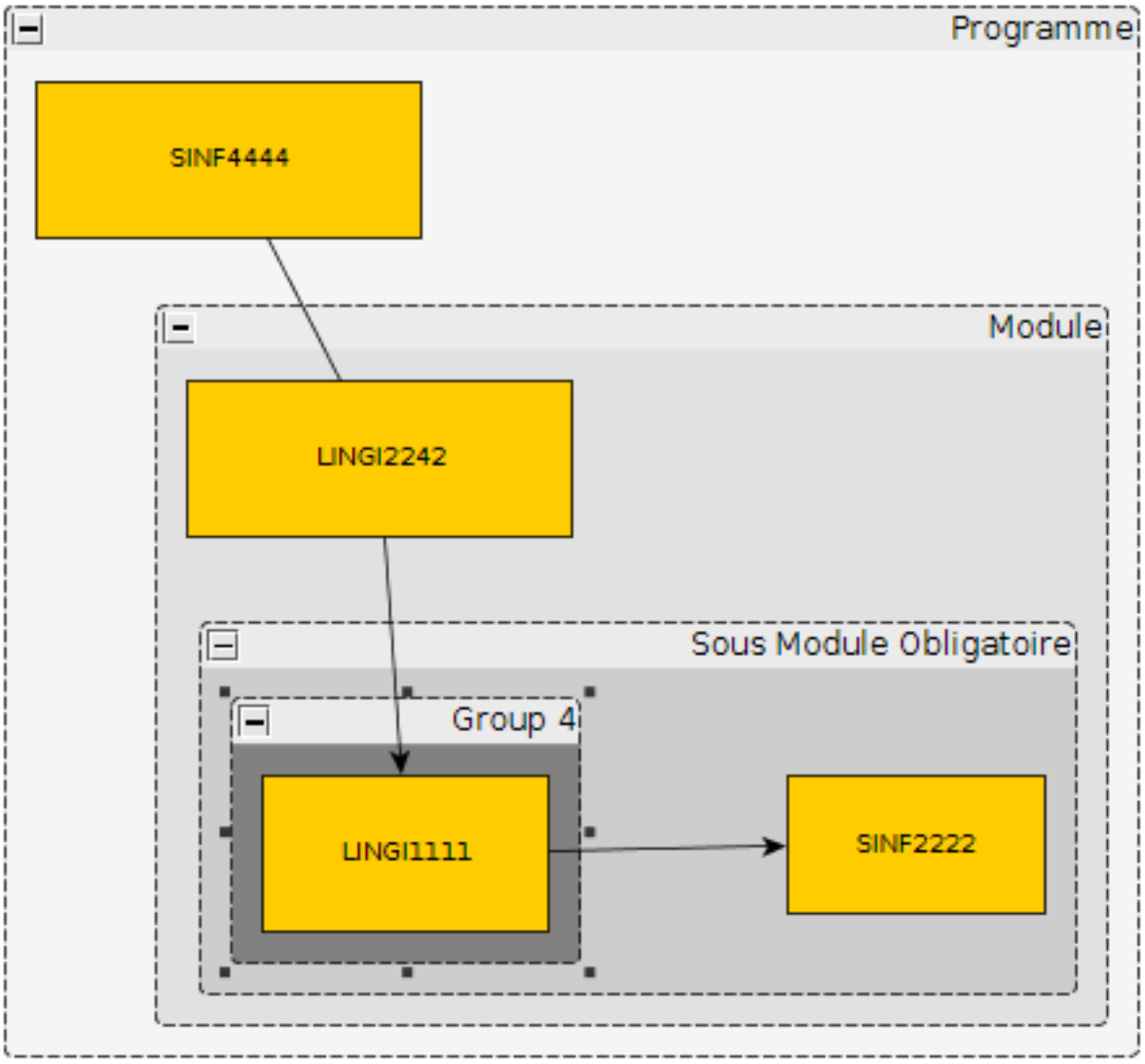
En comparaison, la version *XGML* du même nœud.

```
<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 respectant pas la hiérarchie des différents nœuds et boites.

FIGURE 4.11 – Exemple de graphe hiérarchique




Pour le graph 4.11 par exemple, la structure du fichier sera de la sorte :

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

Lorsque l'on parsera le fichier, on devrait donc une première fois lire le fichier, récupérer les informations, puis retraiter une deuxième fois les données pour ajouter les parents des différents nœuds.

TODO : Insérer l'algorithme tel qu'il devrait être pour un graphe de la sorte ?

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 : [
    ]
  ]
]
(...)
```

Il n'est donc pas nécessaire de retraiter tout 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*[1] pour parser le format *GraphML* de yEd. Les informations contenue 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

Tout ses 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, suivit par toute les arrêtes.

Le module ajoutes 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 contrainte).
2. Il renvoie des objets cours, modules et programmes ainsi que leur différentes dépendances.

Comme expliqué dans l'introduction de cette section, chaque élément peut avoir un enfant, est identifié par un tag, peut avoir des attributs et contient de l'information.

la structure plus détaillée d'un nœud est la suivante :

```
<node id="parentID::nodeID" [GROUP]>
  <(...)>
  </(...)
  <NodeLabel>
    NAME
  </(...)
  <(...)>
  </(...)>
</node>
<graph id="parentID">
  <node id (...)>
  <node id (...)>
</graph>
```

Il y a donc quatre informations à extraire ;

1. l'identifiant du nœud parent (parentID) ;
2. l'identifiant du nœud (nodeID) ;
3. son nom (NAME) ;
4. est-ce un groupe ? (GROUP)

La structure de graphe imposée à la commission est la suivante ;

- les modules et programmes sont représentés par des nœuds *Group* ;
- les cours sont représentés par des simple nœuds ;

- les dépendances par des arrêtes.

Notez que chaque cour et module **DOIT** se trouver dans un programme. En effet, *nœuds groupes* sans parents sont parsés comme programmes. Cela nous permet de distinguer les modules des programmes, sans devoir ajouter une convention de couleur sur les boites pour les différencier.

L'algorithme de parsing est donc relativement simple.

On a ;

- une méthode qui parse les arrêtes ;
- une methode qui parse les nœuds ;
- une méthode qui aprse les programmes et ses enfants ;
- une méthode qui parse les modules et ses enfants ;
- une méthode qui parse les cours.

On crée au préalable un objet catalogue. Chacune des méthodes listées précédemment renvoie un objet Cours, Module, Programme, Contrainte. Ces objets sont ajoutés au catalogue en respectant leur hiérarchie. Un cours aura donc comme parent un module ou un programme.

```
FOREACH elements in graph do
- IF node ->
  parse node
  - IF program ->
    parse program (extract informations and add to catalog)
    (1) parse childs
      - IF module
        -> parse module (extract informations and add to parent)
        -> parse childs (go_to (1))
      - IF course
        -> parse course (extract informations and add to parent)


- IF edge ->
  parse edge
    retrieve source(s)
    retrieve destinatinations(s)
    retrieve constraint type
```

### 4.2.4 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*.

Ce module est divisé en deux parties.

D'un coté nous avons les contraintes. Tout les types de contraintes héritent de la super classe *Contrainte* qui, en plus de constructeur ne contient qu'une seule méthode **Check**.

De l'autre coté nous avons les entités qui représentent les modèles *Course*, *Module* et *Program*

#### Entités

*Entity* est la classe qui représente les objets traités par le module *ConstraintsChecker*. Entity étends la classe OpenStruct. Openstruct est une librairie qui permet de créer des objets à la volée. Pour chaque paramètre passé à un objet *OpenStruct* [1], un attribut sera créé ainsi que les méthodes pour le modifier et y accéder. On évite ainsi de devoir modifier la classe *Entity* et ses enfants Course, PModule lorsque l'on veut rajouter une contrainte par exemple.

Nos différents objets sont stockés sous forme d'arbre. La racine est l'objet catalogue, et les enfants sont les différents objets *Course* et *PModule*, tous étendant le super-type *Entity*.

Un objet *entity* contient essentiellement ;

- un attribut **constraints** qui contienne les différentes contraintes de l'objet ;
- un attribut **childrens** qui contient les références vers ses enfant dans l'arbre ;
- un attribut **parent** qui contient la référence vers son parent dans l'arbre ;
- une méthode **find_children(children_id, children_type)** qui permet d'effectuer une recherche sur ses enfants. Le paramètre *children_type* permets de spécifier le type d'enfant que nous recherchons (Cours, PModule) ;
- une méthode **search(children_id, children_type)** qui permet d'effectuer une recherche dans tout l'arbre. Cette méthode est utilisé dans les methodes *check* des

différentes contraintes pour retrouver des objets *Course*. (Retrouver une dépendance par exemple). L'algorithme est le suivant ;

```
- Retrouver la racine de l'arbre
- Appeler find_children sur ses enfants
```

- une méthode **check** qui vérifie ses contraintes et celle de ses enfants ;
- des méthodes relatives aux différentes contraintes, comme *count_ credits*, *check_ max*, *check_ min*, ...

Ces objets sont construits dans les différents modèles de l'application. L'idée, dans chacun de ces modèles (Course, PModule, Constraint) est d'avoir une méthode get_[model_name]object qui va créer l'objet correspondant.

Dans le cas du modèle *Course*, la méthode est la suivante :

**Course**

```
def get_course_object(passed)
  course = ConstraintsChecker::Entities::Course.new(name: self.name, id: self.id,
     passed: passed, parent_id: self.p_module_id, credits: self.credits,
    mandatory: self.mandatory?)
  self.constraints.each do |c|
    course.add_constraint(c.get_constraint_object(course))
  end
  return course
end
```

Le paramètre *passed* sert à identifier si un cours a été réussi ou non. Cela est utile notamment lors de la vérification des contraintes de type *Prérequis*.

Comme expliqué plus haut, on peut passer au constructeur de *Course* tout ce que l'on veut, grâce à OpenStruct [1].

Rien ne nous oblige à passer par le modèle *Constraint* pour ajouter les contraintes. Comme expliqué dans le chapitre 3, certaines contraintes, comme les contraintes sur les propriétés, n'ont pas d'objet pour les représenter en base de données. En effet la plupart des données relatives à ces contraintes sont, en plus d'être présentes en base de données, assez volatiles, car elles peuvent être modifiées régulièrement via l'import de fichiers excels. Nous évitons ainsi de surcharger notre architecture avec des objets dont l'utilité est plus que relative .... Pour revenir à ce type de contraintes, il est préférable de les ajouter dans la méthode *get_ [model_ name]object*.

### PModule

```
def get_p_module_object(mandatory)
  p_module = ConstraintsChecker::Entities::PModule.new(id: self.id, name: self.
  name)
  p_module.add_constraint(ConstraintsChecker::Constraints::Min.new(p_module,
  self.min))-
  p_module.add_constraint(ConstraintsChecker::Constraints::Max.new(p_module,
  self.max))
  self.sub_modules.each do |m|
    p_module.add_children(m.get_p_module_object(true))
  end

  if mandatory
    course_ids = []
    self.courses.each do |course|
      course_ids << course.id
    end
    p_module.add_constraint(ConstraintsChecker::Constraints::Mandatory.new(
  p_module, course_ids))
  end

  return p_module
end
```

Le paramètre *mandatory* sert à identifier si le module est obligatoire ou non. Cela est utile lors de la vérification des contraintes sur les propriétés notamment.

Nous avons ici un exemple de contraintes (Min & Max) qui ne sont pas ajoutées en passant par le modèle *Constraint*.

### Contraintes

L'idée, pour chaque type de contraintes, est d'étendre la super-classe *Constraint* en réimplémentant la méthode *check* pour qu'elle corresponde au comportement recherché. Cette méthode renvoie un *hash*, spécifique à chaque type de contraintes, contenant les résultats de la vérification.

Par exemple, pour les dépendances entre les cours nous avons deux types. Les dépendances binaires, qui correspondent à un prérequis ou corequis entre deux cours, et les dépendances n-aire qui correspondent à un ensemble de prérequis ou corequis entre plusieurs cours avec une condition sur cet ensemble de contraintes. Cet ensemble peut être une disjonction (OR) de contraintes par exemple, exprimant qu'il faut choisir au moins

une des dépendances, ou une disjonction exclusive (XOR), exprimant qu'il faut choisir une et une seule dépendance.

Prenons l'exemple des dépendances binaires. Nous avons une classe *BinaryConstraint* qui ne ré-implémente pas la méthode check de sa super-classe *Constraint* car elle sert de super-classe pour les deux classes représentant les deux types de contraintes ; *Prerequisite* et *Corequisite*.

**Corequisite** La méthode check va appeler la méthode *find_ course* du cours en question, qui va remonter jusqu'à la racine (le catalogue), et chercher si le corequis du cours est présent dans le catalogue. S'il n'est pas présent, le message "corequisite_missing :[course_id]" est envoyé.

**Prerequisite** La méthode check se comporte comme celle de *Corequisite*, à la différence qu'elle va vérifier si l'attribut *passed* du cours est *true*, conformément au comportement d'un pré-requis.

Dans le cas d'un ensemble n-aire de contraintes, il y a essentiellement deux points qui diffèrent ;

1. l'existence d'une méthode *find_ dependancies* qui récupère les *course_ id* manquants pour la contrainte en question ;
2. une vérification sur la taille de la liste, correspondant à la condition qui régit cet ensemble de contrainte. Dans le cas d'un ensemble disjonctif (OR), il faut vérifier que le nombre de *course_ id* renvoyé soit strictement inférieure au nombre de dépendances du cours, pour vérifier qu'il y ai au moins une dépendance qui est choisie, conformément à la logique d'une disjonction. Dans le cas d'un ensemble disjonctif exclusif (XOR), il faut vérifier qu'il n'y aie qu'une et une seule dépendance choisie, conformément à la logique d'une disjonction exclusive.

Pour vérifier ces contraintes, l'objet *Catalog* appelle sur chacune des contraintes de ses enfants et de leur enfants leur méthode *check* et récupère les messages qu'elles renvoient. Ces messages sont traités par le modèle *StudentProgram* et affichés sur les vues correspondantes.

À ce jour, la liste des messages renvoyés par les méthodes *check* des différents types de contraintes est la suivante :

**or_corequisites_missing** Ce message concerne la contrainte n-aire *OR-Corequisite*. Il contient la liste des ids des cours concernés par la contrainte si elle n'est pas vérifiée ;

**xor_corequisites_missing** Ce message concerne la contrainte n-aire *XOR-Corequisite*. Il contient la liste des ids des cours concernés par la contrainte si elle n'est pas vérifiée ;

**or_prerequisites_missing** Ce message concerne la contrainte n-aire *OR-Prerequisite*. Il contient la liste des ids des cours concernés par la contrainte si elle n'est pas vérifiée ;

**xor_prerequisites_missing** Ce message concerne la contrainte n-aire *XOR-Prerequisite*. Il contient la liste des ids des cours concernés par la contrainte si elle n'est pas vérifiée ;

**prerequisites_missing** Ce message concerne la contrainte binaire *Prerequisite*. Il contient l'id du cours concerné par la contrainte si elle n'est pas vérifiée ;

**corequisites_missing** Ce message concerne la contrainte binaire *Corequisite*. Il contient l'id du cours concerné par la contrainte si elle n'est pas vérifiée ;

**to_few_credits** Ce message concerne la contrainte sur la propriété (Crédits) *Min*. Il contient l'id de l'entité concerné par la contrainte si elle n'est pas vérifiée ;

**to_many_credits** Ce message concerne la contrainte sur la propriété (Crédits) *Max*. Il contient l'id de l'entité concerné par la contrainte si elle n'est pas vérifiée ;

**courses_missing_in_module** Ce message concerne la contrainte sur la propriété *Mandatory* d'un objet *Module*. Il contient les ids des entités manquantes d'un Module ; obligatoire si la contrainte n'est pas vérifiée ;

**mandatory_courses_missing** Ce message concerne la contrainte sur la propriété *Mandatory* d'un objet *Course*. Il contient l'id du cours en question si la contrainte n'est pas vérifiée.

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. Il ne faut pas oublier de renvoyer à la fin de cette méthode un message qui *explique* pourquoi la contrainte n'est pas vérifiée, en cas d'échec
3. Dans la méthode *get_ object* du modèle concerné par la contrainte, créer et ajouter l'objet contrainte et ajouter les informations nécessaire 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*

### 4.2.5 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 *Writter* 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 *Writter* est appelé lorsque l'utilisateur télécharge un template de formulaire excel après avoir créer le catalogue.

Le *Reader* est appelé à chaque fois que l'utilisateur mets à jours 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*

# Chapitre 5

# Validation

## 5.1 Introduction

Ce document présente un scénario typique d'utilisation pour la **commission INFO** ainsi qu'un scénario d'utilisation pour un *Étudiant*. L'application au moment où ce scénario a été conçu n'est pas encore finie. Il risque d'y avoir des changements au niveau de son design et l'ajout de certaines *features* non encore implémentées. Cependant, la base de l'application est suffisamment présente que pour permettre à ce test d'être pertinent.

Le scénario *Commission INFO* consiste à créer un catalogue de cours, mettre à jours ses informations et créer un programme de cours à la carte.

Le scénario *Étudiant* consiste à se créer un compte, se connecter avec sur l'application, créer un programme de cours et à le configurer

Notez que

1 - Le catalogue présenté aux étudiants pour construire leur programme de cours est le dernier en date à voir été créé en base de donnée (la feature pour sélectionner le programme de cours principal n'ayant pas encore été implémentée)

2 - Si la commission ne rajoute pas l'information relative au semestre durant lequel sont dispensés les cours, l'étudiant ne verra aucun cours lorsqu'il voudra créer une année (La propriété *Semester* étant initialisée à *NONE*)

3 - L'étudiant ne peut pas accéder aux informations relatives au programme qu'il veut suivre depuis son interface (Ces vues n'ont pas encore été implémentées)

Il est demandé de ne pas regarder dans le manuel pour réaliser l'expérience, afin que le feedback soit le plus complet possible.

Bon amusement :-)

## 5.2 Ressources

Tout d'abord, voici l'url de l'application : : `http://curriculum-mgmt.herokuapp.com/`

Ensuite, voici les informations relatives au compte avec lequel il faut se connecter pour accéder à l'application

- USERNAME : commission@gmail.com
- PASSWORD : coucou42

(Il n'est pas possible de se créer un compte admin via l'application pour des raisons de sécurité évidentes)

## 5.3 Scénario Commission INFO

1. Connectez vous à l'application
2. Accédez à l'onglet **Catalogues**
3. Créer un graphe avec yEd, exporter le graph en .graphml ou utiliser un fichier de graphe déjà existant
4. Créer un catalogue en utilisant le fichier de graphe précédemment créer
5. Mettre à jour les informations du catalogue (En commençant par télécharger le fichier excel depuis l'application, comme expliqué sur la vue)
6. Se rendre dans l'onglet **Programmes** pour accéder aux programmes de cours
7. Créer un nouveau programme de cours avec les modules & cours désirés (Attention, tout les cours des modules sélectionnés seront ajoutés automatiquement, vous ne pouvez que choisir les cours qui ne sont dans aucun modules)
8. Supprimer le programme de cours précédemment créer
9. Naviguer dans les différents menus

## 5.4 Scénario Étudiant

- Créer un comte sur l'application. Vous pouvez mettre n'importe quelle adresse email, aucun mails ne sera envoyé.
- Se rendre dans le menu à droite, cliquez sur mon compte et changer votre mot de passe. Vous pouvez aussi supprimer votre compte si vous le désirer
- Se rendre dans le menu *Mes programmes de cours* et se créer un nouveau programme
- Configurer son programme de cours, en choisissant des modules par exemple, et en ajoutant une année avec les cours désirés. Aucun cours ne sera afficher si la note 2 5.1 de la section 5.1 n'a pas été suivie.
- Vérifier les contraintes de son programme
- Envoyer son programme à la validation
- Naviguer dans les différents menus

# Chapitre 6

# Travaux futurs

Chapitre 7

# Conclusion

# Bibliographie


[1] Openstruct api documentation.

[2] Rails Casts. Migrating to postgresql, 2012.

[3] Université Catholique de Louvain. Année d'études préparatoire au master en sciences informatiques - sinf1pm, 2013-2014.

[4] Université Catholique de Louvain. Bachelier en sciences de l'ingénieur, orientation ingénieur civil - fsa1ba (majeure et mineure info), 2013-2014.

[5] Université Catholique de Louvain. Bachelier en sciences informatiques - sinf1ba, 2013-2014.

[6] Université Catholique de Louvain. Master [120] : ingénieur civil en informatique - info2m, 2013-2014.

[7] Université Catholique de Louvain. Master [120] en sciences informatiques - sinf2m, 2013-2014.

[8] Université Catholique de Louvain. Master [60] en sciences informatiques - sinf2m, 2013-2014.

[9] Rails Guides. Active record querying interface, 2014.

[10] Bryan Helmkamp. 7 ways to decompose fat activerecord models, 2012.

[11] Jeremy Miller. Cohésion et couplage, 2008.

[12] Eugene Wang. How (and when) to use single table inheritance in rails, 2013.

# Manuel - Commission INFO

## .1 Introduction

Le but de présenter un scénario typique d'utilisation pour la **commission INFO**. L'application au moment où ce scénario a été conçu n'est pas encore finie. Il risque d'y avoir des changements au niveau de son design et l'ajout de certaines *features* non encore implémentées (comme la gestion des utilisateurs, ou certaines contraintes spécifiques). Cependant, la base de l'application est suffisamment présente que pour permettre à ce test d'être pertinent.

Le scénario est le suivant. Tout d'abord, le lecteur va être amené à créer un graph avec l'outil yEd. Ensuite, il va l'importer dans l'implication. Après, il va rajouter des données au catalogue de cours précédemment créé. Il sera ensuite expliquer comment accéder aux différents vues montrant les données qui ont été importée (comme les cours, les modules, ...). Enfin, il sera présenter comment créer des programmes de cours à partir des données importées.

Bon amusement :-)

P.S. Le lien vers l'application : `http://curriculum-mgmt.herokuapp.com/`

### .1.1 Feedback recherché

- Des bugs s'il y en a
- Critiques au niveau de l'interface, des étapes qui ne sont pas claires, etc
- Features manquantes dans l'application
- Attributs identifiants certains modèles qui ne sont pas adéquats. (Pour les catalogue par exemple, je ne suis pas sûr que la faculté et le département soit pertinents

- ...

## .2 Gestion des catalogues de cours

### .2.1 Page d'accueil


FIGURE 1 – Accès de gestion des catalogues

Sur l'image 1 apparaît la page d'accueil Le menu pour accéder aux différents catalogues de cours est entouré d'un cadre rouge sur la capture d'écran.

### .2.2 Accéder aux catalogues

Après avoir cliquer sur le menu catalogue, nous arrivons sur la page montrant les différents cours présents dans l'application. Comme vous pouvez le voir sur la capture d'écran 2, il n'y a qu'un seul catalogue pour le moment dans l'application. Vous pouvez cliquer sur le bouton *Plus d'informations* pour accéder au catalogue en question, ou encore sur *supprimer* pour le supprimer de l'application. Cependant, nous allons commencer par créer un graphe pour pouvoir créer un catalogue en l'important

FIGURE 2 – Liste des catalogues

### .2.3 Création d'un graphe de cours avec yEd

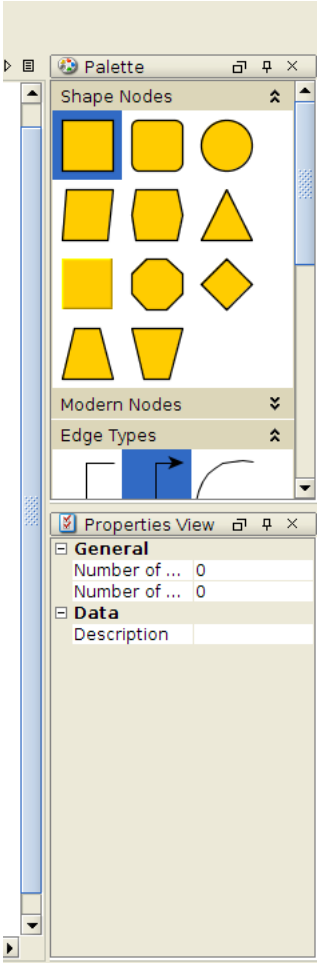
#### Le menu de yed

Dans cette section nous allons pas à pas construire un graph de cours avec yed (L'outil utilisé pour générer des graphes).

Yed est disponnible ici `http://www.yworks.com/en/downloads.html#yEd`

Nous allons créer un petit catalogue de cours, composé d'un programme, un module quelques cours et quelques dépendances. Tout d'abord, ouvrez le programme yEd et créer un nouveau document. Une fois le yEd ouvert et le nouveau graph créer, vous verrez à votre droite le menu suivant 3.

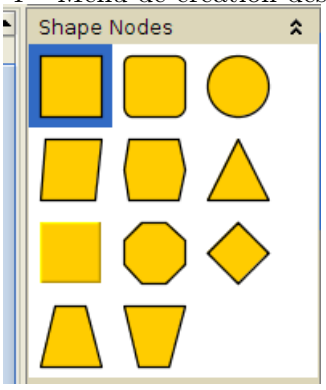
FIGURE 3 – Menu de yEd



### Créer des cours

Pour créer les objets représentant les cours, il faut utiliser ce menu 4 et sélectionner le carré (en bleu sur la capture d'écran) puis cliquer sur le document pour l'ajouter au graphe.

FIGURE 4 – Menu de création des noeuds



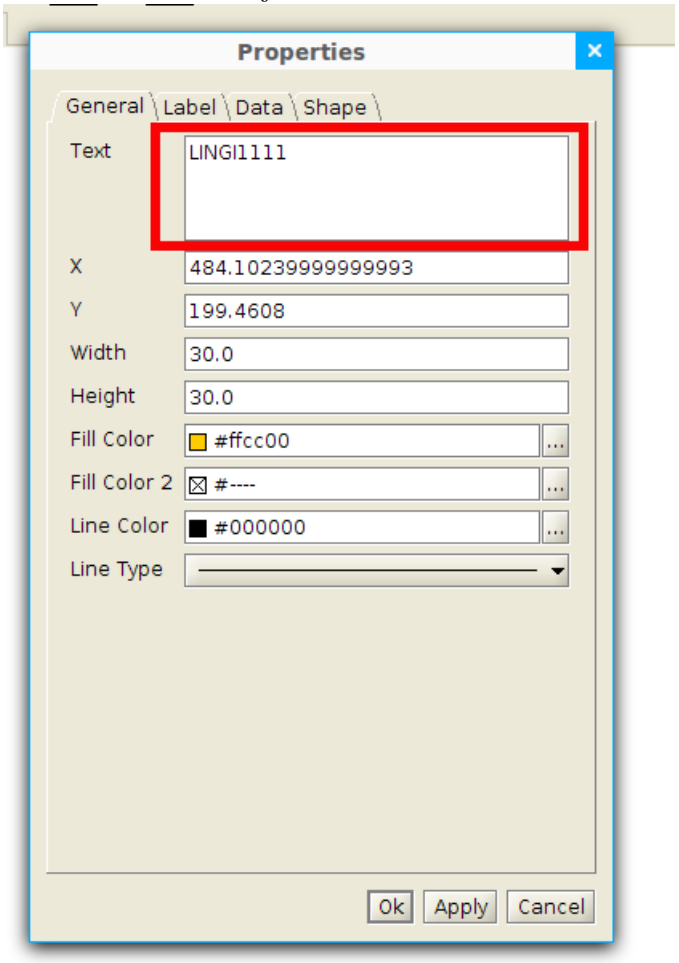

Pour nommer le nœud :

1. Clique droit sur le nœud

2. Cliquez sur *properties*

Le menu suivant apparaîtra 5. Il suffit de remplir le champ *Texte* avec le nom désiré.

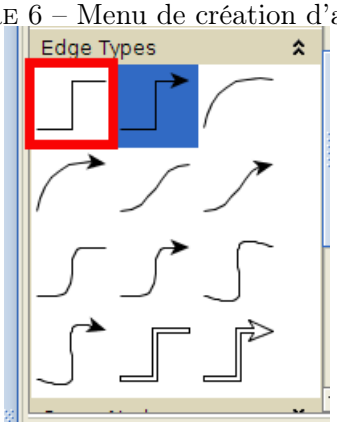
FIGURE 5 – Ajouter un nom à un noeud




### Ajouter des dépendances

Pour représenter les dépendances entre les cours, il faut utiliser le menu présent sur l'image 6 qui permet de dessiner des arrêtes entre les nœuds.

FIGURE 6 – Menu de création d'arrêtes



Nous utilisons deux type d'arrêtes ;

- en bleu sur l'image 6, les arrêtes pour représenter les prérequis
- en rouge sur l'image 6, les arrêtes pour représenter les corequis

En créant un catalogue de deux cours, avec l'un étant le prérequis de l'autre, nous obtenons le graph présent sur l'image suivante .2.3.

FIGURE 7 – Deux cours avec une dépendance

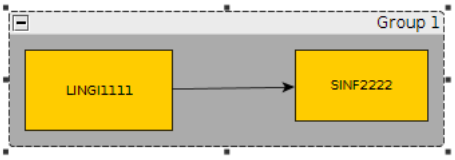

**Insérer des cours dans des modules**

Après avoir créer plusieurs cours, il est possible de regrouper ces nœuds dans une boite.

1. Sélectionner les noeuds à regrouper
2. Clique droit
3. Cliquer sur *grouping*

Vous obtiendrez une boite comme sur la capture 8

FIGURE 8 – Mettre des nœuds dans une boite




La démarche à suivre pour nommer les boites est la même que celle pour nommer les nœuds.

Avant de poursuivre, il est nécessaire de préciser la structure de graph reconnue par l'application.

- Un **catalogue** est représenté par un **graphe** et est composé exclusivement de **programmes**

- Un **programme** est représenté par une **boite** et est composé de **modules** et de **cours**
- Un **module** est représenté par une **boite** et est composé de plusieurs **sous-modules** et **cours**
- Un **sous-module** marquera dans l'application tout les **cours** qu'il contient comme obligatoire pour le **module** parent.
- Un **cours** est représenté par un **nœud** et peut avoir plusieurs dépendances.
- Une **dépendance** est représenté par une **arrête** entre deux **nœuds**.(Elle peut avoir deux type comme expliqué plus tôt

Sur l'image .2.3, vous pouvez voir un exemple de catalogue valide.

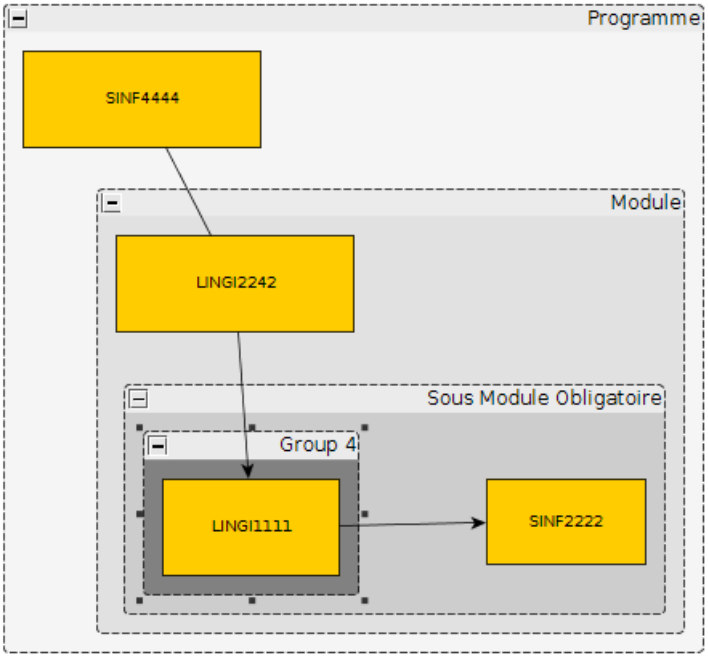


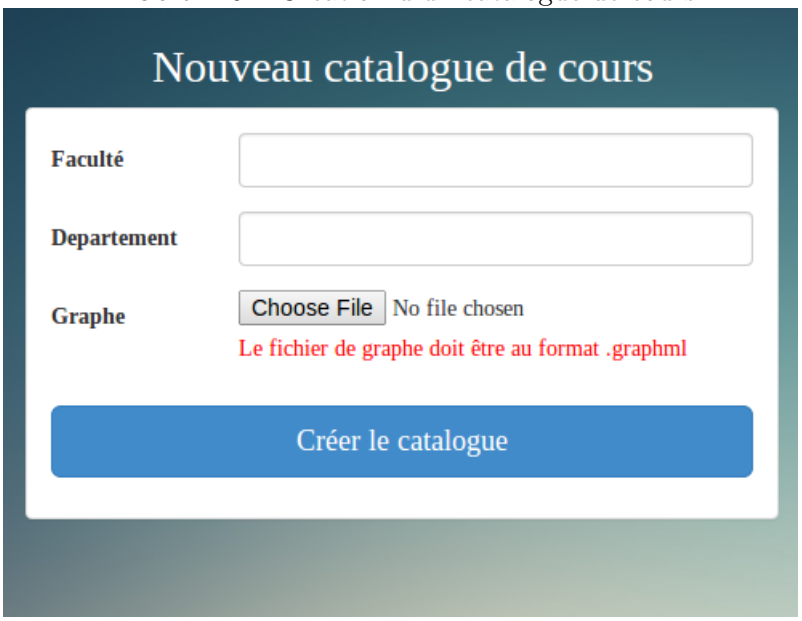

FIGURE 9 – Un (petit) catalogue de cours valide

### .2.4 Import du graph dans l'application

Pour poursuivre, vous pouvez soit utiliser le graphe que vous venez de créer avec yEd, soit télécharger un graph complet à l'adresse suivante : `http://xavier.ethylix.be/graph.graphml`

Vous êtes donc sur la page comme illustré sur l'image 10.

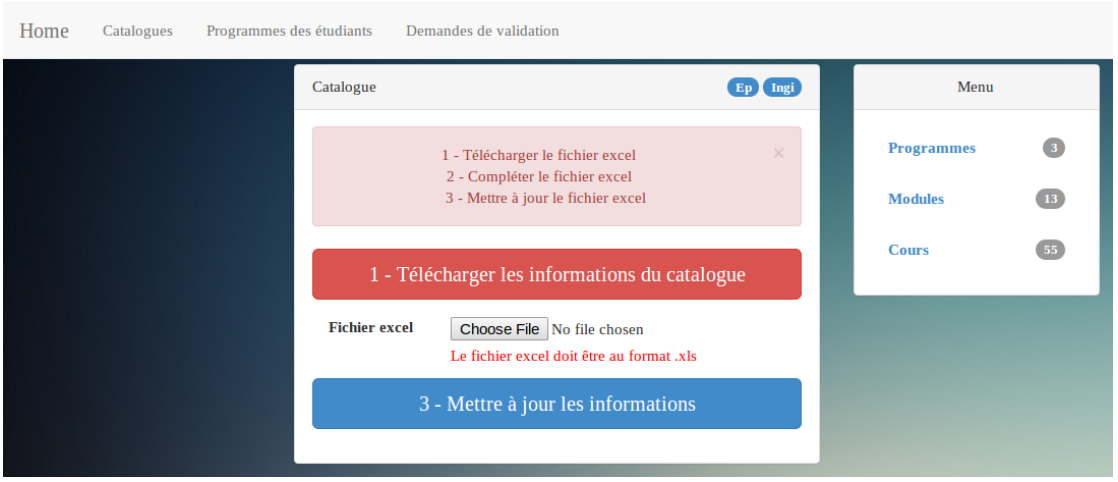
FIGURE 10 – Création d'un catalogue de cours




Remplissez, les champs, sélectionner le fichier de graph désiré et cliquez sur *Créer le catalogue*

Vous arrivez sur la page présentée sur l'image suivante 11

FIGURE 11 – Le catalogue une fois créé




### Mise à jour des informations contenues dans le catalogue

Les informations contenues dans le graphe ne sont pas complète. C'est pourquoi il est possible (et même fortement conseillé d'en ajouter via un formulaire excel

Pour ce faire il est conseillé de commencer par télécharger le formulaire excel (en cliquant

sur le bouton en rouge sur l'image 11) contenant les informations du catalogue ainsi que certaines proposition d'attributs à ajouter au catalogue. Il est préférable de ne pas passer cette étape. Dans le cas contraire, vous serez obligé d'écrire les identifiants de tout les différents programmes, modules et cours un à un dans le formulaire. Pour un catalogue de 55 cours, cela peut être long.

Une fois le *template* de formulaire téléchargé et complété , sélectionnez le et cliquez sur le bouton *Mettre à jour les informations* (en bleu sur l'image 11).

### .2.5 Création de programme de cours

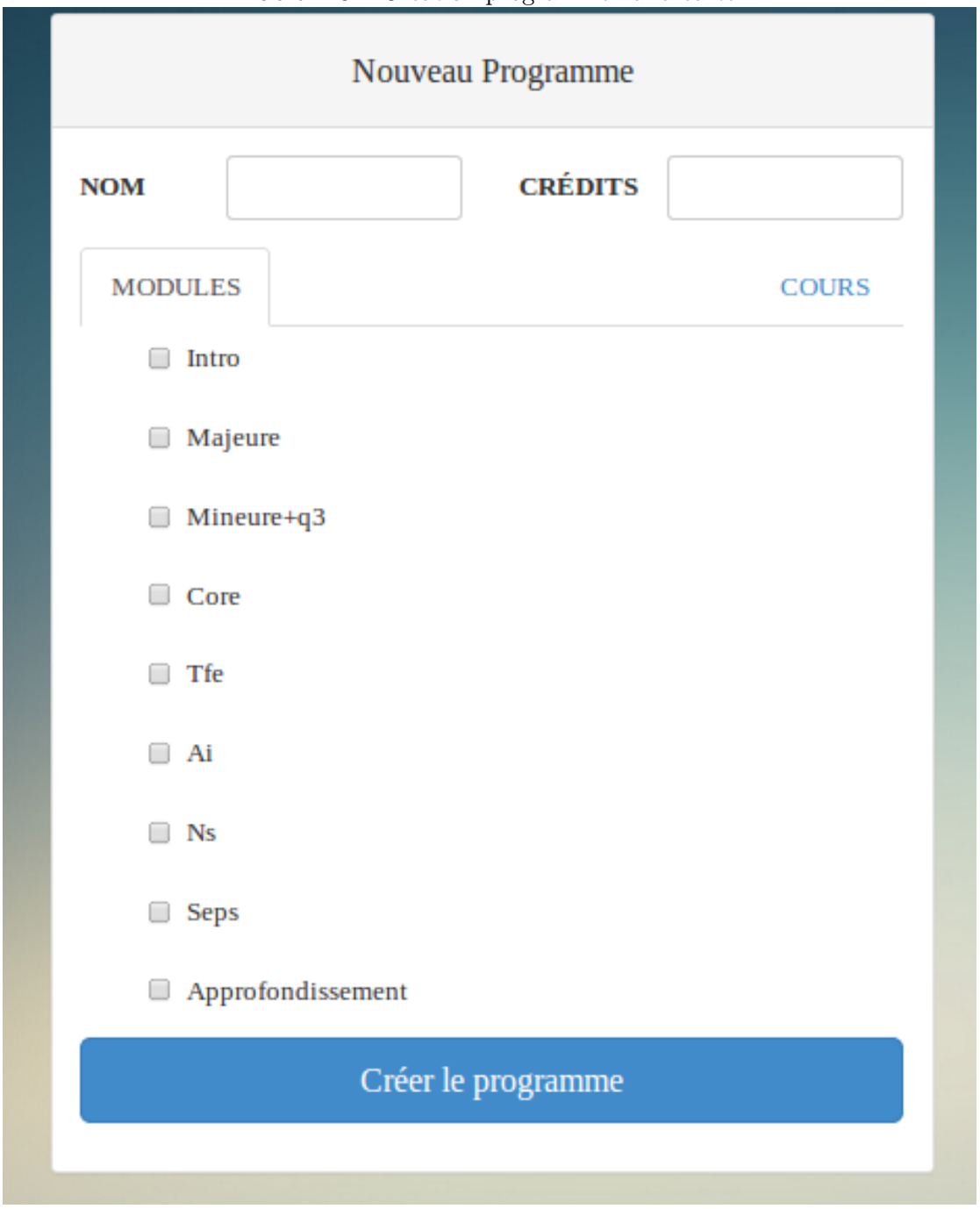
Il est possible de créer des programmes *à la carte* avec les modules et différents cours importés dans l'application lors de l'import de graphe. Pour ce faire, cliquer, dans le menu à droite sur le lien intitulé *Programme* (Image 11).

Vous arriverez sur une page comme présentée sur l'image 12 qui liste tout les programmes présent dans le catalogue. Vous pouvez inspecter les modules et les cours dont il est composé en cliquant sur le bouton *Plus d'informations*, ou même le supprimer.

FIGURE 12 – Programmes créés par défaut

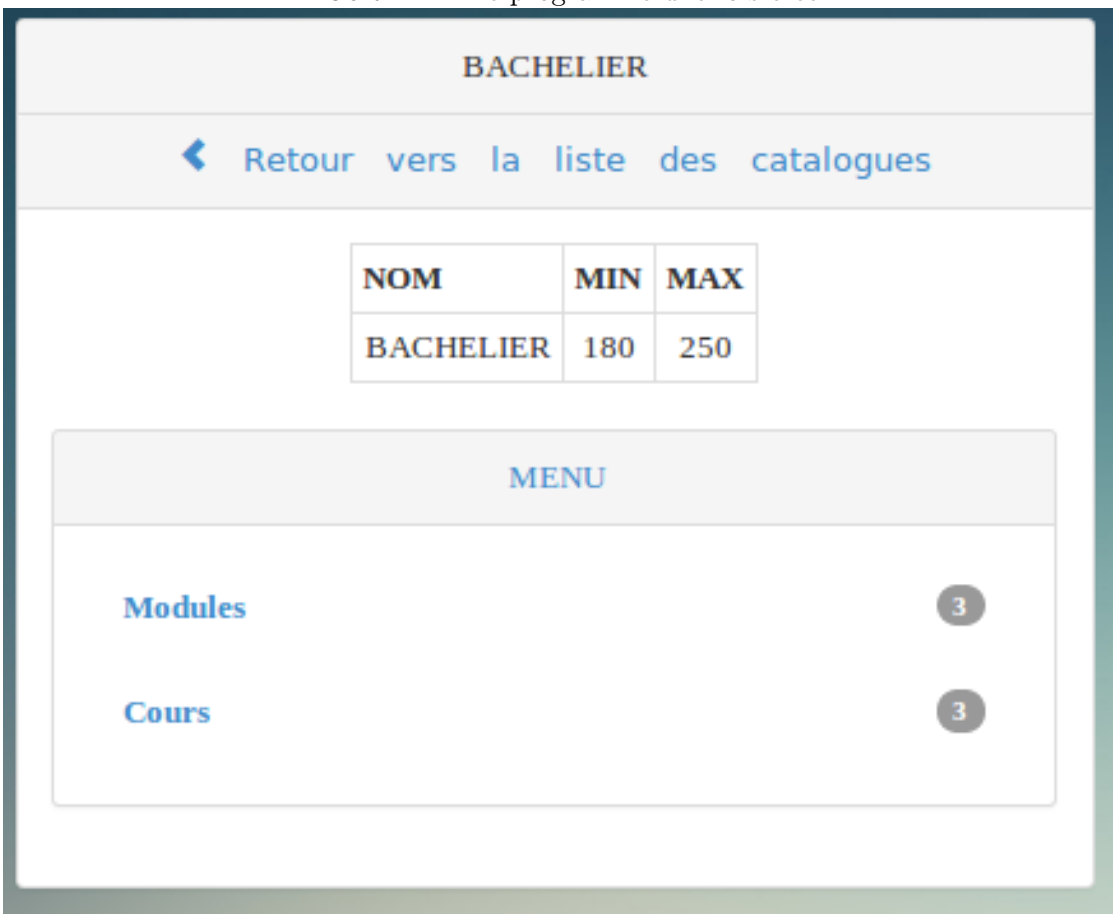

Pour créer un programme, il suffit de cliquer sur le bouton *Créer un programme*. Vous arriverez sur la page illustré sur l'image 10

FIGURE 13 – Création programme "à la carte"




Remplissez les différents champs et naviguez entre les onglets **MODULES** et **COURS** pour choisir les modules et cours dont le nouveau programme sera composé. Cliquez ensuite sur *Créer le programme*. Vous arriverez à la page illustrée sur l'image 14. En cliquant sur le menu déroulant intitulé *Menu*, vous pouvez accéder aux différents modules et sous-modules qui le compose.

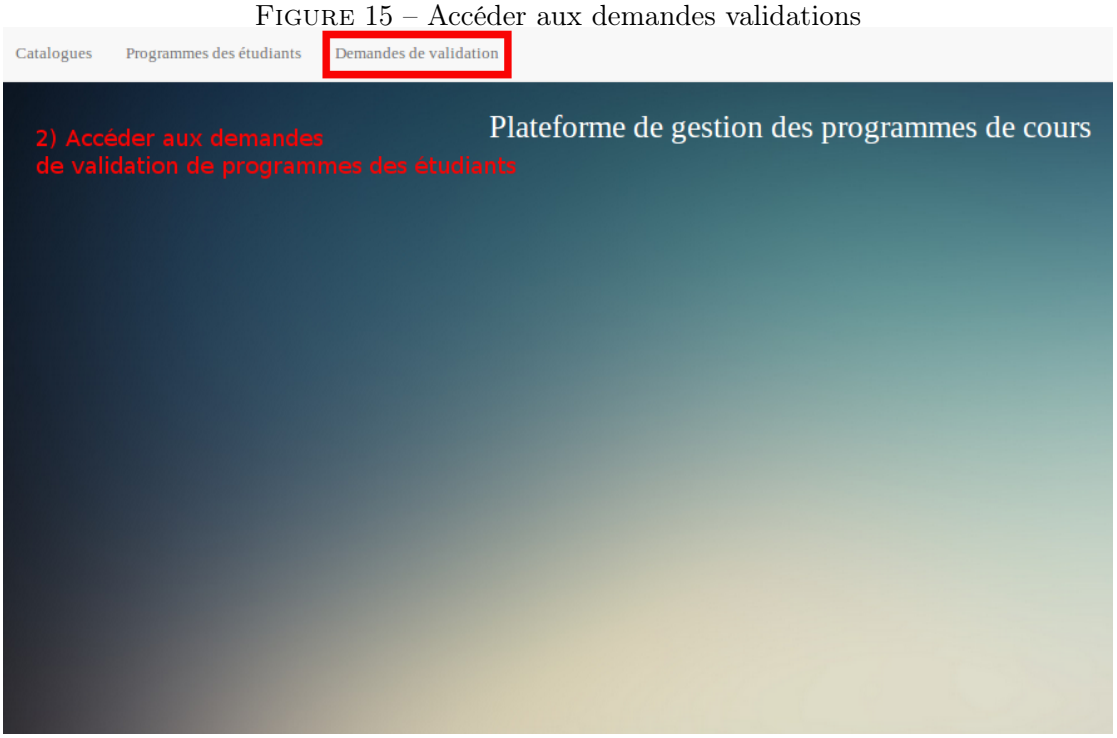
FIGURE 14 – Le programme une fois créé




## .3 Gestion des requêtes de validations

Lorsqu'un étudiant désire envoyer le programme qu'il s'est créé à la validation, une demande de validation est envoyé à la **commission INFO**. Pour accéder à ces demandes de validations, il suffit de cliquer sur le menu intitulé *Validations* entouré en rouge sur l'image 15.

FIGURE 15 – Accéder aux demandes validations




Sur la page des validations (Image 16) vous pouvez

- inspecter le programme que l'étudiant demande de valider en cliquant sur *Programme* (en bleu sur l'image 16)
- valider la requête en cliquand sur *Valider* (en vert sur l'image 16)
- refuser la requête en cliquant sur *Refuser* (en rouge sur l'image 16)

FIGURE 16 – Quelques demandes de validations