Laboratoire de Programmation en C++

2ème informatique et systèmes :
option(s) industrielle et réseaux (1er quadrimestre)
et 2ème informatique de gestion (1er quadrimestre)

Année académique 2021-2022

Gestion d'horaires de cours

Anne Léonard
Patrick Quettier
Claude Vilvens
Jean-Marc Wagner

Introduction

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

Cet énoncé de laboratoire concerne les unités d'enseignement (UE) suivantes :

a) 2ème Bach. en informatique de Gestion : « Développement Système et orienté objet »

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

- **Base de la programmation orientée objet – C++** (45h, Pond. 30/90)
- **Principes fondamentaux des Systèmes d'exploitation** (15h, Pond. 10/90)
- **Système d'exploitation et programmation système UNIX** (75h, Pond. 50/90)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

b) 2ème Bach. en informatique et systèmes : « Développement Système et orienté objet »

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

- **Base de la programmation orientée objet – C++** (45h, Pond. 45/101)
- **Système d'exploitation et programmation système UNIX** (56h, Pond. 56/101)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

Quel que soit le bachelier, la cote de l'AA « Base de la programmation orientée objet – C++ » est construite de la même manière :

♦ théorie : un examen écrit en janvier 2022 (sur base d'une liste de points de théorie fournis en novembre et à préparer) et coté sur 20

♦ laboratoire **(cet énoncé)** : une évaluation globale en janvier, accompagné de « check-points » réguliers pendant tout le quadrimestre. Cette évaluation fournit une note de laboratoire sur 20

♦ note finale : **moyenne arithmétique 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 via leur centre de ressources.

2. Le contexte : Gestion d'horaires de cours

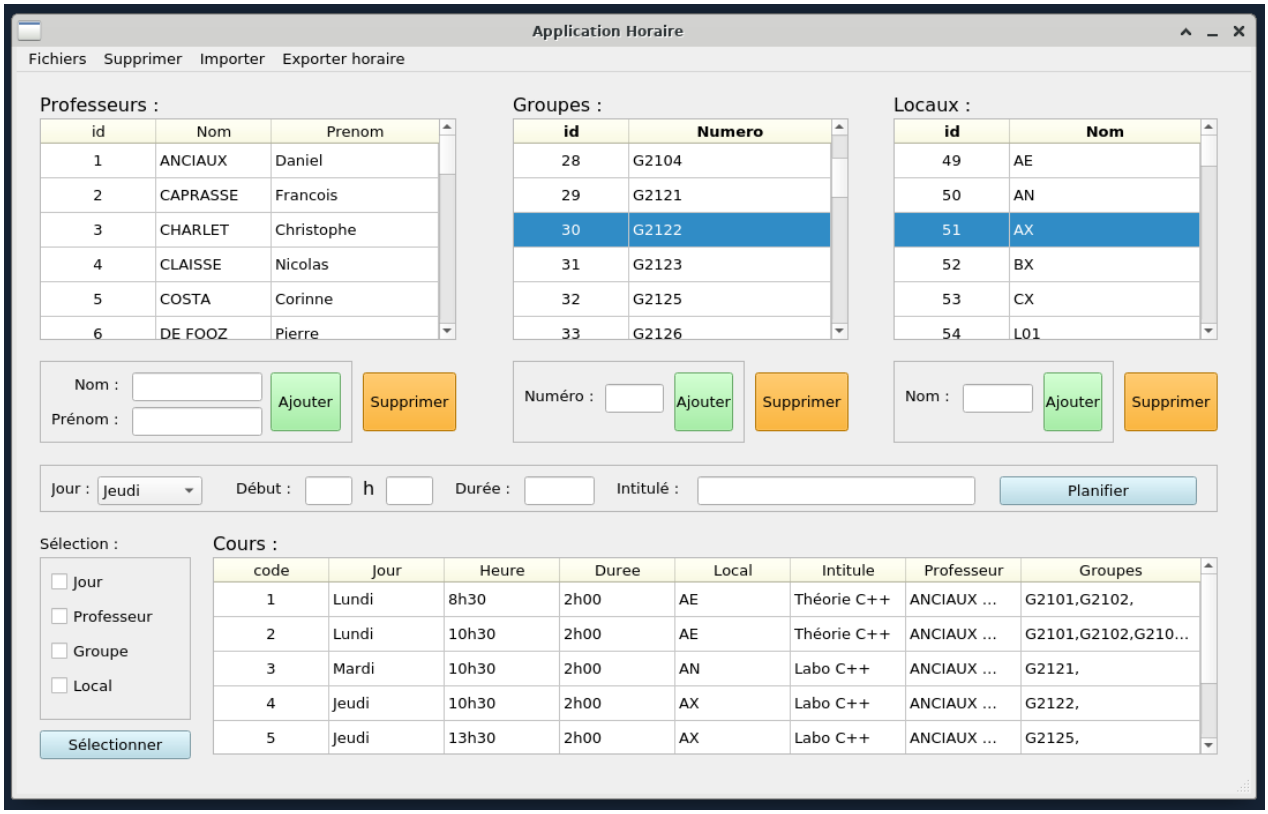
Les travaux de Programmation Orientée Objets (POO) se déroulent dans le contexte de la gestion d'un horaire de cours. Plus particulièrement, il s'agit de gérer la planification de cours, ainsi que d'éviter les conflits entre les ressources tels que les professeurs, les groupes d'étudiants et les locaux, 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 » (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. La machine de développement sera Jupiter (10.59.28.2 sur le réseau de l'école). Néanmoins, une machine virtuelle (VMWare) possédant exactement la même configuration que celle de Jupiter sera mise à la disposition des étudiants lors des premières séances de laboratoire. Mais attention, **seul le code compilable sur Jupiter sera pris en compte !!!**

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.

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. Planning et contenu des évaluations

a) Evaluation 1 (continue) :

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.

b) Evaluation 2 (examen de janvier 2022) :

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 de l'école, selon les modalités fixées par le professeur de laboratoire.

Plan des étapes à réaliser

CONTRAINTES : Tout au long du laboratoire de C++ (évaluation 1 et 2, et seconde session), il vous est interdit, pour des raisons pédagogiques, d'utiliser la classe **string** et les **containers génériques template de la STL**.

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 (**int**) permettant d'identifier de manière unique un événement dans l'horaire.
- Un **intitulé** : une chaine de caractères 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. Les variables de type chaîne de caractères seront donc des **char***. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé du tout dans TOUT ce dossier de C++.** Vous aurez l'occasion d'utiliser la classe string dans votre apprentissage du C# et du Java.

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 **Temps** contenant

- Une variable **heure** 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 Affiche() 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 :

- Un **jour** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé, et contenant « Lundi » ou « Mardi », …
- Une **heure** (variable de type **Temps**) représentant l'heure du début de l'événement.
- Une **durée** (variable de type **Temps**) 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 Affiche().

Bien sûr, les classes Temps 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 Heure 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 NULL si l'événement n'a pas encore été planifié.

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

Notez que la méthode Affiche() 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 jour 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 chaîne de caractères (**char ***) : **LUNDI** contenant « Lundi », **MARDI** 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** **codeCourant** de type **int**. Cette variable sera initialisée à 1 et incrémentée au besoin.

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 Temps

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

```
Temps 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.Affiche() ;
h4 = 10 + h3 ;  // h4 vaut à présent 14h30
h4 = h4 + d ;   // h4 vaut à présent 15h00

h3 = h3 - 10 ;             // h3 vaut à présent 14h10
Temps duree = 50 - d ;     // duree vaut à présent 0h20
Temps duree2 = h3 - h1 ;   // duree2 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 Temps

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 Temps, permettant d'exécuter un code du genre

```
Temps 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 Temps

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

```
Temps 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 classe Temps

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

```
Temps 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 Temps, 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 numéro (ex : 2201, 2222, 2225,…) et un « horaire », c'est-à-dire une liste de cours à suivre.
- Les locaux qui ont un nom (ex : AN, PV11, LE0, …), et un « horaire », c'est-à-dire une liste des cours ayant lieu dans chaque local.

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 « planifiable ». Ces planifiables 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** **Planifiable** et contiendra, pour l'instant :

- un identifiant **id** (de type **int**) qui identifie de manière unique un planifiable dans l'application. Cette variable doit disposer de ces accesseurs classiques.
- la méthode **const char* Txt()** : qui retourne une chaine de caractères décrivant un planifiable (cette chaine sera construite de manière dynamique en fonction des données propres à chaque planifiable ; 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 **const char* Tuple()** : qui retourne une chaine de caractères correspondant à un tuple (comme dans une BD) décrivant le planifiable. Ce tuple sera très utile lorsqu'il faudra afficher un planifiable dans une table de la future interface graphique de l'application. Cette fonction **doit également être une méthode virtuelle pure**.

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 planifiable qui a, en plus, un nom et un prénom. On vous demande donc de programmer la classe **Professeur**, qui hérite de la classe Planifiable, et qui présente, en plus, les variables membres suivantes :

- Un **nom** : chaine de caractères (**char ***),
- Un **prenom** : chaine de caractères (**char ***).

Un groupe d'étudiants est un planifiable qui a, en plus, un numéro. Donc, on vous demande de programmer la classe **Groupe**, qui hérite de la classe Planifiable, et qui présente, en plus, la variable membre suivante :

- Un **numero** : un entier (**int**).

Enfin, un local est un planifiable qui a, en plus, un nom. Vous devez donc programmer la classe **Local**, qui hérite de la classe Planifiable, et qui présente, en plus, la variable membre suivante :

- Un **nom** : chaine de caractères (**char ***).

On redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire, par exemple les opérateurs d'affectation = ainsi que <<.

c) Mise en évidence de la virtualité

Les trois classes dérivées (Professeur, Groupe, Local) devront redéfinir les méthodes suivantes :

- **const char* Txt()** : qui retourne une chaîne de caractères propre à chaque intervenant
  Professeur → « nom prenom» (Ex : « Wagner Jean-Marc »)
  Groupe → « Gnumero » (Ex : « G2201 »)
  Local → « nom » (Ex : « AE »)

- **const char* Tuple()** : qui retourne une chaîne de caractères propre à chaque intervenant
  Professeur → « id;nom;prenom» (Ex : « 3;Wagner;Jean-Marc »)
  Groupe → « id;Gnumero » (Ex : « 2;G2201 »)
  Local → « id;nom » (Ex : « 6;AE »)

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

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 abstraite d'exception de base** ne contenant qu'une seule variable membre **message** de type chaîne de caractères (**char***) destinée à contenir un message d'erreur à l'intention de l'utilisateur et la **méthode virtuelle pure int getCode() const**. 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.

- **TempsException** : **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 **HEURE_INVALIDE, MINUTE_INVALIDE, DEPASSEMENT ou FORMAT_INVALIDE** qui sont 4 **variables membres statiques constantes** de type **int** de la classe TempsException. Cette exception sera lancée avec le code d'erreur
  - HEURE_INVALIDE si l'on tente de créer ou modifier un objet de la classe Temps 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 Temps, h.setHeure(-2) lancera une exception.
  - MINUTE_INVALIDE si l'on tente de créer ou modifier un objet de la classe Temps 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 Temps, h.setMinute(70) lancera une exception.
  - DEPASSEMENT si le résultat d'une opération + (++) ou – (--) est antérieur à 0h00 ou postérieur à 23h59,
  - FORMAT_INVALIDE lors d'une extraction du flux en cas de mauvaise saisie (ex : « xh56 »).

- **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 **JOUR_INVALIDE, AUCUN_TIMING** qui sont 2 **variables membres statiques constantes** de type **int** de la classe TimingException. Cette exception sera lancée avec le code d'erreur
  - JOUR_INVALIDE 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.setJour(« Vendremanche ») lancera une exception.
  - AUCUN_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 NULL.

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) : Les containers et les templates

a) L'utilisation future des containers

On conçoit sans peine que notre future application va utiliser des containers mémoire divers qui permettront par exemple de contenir les professeurs, les groupes, ou les locaux. Nous allons ici mettre en place une base pour nos containers. Ceux-ci seront construits via une hiérarchie de classes templates.

b) Le container typique : la liste (Essai1() et Essai2())

Le cœur de notre hiérarchie va être une liste chaînée dynamique. Pour rappel, une liste chaînée dynamique présente un pointeur de tête et une succession de cellules liées entre elles par des pointeurs, la dernière cellule pointant vers NULL. Cette liste va être encapsulée dans une **classe Liste template** contenant comme seule variable membre le pointeur de tête de la liste chaînée. Elle aura donc la structure de base suivante :

```
template<class T> class Liste
{
	protected :
		Cellule<T> *pTete ;
	…
}
```

où les cellules de la liste chaînée auront la structure suivante :

```
template<class T> struct Cellule
{
	T valeur ;
	Cellule<T> *suivant ;
}
```

La classe **Liste** devra disposer des méthodes suivantes :

- Un **constructeur par défaut** permettant d'initialiser le pointeur de tête à NULL.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- La méthode **estVide()** retournant le booléen true si la liste est vide et false sinon.
- La méthode **getNombreElements()** retournant le nombre d'éléments présents dans la liste.
- La méthode **Affiche()** permettant de parcourir la liste et d'afficher chaque élément de celle-ci.
- La méthode **void insere(const T & val)** permettant d'insérer un nouvel élément **à la fin de la liste**.
- La méthode **T retire(int ind)** qui permet de supprimer et de retourner l'élément situé à l'indice **ind** dans la liste. Les valeurs possibles pour ind sont 0, …, getNombreElements()-1.

- Un **opérateur =** permettant de réaliser l'opération « liste1 = liste2 ; » sans altérer la liste2 et de telle sorte que si la liste1 est modifiée, la liste2 ne l'est pas et réciproquement.

Dans un premier temps, vous testerez votre classe Liste avec des **entiers**, puis ensuite avec des objets de la classe **Professeur**.

Bien sûr, on travaillera, comme d'habitude, en fichiers séparés afin de maîtriser le problème de l'instanciation des templates.

c) La liste triée (Essai3() et Essai4())

On vous demande à présent de programmer la **classe template ListeTriee** qui hérite de la classe Liste et qui redéfinit la méthode **insere** de telle sorte que l'élément ajouté à la liste soit inséré au bon endroit dans la liste, c'est-à-dire en respectant l'ordre défini par les opérateurs de comparaison de la classe template.

Dans un premier temps, vous testerez votre classe ListeTriee avec des **entiers**, puis ensuite avec des objets de la classe **Professeur**. Ceux-ci devront être triés par ordre alphabétique.

d) Parcourir et récupérer les éléments d'une liste : l'itérateur de liste

Dans l'état actuel des choses, nous pouvons ajouter/supprimer des éléments à une liste ou à une liste triée mais nous n'avons aucun moyen de les parcourir, élément par élément. La notion d'itérateur va nous permettre de réaliser cette opération.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **Liste** (elle permettra donc de parcourir tout objet instanciant la classe **ListeTriee**), et qui comporte, au minimum, les méthodes et opérateurs suivants :

- **reset()** qui réinitialise l'itérateur au début de la liste.
- **end()** qui retourne le booléen true si l'itérateur est situé au bout de la liste.
- **Opérateur ++** qui déplace l'itérateur vers la droite.
- **Opérateur de casting ()** qui retourne (par valeur) l'élément pointé par l'itérateur.

On vous demande donc d'utiliser la classe Iterateur afin de vous faciliter l'accès aux containers. Son usage correct sera vérifié lors de l'évaluation finale.

Etape 7 (Test7.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 bytes.

La classe Event se sérialise elle-même

On demande de compléter la classe **Event** avec les deux méthodes suivantes :

- **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données de l'événement (identifiant, intitulé, Timing) et cela **champ par champ**. Le fichier obtenu sera un fichier **binaire** (utilisation des méthodes **write** et **read**).

- **Load(ifstream & fichier)** permettant de charger toutes les données relatives à un événement enregistré sur le flux fichier passé en paramètre.

Afin de vous aider dans le développement, on vous demande d'utiliser l'encapsulation, c'est-à-dire de laisser chaque classe gérer sa propre sérialisation. En d'autres termes, on vous demande d'ajouter aux classes **Timing** et **Temps** les méthodes suivantes :

- **void Save(ofstream & fichier) const** : méthode permettant à un objet de s'écrire lui-même, **champ par champ**, sur le flux fichier qu'il a reçu en paramètre.
- **void Load(ifstream & fichier)** : méthode permettant à un objet de se lire lui-même, **champ par champ**, sur le flux fichier qu'il a reçu en paramètre.

Ces méthodes seront appelées par les méthodes Save et Load de la classe Event lorsqu'elle devra enregistrer ou lire sa variable membre de type Timing qui fera de même pour lire ses variables de type Temps.

Important

Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une **chaine de caractères** « chaine » (type **char ***), on enregistrera tout d'abord le **nombre de caractères** de la chaine (strlen(chaine)) puis ensuite la **chaine elle-même**. Ainsi, lors de la lecture dans le fichier, on lit tout d'abord la taille de la chaine et on sait directement combien de caractères il faut allouer et lire ensuite.

Etape 8 (Test8.cpp) :

Un conteneur pour nos conteneurs : La classe Horaire

Un horaire est avant tout composé de professeurs, de groupes, de locaux. On vous demande donc de regrouper toutes ces données au sein de la même classe :

```
class Horaire
{
  private :
    ListeTriee<Professeur> professeurs;
    ListeTriee<Groupe>     groupes;
    ListeTriee<Local>      locaux;

  public:
    Horaire();
    ~Horaire();

    void ajouteProfesseur(const char* nom,const char* prenom);
    void afficheProfesseurs();
    void supprimeProfesseurParIndice(int ind);
    void supprimeProfesseurParId(int id);

    void ajouteGroupe(int numero);
    void afficheGroupes();
    void supprimeGroupeParIndice(int ind);
    void supprimeGroupeParId(int id);

    void ajouteLocal(const char* nom);
    void afficheLocaux();
    void supprimeLocalParIndice(int ind);
    void supprimeLocalParId(int ind);
};
```

Cette classe comporte notamment :

- Les **listes triées** de professeurs (**ordre alphabétique**), groupes (**ordre alphabétique**), locaux (**ordre croissant** de leur numéro).

- Un **constructeur par défaut**. Inutile de faire un constructeur de copie car il n'existera qu'un seul objet instance de la classe Horaire dans l'application, on parle alors de **« singleton ».**

- Les méthodes **void ajouteXXX(…)** permettent d'ajouter un XXX (XXX = Professeur, Groupe ou Local) 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 Planifiable une **variable membre statique entière appelée idCourant**

qui sera utilisée et incrémentée de 1 à chaque ajout. Ceci assurera donc l'unicité de l'identifiant des planifiables.

- Les méthodes **void afficheXXX()** (XXX = Professeurs, Groupes ou Locaux) qui permettent d'afficher le contenu des différents conteneurs. Rappelez-vous que la classe ListeTriee dispose d'une méthode Affiche()…

- Les méthodes **void supprimeXXXParIndice(int ind)** qui permettent de supprimer le XXX (XXX = Professeur, Groupe ou Local) dont l'indice dans sa liste triée est **ind**.

- Les méthodes **void supprimeXXXParId(int id)** qui permettent de supprimer le XXX (XXX = Professeur, Groupe ou Local) dont l'identifiant est **id**. Ceci peut se faire très facilement à l'aide de l'itérateur…

Etape 9 (utilisation de InterfaceQt.tar fourni)

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

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 un **objet de classe Horaire**. 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. Un makefile est également fourni pour la compilation. A vous de combiner votre makefile actuel et le makefile fourni.

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 dialogueMessage(const char* titre,const char* message);
- void dialogueErreur(const char* titre,const char* message);
- const char* dialogueDemandeTexte(const char* titre,const char* question);
- int dialogueDemandeInt(const char* titre,const char* question);

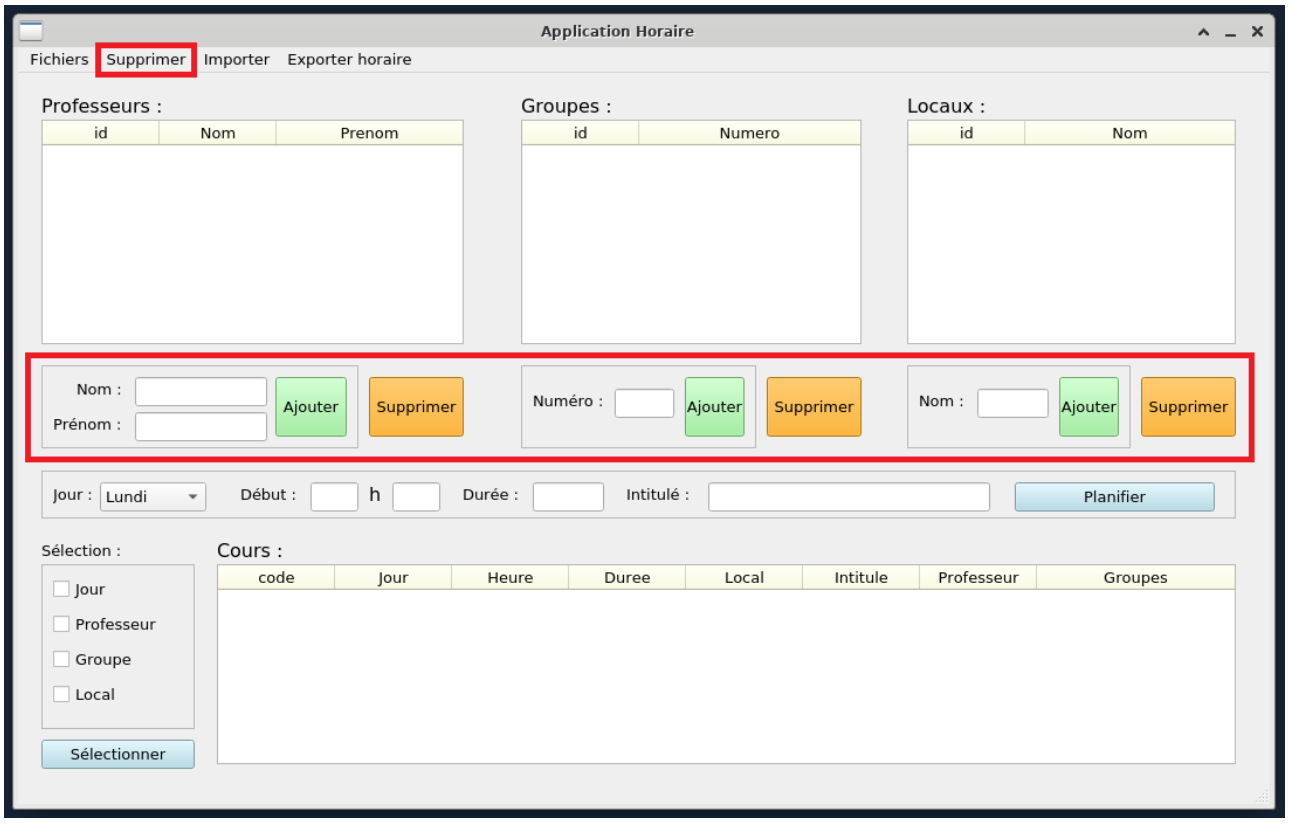
Etant une classe C++ au sens propre du terme, vous pouvez ajouter, à la classe **ApplicHoraireWindow**, des variables et des méthodes membres en fonction de vos besoins.

b) 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 programmée à l'étape précédente dans la classe Horaire. Il suffit donc d'instancier un objet de la classe Horaire et de le manipuler à l'aide de l'interface graphique. Toute la logique de l'application, dite « logique métier », se trouve dans la classe Horaire.

La première chose à faire est donc d'ajouter un **objet horaire de la classe Horaire** en tant que **variable membre** de la classe **ApplicHoraireWindow**. Ainsi l'objet horaire sera accessible dans toutes les méthodes de la classe.

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 **const char* getNomProfesseur()** et **const char* getPrenomProfesseur()**. Si une des chaînes de caractères est vide, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char* titre,const char* message).**
2. appeler la méthode **void ajouteProfesseur(const char* nom, const char* premom)** de l'objet **horaire**.
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 videTableProfesseurs()**.
    b. récupérer la liste triée des professeurs. L'**itérateur** vous permet de parcourir cette liste et d'en récupérer chaque élément. La fonction **Tuple()** de chaque objet Professeur retournera le tuple qui pourra être inséré dans la table des professeurs à l'aide de la méthode **void ajouteTupleTableProfesseurs(const char *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 getIndiceProfesseurSelectionne()** 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 dialogueErreur(const char* titre,const char* message)**.
2. appeler la méthode **void supprimeProfesseurParIndice(int ind)** de l'objet horaire.
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 dialogueDemandeInt(const char* titre,const char* question)**.
2. appeler la méthode **void supprimeProfesseurParId(int id)** de l'objet horaire.
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 :

La planification des cours : la classe Cours

a) Une classe spécialisée pour les Cours

Il s'agit tout d'abord de développer la classe Cours 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 **Cours**, qui hérite de la classe Event, et qui présente les variables membres supplémentaires suivantes :

- **idProfesseur** : l'identifiant (de type **int**) du professeur concerné par ce cours.
- **idLocal** : l'identifiant (de type **int**) du local concerné par ce cours.
- Une liste « **idGroupes** » : variable du type **Liste<int>** contenant les identifiants des groupes concernés par ce cours.

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

- **code** (entier qui identifie de manière unique un cours donné)
- **intitulé** 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 **Cours** devra être munie des **opérateurs** de comparaison de telle sorte que l'ordre chronologique soit respecté, ainsi que les opérateurs nécessaires pour que l'on puisse utiliser le conteneur ListeTriee avec des objets de type Cours.

Outre ses accesseurs classiques, la classe **Cours** devra disposer des méthodes :

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

b) Mise à jour de la classe Horaire

La première chose est d'ajouter à la classe **Horaire** une variable membre **courss** de type **ListeTriee<Cours>** et qui contiendra **l'ensemble des cours planifiés** par ordre chronologique.

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

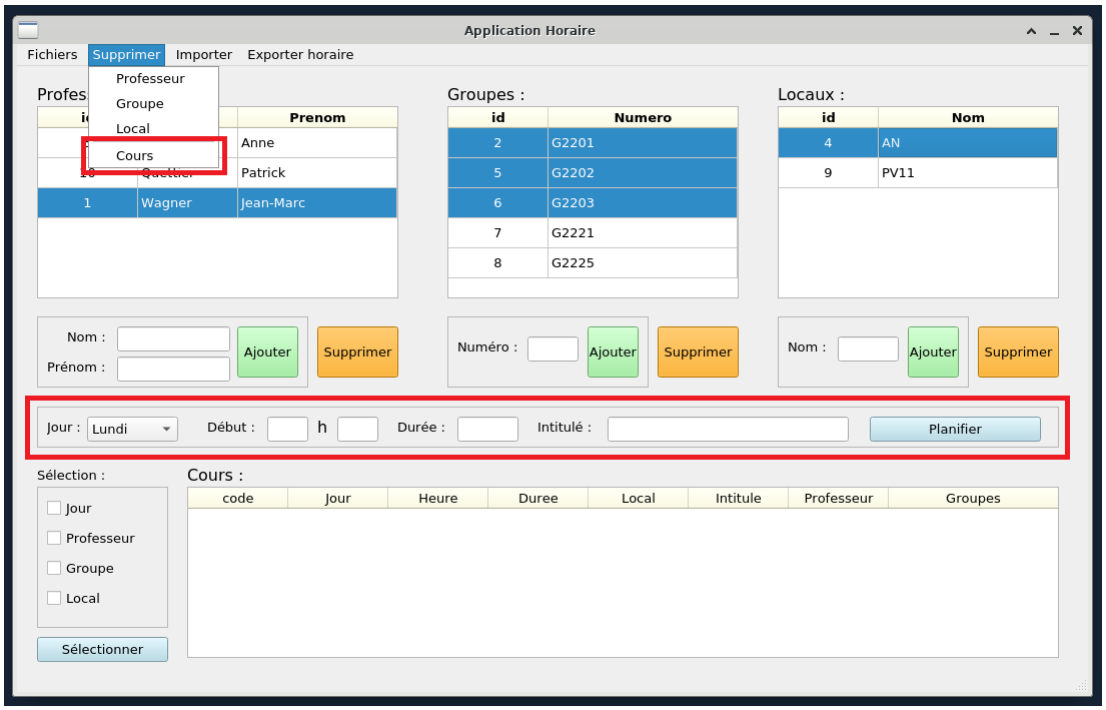
- **bool professeurDisponible(int id,const Timing& timing)** : qui retourne true si le professeur d'identifiant **id** est disponible (non occupé) pendant le laps de temps défini par **timing**, et false sinon. Il suffit pour cela de balayer le conteneur courss et de voir si timing « intersecte » temporellement 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 intersecte(const Timing& t)** à la classe **Timing** et permettant de tester si deux timing entrent en conflit temporellement.

- **bool groupeDisponible(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 courss et de voir si le groupe id est concerné par un cours déjà planifié (vous disposez pour cela de la méthode **contientIdGroupe** de **Cours**) 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 localDisponible(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 courss 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 planifie(Cours& c,const Timing& t)** qui
  - reçoit un objet c de type **Cours** contenant déjà un intitulé, un id de professeur, un id de local et contenant au moins un id de groupe dans sa liste idGroupes. Par contre, sa variable membre timing est NULL.
  - 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 courss 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 planifie de l'objet **horaire** afin de planifier le cours.

L'instanciation des objets de type **Temps** et **Timing**, ainsi que la méthode planifie de la classe **Horaire** sont susceptibles de lancer des **exceptions de type TempsException 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 ajouteTupleTableCourss(const char *tuple)** de la classe **ApplicHoraireWindow** (exactement comme c'est déjà fait pour les tables des professeurs, des groupes et des locaux). Cependant, la classe **Cours** 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 **Horaire** la méthode

- **const char* getTuple(const Cours& c)** qui reçoit en paramètre un objet c de type Cours déjà planifié et présent dans le conteneur **courss**. Vu que la classe Horaire 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;G2201,G2203 »

  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 dialogueDemandeInt(const char* titre,const char* question)**.
- supprime le cours correspondant du conteneur **courss** et met à jour la table des cours.

Etape 11 :

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

a) Des méthodes Save() et Load() pour la classe Horaire

Comme nous l'avons dit précédemment, un horaire devra être sérialisé dans un **fichier binaire** (utilisation des méthodes write et read) d'extension « **.hor** ». Un tel fichier sera structuré de la manière suivante :

1. la variable statique **idCourant** de la classe **Planifiable**. En effet, lorsque nous ouvrirons un fichier, l'ajout futur d'un nouveau planifiable devra générer un identifiant non encore utilisé.
2. La liste triée des professeurs : pour cela, on écrira tout d'abord le **nombre de professeurs**, puis ensuite **chacun des objets Professeur** présents dans le conteneur professeurs. Lors de la lecture, on saura alors exactement combien d'objets Professeur lire.
3. la liste triée des groupes : pour cela, on écrira tout d'abord le **nombre de groupes**, puis ensuite **chacun des objets Groupe** présents dans le conteneur groupes. Lors de la lecture, on saura alors exactement combien d'objets Groupe lire.
4. la liste triée des locaux : pour cela, on écrira tout d'abord le **nombre de locaux**, puis ensuite **chacun des objets Local** présents dans le conteneur locaux. Lors de la lecture, on saura alors exactement combien d'objets Local lire.
5. la variable statique **codeCourant** de la classe **Event**. En effet, lorsque nous ouvrirons un fichier, la planification future d'un nouveau cours devra générer un code non encore utilisé.
6. la liste triée des cours : pour cela, on écrira tout d'abord le **nombre de cours**, puis ensuite **chacun des objets Cours** présents dans le conteneur courss. Lors de la lecture, on saura alors exactement combien d'objets Cours lire.

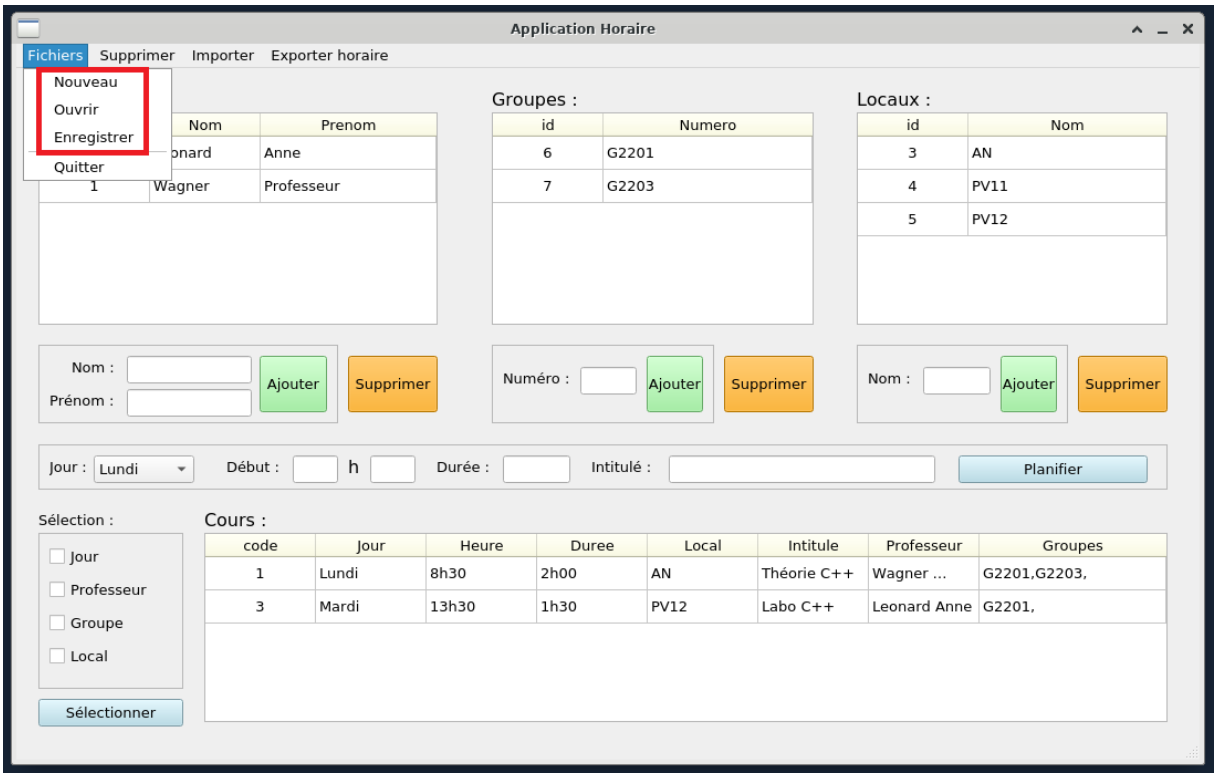
On vous demande donc d'ajouter à la classe **Horaire** :

- La méthode **Save(const char* nomFichier)** qui enregistrera dans le fichier « nomFichier » toutes les données de l'horaire selon la structure décrite ci-dessus.
- La méthode **Load(const char* nomFichier)** qui lira sur disque toutes les données d'un horaire.

Ces deux méthodes devront appeler les méthodes Save et Load de tous les objets intervenants (Professeur, Groupe, Local, Cours, …). Chaque objet va donc se sérialiser lui-même.

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 du fichier dans lequel l'horaire va être enregistré. Pour cela, vous disposez de la méthode **const char* dialogueDemandeTexte(const char* titre,const char* question)**.
- enregistre l'horaire sur disque en appelant la méthode **Save** de l'objet horaire.

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

- demande à l'utilisateur le nom du fichier de l'horaire qui doit être lu sur disque.
- lit l'horaire sur disque en appelant la méthode **Load** de l'objet horaire.
- met à jour les 4 tables 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 »** :

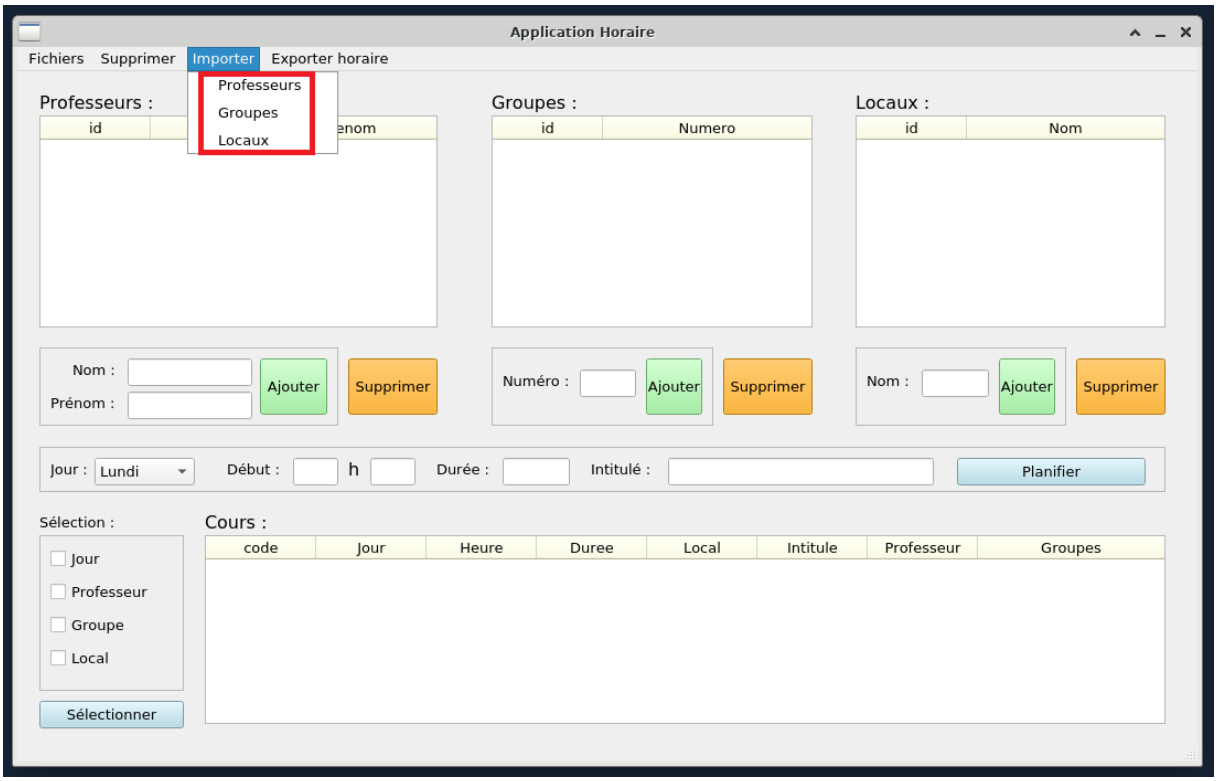
- vide tous les conteneurs de l'objet horaire.
- remet les variables statiques **Planifiable::idCourant** et **Event::codeCourant** à 1.
- met à jour les 4 tables de la fenêtre.

Etape 12 :

Importation de planifiables et exportation d'horaires : Fichiers textes

a) Importation de planifiables : 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

```
ANCIAUX;Daniel;
CAPRASSE;Francois;
CHARLET;Christophe;
CLAISSE;Nicolas;
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 **Horaire** les méthodes suivantes :

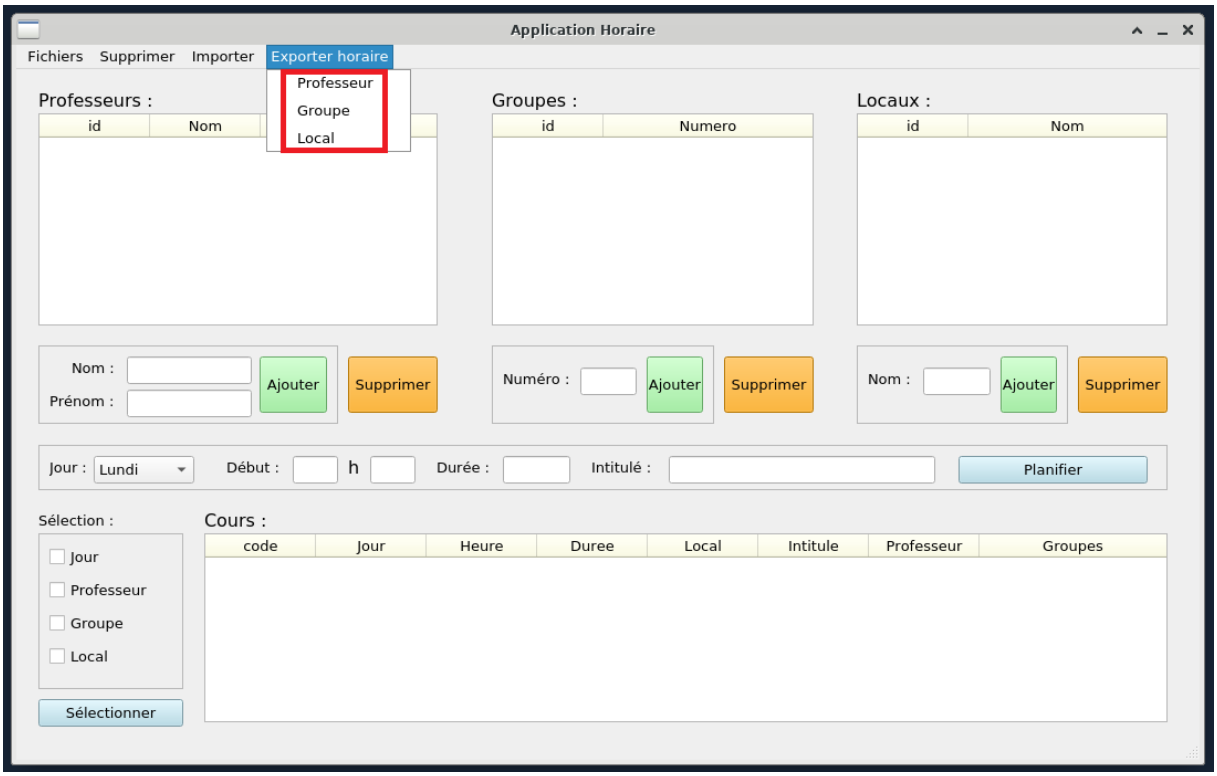
- **importeProfesseurs(const char* nomFichier)** : qui importe les professeurs présents dans le fichier csv dont le nom est passé en paramètre.
- **importeGroupes(const char* nomFichier)** : qui importe les groupes présents dans le fichier csv dont le nom est passé en paramètre.
- **importeLocaux(const char* nomFichier)** : 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 (**Professeurs.csv**, **Groupes.csv** et **Locaux.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.

b) Exportation de l'horaire d'un planifiable : 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.txt** » contenant

```
Horaire de Wagner Jean-Marc :

Lundi       8h20   2h00   AE    Théorie C++   G2201,G2221
Lundi       13h30  1h30   LE0   Labo UNIX     G2223
Mardi       10h30  2h00   PV11  Labo C++      G2221
…
```

Et par exemple pour le groupe 2221, un fichier "**G2221.txt**" :

```
Horaire de G2221 :

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 **Horaire** les méthodes suivantes :

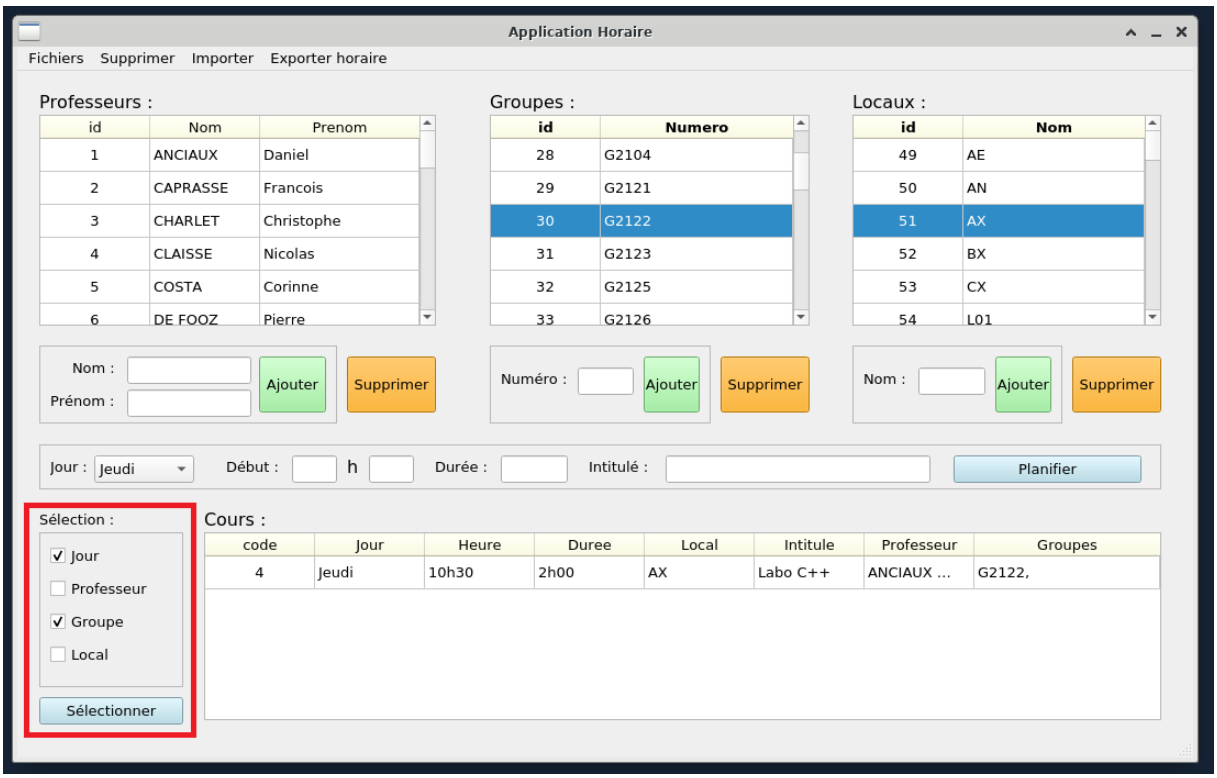
- **exporteHoraireProfesseur(int id)** : qui exporte l'horaire du professeur dont l'identifiant est passé en paramètre.
- **exporteHoraireGroupe(int id)** : qui exporte l'horaire du groupe dont l'identifiant est passé en paramètre.
- **exporteHoraireLocal(int id)** : qui exporte l'horaire du local dont l'identifiant est passé en paramètre.

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

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 courss de l'objet **horaire**) en tenant comptes des checkboxs sélectionnés :

- **Si le checkbox « Jour » est sélectionné** : 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 du jour pour la sélection et on les affiche tous.
- **Si le checkbox « Professeur » est sélectionné** : 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 le checkbox « Groupe » est sélectionné** : 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 « 2122 »), sinon on ne tient pas compte du groupe et on les affiche tous.
- **Si le checkbox « Local » est sélectionné** : 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 !