Laboratoire de Programmation en C++

2ème bachelier en informatique :
orientations « informatique industrielle », « réseaux et télécommunications » et « développement d'applications »
(1er quadrimestre)
Année académique 2024-2025

« Gestion d'horaires de cours »

Anne Léonard
Patrick Quettier
Jean-Marc Wagner

Introduction

1. Informations générales : UE, AA et règles d'évaluation

Cet énoncé de laboratoire concerne l'unité d'enseignement (UE) suivante :

« Développement Système et orienté objet »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Programmation orientée objet en C++** (45h, Pond. 45/100)
- **Développement système en C sous Linux** (60h, Pond. 55/100)

Ce laboratoire intervient dans la construction de la côte de l'AA « Programmation orientée objet en C++ ».

La cote de l'AA « Programmation orientée objet en C++» est construite selon :

♦ théorie : un examen écrit en janvier 2025 et coté sur 20

♦ laboratoire **(cet énoncé)** : une évaluation globale en janvier fournissant une note de laboratoire sur 20. Des « check-points » réguliers formatifs auront lieu pendant tout le quadrimestre.

♦ note finale : **moyenne géométrique de la note de théorie (50%) et de la note de laboratoire (50%)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session.

1) Chaque étudiant doit être capable d'expliquer et de justifier l'intégralité du travail.

2) En 2ème session, un **report de note** est possible pour la théorie ou le laboratoire **pour des notes supérieures ou égales à 10/20**. Les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

3) Les consignes de présentation des travaux de laboratoire sont fournies par les différents professeurs de laboratoire

Consigne importante concernant ChatGPT :

« **ChatGPT** (ou toute aide « IA » à la production automatique de code) est ton ami mais… Pour qu'il le reste, tu devras être capable d'expliquer tout le code généré par **ChatGPT**. Tout code généré par **ChatGPT** que tu seras incapable d'expliquer en détails débouchera sur une cote nulle concernant la question sur ce code (**ET** pour la partie concernée du projet), même si celui-ci est correct et fonctionnel. »

2. Le contexte : Gestion d'horaires de cours

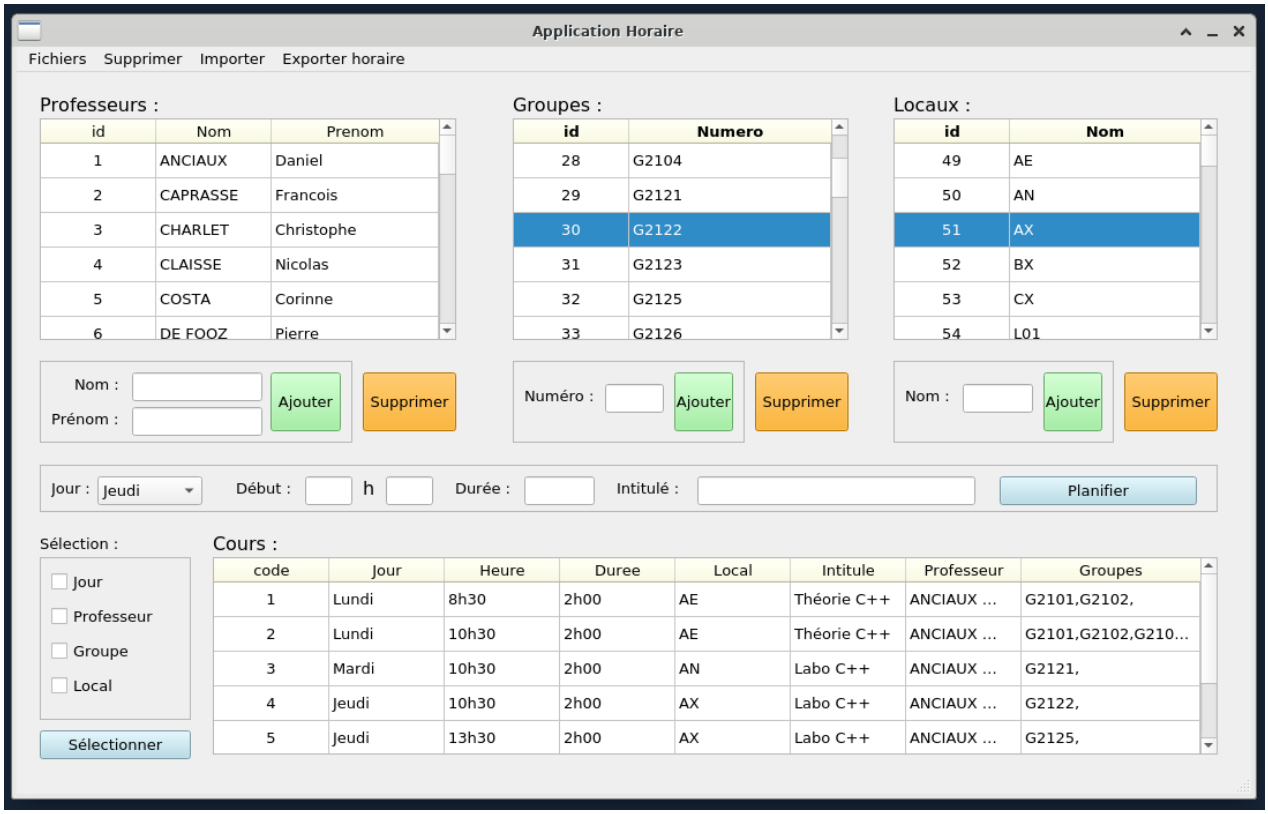
Les travaux de Programmation Orientée Objets en C++ (POO) se déroulent dans le contexte de la gestion d'un horaire de cours. Plus particulièrement, il s'agit de gérer la planification de cours, ainsi que d'éviter les conflits entre les ressources tels que les professeurs, les groupes d'étudiants et les locaux, principaux intervenants dans l'application.

Dans un premier temps, il s'agira de modéliser la gestion d'un horaire en général, ce qui fera intervenir la notion d'événement (Quoi ? Où ? Quand ?) et de timing (Quel jour ? Quelle heure ? Combien de temps ?) Nous aborderons ensuite, la problématique de la gestion des « planifiables (schedulable) » (Qui peut avoir un horaire ?), et des cours donnés par les professeurs à un ou plusieurs groupes en même temps (cours théorique/laboratoire).

L'application finale sera utilisée par toute personne réalisant la confection d'un horaire de cours. Elle lui permettra

- de concevoir un horaire de cours sur une semaine (horaire qui se répète donc de semaine en semaine),
- d'ajouter des professeurs, locaux, groupes à un horaire (ajout manuel ou importation),
- de planifier des cours pour ces différents intervenants en évitant les conflits,
- d'exporter l'horaire pour n'importe quel intervenant.

L'application aura l'aspect visuel suivant :



Il s'agit d'une interface graphique basée sur la librairie C++ Qt. Celle-ci vous sera fournie telle quelle. Vous devrez programmer la logique de l'application et non concevoir l'interface graphique.

3. Philosophie du laboratoire

Le laboratoire de programmation C++ sous **Linux** a pour but de vous permettre de faire concrètement vos premiers pas en C++ au début du quadrimestre (septembre-octobre) puis de conforter vos acquis à la fin du quadrimestre (novembre-décembre). Les objectifs sont au nombre de trois :

- mettre en pratique les notions vues au cours de théorie afin de les assimiler complètement,
- créer des "briques de bases" pour les utiliser ensuite dans une application de synthèse,
- vous aider à préparer l'examen de théorie du mois de janvier.

Il s'agit bien d'un laboratoire de C++ sous Linux. Il a également pour but de vous familiariser à un environnement de développement autre que Windows mais surtout en dehors d'un IDE totalement fenêtré. Pour cela, on vous fournira une **machine virtuelle (VMWare)** dès le premier laboratoire.

4. Méthodologie de développement

La programmation orientée objet permet une approche modulaire de la programmation. En effet, il est possible de scinder la conception d'une application en 2 phases :

1. La programmation des classes de base de l'application (les briques élémentaires) qui rendent un service propre mais limité et souvent indépendant des autres classes. Ces modules doivent respecter les contraintes imposées par « un chef de projet » qui sait comment ces classes vont interagir entre elles. Cette partie est donc réalisée par « le programmeur créateur de classes ».
2. La programmation de l'application elle-même. Il s'agit d'utiliser les classes développées précédemment pour concevoir l'application finale. Cette partie est donc réalisée par « le programmeur utilisateur des classes ».

Durant la première partie de ce laboratoire (**de la mi-septembre à mi-novembre**), vous vous situez en tant que « programmeur créateur de classes ». On va donc vous fournir une série de jeux de tests (les fichiers **Test1.cpp**, **Test2.cpp**, **Test3.cpp**, …) qui contiennent une fonction main() et qui vous imposeront le comportement (l'interface) de vos classes. Cette méthodologie de développement est bien connue en entreprise, il s'agit de la méthodologie appelée « **Test-Driven Development (TDD)** » :

> Le **Test-Driven Development (TDD)**, ou développement piloté par les tests en français, est une méthode de développement de logiciel qui consiste à concevoir un logiciel par petites étapes, de façon progressive, en écrivant avant chaque partie du code source propre au logiciel les tests correspondants et en remaniant le code continuellement. (wikipedia)

Dans la deuxième partie du laboratoire (**de mi-novembre à fin décembre**), vous vous situerez en tant que « programmeur utilisateur des classes » utilisant les classes que vous aurez développées précédemment. C'est dans cette seconde phase que vous développerez l'application proprement dite.

5. Code source de base

Tous les fichiers sources de base fournis peuvent être obtenus par

- Clonage du repository **GitHub** : https://github.com/hepl-dsoo/LaboCpp2024_Enonce

6. Convention de nommage « camelCase »

Toutes les variables que vous utiliserez devront être en anglais et respecter la convention de nommage **« camelCase »**. Dans **camelCase**, le premier mot commence par une lettre minuscule, et chaque mot subséquent commence par une lettre majuscule, sans espaces ni caractères de soulignement entre les mots. Par exemple : « numberOfStudents », « seatingCapacity », …

7. Planning, modalités et contenu des évaluations

a) Evaluation 1 (formative → donc non certificative) :

Le développement de l'application, depuis la création des briques de base jusqu'à la réalisation de l'application avec ses fonctionnalités a été découpé en une **série d'étapes à réaliser dans l'ordre**. A chaque nouvelle étape, vous devez rendre compte de l'état d'avancement de votre projet à votre professeur de laboratoire qui validera (ou pas) l'étape. On vous demande également :

- **Jusque l'étape 7 incluse** : vous travaillez seul et validez les étapes seul
- **A partir de l'étape 8** : vous travaillez par équipe de 2 et l'évaluation finale (voir point **b**) se fera par équipe de 2

b) Evaluation 2 (certificative → examen de janvier 2025) :

Porte sur :

- la validation des étapes non encore validées le jour de l'évaluation,
- le développement et les tests de l'application finale.
- Vous devez être capable d'expliquer l'entièreté de tout le code développé.

Date d'évaluation : jour de votre examen de Laboratoire de C++ (selon horaire d'examens)

Modalités d'évaluation : Sur la machine Linux fournie, selon les modalités fixées par le professeur de laboratoire.

Plan des étapes à réaliser

Etape 1 : Une première classe (Test1.cpp)

a) Description des fonctionnalités de la classe

Un des éléments de base de l'application est la notion d'événement (« Event »). Un événement est « quelque chose » qui se produit un certain jour, à une certaine heure, à un certain endroit et qui dure un certain temps. Dans cette première étape, nous allons commencer par aborder le « quoi ? ». Les notions de « où ? » et « quand ? » seront abordées plus loin.

Notre première classe, la classe **Event**, sera donc caractérisée par :

- Un **code** : un entier **code** (**int**) permettant d'identifier de manière unique un événement dans l'horaire.
- Un **intitulé** : une chaine de caractères **title** allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Cet intitulé représente le « quoi » d'un événement. Exemples : « Cinéma avec les potos », « Resto avec David », « Labo C++ », …

Comme vous l'impose le premier jeu de test (Test1.cpp), on souhaite disposer au minimum des trois formes classiques de constructeurs et d'un destructeur, des méthodes classiques getXXX() et setXXX() et une méthode pour afficher les caractéristiques de l'objet. La variable **title** de type chaîne de caractères sera actuellement un char*. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé dans cette 1ère étape de ce dossier de C++.** Vous aurez l'occasion d'utiliser la classe string dans la suite de ce projet.

b) Méthodologie de développement

Veillez à tracer (cout << …) vos constructeurs et destructeurs pour que vous puissiez vous rendre compte de quelle méthode est appelée et quand elle est appelée.

On vous demande de créer pour la classe **Event** (ainsi que pour chaque classe qui suivra) les **fichiers .cpp et .h** et donc de travailler en fichiers séparés. Un **makefile** permettra d'automatiser la compilation de votre classe et de l'application de tests.

Etape 2 (Test2.cpp) : Associations entre classes : agrégations + Variables statiques

Nous allons à présent ajouter à notre classe Event tout ce qui est nécessaire à la gestion du temps, c'est-à-dire « Quel jour ? », « A quelle heure ? » et « Combien de temps ? ».

a) Une agrégation par valeur (Essai1() et Essai2())

Il s'agit à présent de créer une classe modélisant la notion d'heure de la journée/de durée. On vous demande de créer la classe **Time** contenant

- Une variable **hour** de type **int** contenant l'heure de la journée/le nombre d'heures entières de la durée. Exemple : pour « 15h39 », heure contient 15.
- Une variable **minute** de type **int** contenant les minutes. Exemple : pour « 15h39 », minute contient 39.
- Un constructeur par défaut, un de copie et deux d'initialisation (voir jeu de tests), ainsi qu'un destructeur (dans la suite, nous n'en parlerons plus → toute classe digne de ce nom doit au moins contenir un constructeur par défaut et un de copie, ainsi qu'un destructeur).
- Les méthodes getXXX()/setXXX() associées,
- Une méthode display() permettant d'afficher l'heure/la durée sous la forme XXhXX.

Ensuite, on vous demande de créer une classe **Timing** contenant toutes les informations temporelles d'un événement :

- Une variable **day** de type **string** représentant le jour de l'événement, c'est-à-dire « Lundi », « Mardi », …
- Une variable **start** de type **Time** représentant l'heure du début de l'événement.
- Une variable **duration** de type **Time** représentant la durée de l'événement.

De nouveau, cette classe doit contenir les méthodes getXXX()/setXXX() associées à ses variables membres (ce que nous ne répéterons plus par la suite pour les autres classes), ainsi qu'une méthode display().

Bien sûr, les classes **Time** et **Timing** doivent posséder leurs propres fichiers .cpp et .h. La classe **Timing** contenant des variables membres dont le type est une autre classe, on parle d'**agrégation par valeur**, les objets de type Time font partie intégrante de l'objet Timing.

b) Une agrégation par référence (Essai3())

Nous pouvons à présent compléter la classe **Event** en ajoutant

- la variable **timing** (du type **Timing***) : il s'agit d'un pointeur vers un objet du type Timing et représentant donc toute l'information temporelle de l'événement. Ce pointeur peut être égal à **nullptr** si l'événement n'a pas encore été planifié.

- les méthodes setTiming(…) et getTiming(). Si le pointeur est **nullptr**, la méthode getTiming() ne sait rien retourner, il s'agit d'un cas d'erreur qui sera traité plus loin.

Notez que la méthode display() de la classe **Event** doit être mise à jour pour tenir compte du timing de l'événement.

La classe **Event** possède à présent un pointeur vers un objet de la classe **Timing**. Elle ne contient donc pas l'objet Timing en son sein mais seulement un pointeur vers un tel objet. On peut parler ici d'**agrégation par référence**, même si la classe **Event** reste responsable de l'allocation et de la destruction de l'objet pointé par le pointeur timing.

c) Mise en place de quelques variables statiques utiles (Essai4())

On se rend bien compte que la variable **day** de la classe **Timing** ne peut contenir que des chaînes de caractères bien précises, à savoir les jours de la semaine. Dès lors, on vous demande d'ajouter, à la classe Timing, **7 variables membres statiques constantes** de type **string** : **MONDAY** contenant « Lundi », **TUESDAY** contenant « Mardi », etc…

Afin d'assurer l'unicité de chaque événement (càd de la variable code de la classe **Event**), nous allons devoir gérer un indice courant « global » qui sera utilisé et incrémenté chaque fois que l'on créera un nouvel événement. Pour cela, on vous demande d'ajouter, à la classe **Event**, une **variable membre statique currentCode** de type **int**. Cette variable sera initialisée à 1 et incrémentée au besoin.

d) Mise en place d'un namespace

On se rend bien compte que certaines classes développées jusqu'ici ont des noms (comme **Time** par exemple) qui pourraient entrer en conflit avec celui de classes d'autres librairies. Pour éviter ces conflits, on vous demande de placer les classes **Time**, **Timing** et **Event** dans un **namespace** appelé **planning**.

Etape 3 (Test3.cpp) : Extension des classes existantes : surcharges des opérateurs

Il s'agit ici, de surcharger un certain nombre d'opérateurs des classes développées ci-dessus afin de faciliter la gestion des heures et durées.

a) Surcharge des opérateurs =, + et – de la classe Time

Dans un premier temps, on vous demande de surcharger les opérateurs =, + et - de la classe **Time**, permettant d'exécuter un code du genre :

```
Time h1,h2(13,30),h3,h4,d(30);

h1 = h2 ;       // h1 vaut à présent 13h30

h3 = h2 + 20 ;  // attention, h2 doit rester inchangé, h3 vaut 13h50
h3 = h3 + 30 ;  // h3 vaut à présent 14h20
h3.display() ;
h4 = 10 + h3 ;  // h4 vaut à présent 14h30
h4 = h4 + d ;   // h4 vaut à présent 15h00

h3 = h3 - 10 ;             // h3 vaut à présent 14h10
Time duration = 50 - d ;   // duration vaut à présent 0h20
Time duration = h3 - h1 ;  // duration vaut 0h40
```

On supposera que le résultat d'une opération d'addition ou de soustraction donne un résultat compris entre 0h00 et 23h59. Les dépassements ne sont pas acceptés et seront gérés plus tard.

b) Surcharge des opérateurs de comparaison de la classe Time

A terme, il sera nécessaire de classer les événements par ordre chronologique. Dans ce but, on vous demande de surcharger les opérateurs <, > et == de la classe **Time**, permettant d'exécuter un code du genre

```
Time h1,h2 ;
…
if (h1 > h2) cout << "h1 est posterieure h2" ;
if (h1 == h2) cout << autre message;
if (h1 < h2) …
```

c) Surcharge des opérateurs d'insertion et d'extraction de la classe Time

On vous demande à présent de surcharger les opérateurs << et >> de la classe **Time**, ce qui permettra d'exécuter un code du genre :

```
Time h1,h2;

cin >> h1 ; // permet d'encoder une heure en tapant « 14h20 »
cin >> h2 ;
cout << "De" << h1 << "a" << h2 << endl ; // Affiche l'heure sous forme XXhXX
```

d) Surcharge des opérateurs ++ et -- de la classe Time

On vous demande donc de programmer les opérateurs de post et pré-in(dé)crémentation de la classe **Time**. Ceux-ci in(dé)crémenteront l'heure/la durée de **30 minutes**. Cela permettra d'exécuter le code suivant :

```
Time h1(13,10), h2(16,50), h3(16,0), h4(9,50) ;

cout << ++h1 << endl ; // h1 vaut a présent 13h40
cout << h2++ << endl ; // h2 vaut à présent 17h20
cout << --h3 << endl ; // h3 vaut à présent 15h30
cout << h4-- << endl ; // h4 vaut à présent 9h20
```

e) Surcharge des opérateurs de comparaison de la classe Timing

Comme cela a été fait pour la classe **Time**, on vous demande de surcharger les opérateurs <, > et == de la classe **Timing** afin d'assurer l'ordre chronologique. Ceci permettra d'exécuter un code du genre

```
Timing t1,t2 ;
…
if (t1 > t2) cout << "t1 est posterieur que t2" ;
if (t1 == t2) cout << autre message;
if (t1 < t2) …
```

La comparaison porte tout d'abord sur le jour de la semaine. A jour égal, on compare l'heure de début. A jour et heure identiques, on dira que l'événement le plus court (en durée) est antérieur à l'autre.

Etape 4 (Test4.cpp) :

Associations de classes : héritage et virtualité

Nous allons à présent aborder la modélisation des intervenants du type « Qui peut avoir un horaire ? Qui est occupé pendant cette tranche horaire ? etc… » Dans notre application de gestion d'horaires de cours, trois types d'intervenant existent :

- Les professeurs, qui ont un nom, un prénom et un « horaire », c'est-à-dire une liste de cours à donner.
- Les groupes d'étudiants qui ont un nom (ex : « INFO2 D201 », « INFO2 R202 », « INFO2 I202 »,…) et un « horaire », c'est-à-dire une liste de cours à suivre.
- Les classes de cours qui ont un nom (ex : AN, PV11, LE0, …), et un « horaire », c'est-à-dire une liste des cours ayant lieu dans chaque classe.

Nous remarquons que tous ces intervenants ont au moins un point commun : un horaire, c'est-à-dire une liste de cours (type d'événement particulier). Nous donnerons à tous ces intervenants, pouvant avoir un « horaire », « être planifiés », l'appellation de « schedulable ». Ces schedulables se différencient par les autres variables membres.

a) Héritage : une classe Abstraite de base

Etant donné les considérations évoquées ci-dessus, l'idée est de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes aux trois intervenants, à savoir une liste de cours (mais celle-ci sera gérée plus tard). Cette classe de base sera la **classe abstraite** **Schedulable** et contiendra, pour l'instant :

- un identifiant **id** (de type **int**) qui identifie de manière unique un schedulable dans l'application. Cette variable doit disposer de ces accesseurs classiques.
- la méthode **string toString()** : qui retourne une chaine de caractères décrivant un schedulable (cette chaine sera construite en fonction des données propres à chaque schedulable ; voir plus bas) et **qui doit être une méthode virtuelle pure**. Cette fonction n'a donc pas de corps et devra obligatoirement être redéfinie dans les classes dérivées.
- La méthode **string tuple()** : qui retourne une chaine de caractères correspondant à un tuple (comme dans une BD) décrivant le schedulable. Ce tuple sera très utile lorsqu'il faudra afficher un schedulable dans une table de la future interface graphique de l'application ou lors de son enregistrement sur disque. Cette fonction **doit également être une méthode virtuelle pure**. → des explications plus précises sur cette méthode sont fournies plus loin

b) Héritage : les classes dérivées de la hiérarchie

Maintenant que nous disposons de la classe mère de la hiérarchie, on vous demande de programmer les classes dérivées décrites ci-dessous.

Un professeur est un schedulable qui a, en plus, un nom et un prénom. On vous demande donc de programmer la classe **Professor**, qui hérite de la classe **Schedulable**, et qui présente, en plus, les variables membres suivantes :

- **lastName** : chaine de caractères (**string**),
- **firstName** : chaine de caractères (**string**).

Un groupe d'étudiants est un schedulable qui a, en plus, un nom. Donc, on vous demande de programmer la classe **Group**, qui hérite de la classe **Schedulable**, et qui présente, en plus, la variable membre suivante :

- **name** : un entier (**int**).

Enfin, un local de classe est un schedulable qui a, en plus, un nom et un nombre de places assises. Vous devez donc programmer la classe **Classroom**, qui hérite de la classe **Schedulable**, et qui présente, en plus, les variables membres suivantes :

- **name** : chaine de caractères (**string**),
- **seatingCapacity** : un entier (**int**).

On redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire, par exemple les opérateurs d'affectation = ainsi que << (qui devra afficher ce que retourne la méthode **toString()** décrite ci-dessous).

c) Mise en évidence de la virtualité

Les trois classes dérivées (**Professor**, **Group**, **Classroom**) devront redéfinir les méthodes suivantes :

- **string toString()** : qui retourne une chaîne de caractères propre à chaque intervenant
  - **Professor** → « lastName firstName» (Ex : « Wagner Jean-Marc »)
  - **Group** → « name » (Ex : « INFO2 D201 »)
  - **Classroom** → « name (seatingCapacity) » (Ex : « AE (90) »)

- **string tuple()** : qui retourne une chaîne de caractères propre à chaque intervenant
  - **Professor** → « id;lastName;firstName» (Ex : « 3;Wagner;Jean-Marc »)
  - **Group** → « id;name» (Ex : « 2;INFO2 D201 »)
  - **Classroom** → « id;name (seatingCapacity)» (Ex : « 6;AE (90) »)

Le **caractère de séparation** entre les différents champs d'un tuple doit obligatoirement être un **;** pour la suite du projet.

Pour rappel, **l'opérateur <<** de chaque **« entité »** (professeur, groupe ou local) devra appeler la méthode **toString()** de cet objet.

Etape 5 (Test5.cpp) :

Les exceptions

On demande de mettre en place une structure minimale de gestion des erreurs propres aux classes développées jusqu'ici. On va donc imaginer la hiérarchie de classes d'exception suivante :

- **Exception** : **classe d'exception de base** ne contenant qu'une seule variable membre **message** de type **string** destinée à contenir un message d'erreur à l'intention de l'utilisateur. Cette classe va servir de base aux classes d'exceptions suivantes et à toute autre classe d'exception qui pourrait être utile à l'application finale.

- **TimeException** : **classe héritée de Exception** qui contient en plus une variable membre **code** de type **int** contenant un code d'erreur à l'intention du programmeur qui pourrait agir en conséquence. Ce code d'erreur pourra prendre l'une des 4 valeurs **INVALID_HOUR, INVALID_MUNUTE ou OVERFLOW** qui sont 3 **variables membres statiques constantes** de type **int** de la classe **TimeException**. Cette exception sera lancée avec le code d'erreur
  - INVALID_HOUR si l'on tente de créer ou modifier un objet de la classe **Time** avec des heures invalides, c'est-à-dire si heure est inférieure à 0 ou supérieure à 23. Par exemple, si h est un objet de la classe **Time**, h.setHour(-2) lancera une exception.
  - INVALID_MINUTE si l'on tente de créer ou modifier un objet de la classe **Time** avec des minutes invalides, c'est-à-dire si minute est inférieure à 0 ou supérieure à 59. Par exemple, si h est un objet de la classe **Time**, h.setMinute(70) lancera une exception.
  - OVERFLOW si le résultat d'une opération + (++) ou -- (--) est antérieur à 0h00 ou postérieur à 23h59.

- **TimingException** : **classe héritée de Exception** qui contient en plus une variable membre **code** de type **int** contenant un code d'erreur à l'intention du programmeur qui pourrait agir en conséquence. Ce code d'erreur pourra prendre l'une des 2 valeurs **INVALID_DAY, NO_TIMING** qui sont 2 **variables membres statiques constantes** de type **int** de la classe **TimingException**. Cette exception sera lancée avec le code d'erreur
  - INVALID_DAY si l'on tente d'affecter à un objet de la classe **Timing** un jour de la semaine qui ne fait pas partie des 7 variables membres statiques de la classe. Par exemple, si t est un objet de la classe **Timing**, t.setDay(« Vendremanche ») lancera une exception.
  - NO_TIMING lorsque l'on appelle la méthode **getTiming()** d'un objet **Event** qui n'a pas été planifié, c'est-à-dire dont le pointeur timing est **nullptr**.

Les classes **TimeException** et **TimingException** devront faire partie du namespace **planning**.

Le fait d'insérer la gestion d'exceptions implique qu'elles soient récupérées et traitées lors des tests effectués en première partie d'année (**il faudra donc compléter le jeu de tests Test5.cpp** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

Etape 6 (Test6.cpp) :

Première utilisation des flux

Il s'agit ici d'une première utilisation des flux en distinguant **les flux caractères** (manipulés avec les opérateurs << et >>) et **les flux bytes (méthodes write et read**). Dans cette première utilisation, nous ne traiterons que des **flux caractères**.

Le but de cette étape est de mettre en place la **sérialisation des objets** de notre application :

La **sérialisation** est un processus qui permet de convertir l'état d'un objet en une forme qui peut être sauvegardée (sur disque par exemple) et ensuite rechargée pour restaurer l'objet.

A partir de maintenant, ce sont les **opérateurs** << et >> de nos différents objets qui assureront respectivement la sérialisation et la désérialisation de nos objets sur un **flux**. Dans la suite, si on désire afficher un objet en console à l'aide de « cout << … », on utilisera plutôt la méthode **toString()** de chaque objet.

a) La classe Time se sérialise elle-même

On vous demande de modifier l'opérateur << de la classe **Time** de telle sorte que celui-ci écrive au **format texte** les heures et les minutes contenus dans un objet de ce type. Le format suivant devra alors être utilisé (si on prend comme exemple « 8h20 ») :

```
<Time>
<hour>
8
</hour>
<minute>
20
</minute>
</Time>
```

On remarque la présence de **balises ouvrantes et fermantes** pour le type d'objet (ici **Time**) et pour chaque variable membre de l'objet.

On vous demande également de modifier l'operateur >> de la classe **Time** afin qu'il relise ligne par ligne les données (et balises). Pour lire une ligne, on utilisera la fonction **getline(istream & flux, string & chaine)** de la std du C++. Etant donné ce qui a été écrit ici dans le fichier, l'opérateur >> devra ici faire appel 8 fois à la fonction getline (car 8 lignes écrites).

Attention que la lecture devra se faire dans le même ordre que l'écriture !

Notez que le format utilisé ici n'est rien d'autre que du **XML** (**Extensible Markup Language**). Il s'agit d'un format de fichier texte que vous pouvez lire avec n'importe quel éditeur de texte mais également dans un browser internet.

b) La classe Classroom se sérialise elle-même

On vous demande de modifier l'opérateur << de la classe **Classroom** de telle sorte que celui-ci écrive au **format texte** l'id, le nom et le nombre de places contenus dans un objet de ce type. Le format suivant devra alors être utilisé (si on prend comme exemple « 5;AN;110 ») :

```
<Classroom>
<id>
5
</id>
<name>
AN
</name>
<seatingCapacity>
110
</seatingCapacity>
</Classroom>
```

Il s'agit donc simplement de faire le même travail que pour la classe **Time** mais adapté à la classe **Classroom**. Il en est donc de même pour l'opérateur >> de la classe **Classroom**.

c) La classe Timing se sérialise elle-même

On vous demande de réaliser le même travail ici pour la classe **Timing**. Par exemple, pour le timing « Mardi a 8h20 (2h00) », le contenu du fichier texte généré sera :

```
<Timing>
<day>
Mardi
</day>
<start>
<Time>
<hour>
8
</hour>
<minute>
20
</minute>
</Time>
</start>
<duration>
<Time>
<hour>
2
</hour>
<minute>
0
</minute>
```

```
</Time>
</duration>
</Timing>
```

On notera qu'un objet de la classe **Timing** contient deux objets de la classe **Time**. Dès lors,

- L'**opérateur <<** de la classe **Timing** peut appeler avantageusement l'**opérateur <<** des objets **start** et **duration** contenu dans l'objet Timing
- L'**opérateur >>** de la classe **Timing** peut appeler avantageusement l'**opérateur >>** de la classe Time

d) La classe Event se sérialise elle-même

On vous demande de réaliser le même travail ici pour la classe **Event**. Par exemple, pour l'event « [code=17] Labo C++ (Timing : Mardi a 10h30 (2h00)) », le contenu du fichier texte généré sera :

```
<Event>
<code>
17
</code>
<title>
Labo C++
</title>
<timing>
<Timing>
<day>
Mardi
</day>
<start>
<Time>
<hour>
10
</hour>
<minute>
30
</minute>
</Time>
</start>
<duration>
<Time>
<hour>
2
</hour>
<minute>
0
</minute>
</Time>
</duration>
</Timing>
```

```
</timing>
</Event>
```

A nouveau, on notera qu'un objet de la classe **Event** contient un objet de la classe **Timing**. Dès lors,

- L'**opérateur <<** de la classe **Event** peut appeler avantageusement l'**opérateur <<** de l'objet ***timing** de l'objet Event
- L'**opérateur >>** de la classe **Event** peut appeler avantageusement l'**opérateur >>** de la classe Timing

Attention ! Un objet **Event** peut ou non avoir un objet **Timing**, reflété par la valeur du pointeur **timing** présent dans l'objet **Event** :

- Si ce pointeur est différent de **nullptr**, on appellera l'**opérateur <<** de ***timing** afin de l'écrire sur disque
- Lors de la lecture, l'**opérateur >>** devra vérifier s'il lit la balise **<timing>**. Si c'est le cas, l'**opérateur >>** de **Timing** sera appelé. Sinon, cela signifie que l'objet **Event** n'a pas de timing.

Etape 7 (Test7.cpp) : Une classe template permettant la sérialisation

Nous disposons à présent des opérateurs << et >> permettant d'écrire quelques-unes de nos classes au format **XML**. L'idée à présent est de construire une classe permettant de créer un fichier XML digne de ce nom et comportant non pas un objet (comme dans le test 6) mais une **collection d'objets** de même type. Cette classe devra être capable de gérer tous les types de nos données.

On vous demande donc de créer la **classe template XmlFileSerializer**

```
template<typename T>
class XmlFileSerializer {
…
};
```

ayant comme variable membre :

- La variable **file** (de type **fstream**) qui représente le lien avec le fichier sur disque
- La variable **filename** (de type **string**) qui contient le nom du fichier
- La variable **mode** (de type **char**) qui représente le mode d'ouverture du fichier : en lecture seule (dans ce cas mode aura la valeur **READ** qui est une variable membre statique constante de type char contenant **'R'** de la classe **XmlFileSerializer**) ou en écriture seule (dans ce cas mode aura la valeur **WRITE** qui est une variable membre statique constante de type char contenant **'W'** de la classe **XmlFileSerializer**). A vous à créer ces deux variables membres statiques.
- La variable **collectionName** (de type **string**) qui représente le nom de la collection de données. Par exemple, si le fichier est voué à contenir des objets de la classe **Time**, le nom de la collection pourrait être **« times ».** Cette variable servira également à mettre en place la balise racine du fichier XML.

Le format d'un fichier XML, dans le cas d'une collection de 3 objets de la classe **Time** est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<times>
<Time>
<hour>
8
</hour>
<minute>
45
</minute>
</Time>
<Time>
<hour>
9
</hour>
<minute>
15
```

```
</minute>
</Time>
<Time>
<hour>
9
</hour>
<minute>
45
</minute>
</Time>
</times>
```

Où on observe que

- La première ligne correspond à l'entête du fichier XML avec sa version et son format d'encodage. En ce qui nous concerne, il s'agit d'une chaine de caractères fixe qu'il faudra écrire et lire en début du fichier.
- Les balises **<times>** et **</times>** représentent le début et la fin de la collection. Le nom apparaissant dans cette balise (« **times** ») correspond à la variable **collectionName** de la classe **XmlFileSerializer**

La classe **XmlFileSerializer** sera donc chargée de lire ou écrire un tel fichier. Elle sera responsable de l'écriture et de la lecture de l'entête du fichier, et des balises de début et de fin de collection. Entre les deux, les opérateurs << et >> déjà développés pour nos différentes classes seront appelés.

Concernant les méthodes de la classe **XmlFileSerializer** :

- Elle disposera des accesseurs **getFilename()**, **getCollectionName()**, **isReadable()** et **isWritable()** mais pas des accesseurs permettant de modifier les variables membres. En effet, **les variables membres seront fixées une fois pour toute par le constructeur et ne pourront pas être remodifiées par la suite**.
- Elle disposera du constructeur d'initialisation **XmlFileSerializer(const string &fn, char m, const string& cn = « entities »)** où fn est le nom du fichier, m le mode d'ouverture et cn le nom de la collection. Ce dernier paramètre aura une **valeur par défaut « entities »** et ne sera utilisé que dans le cas d'une création et écriture dans le fichier. Ce constructeur est décrit en détails ci-dessous.
- Afin d'éviter des ouvertures multiples du même fichier, cette classe ne pourra pas avoir de constructeur par défaut, de constructeur de copie et d'opérateur =. Pour cela, vous utiliserez le mot clé **« delete »**.
- Elle disposera des méthodes template
  - **void write(const T& val)** permettant de sérialiser et d'écrire un objet dans le fichier (voir ci-dessous)
  - **T read()** permettant de lire et de désérialiser un objet à partir du fichier (voir ci-dessous)
- Elle disposera d'un destructeur qui écrira la balise de fin </…> de la collection.

Le constructeur **XmlFileSerializer(const string &fn, char m, const string& cn = « entities »)** :

- Si mode == WRITE : ouvrira le fichier en écriture seule (si le fichier existe déjà, il sera écrasé), écrira l'entête du fichier XML, la balise de début de collection <…> grâce au paramètre cn reçu.

- Si mode == READ : ouvrira le fichier en lecture seule, lira l'entête du fichier XML (mais n'en fera rien ici), lira la balise de début de collection <…> et récupéra donc le nom de la collection.

Pour rappel, c'est le destructeur de la classe qui écrira la balise de fin de collection.

La méthode **void write(const T& val)** utilisera l'opérateur << de la classe T pour sérialiser et écrire la variable val dans le fichier.

La méthode **T read()** utilisera l'opérateur >> de la classe T pour lire et désérialiser un objet à partir du fichier. La détection de fin de fichier lors de lecture est expliquée ci-dessous.

On vous demande également de créer la classe d'exception **XmlFileSerializerException** (cette classe n'est pas un template !) qui hérite de la classe **Exception** développée auparavant. Comme pour les autres classes d'exception déjà développés, cette classe disposera

- D'une variable **code** (de type **int**) correspondant au code de l'erreur, avec ses accesseurs classiques.
- Des variables membres statiques constantes **NOT_ALLOWED**, **FILE_NOT_FOUND** et **END_OF_FILE**, valeurs que peut prendre la variable **code** cette classe.

Différents cas peuvent se produire :

- L'exception de code **NOT_ALLOWED** sera lancée si on appelle la méthode **write()** alors que mode est égal à **READ**
- L'exception de code **NOT_ALLOWED** sera lancée si on appelle la méthode **read()** alors que mode est égal à **WRITE**
- L'exception de code **FILE_NOT_FOUND** sera lancée dans le constructeur de **XmlFileSerializer** si le mode est **READ** mais que le fichier dont le nom passé en paramètre n'existe pas sur disque.

Venons-en enfin à la détection de la fin du fichier représenté par la **balise </…>.** La méthode **read()**

- Doit lire une ligne via « **file >> ligne ;** »
- Si cette ligne est égale à **« </collectionName> »** (à vous de remplacer collectionName par son contenu), c'est qu'on a atteint la fin de fichier. On lancera alors l'exception de code **END_OF_FILE**
- Sinon, on a déjà lu la première d'un de nos objets sérialisés… Il faut donc faire marche arrière. Pour cela, vous utiliserez les méthodes **tellg()** et **seekg()** du flux **fstream file**
- Vous pouvez ensuite faire appel à l'opérateur >> de l'objet à désérialiser pour le lire correctement et complètement dans le fichier.

Dans le jeu de tests fourni, vous pourrez tester votre classe template tout d'abord sur des entiers et ensuite sur les classes dont vous avez créé les opérateurs << et >> en conséquence.

Tous les fichiers XML d'extension .xml peuvent être lu par n'importe quel éditeur de texte ou browser internet. Ceci vous permettra de vérifier facilement le contenu de vos fichiers de données créés.

Etape 8 (Test8a.cpp et Test8b.cpp) :

Un conteneur pour toutes nos données : La classe Timetable
Les conteneurs génériques de la STL

a) Mise en place de la classe Timetable (Test8a.cpp)

Afin d'éviter que les données de l'application soient dispersées et déclarées à différents endroits du code, on vous demande de regrouper toutes ces données au sein de la même classe appelée **Timetable** :

```
class Timetable
{
  private:
    set<Classroom> classrooms;
    set<Professor> professors;
    set<Group>     groups;

  public:
      Timetable();
      ~Timetable();

      int addClassroom(const string& name,int seatingCapacity);
      void displayClassrooms() const;
      Classroom findClassroomByIndex(int index) const;
      Classroom findClassroomById(int id) const;
      void deleteClassroomByIndex(int index);
      void deleteClassroomById(int id);

      int addProfessor(const string& lastName,const string& firstName);
      void displayProfessors() const;
      Professor findProfessorByIndex(int index) const;
      Professor findProfessorById(int id) const;
      void deleteProfessorByIndex(int index);
      void deleteProfessorById(int id);

      int addGroup(const string& name);
      void displayGroups() const;
      Group findGroupByIndex(int index) const;
      Group findGroupById(int id) const;
      void deleteGroupByIndex(int index);
      void deleteGroupById(int id);
};
```

Cette classe comporte notamment :

- Les **listes triées** (**ordre alphabétique**) de professeurs, groupes, et locaux. Ces listes seront des instances de la **classe template set<> de la STL du C++**.

- Un **constructeur par défaut**. Inutile de faire un constructeur de copie car il n'existera qu'un seul objet instance de la classe **Timetable** dans l'application.

- Les méthodes **void addXXX(…)** permettent d'ajouter un XXX (XXX = Professor, Group ou Classroom) au bon conteneur en recevant en paramètre les paramètres adéquats. Remarquez que **l'identifiant entier n'est pas passé en paramètre**. On vous demande donc d'ajouter à la classe **Schedulable** une **variable membre statique entière appelée currentId** qui sera utilisée et incrémentée de 1 à chaque ajout. Ceci assurera donc l'unicité de l'identifiant des schedulables.

- Les méthodes **void displayXXX()** (XXX = Professors, Groups ou Classrooms) qui permettent d'afficher le contenu des différents conteneurs.

- Les méthodes **XXX findXXXById(int id)** qui permettent de rechercher et de retourner un objet XXX (XXX = Professor, Group ou Classroom) dont l'identifiant **id** est passé en paramètre.

- Les méthodes **XXX findXXXByIndex(int index)** qui permettent de rechercher et de retourner un objet XXX (XXX = Professor, Group ou Classroom) dont l'indice dans sa liste est **index**.

- Les méthodes **void deleteXXXByIndex(int index)** qui permettent de supprimer le XXX (XXX = Professor, Group ou Classroom) dont l'indice dans sa liste triée est **index**.

- Les méthodes **void deleteXXXById(int id)** qui permettent de supprimer le XXX (XXX = Professor, Group ou Classroom) dont l'identifiant est **id**.

Les méthodes de recherche (find…) et de suppression (delete…) devront utiliser les **itérateurs fournis par la classe set<> de la STL**.

b) La classe Timetable en tant que singleton (Test8b.cpp)

La classe **Timetable** contenant toutes les données de notre future application, on se rend bien compte qu'elle ne sera **instanciée qu'une seule fois**. Une telle classe est appelée un « **singleton** ».

« Le **singleton** est un **patron de conception** (« **Design pattern** ») dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. On fournira un accès global à cet objet. Il est utilisé lorsqu'on a besoin d'exactement un objet pour coordonner les opérations d'un système. » (wikipedia)

En C++ (et donc dans notre cas pour la classe **Timetable**), une manière de faire est

- Rendre le **constructeur par défaut privé** : cela empêchera un programmateur d'instancier un ou plusieurs objets de classe **Timetable**.

- Placer dans la classe **Timetable** une **variable statique privée de type Timetable**, que l'on appellera **instance** (il s'agira de l'unique instance de la classe **Timetable**). Cet objet sera instancié en utilisant le constructeur par défaut.
- Cet objet étant privé, on lui donnera un accès « global » en écrivant une **méthode statique publique Timetable& getInstance()** qui retourne une référence vers l'objet **instance** qui pourra donc être manipulé de n'importe où dans l'application.
- Afin d'éviter qu'un programmateur puisse obtenir une copie de cette instance, on déclarera en **privé** (mais on ne les définira pas dans Timetable.cpp) un **constructeur de copie** et un **opérateur =** dans Timetable.h → on utilisera également le **mot clé « delete »** pour prévenir le compilateur que ces deux méthodes n'auront pas de corps.

Remarquez que lorsque vous aurez terminé les modifications de la classe **Timetable** afin de la transformer en singleton, le programme Test8a.cpp ne compilera plus. (Pourquoi 😊 ?)

Etape 9 (utilisation de InterfaceQt.tar fourni)

Mise en place de l'interface graphique : Introduction à Qt et à CMake

a) Introduction

Nous allons à présent mettre en place l'interface graphique. Celle-ci sera construite en utilisant la librairie graphique Qt. Sans entrer dans les détails de cette librairie, il faut savoir qu'une **fenêtre** est représentée par une **classe** dont

- les variables membres sont les composants graphiques apparaissant dans la fenêtre : les boutons, les champs de texte, les checkboxs, les tables, …
- les méthodes publiques permettent d'accéder à ses composants graphiques, soit en récupérant les données qui sont encodées par l'utilisateur, soit en y insérant des données.

L'interface graphique de notre application sera la classe **ApplicHoraireWindow** qui a été construite, pour vous, à l'aide de l'IDE QtCreator. Cette classe est fonctionnelle mais il ne s'agit que d'une **coquille vide** qui va permettre de manipuler **le seul objet de classe Timetable** de notre application. La classe **ApplicHoraireWindow** est fournie par les fichiers

- **applichorairewindow.h** qui contient la définition de la classe et la déclaration de ses méthodes
- **applichorairewindows.cpp** qui contient la définition de ses méthodes.

Seuls ces deux fichiers pourront être modifiés par vous. Les autres fichiers fournis (dans InterfaceQt.tar) ne devront en aucun cas être modifiés. Le main de votre application est le fichier main.cpp également fourni et ne devra pas être modifié.

La classe **ApplicHoraireWindow** contient déjà un ensemble de méthodes fournies (et que vous ne devez donc pas modifier) afin de vous faciliter l'accès aux différents composants graphiques.

Les méthodes que vous devez modifier contiennent le commentaire // TO DO. Elles correspondent aux différents boutons et items de menu de l'application et correspondent toutes à une action demandée par l'utilisateur, comme par exemple « Ajouter un professeur », « Supprimer un local », …

Pour l'affichage de messages dans des **boîtes de dialogue** ou des saisies de int/chaînes de caractères via des boîtes de dialogues, vous disposez des méthodes (**que vous ne devez pas modifier mais juste utiliser**) :

- void dialogMessage(const string& title, const string& message);
- void dialogError(const string& title, const string& message);
- string dialogInputText(const string& title, const string& question);
- int dialogInputInt(const string& title, const string& question);
- string dialogInputFileForLoad(const string& question);
- string dialogInputFileForSave(const string& question);

Etant une classe C++ au sens propre du terme, vous pouvez ajouter, à la classe **ApplicHoraireWindow**, des variables et des méthodes membres en fonction de vos besoins.

b) Structuration correcte du projet et compilation avec cmake

A partir de maintenant, on vous demande de structurer correctement votre projet C++ en sous-répertoires :

```
/Etape10/sources
/Etape10/includes
/Etape10/build
/Etape10/CMakeLists.txt
```

Le nom du répertoire (« Etape10 ») peut être adapté selon votre choix. Pour le reste :

- le répertoire **sources** contiendra tous les fichiers .cpp de vos classes mais également ceux fournis pour l'interface Qt
- le répertoire **includes** contiendra tous les fichiers .h de vos classes (y compris .hpp et .ipp pour les templates) mais également ceux fournis pour l'interface Qt
- le répertoire **build** contiendra l'exécutable créé (qui devra s'appeler « **ApplicHoraire** ») mais également tous les fichiers générés par l'exécution de **cmake** à partir de ce répertoire
- le fichier **CMakeLists.txt** contiendra les directives de compilation. A vous à créer ce fichier.

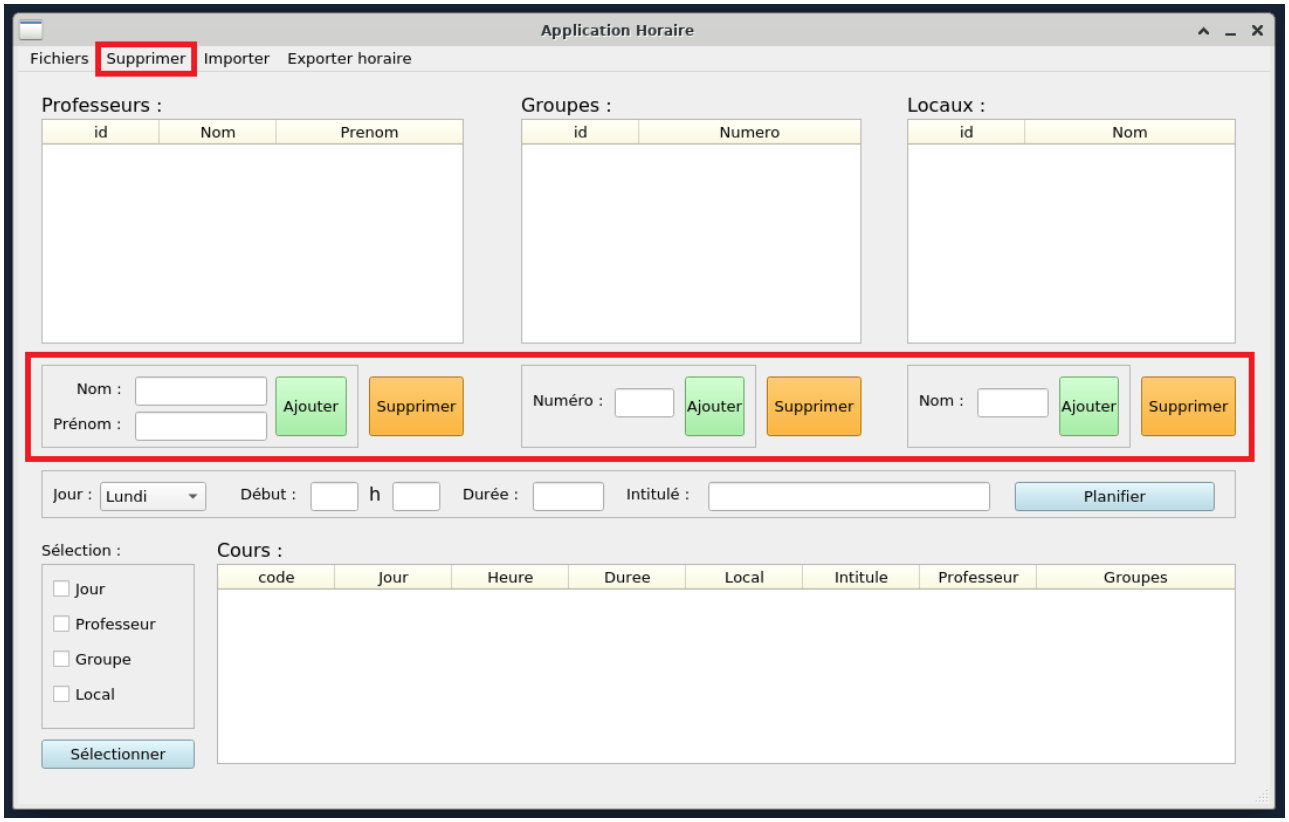
c) Mise en place des premières fonctionnalités

Les premières fonctionnalités demandées ici correspondent à l'ajout et à la suppression de professeurs/groupes/locaux. Ces fonctionnalités ont déjà été programmées à l'étape précédente dans la classe **Timetable**. Il suffit donc de manipuler l'unique instance statique de cet objet à l'aide de l'interface graphique. Toute la logique de la gestion de l'horaire de l'application, dite « logique métier », se trouve dans la classe **Timetable**.

Etant un objet statique, l'unique instance de la classe **Timetable** sera accessible partout dans l'application à partir de la classe **Timetable** elle-même :

- **Timetable& Timetable::getInstance()** : qui retourne la référence vers le singleton de notre application.

Les fonctionnalités demandées ici correspondent aux encadrés rouges ci-dessous :



Il s'agit donc de modifier les méthodes suivantes de la classe **ApplicHoraireWindow** :

- void on_pushButtonAjouterProfesseur_clicked();
- void on_pushButtonAjouterGroupe_clicked();
- void on_pushButtonAjouterLocal_clicked();
- void on_pushButtonSupprimerProfesseur_clicked();
- void on_pushButtonSupprimerGroupe_clicked();
- void on_pushButtonSupprimerLocal_clicked();

pour les boutons, et

- void on_actionSupprimerProfesseur_triggered();
- void on_actionSupprimerGroupe_triggered();
- void on_actionSupprimerLocal_triggered();

pour les items de menu.

Pour **ajouter un nouveau professeur** en cliquant sur le bouton « Ajouter » correspondant, vous devez (dans la méthode on_pushButtonAjouterProfesseur_clicked)

1. récupérer le nom et le prénom encodés par l'utilisateur. Pour cela, vous disposez des méthodes **string getProfessorLastName()** et **string getProfessorFirstName()**. Si une des chaînes de caractères est vide, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogError(const string& title, const string& message).**
2. appeler la méthode **void addProfessor(const string& lastName, const string& firstName)** de notre singleton.
3. mettre à jour la table des professeurs. Pour cela, vous devez
    a. vider la table des professeurs à l'aide de la méthode **void clearTableProfessors()**.
    b. récupérer la liste triée des professeurs. Un **itérateur** vous permettra de parcourir cette liste et d'en récupérer chaque élément. La fonction **tuple()** de chaque objet Professor retournera le tuple qui pourra être inséré dans la table des professeurs à l'aide de la méthode **void addTupleTableProfessors(const string& tuple).**

Le clic sur le **bouton « Supprimer »** correspondant aux professeurs **supprime le professeur sélectionné dans la table**. Pour ce faire, vous devez (dans la méthode void on_pushButtonSupprimerProfesseur_clicked) :

1. récupérer l'indice du professeur sélectionné dans la table. Pour cela vous disposez de la méthode **int getIndexProfessorSelection()** qui retourne l'indice dans la table du professeur sélectionné (il s'agit également de son indice dans la liste triée), et -1 si aucun professeur n'est sélectionné. Si aucun professeur n'est sélectionné, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogError(const string& title, const string& message)**.
2. appeler la méthode **void deleteProfessorByIndex(int ind)** de notre singletion.
3. mettre à jour la table des professeurs (voir ci-dessus).

Le clic sur **l'item de menu « Supprimer » → « Professeur » supprime un professeur dont on demandera l'id à l'utilisateur**. Pour ce faire, vous devez (dans la méthode void on_actionSupprimerProfesseur_triggered) :

1. demander à l'utilisateur d'encoder un identifiant. Pour cela, vous devez afficher une boîte de dialogue de saisie d'entier en utilisant la méthode **int dialogInputInt(const string& title, const string& question)**.
2. appeler la méthode **void deleteProfessorById(int id)** de notre singleton.
3. mettre à jour la table professeurs (voir ci-dessus).

L'ajout/suppression des groupes/locaux doit se faire de manière tout à fait analogue.

Etape 10 :

Enregistrement sur disque : La classe Timetable se sérialise elle-même

a) Des méthodes save() et load() pour la classe Timetable

Comme nous l'avons vu précédemment, un horaire devra être sérialisé dans différents fichiers XML mais pas uniquement. Dans l'état actuel des choses, il va falloir sérialiser les professeurs, les groupes et les locaux dans **3 fichiers XML distincts**. Pour les locaux (classe **Classroom**), le travail a déjà été bien amorcé à l'étape 7 avec la classe **XmlFileSerializer**. Vous devez donc dans un premier temps compléter les classes **Professor** et **Group** pour qu'elles disposent des opérateurs << et >> nécessaires à leur sérialisation au format XML.

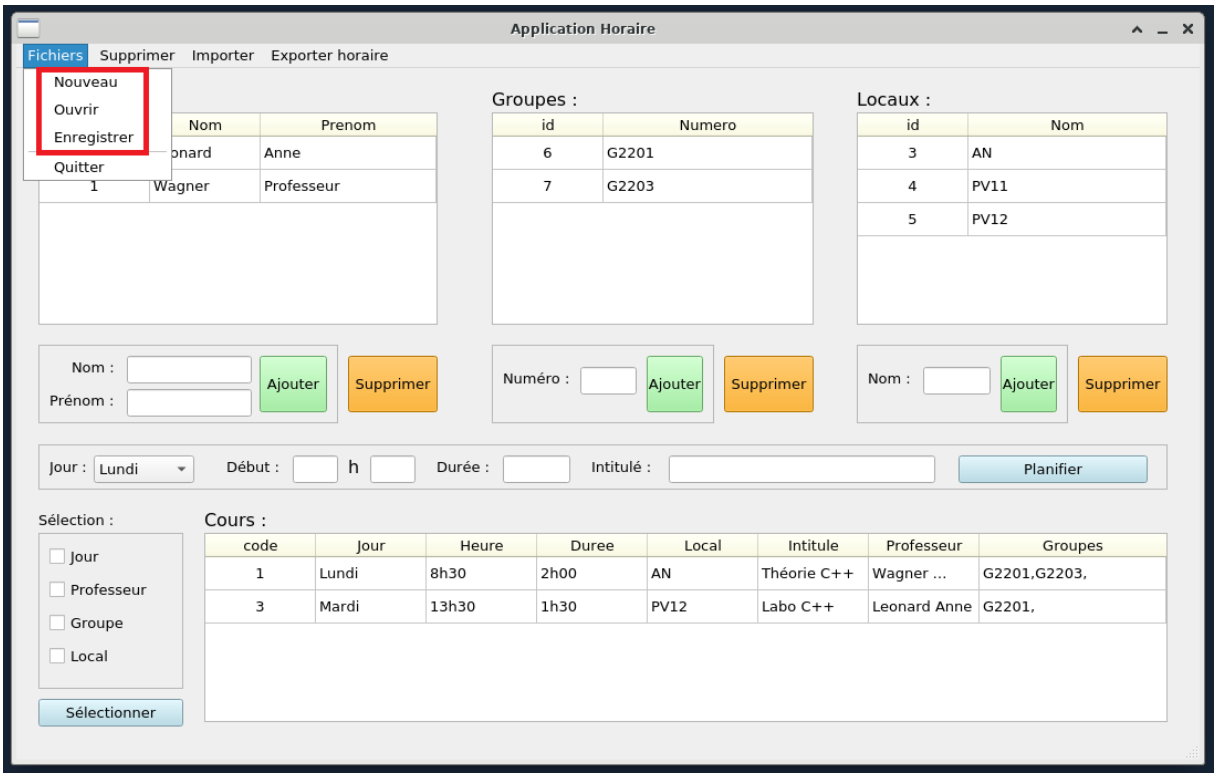
Un **4ème fichier** (**binaire à présent !**) devra être créé et contenir la variable **currentId** de la classe **Schedulable**. En effet, lorsque nous relirons un horaire sur disque, l'ajout futur d'un nouveau Schedulable devra générer un identifiant non encore utilisé, et c'est justement la valeur de **currentId** qui assure cette unicité.

On vous demande donc d'ajouter à la classe **Timetable** :

- La méthode **save(const string& timetableName)** qui sérialisera toutes les données de l'horaire dans différents fichiers. Pour cela :
  - Elle créera les noms des 4 fichiers construits par concaténation de la variable **timetableName** avec **« _professors.xml », « _groups.xml », « _classrooms.xml »** et **« _config.dat »**
  - Elle enregistrera, au format **binaire** (utilisation de **write(…)**) dans le fichier de configuration (**« …_config.dat »**) l'entier **Schedulable ::currentId**
  - Elle utilisera la classe template **XmlFileSerializer** pour sérialiser au format XML les professeurs, les groupes et les locaux dans les 3 fichiers XML dont les noms ont été créés précédemment.
- La méthode **load(const string& timetableName)** qui désérialisera toutes les données d'un horaire dont le nom est passé en paramètre. Pour cela, elle réalisera les opérations de lecture nécessaires dans les 4 fichiers associés à un horaire. Notez qu'avant de lire, la méthode load() devra vider les différents conteneurs présents dans la classe **Timetable**.

b) Ajout de nouvelles fonctionnalités à l'application

Nous allons à présent ajouter les fonctionnalités encadrées en rouge ci-dessous :



Un clic sur **l'item de menu « Fichier » → « Enregistrer »** :

- demande à l'utilisateur le nom de l'horaire qui va être enregistré. Pour cela, vous disposez de la méthode **string dialogInputText(const string& title, const string& question)**.
- enregistre l'horaire sur disque en appelant la méthode **save** de l'objet **Timetable**.

Un clic sur **l'item de menu « Fichier » → « Ouvrir »** :

- demande à l'utilisateur le nom de l'horaire qui doit être lu sur disque.
- lit l'horaire sur disque en appelant la méthode **load** de l'objet **Timetable**.
- met à jour les tables professeurs, groupes et locaux de la fenêtre.

Toutes les anciennes données sont effacées lors de la lecture d'un fichier horaire.

Un clic sur **l'item de menu « Fichier » → « Nouveau »** :

- vide tous les conteneurs de l'objet horaire.
- remet la variable statique **Schedulable::currentId** à 1.
- met à jour les tables professeurs, groupes et locaux de la fenêtre.

Etape 11 :

La planification des cours : la classe Course

a) Une classe spécialisée pour les Cours

Il s'agit tout d'abord de développer la classe **Course** modélisant un cours donné par un professeur, dans un certain local, à un certain nombre de groupes, à un jour et une heure donnés. Il s'agit d'un événement spécialisé. On vous demande donc de développer la classe **Course**, qui hérite de la classe **Event**, et qui présente les variables membres supplémentaires suivantes :

- **professorId** : l'identifiant (de type **int**) du professeur concerné par ce cours.
- **classroomId** : l'identifiant (de type **int**) du local concerné par ce cours.
- Une liste **groupsIds** : variable du type **set<int>** (de la STL du C++) contenant les identifiants des groupes concernés par ce cours.

Héritant de la classe **Event**, la classe **Course** dispose des variables membres

- **code** (entier qui identifie de manière unique un cours donné)
- **title** qui représente ici le nom du cours donné (Exemple : « Labo C++ »)
- **timing** qui contient toute l'information temporelle du cours (jour, heure, durée)

La classe **Course** devra être munie des **opérateurs** de comparaison de telle sorte que l'ordre chronologique soit respecté. Outre ses accesseurs classiques, la classe **Course** devra disposer des méthodes :

- **void addGroupId(int id)** qui permet d'ajouter un identifiant de groupe à la liste **groupsIds**.
- **bool isGroupIdPresent(int id)** qui retourne true si l'identifiant **id** se trouve dans la liste **groupsIds** et false sinon. Cette fonction permettra de tester si un groupe est concerné par un cours.

b) Mise à jour de la classe Timetable

La première chose est d'ajouter à la classe **Timetable** une variable membre **courses** de type **list<Course>** (de la STL du C++) et qui contiendra **l'ensemble des cours planifiés** par ordre chronologique.

On demande ensuite d'ajouter à la classe **Timetable** les méthodes :

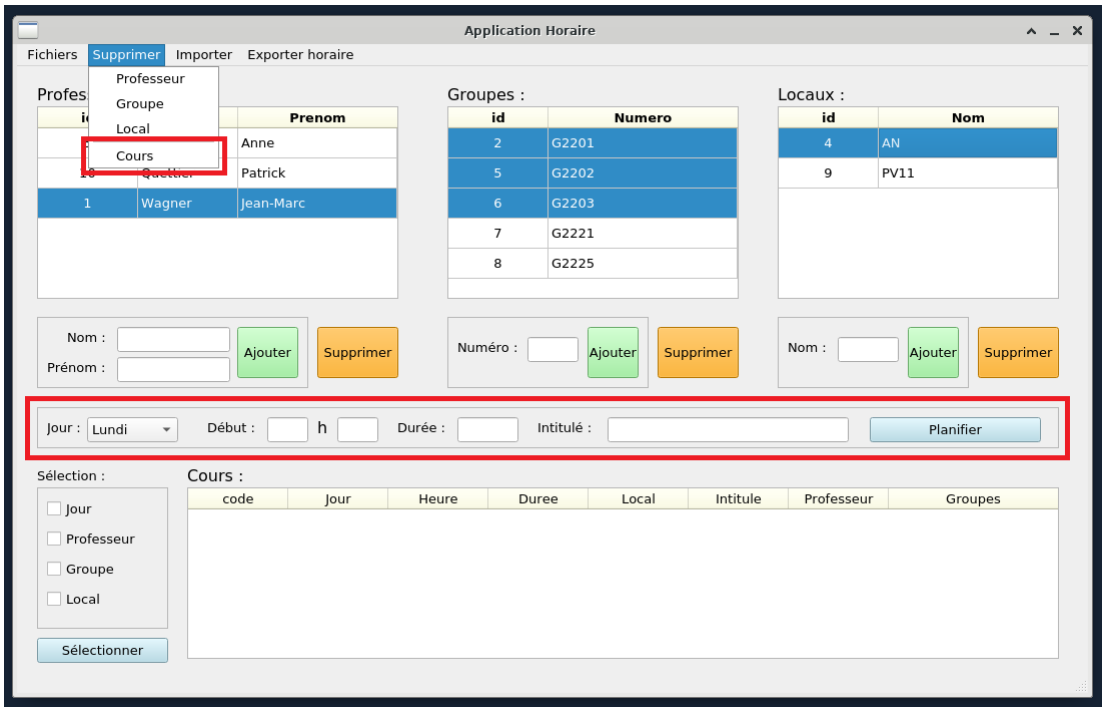
- **bool isProfessorAvailable(int id, const Timing& timing)** : qui retourne true si le professeur d'identifiant **id** est disponible (non occupé) pendant le laps de temps défini par **timing**, et false sinon. Il suffit pour cela de balayer le conteneur **courses** et de voir si timing « intersecte » temporellement le timing d'un des cours déjà planifiés pour ce professeur. Si un des cours planifiés « intersecte » l'objet timing, le professeur n'est pas disponible.

  Une idée intéressante serait donc d'ajouter la méthode **bool intersect(const Timing& t)** à la classe **Timing** et permettant de tester si deux timing entrent en conflit temporellement.

- **bool isGroupAvailable(int id, const Timing& timing)** : qui retourne true si le groupe d'identifiant **id** est disponible (non occupé) pendant le laps de temps défini par **timing**, et false sinon. Il suffit pour cela de balayer le conteneur **courses** et de voir si le groupe id est concerné par un cours déjà planifié (vous disposez pour cela de la méthode **isGroupIdPresent** de **Course**) et si timing « intersecte » temporellement le timing de ce cours. Si un des cours planifiés « intersecte » l'objet timing, le groupe n'est pas disponible.
- **bool isClassroomAvailable(int id, const Timing& timing)** : qui retourne true si le local d'identifiant **id** est disponible (non occupé) pendant le laps de temps défini par **timing**, et false sinon. Il suffit pour cela de balayer le conteneur **courses** et de voir si timing « intersecte » temporellement le timing d'un des cours déjà planifiés pour ce local. Si un des cours planifiés « intersecte » l'objet timing, le local n'est pas disponible.
- **void schedule(Course& c, const Timing& t)** qui
  - reçoit un objet c de type **Course** contenant déjà un intitulé, un id de professeur, un id de local et contenant au moins un id de groupe dans sa liste **groupsIds**. Par contre, sa variable membre timing est égale à **nullptr**.
  - reçoit un objet t de type **Timing** contenant le moment (jour, heure et durée) où l'on veut planifier le cours c.
  - lance une exception de type **TimingException** (avec un message adéquat) si le professeur n'est pas disponible, si le local n'est pas disponible ou si un des groupes n'est pas disponible. Pour tester cela, vous disposez des méthodes ci-dessus.
  - associe l'objet timing t au cours c et insère le cours planifié dans le conteneur **courses** en lui attribuant le code courant (qui sera incrémenté juste après).

c) Ajout de nouvelles fonctionnalités à l'application

Nous allons à présent ajouter les fonctionnalités encadrées en rouge ci-dessous :



Lors d'un **clic sur le bouton « Planifier »,** il faut

- vérifier qu'il y a un professeur sélectionné, un local sélectionné et au moins un groupe sélectionné et en récupérer les identifiants. Si ce n'est pas le cas, on fera apparaître une boîte de dialogue d'erreur.
- récupérer les informations temporelles du cours à planifier dans la fenêtre (jour, heure et durée) et instancier un objet de type **Timing**.
- récupérer l'intitulé du cours dans la fenêtre, instancier un objet de classe **Cours** et lui attribuer les identifiants du professeur, du local et des groupes concernés.
- appeler la méthode **schedule()** de notre singleton afin de planifier le cours.

L'instanciation des objets de type **Time** et **Timing**, ainsi que la méthode **schedule()** de la classe **Timetable** sont susceptibles de lancer des **exceptions de type TimeException et TimingException** qu'il faudra gérer par un **try…catch** dans la méthode **on_pushButtonPlanifier_clicked** de la classe **ApplicHoraireWindow**.

Une fois le cours planifié, il faut encore mettre à jour la table des cours. Pour cela, vous disposez de la méthode **void addTupleTableCourses(const string& tuple)** de la classe **ApplicHoraireWindow** (exactement comme c'est déjà fait pour les tables des professeurs, des groupes et des locaux). Cependant, la classe **Course** ne dispose pas de méthode **tuple()** et il est impossible de lui en créer une car elle ne contient que les identifiants des planifiables concernés. On vous demande donc d'ajouter à la classe **Timetable** la méthode

- **string tuple(const Course& c)** qui reçoit en paramètre un objet c de type **Course** déjà planifié et présent dans le conteneur **courses**. Vu que la classe **Timetable** contient tous les objets planifiables, elle peut récupérer toutes les informations nécessaires et générer la chaîne de caractères qui sera retournée par la méthode. Par exemple :

« 1;Mardi;8h30;2h00;AN;Théorie C++;Wagner Jean-Marc;INFO2 D201, INFO2 D202 »

Remarquez que ce tuple comprend 8 champs (la table des cours comporte 8 colonnes).

Enfin, un clic sur **l'item de menu « Supprimer » → « Cours »** :

- demande à l'utilisateur le code d'un cours à supprimer. Pour cela, vous disposez de la méthode **int dialogInputInt(const string& title, const string& question)**.
- supprime le cours correspondant du conteneur **courses** et met à jour la table des cours.

d) Enregistrement des cours sur disque

Comme cela a été fait pour les professeurs, les locaux et les groupes, il est maintenant nécessaire de mettre en place la **sérialisation des objets Course** contenus dans le conteneur **courses** de la classe **Timetable**. Pour cela, vous devez

- implémenter les opérateurs << et >> de la classe **Course** afin de gérer la sérialisation de ce type d'objet au format XML
- mettre à jour les méthodes **save()** et **load()** de la classe **Timetable** afin de :

- enregistrer (et relire) la variable **Event::currentCode** dans le fichier de configuration de l'horaire juste après la variable **Schedulable::currentId**
- enregistrer (et relire) tous les cours dans le fichier **« …_courses.xml »** grâce à la classe template **XmlFileSerializer**

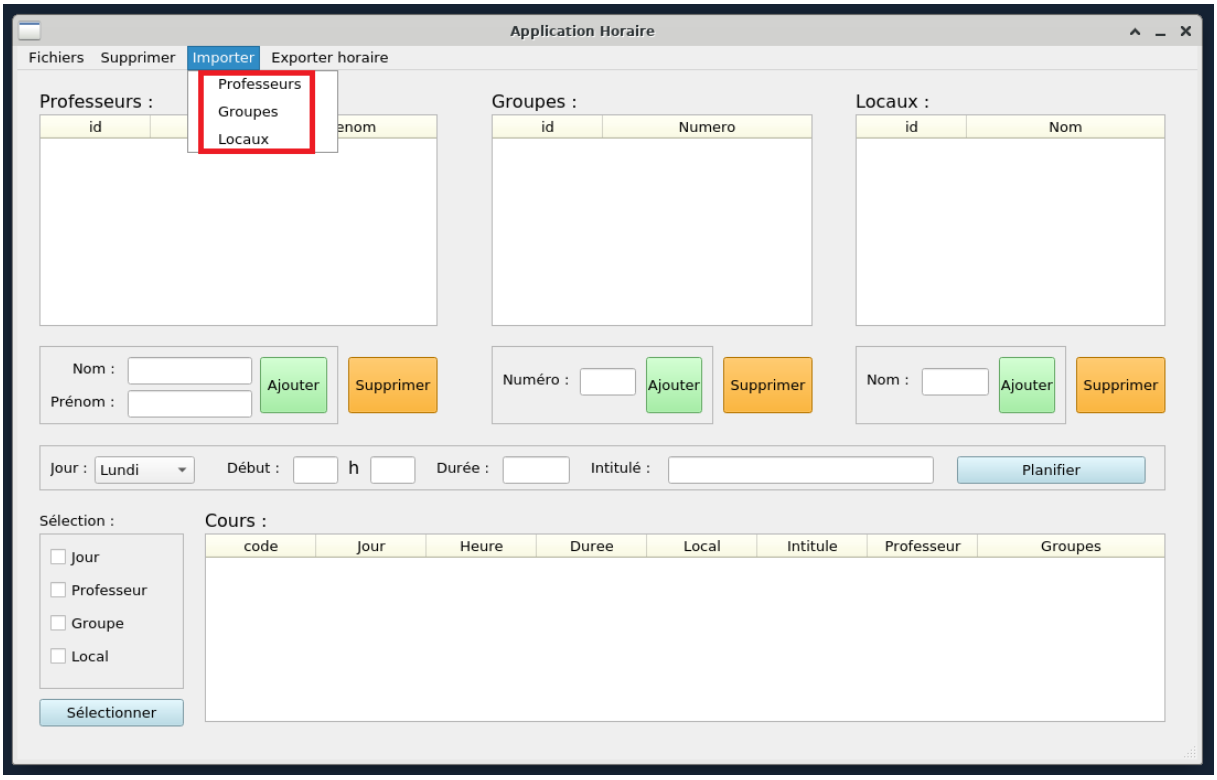
- mettre à jour la classe **ApplicHoraireWindow** afin de mettre à jour la table de cours lorsque l'on charge un horaire à partir du disque ou que l'on crée un nouvel horaire vide.

Etape 12 :

Importation de schedulables et exportation d'horaires : Fichiers textes

a) Importation de « schedulables » : lecture de fichiers textes au format csv

Nous allons à présent ajouter les fonctionnalités encadrées en rouge ci-dessous :



Ces 3 items de menu permettent d'importer rapidement des professeurs, des groupes et des locaux à partir de fichiers csv. Ces **fichiers textes** peuvent être créés/lus à l'aide de Excel mais également dans n'importe quel éditeur de texte. Par exemple, pour les professeurs nous pourrions avoir

```
CAPRASSE;Francois;
CHARLET;Christophe;
COSTA;Corinne;
…
```

Chaque ligne représente donc un professeur et le caractère **';'** est appelée le ***séparateur***. Celui-ci pourrait être **':'** ou encore **','**.

On vous demande donc d'ajouter à la classe **Timetable** les méthodes suivantes :

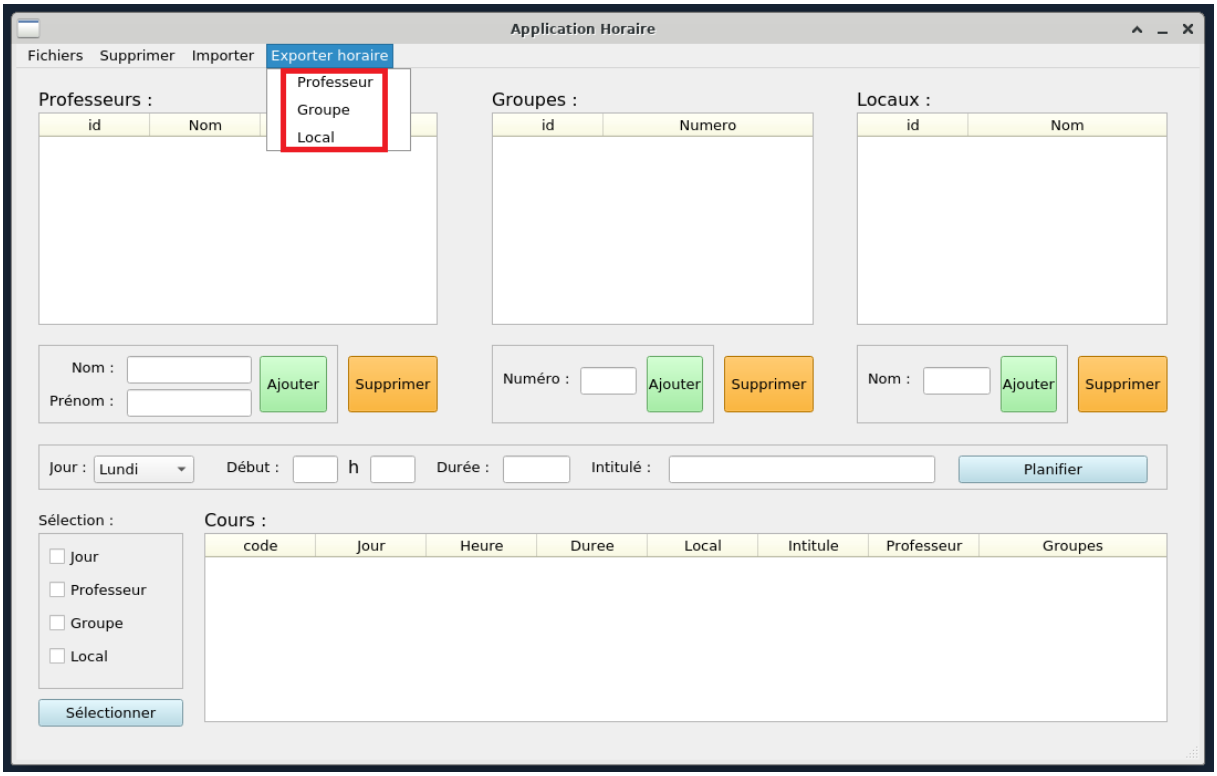
- **importProfessorsFromCsv(const string& filename)** : qui importe les professeurs présents dans le fichier csv dont le nom est passé en paramètre.
- **importGroupsFromCsv(const string& filename)** : qui importe les groupes présents dans le fichier csv dont le nom est passé en paramètre.
- **importClassroomsFromCsv(const string& filename)** : qui importe les locaux présents dans le fichier csv dont le nom est passé en paramètre.

Un ensemble de fichiers csv vous seront fournis au laboratoire (**professors.csv**, **groups.csv** et **classrooms.csv**).

Un clic sur un des items du menu « Importer » fera apparaître une boîte de dialogue demandant à l'utilisateur d'encoder le **nom du fichier** csv à lire. Pour cela, vous pouvez utiliser la méthode **string dialogInputFileForLoad(const string& question)** de la classe **ApplicHoraireWindow**.

b) Exportation de l'horaire d'un « schedulable » : écriture de fichiers textes

Nous allons à présent ajouter les fonctionnalités encadrées en rouge ci-dessous :



Ces 3 items de menu permettent d'exporter l'horaire, au **format texte**, d'un professeur/groupe/local dont **on demande l'identifiant à l'utilisateur** à l'aide d'une boîte de dialogue. Par exemple, pour le professeur Wagner, on pourra avoir, en sortie le fichier « **Wagner_Jean-Marc.hor** » contenant

```
Horaire de Wagner Jean-Marc :

Lundi       8h20   2h00   AE    Théorie C++   INFO2_D201,INFO2_I202
Lundi       13h30  1h30   LE0   Labo UNIX     INFO2_R201
Mardi       10h30  2h00   PV11  Labo C++      INFO2_I202
…
```

Et par exemple pour le groupe INFO2_I202, un fichier "**INFO2_I202.hor**" :

```
Horaire de INFO_I202 :

Lundi       8h20   2h00   AE    Théorie C++   Wagner Jean-Marc
Mardi       10h30  2h00   PV11  Labo C++      Wagner Jean-Marc
…
```

On vous demande donc d'ajouter à la classe **Timetable** les méthodes suivantes :

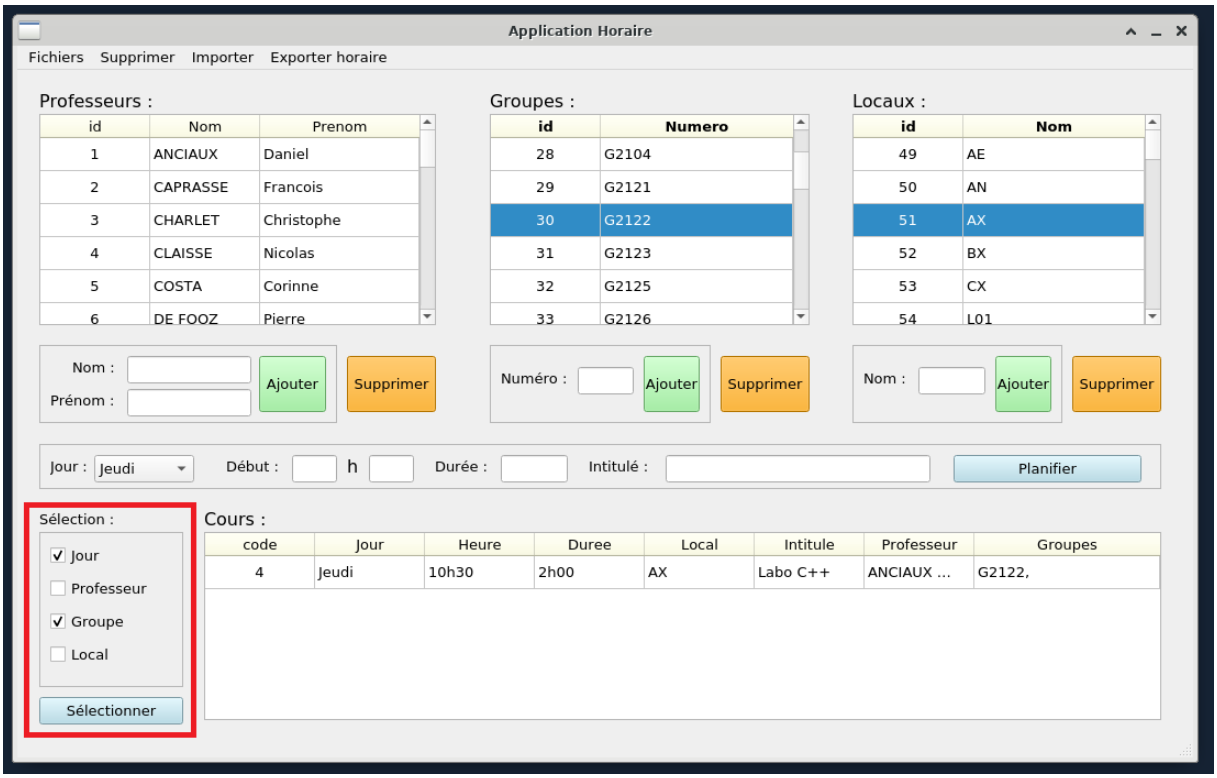
- **exportProfessorTimetable(int id)** : qui exporte l'horaire du professeur dont l'identifiant est passé en paramètre.
- **exportGroupTimetable(int id)** : qui exporte l'horaire du groupe dont l'identifiant est passé en paramètre.
- **exportClassroomTimetable(int id)** : qui exporte l'horaire du local dont l'identifiant est passé en paramètre.

Les **noms des fichiers** textes créés doivent être **générés automatiquement** à partir des données de chaque schedulable.

Etape 13 :

Sélection spécifique dans l'affichage des cours planifiés

Nous allons à présent ajouter les fonctionnalités encadrées en rouge ci-dessous :



Un clic sur le **bouton « Sélectionner »** met à jour l'affichage de la table des cours (sans suppression dans le conteneur **courses** de l'objet **Timetable**) en tenant comptes des checkboxs sélectionnés :

- **Si la checkbox « Jour » est sélectionnée** : on n'affichera que les cours dont le jour est celui apparaissant dans la combobox des jours (ci-dessus dans la capture d'écran : « Jeudi »), sinon on ne tient pas compte du jour pour la sélection et on les affiche tous.
- **Si la checkbox « Professeur » est sélectionnée** : on n'affichera que les cours dont le professeur est sélectionné dans la table des professeurs, sinon on ne tient pas compte du professeur et on les affiche tous (ci-dessus dans la capture c'est le cas).
- **Si la checkbox « Groupe » est sélectionnée** : on n'affichera que les cours dont le groupe est sélectionné dans la table des groupes (on se contentera du premier groupe sélectionné - ci-dessus dans la capture d'écran « G2122 »), sinon on ne tient pas compte du groupe et on les affiche tous.
- **Si la checkbox « Local » est sélectionnée** : on n'affichera que les cours dont le local est sélectionné dans la table des locaux, sinon on ne tient pas compte du local et on les affiche tous (ci-dessus dans la capture d'écran c'est le cas).

Comment faire ? Soyez imaginatif 😊...

Bon travail !