



Documentation technique

Projet de semestre

**Agenda étudiant**

**Étudiants :** Jérôme Moret, Paul Ntawuruhunga,

Thibaud Duchoud, David Kunzmann et Mario Ferreira

**Professeur :** Dr. René Rentsch

**Date :** 1er juin 2015 **Classe :** PRO-1-B

Table des matières

[1 Introduction 3](#_Toc421009364)

[2 Modèles 3](#_Toc421009365)

[3 Vue « Tâche » 3](#_Toc421009366)

[3.1 Structure 3](#_Toc421009367)

[3.2 Récupération et affichage des données 4](#_Toc421009368)

[3.3 QSqlTableModel 4](#_Toc421009369)

[3.4 QSqlRelationTableModel 5](#_Toc421009370)

[3.5 Les slots et les signaux 6](#_Toc421009371)

[3.6 Tri des tâches 6](#_Toc421009372)

[4 Vue « Résumé » 6](#_Toc421009373)

[4.1 Images de l'interface finale 6](#_Toc421009374)

[4.1.1 Lors de la sélection d'un cours 6](#_Toc421009375)

[4.1.2 Lors de la sélection d'un semestre 7](#_Toc421009376)

[4.2 Explication génériques 7](#_Toc421009377)

[4.2.1 L'ajout et la modification d'un cours ou d'un semestre 7](#_Toc421009378)

[4.2.2 Vue détaillée 8](#_Toc421009379)

[4.2.3 La bar latéral (SideBar) et la vue en arbre (TreeView) 8](#_Toc421009380)

[4.2.4 Barre d'outils (ToolBar) 8](#_Toc421009381)

[5 Tests 8](#_Toc421009382)

[6 Problèmes rencontrés 8](#_Toc421009383)

[6.1 Au sein de la vue tâche 9](#_Toc421009384)

[6.1.1 Tri et sélection 9](#_Toc421009385)

[6.1.2 Affichage 9](#_Toc421009386)

[6.2 Au sein de la vue « Résumé » 9](#_Toc421009387)

[6.2.1 Error LNK2019 : symbole externe non résolu 9](#_Toc421009388)

[6.2.2 Garder l'état du TreeView lors d'un rafraîchissement 10](#_Toc421009389)

[6.2.3 Scroll Aera et redimensionnement des widgets 10](#_Toc421009390)

[7 Améliorations possibles 10](#_Toc421009391)

[7.1 Au sein de la vue « Résumé » 10](#_Toc421009392)

[7.1.1 Calcul des moyennes 10](#_Toc421009393)

[7.1.2 Changements par rapport à l'interface initiale 10](#_Toc421009394)

[8 Conclusion 12](#_Toc421009395)

[9 Annexes 12](#_Toc421009396)

Table des figures

[Figure 1 : Structure de la fenêtre pour la vue Tâche 5](#_Toc421009397)

[Figure 2 : Schéma du Model-View-Programming 6](#_Toc421009398)

[Figure 3 : Capture d'écran de l'interface "Résumé" 7](file:///C:\Users\Jérôme\Documents\proagenda\Rapport\Documentation_technique.docx#_Toc421009399)

[Figure 4 : Capture d'écran de la vue "Résumé" lors d'un clic sur un semestre 8](#_Toc421009400)

[Figure 5 : Vue tâche selon le schéma de l'interface 10](#_Toc421009401)

[Figure 6 : Vue générale sur l'application finale 12](file:///C:\Users\Jérôme\Documents\proagenda\Rapport\Documentation_technique.docx#_Toc421009402)

[Figure 7 : Vue générale selon le Mockup de la documentation de développement 12](file:///C:\Users\Jérôme\Documents\proagenda\Rapport\Documentation_technique.docx#_Toc421009403)

[Figure 8 : Gestion des modules selon le mockup 12](file:///C:\Users\Jérôme\Documents\proagenda\Rapport\Documentation_technique.docx#_Toc421009404)

[Figure 9 : Gestion des modules sur l'application finale 13](file:///C:\Users\Jérôme\Documents\proagenda\Rapport\Documentation_technique.docx#_Toc421009405)

# Introduction

Dans ce document vous trouverez toutes les informations qui ont trait à l’aspect technique de la partie développement du projet.

Vous trouverez notamment l’architecture de chaque partie, de quelle manière le problème a été abordé, les problèmes rencontrés au sein de chaque partie ainsi que les améliorations possibles, s’il y en a.

# Modèles

Notre application utilise 3 modèles pour gérer les données. Ces modèles sont des simples liens reliés à leur table correspondante. Chaque modèle est unique (singleton).

Le lien entre la table et son modèle se fait à l’aide d’un attribut du type « **QSqlRelationalTableModel** ». Ce type nous est fourni par Qt et permet de gérer des bases de données **SQLite**. Une méthode existe sur le modèle pour pouvoir récupérer cet attribut.

Selon le modèle, différentes méthodes ont été implémentées. Par exemple, pour le modèle « Task » la méthode « addtask() » a été implémentée de manière à pouvoir ajouter une tâche.

# Vue « Tâche »

## Structure

L’affichage dans le Framework Qt est basé sur des widgets. Ces widget sont simplement des objets affichables. Une fenêtre de notre application est une collection de widget. Ceci est le cas pour notre fenêtre tâche qui compte quatre widgets principaux encapsulant eux-mêmes d’autre widgets.

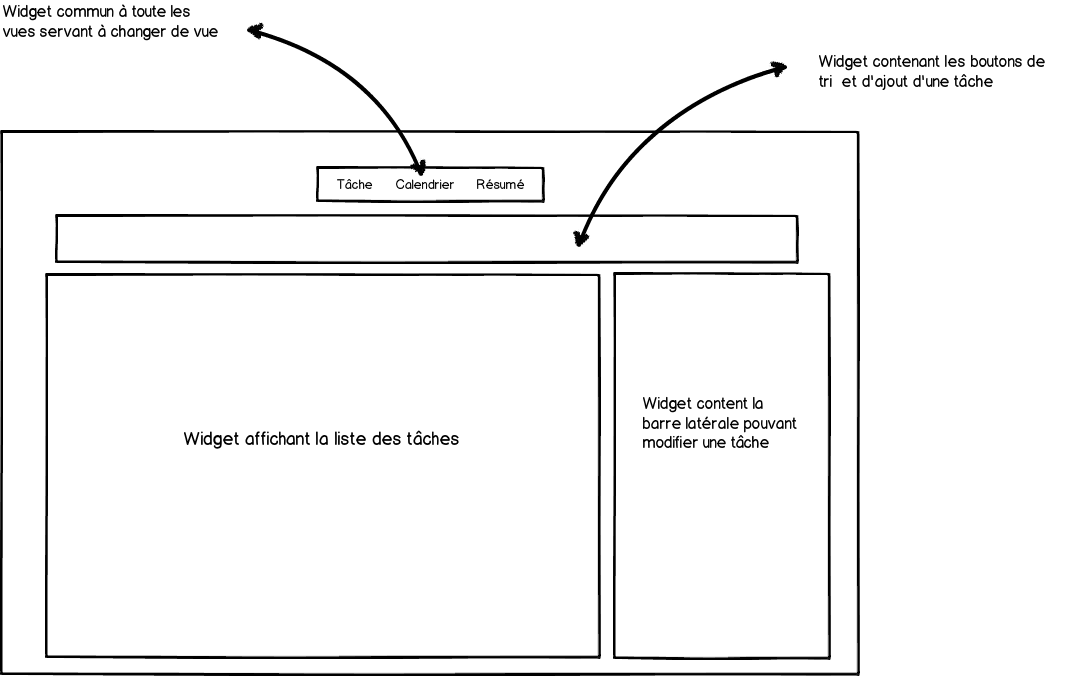


Figure 1 : Structure de la fenêtre pour la vue Tâche

Chacun de ces widgets contient plusieurs autre widgets tels que les boutons ou les champs de texte. Cette structure nous permet de modifier un widget sans devoir modifier la totalité de l’application. Par exemple en passant de la vue Tâche à la vue Résumé nous avons simplement à cacher le widget visible et à le remplacer par un autre widget. Cette opération est réalisée avec une pile de widget, structure propre au Framework Qt **QStackedWidget**.

## Récupération et affichage des données

La vue tâche consiste à afficher les tâches créer par l’utilisateur. Afin de se faire nous avions pour but d’aller récupérer les différentes tâche stockées dans notre base de données à l’aide de requêtes SQL. Pour cela nous avons créé une classe **sqlConnection** qui permet de se connecter à la base de données mais également à réaliser certaine action telles que récupérer toute les tâches. Cette façon de procéder est simple d’implémentation mais demande beaucoup de lignes de code surtout lorsqu’il faut ensuite afficher ces données. C’est pour cette raison que nous avons décidé, dans le mesure du possible, non pas de créer nous-même nos requêtes SQL mais d’utiliser les fonctionnalités offertes par le Framework Qt. En effet ce Framework nous met à disposition un outil très puissant qu’est les classes **QSqlTableModel** ainsi que **QSqlRelationTableModel**.

## QSqlTableModel

Cette classe nous permet de lier un modèle à une base de données. En d’autre mot nous lions chaque champ de notre modèle Task avec un champ dans notre base de données. Ceci a pour but que lorsque nous chargeons une tâche depuis la base de donnée, cette dernière est instanciée dans notre modèle. Il est ensuite extrêmement pratique de réaliser des actions sur notre modèle, actions qui seront répercutées directement sur notre base de données.

L’étape suivante est de pouvoir afficher ces données et une fois encore Qt nous met à disposition des méthodes permettant de lier une vue à un modèle (**mapper**). C’est de cette manière que nous pouvons réaliser des changements depuis l’interface de l’utilisateur (par exemple créer une nouvelle tâche) qui se répercuteront directement dans notre base de données. Cela sans avoir à créer chacune des commandes SQL requises pour ces opérations.

Sur le schéma ci-dessous nous voyons une représentation du lien entre la vue (view) et les données (data). Ces dernières étant stockées dans notre base de données. Le mapper cité plus haut est représenté par le cercle vert (delegate).

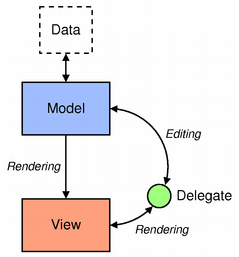


Figure 2 : Schéma du Model-View-Programming[[1]](#footnote-1)

## QSqlRelationTableModel

Ce deuxième outil nous a permis d’améliorer l’affichage de nos données en utilisant le moins possible de requête SQL et donc en gardant un code plus simple et plus lisible. **QSqlRelationTableModel** nous est utile lorsque nous avons une « Foreign Key » dans notre table SQL. En effet si nous utilisons seulement **QSqlTableModel** nous stockerions dans le modèle les informations qui sont dans une table de notre base de données et donc pour le modèle de tâche au lieu de nous d’obtenir le nom du cours auquel notre tâche est lié nous obtiendrons son identifiant. Il faudrait pouvoir ensuite rechercher cette « id » et récupérer le nom auquel elle correspond. C’est exactement à cela que sert **QSqlRelationTableModel**. Nous lions une colonne dans une table SQL à une autre colonne dans une autre table. Ceci a pour effet d’afficher le nom du cours plutôt que son identifiant.

## Les slots et les signaux

Un point important dans le Framework Qt est les slots ainsi que les signaux. Nous pouvons grâce aux signaux passer un message à un slot, ceci permet à différentes classes et/ou objets de communiquer. Nous avons donc utilisé ces signaux lors de modification des mappers et/ou des modèles afin de pouvoir appeler des fonctions pour modifier l’affichage par exemple. Dans la vue tâche, les boutons de tris sont par exemple tous connectés à un slot. Ces boutons émettent un signal lorsque l’on clique dessus, le slot récupère ce signal et le tri sélectionné est appliqué. Nous procédons ainsi car l’affichage des tâches triées et les boutons de tri se situent dans deux widgets différents.

## Tri des tâches

Dans le but de pouvoir trier nos tâches par ordre d’échéance ou par ordre de priorité, nous avons utilisé des fonctionnalités qu’offrent Qt, la classe **QSqlTableModel** et la méthode **sortBy** qui trie notre modèle, lié à une table SQL, en fonction de la colonne dans cette table (par exemple la priorité).

# Vue « Résumé »

## Images de l'interface finale

### Lors de la sélection d'un cours

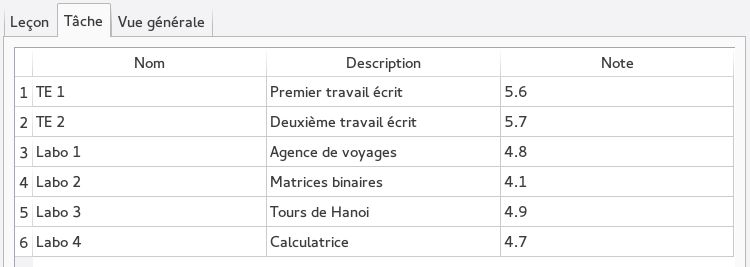
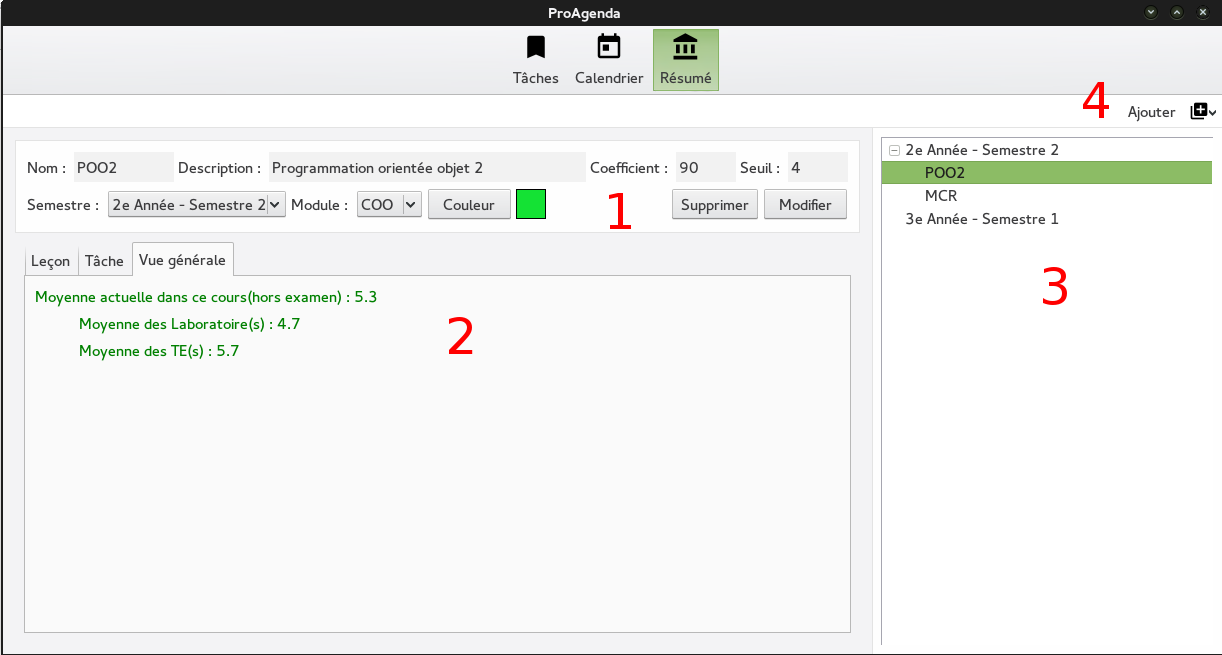
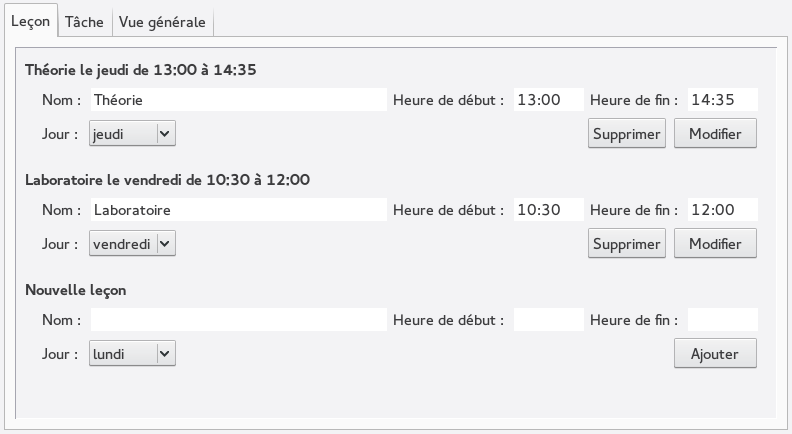


Figure 3 : Capture d'écran de l'interface "Résumé"

### Lors de la sélection d'un semestre

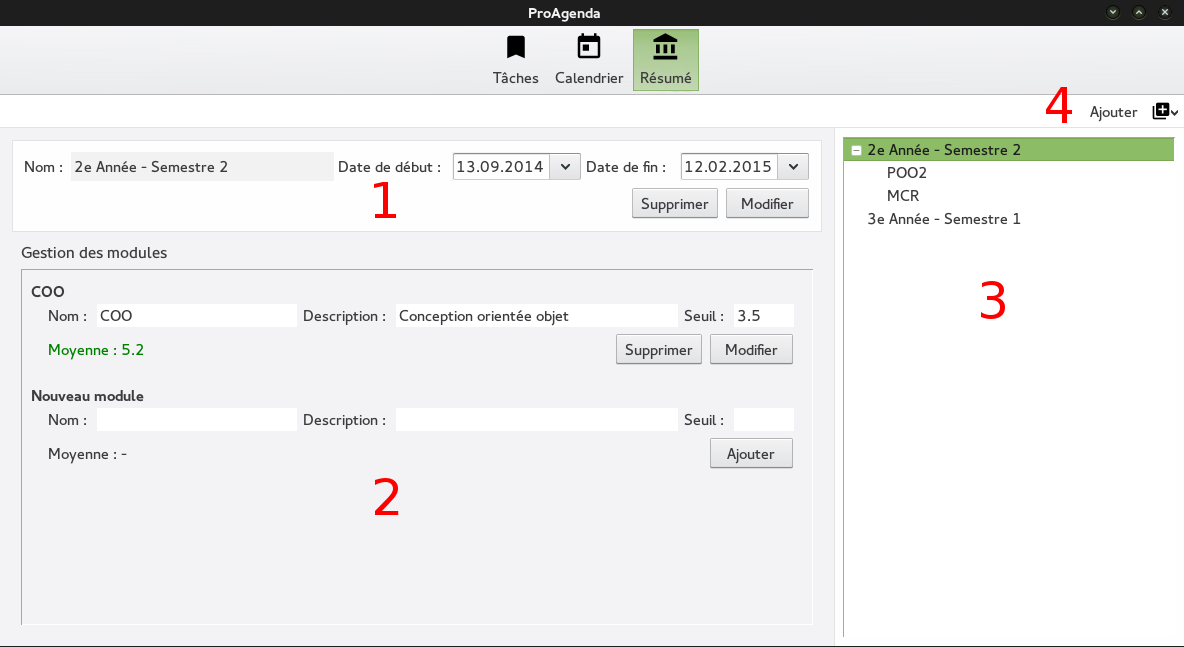


Figure 4 : Capture d'écran de la vue "Résumé" lors d'un clic sur un semestre

## Explication génériques

Comme on peut le voir, il y a des similitudes entre certaines vues (ajout/modification des cours, semestres, modules et leçons).

Ces vues fonctionnent toutes sur le même principe. Lors d'un rafraîchissement ou d'un changement de contexte, la vue précédente est supprimée et une nouvelle est créée avec les nouveaux paramètres.

Il y a également 2 modes à chaque vue, le mode **édition** et le mode **ajout**.

En mode édition le bouton de suppression est présent alors qu'il ne l'est pas en mode ajout.

Lorsqu’on crée une de ces vues, on à la possibilité de la passer en mode édition en fixant les valeurs dans les champs et en changement un booléen dans la vue. Le comportement lors du clic sur le bouton d'ajout/modification dépend donc du mode dans lequel se trouve la vue. On va soit mettre à jour (update) le champ correspondant dans la base de données ou insérer (insert) le nouveau champ.

### L'ajout et la modification d'un cours ou d'un semestre

Dans cette partie, il y a un widget contenant les informations d'un semestre ou d'un cours.

Le widget affiché à cet endroit change en fonction de l'action faite par l'utilisateur. Lors du clic sur le bouton d'ajout d'un cours ou d'un semestre (dans la barre d'outils (4)) on va vider le « layout » contenant ce widget et recréer le widget vide correspondant.

Lorsqu'on sélectionne un cours ou un semestre, on va également vider le « layout » et on va recréer le widget en mode édition. On va fixer les valeurs dans chaque champ avec ceux correspondants au cours ou au semestre sélectionné.

La modification ou suppression d'un cours ou d'un semestre met à jour l'arbre et les modules.

### Vue détaillée

#### Semestre

Lorsqu'on sélectionne un semestre, cette partie affiche la liste des modules et permet de les modifier. Nous avons utilisé une zone de défilement (scroll area) pour gérer l'affichage des modules lorsqu'il y en a trop.

#### Cours

Lorsqu'on sélectionne un cours, cette partie plus détaillée s'affiche sur le cours.

On peut y voir la moyenne actuelle ainsi que celle de ces tâches par type, on peut voir la liste des tâches et enfin les leçons de ce cours.

Les leçons utilisent le même principe que pour les modules.

### La bar latéral (SideBar) et la vue en arbre (TreeView)

Dans cette partie, nous avons créé notre propre classe **SideBarTreeView** pour la création de l'arbre et son affichage (ajout dans le layout) et nous avons également crée notre propre classe **TreeView** (hérité de **QTreeView**) afin de pouvoir gérer les évènements lors d'une sélection dans l'arbre.

Nous avons également crée un modèle personnel (**TreeModel**) afin de rendre publique une méthode qui est protégée pour sauvegarder et restaurer l'état de l'arbre après des suppressions/créations ou autres changements (c. f. problèmes rencontrées point 2 pour plus d'informations).

Ce modèle va contenir les informations sur les cours et les semestres. Il est rempli à l'aide de 2 autres modèles (**QSqlTableModel**), un pour les semestres et un pour les cours.

Les cours et les semestres sont chargés lors de l'instanciation de la classe **SideBarTreeView**.

Lors d'un clic sur un cours ou un semestre, on change simplement la vue en 1 et la vue 2 pour afficher respectivement la modification d'un cours et l'affichage des détails du cours et la modification d'un semestre et l'affichage des modules.

### Barre d'outils (ToolBar)

Cette barre est spécifique à chaque vue. Dans cette vue, nous avons simplement implémenté cette barre en ajoutant un menu contenant 2 boutons : « Ajouter un cours » et « Ajouter un semestre ».

# Tests

# Problèmes rencontrés

## Au sein de la vue tâche

### Tri et sélection

Lors de notre implémentation de la vue tâche nous avons remarqué, que lors d’un tri des tâches, la tâche sélectionnée était différente avant et après le tri. Ceci est dû au fait que la sélection concerne la ligne sur laquelle se trouve la tâche. En raison d’un retard sur le développement de notre application nous n’avons pas eu le temps de régler ce problème de la meilleure manière qui soit. Cependant nous avons cherché un moyen de résoudre cela et avons trouvé que Qt offrait la possibilité de trier nos modèles en gardant la sélection. Ceci se ferai à l’aide de la classe **QSortFilterProxyModel** cette couche se situe entre le modèle et la vue et nous permettrai de garder la sélection entre les différents tris et filtres. Nous n’avons pu implémenter cette classe dans notre application mais nous avons implémenté une solution non définitive mais entièrement fonctionnelle.

### Affichage

Nous n’avons également pas pu obtenir la structure que nous avions prévue dans le mockup présenté dans le cahier des charges. Nous avions pour but d’afficher la liste des tâches triées par cours de la manière suivante :

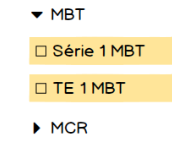


Figure 5 : Vue tâche selon le schéma de l'interface

La représentation sous forme d’arbre n’a pas été implémentée car afin d’obtenir une bonne lisibilité avec la grande quantité d’informations que représente une tâche, il a été plus pratique d’orienté notre design vers une simple liste triable par cours.

## Au sein de la vue « Résumé »

### Error LNK2019 : symbole externe non résolu

Ce problème est survenu souvent sans vraiment de raison apparente, au début. Après recherche, on a trouvé que le problème venait d'inclusions cycliques.

Dans la vue résumé, il y a besoin d'inclusions cycliques. La vue principale (**VSummary**) doit connaître les sous-vues pour pouvoir les créer et les sous-vues doivent connaître la vue principale pour pouvoir la rafraîchir lors de la modification d'une de ces sous-vues.

Pour contourner le problème, il nous a fallu changer la manière d'inclure les fichiers lorsque ce cas se présentait. Dans la vue résumé, nous avons simplement laissé les inclusions des sous-vues dans le .h. Dans le .h des sous-vues, nous avons déclaré la classe (Class **VSummary**;) pour permettre la déclaration. Et dans le .cpp de la sous-vue, nous avons inclus la vue résumé normalement (#include « vsummary.h »).

### Garder l'état du TreeView lors d'un rafraîchissement

Lors du rafraîchissement de l'arbre d'affichage des semestres et cours, l'ancien est détruit et un nouveau est créé. Le problème est que le nouvel arbre n'est pas « ouvert » et ne garde pas les configurations de l'ancien.

Il fallait, en premier, créer un modèle spécifique pour le **TreeView** (ce n'est pas obligatoire qu'il soit exclusivement spécifique au **TreeView** mais nous avons choisi de le faire ainsi) afin de pouvoir définir une méthode *protected* en public pour y accéder de l'extérieur.

Avant la destruction de l'ancien arbre, on le parcourt afin de sauvegarder les valeurs de chacune de ses lignes ouvertes.

Après la création du nouvel arbre, on a simplement à parcourir la liste des valeurs qui étaient ouvertes et on ouvre les lignes dans le nouvel arbre.

### Scroll Aera et redimensionnement des widgets

L'affichage des modules et des leçons se fait dans une scroll area (une zone de défilement). Nous ajoutons des widgets dans cette zone afin de pouvoir défiler lorsqu'il y en a trop.

Malheureusement au début, nous ajoutions nos widgets dans la scroll area mais aucune affiche ne se faisait.

Nous avons dû créer un widget « parent » et ajouter les sous-widgets dans ce widget « parent » et ensuite seulement ajouter ce widget parent dans la scroll area.

Et nous avons également dû accepter le redimensionnement des widgets dans la scroll area afin que celle-ci se charge de ne laisser qu'un espace vide minimum entre les widgets et qu'il y ait un affichage correct.

# Améliorations possibles

## Au sein de la vue « Résumé »

### Calcul des moyennes

Il aurait peut-être fallu plus réfléchir sur la manière de calculer et stocker les moyennes des cours et modules.

Pour l'instant, la moyenne des cours et modules n'est pas stockée dans la base de données mais est calculée directement dans l'application, à chaque changements sur un cours ou un module (sélection, modification et rafraîchissement de la vue correspondante). Ça demande à chaque fois du calcul « inutile » et un accès à la base de données.

Nous aurions pu stocker ces moyennes directement dans la base de données, dans la table Course et Module. Créer un Trigger qui mettrait à jour cette moyenne lors d'un changement dans les tables Type, Task et/ou Course. Le calcul se ferait au niveau de la base de données.

La 2ème méthode à l'avantage de ne calculer la nouvelle moyenne que lors d'un changement dans une table et que le calcul se fait au niveau de la base de données. Lors de la sélection d'un cours, la moyenne n'est pas recalculée et lors d'une modification ou autre, la moyenne n'est pas systématiquement recalculée non plus.

### Changements par rapport à l'interface initiale

Il y a eu des changements de design dans toutes les vues mais certains sont plus mineurs que d'autres. Quand c'est simplement une réorganisation de l'affichage ou autre, ce n'est pas important de signaler un changement dans l'interface finale.

#### Détails des cours

##### Vue générale

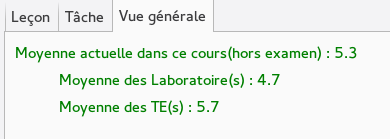
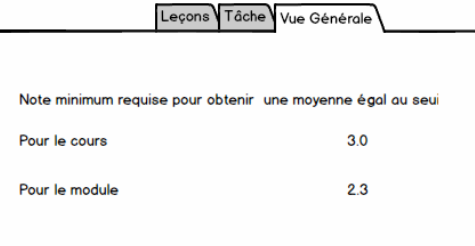
****

Figure 6 : Vue générale sur l'application finale

Figure 7 : Vue générale selon le Mockup de la documentation de développement

Plutôt que de calculer la note minimale à obtenir pour être au-dessus du seuil, nous avons préféré afficher la moyenne du cours à cet endroit ainsi que la moyenne des types de tâches (TE, Labo …).

Des couleurs dans le texte ont également été ajoutées pour permettre de repérer de suite les endroits qui poseraient problème. Si le texte est en vert c'est qu'il n'y a pas de problème (moyenne au-dessus du seuil et de la moyenne scolaire (4)), si le texte est en orange, il est entre le seuil et la moyenne scolaire et sinon il est en rouge.

##### Gestion des modules

Figure 8 : Gestion des modules selon le mockup

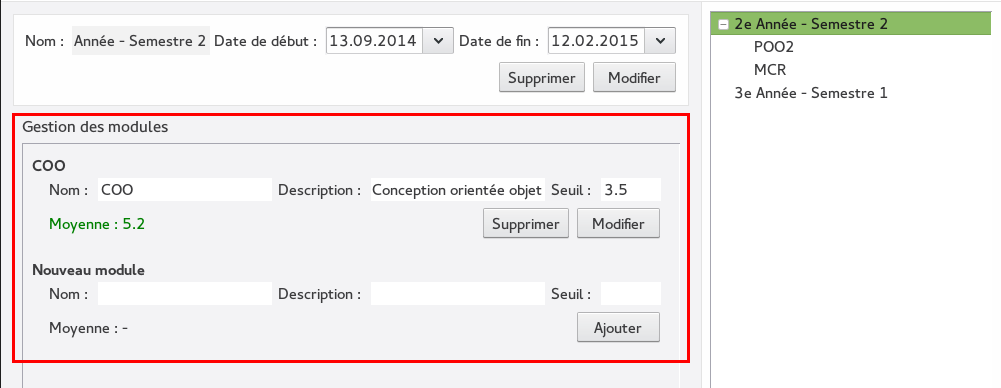
****Au début, lorsqu'on cliquait sur un semestre, il y avait affiché la liste des cours ainsi que la moyenne de chacun. Nous n'avions pas prévu de gestion de modules dans cette fenêtre.

Figure 9 : Gestion des modules sur l'application finale

Pour remédier au problème, la gestion des modules est faite lorsqu'on clique sur un semestre. Et elle a été créée sur le même design que la vue d'affichage des leçons.

L'ajout des modules a également été déplacé dans la vue d'affichage des modules pour que les gestions soient faites au même endroit et que, si un besoin futur se présente, il soit facile de déplacer la gestion des modules dans une autre fenêtre ou une autre vue.

# Documentation C++ (Qt)

La documentation C++ (Qt) de notre application a été générée à l’aide de Doxygen. Nous ne l’avons pas fourni dans un format papier imprimé dû au nombre de pages colossales mais aussi pour protéger l’environnement.

Vous pouvez néanmoins la retrouver en version PDF et HTML sur le CD fourni à la fin de cette documentation. (cf **Annexe A :** CD contenant les fichiers du projet.)

# Conclusion

Comme on a pu le voir le Framework Qt offre énormément de fonctionnalités. Lorsqu’on développe une partie du projet il faut impérativement se référer à la documentation Qt et son forum car pour chaque manière que l’on aurait de coder une partie, il existe souvent une manière simplifiée de le faire, comme le sont, par exemple, les signaux et les slots, assez déroutant au début mais si pratique au final.

En parallèle, il a fallu gérer nos sources et cela a été effectué à l’aide des commandes git sur un dépôt Bitbucket. Lorsque plusieurs développeurs s’occupaient de partie commune, nous avions créé des branches. Mais lorsque les « commits » commençaient à se faire nombreux, il devenait très difficile de gérer les « merge » vers le master. Au final, on peut dire que toutes ces difficultés nous ont beaucoup appris à gérer ce gestionnaire de version.

Pour finir, le travail de groupe nous a fait prendre conscience de la difficulté qu’engendre la répartition des tâches. En effet, distribuer des tâches sur 5 personnes fût compliqué, il faut toujours garder un œil sur le travail qu’effectue chaque collaborateur pour réagir directement lorsqu’un collaborateur part dans une mauvaise voie ou qu’il n’effectue pas son travail. Quand on voit la difficulté qu’implique un partage des tâches sur 5 personnes, on n’ose imaginer celle sur une équipe d’une vingtaine de personnes.

# Annexes

Le média suivant est encore joint ci-après :

* **Annexe A :** CD contenant les fichiers du projet.

1. http://doc.qt.io/qt-4.8/model-view-programming.html [↑](#footnote-ref-1)