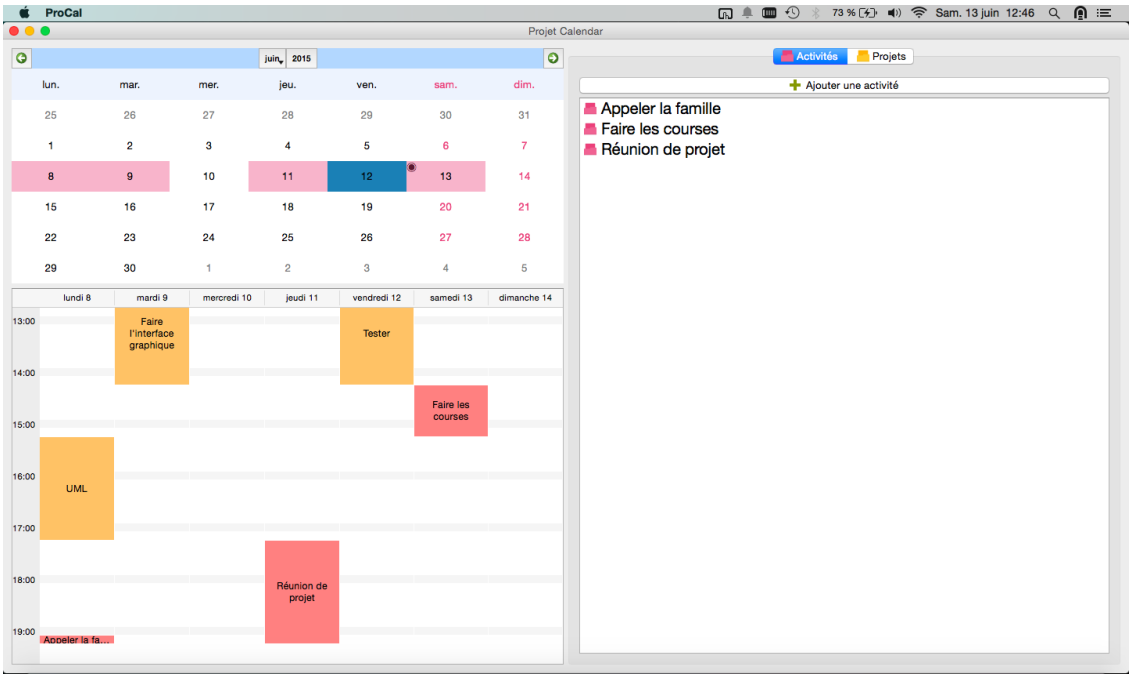
LO21 - ProCal

Développement d'un calendrier gestionnaire de projet



Printemps 2015

Sylvain DOIGNON, GI02
Léo MAISONNEUVE, GI02

Introduction

Le but de ce projet est de réaliser une application destinée à mixer les fonctionnalités d'un agenda avec celle d'un gestionnaire de projet.
Ainsi, grâce à cette application, l'utilisateur doit pouvoir avoir une vue hebdomadaire de son agenda, avec dans celle-ci les évènements qui y sont programmés. Il doit également pouvoir ajouter des activités dans son calendrier.
De plus, il peut créer un projet et lui attribuer des tâches à effectuer. Ces tâches pourront ensuite être programmées sur le calendrier.

Nous avons tenté de réaliser ce programme de manière correcte et réaliste. En effet, nous avons au maximum tiré parti de Qt de ses innombrables possibilités. Nous voulons souligner aussi que nous avons utilisé du C++11 dès que possible (*initializer list, nullptr, =delete, etc*).

I. Architecture

A. Générale

Devant les nombreuses classes à réaliser, il nous a semblé futé de bien ranger les fichiers du projet. Ainsi, il y a une partie “CORE” qui s’utilise sans interface graphique. L’autre partie de l’application est “UI” et concerne l’interface graphique. On peut décider de lancer cette interface en mettant la directive préprocesseur launchGui à *true* dans le main.cpp (comportement par défaut).

Nous avons aussi séparé les entêtes des fichiers sources. Le projet est donc bien organisé et il a été facile le développer en sachant, par exemple, exactement quel fichier correspond à quelle fonctionnalité.

Dans la partie core, beaucoup de méthodes ont des arguments par défaut ce qui permet de rapidement utiliser cette partie sans interface graphique.
p1->creerTacheUnitaire("Tache1") va, par exemple, créer une tache non préemptive, sans prédécesseur avec une date de disponibilité égale à maintenant et une tache d’échéance dans un mois.

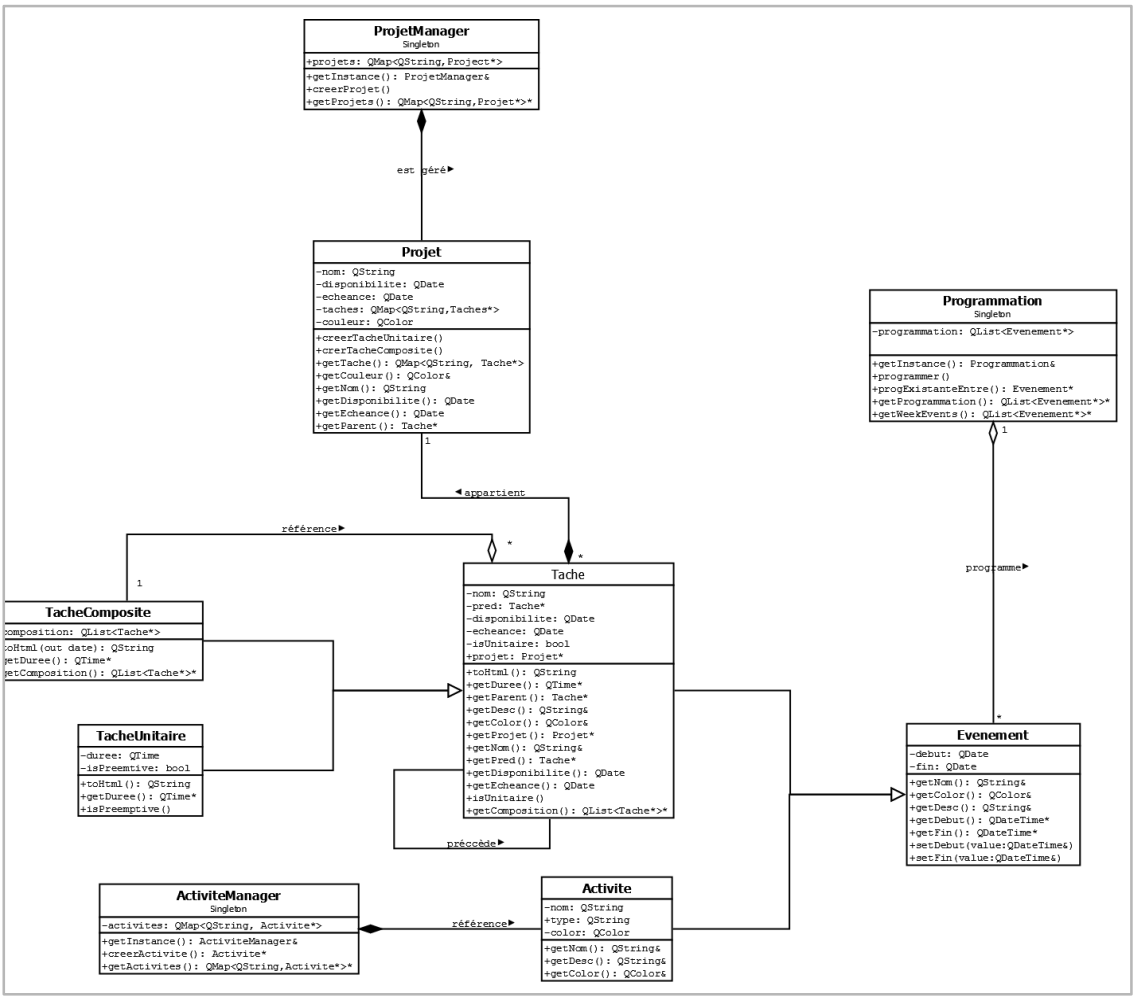
Nous allons y revenir, mais plusieurs classes utilisent un design pattern singleton. Il a été choisi de le réaliser de manière statique. C’est à dire :

```
static ProjetManager& getInstance()
{
    static ProjetManager instance;
    return instance;
}
```

Et non avec une création de pointeur vers l’instance. De cette manière, on évite la création d’un attribut et surtout, on évite le mot clé “new” qui alloue de la mémoire qui n’est pas gérée automatiquement. Voici pourquoi nous avons préféré implémenter nos singletons de cette manière.

A. UML

Nous avons choisi de développer notre architecture comme décrit dans l'UML suivant.



Voir ANNEXE 1

On retrouve à première vue les grandes lignes du sujet :

- Une partie qui va gérer des évènements en les programmant
- Une partie "gestion de projet" qui va permettre la création de projet, puis de tâche.

Ces deux parties étant liées par l'héritage entre Tache/Activité et Évènement.

B. Partie programmation

Tout d'abord la classe **Programmation**, cette classe va permettre de gérer tous les évènements programmés par l'utilisateur. Il est donc essentielle de cette classe soit unique au sein du programme. Nous avons donc choisie de l'implémenter avec le design patern **Singleton**.

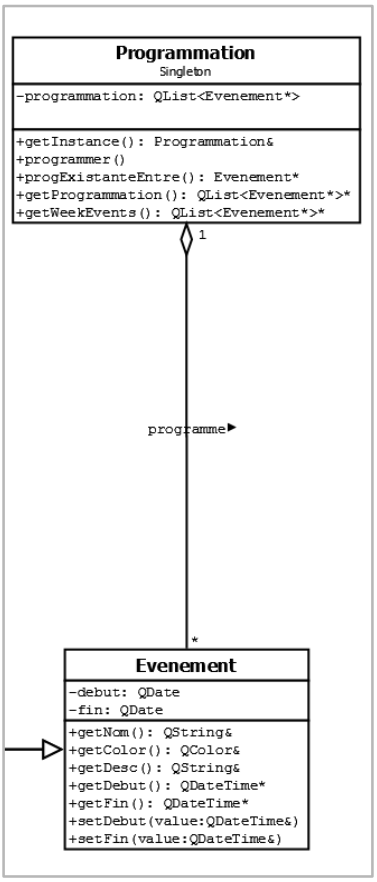
Cette classe à donc comme unique attribut une QList référençant tout les évènements.

La méthode **programmer** permet de programmer un évènement, et donc de le mettre dans la QList (nous avons beaucoup utilisée les structures de données proposées par Qt) :

```
programmation.append(e)
```

La méthode **progExistanteEntre** permet de renvoyer l'évènement existant entre deux dates.

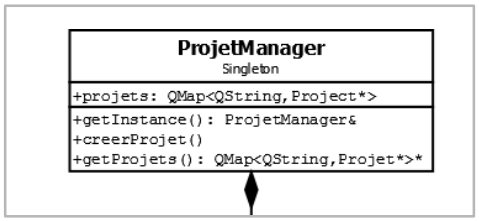
La méthode **getWeekEvents** permet d'avoir la liste des évènements d'une semaine à partir d'une date. Cette méthode sera utilisée pour afficher la vue hebdomadaire.



La classe **évènement** est une classe abstraite, ainsi que la classe Tache, seul les classes héritées **TacheUnitaire** et **Activite** pourront être instanciées.

A part les setters, toutes les méthodes de cette classe, sont des méthodes virtuelles. Elles permettent de récupérer les attributs, des classes hérités.

C. Partie gestion de projet



La classe, **ProjectManager** doit gérer tout les projets crées pas l'utilisateur. Il est donc essentielle de cette classe soit unique au sein du programme, au même titre que la classe **Programmation**. Nous avons donc également choisi de l'implémenter avec le design patern **Singleton**.

Cette classe comme unique attribue une QMap référençant tout les projets et le nom qui leur est associé.

L'utilisation d'une QMap facilite ici énormément l'accès aux projets stockés. En effet pour obtenir la liste des projets, il nous suffit d'utiliser un "foreach" (macro proposée par Qt) de la manière suivante :

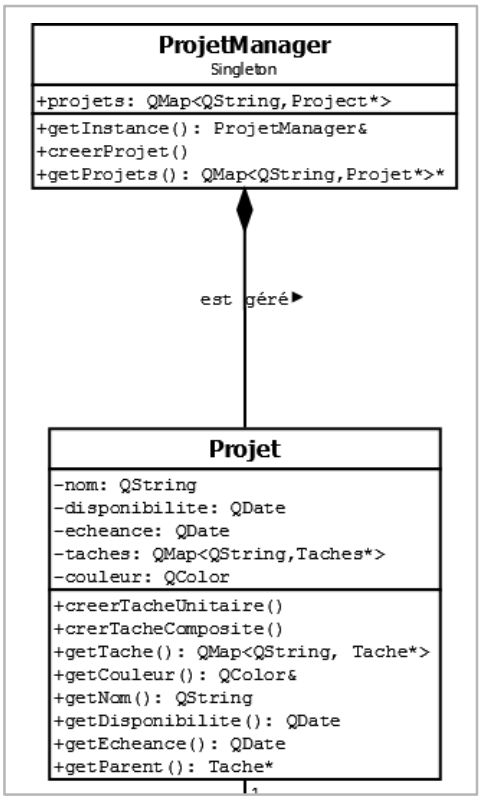
```
//Récupération de l'instance de ProjectManager
        ProjetManager& myProjetManager = ProjetManager::getInstance();
//Parcours de la liste des projets
        foreach(Projet* projet, *myProjetManager.getProjets())
```

Cette classe gère ainsi les projets, une association de composition entre **ProjetManager** et **Projet** est donc de mise.

La classe **Projet,** quant a elle, doit pouvoir stockée toutes les informations concernant un projet :

- Son titre
- Ses dates de disponibilité et d'échéance
- Toutes les tâches qui lui sont liées

Nous avons également ajouté un attribut couleur pour pouvoir repérer plus aisément, sur la vue hebdomadaire, les tâches appartenant au même projet.



A la même manière que le ProjectManager, nous avons utilisé une QMap pour stocker les Tâches appartenant à un projet.

Parmi les méthodes de cette classe, nous avons des getters pour les attributs mais aussi :

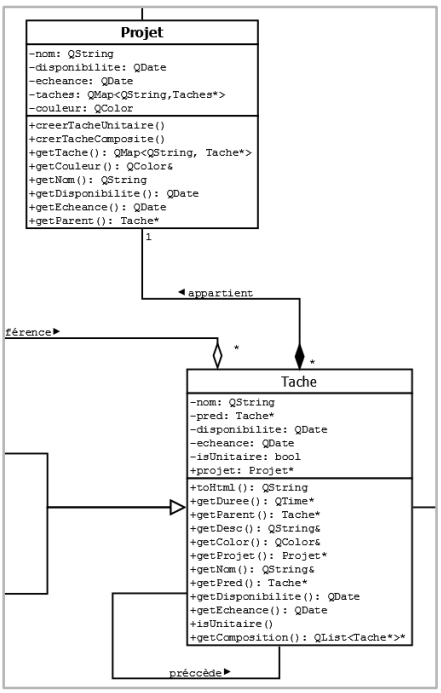
- Deux méthodes **creerTacheUnitaire** et **CreerTacheComposite**, pour créer des taches unitaires et composites

Ces méthodes créent des tâches puis les insèrent dans la QMap de stockage des tâches de la manière suivante :

```
taches.insert(tache_nom, Tache);
```

- Une méthode **getParent** qui renvois la tache parente dans le cas d'une tache sous tâche dans une tache composite.

Un projet est donc formé de plusieurs tâches et celle-ci ne peuvent appartenir qu'à ce projet. La encore une association de composition entre la classe **Projet** et **Tache** est donc nécessaire.



La classe **Tache** est une classe abstraite, seul les classes héritées, **TacheComposite** et **TacheUnitaire** pourront être instanciées.

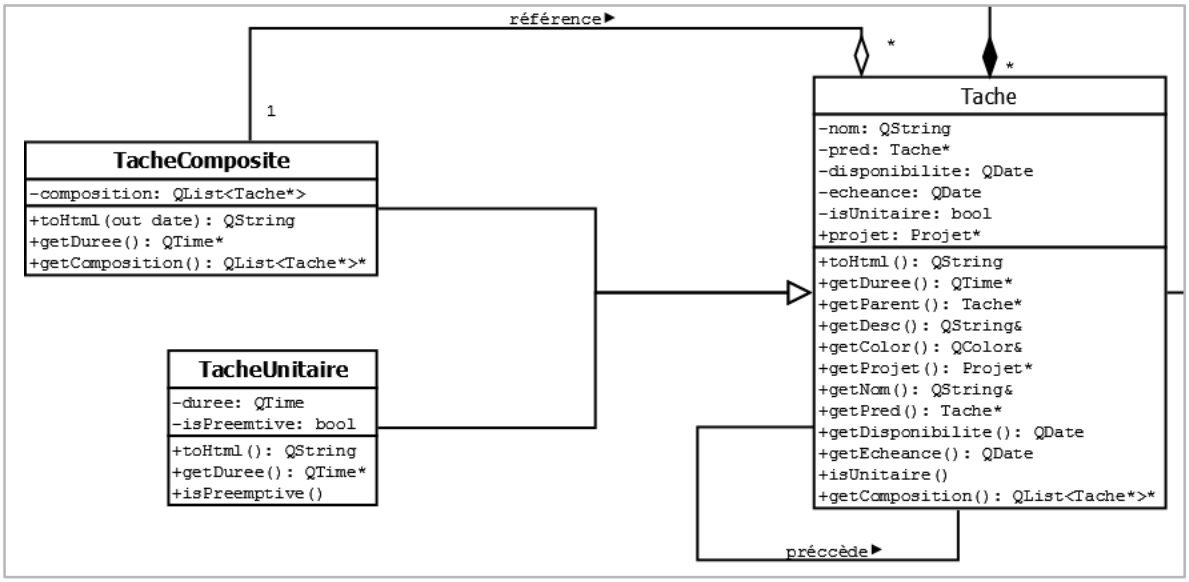
Néanmoins, plusieurs attributs sont communs à ces deux classes :

- Son nom
- Sa contrainte de précédence (de type Tache*)
- Ses dates de disponibilité et d'échéance
- Son type (Unitaire/Composite)
- Le projet auquel elle appartient (de type Projet *)

Le constructeur de Tache est mis ici en "protected". Le constructeur sera déclaré en public dans les sous classe, pour permettre l'initialisation de l'attribut type.

En plus des getter habituels, trois méthodes virtuelles sont également implémentées :

- La méthode **toHtml** qui renvois une QString formatée en fonction du type de la tache, décrivant ses caractéristiques.
- La méthode **getDuree** qui renvois, dans le cas d'une tache unitaire, sa durée.
- La méthode **getComposition** qui renvois, dans le cas d'une tache composite, une Map de toutes les tâches dont elle est composée.



La classe **TacheUnitaire** hérite donc de **Tache**, celle-ci à deux attributs en plus :

- La durée de la tâche
- La préemptivité de la tâche

La classe **TacheComposite** hérite également de **Tache** avec en plus un attribut QMap qui stock les taches dont elle est composée.

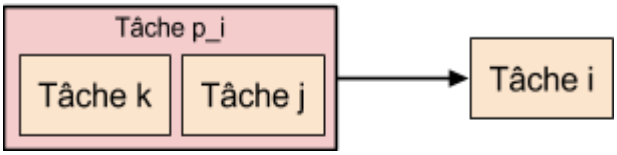
Il nous est aussi demandé de gérer les contraintes de précédences :

"Une tâche i (unitaire ou composite) est caractérisée [...] par un ensemble de taches (unitaires ou composites) qui doivent avoir été terminées avant de commencer la tache i."

Une interprétation de cette phrase étant possible, voici celle que nous avons choisie :

Une tâche, i, ne peut avoir qu'une seul tâche précédente, j. Mais, comme cette tâche précédente, j, peut elle aussi avoir une tâche précédente, k, alors la tâche i a *un ensemble de tâches* [j et k, ici] *qui doivent avoir été terminées avant* d'être commencé.

Si toute fois l'utilisateur veut faire qu'une tâche, i, ai plusieurs tâches précédentes, j et k , "de même niveau" alors, il est invité à programmer une tâche composite, p_i, contenant les tâches j et k, qui pourra alors être précédent de la tâche i.



D. Interface graphique

Nous avons utilisé au maximum les possibilité de Qt, ainsi nos interfaces sont crées avec Qt Designer.

Néanmoins, nous avons parfois voulu étendre les capacités des widgets proposés par Qt, ainsi nous avons dérivé l'objet QCalendarWidget pour créer notre propre widget de calendrier. Celui-ci indique par un rond le jour actuel et, surtout, notifie qu'un jour possède un évènement programmé.

Quelques icônes et caractères Unicode ont été utilisés, pour remplir la même fonction. L'avantage est évidement la taille du projet qui n'augmente pas avec l'utilisation d'un caractère Unicode.

Enfin, les contraintes sur les taches, projets, activités ont toujours été implémentées également visuellement. Typiquement, lors de l'ajout d'une tache, le widget de sélection de date de disponibilité est borné visuellement (par Qt) par les dates du projet.

L'idée de base pour cette interface est de faire une interface ergonomique. Nous avons donc décidé de ne pas afficher toute les fonctionnalité d'un coup à l'utilisateur. Mais plutôt de les montrer lorsqu'il en a besoin. Il en résulte une interface sobre où une grande place est dédiée à l'affichage de l'agenda.

E. Utilisation pratique

Voici le déroulement d'une utilisation, avec notre architecture.

1. L'utilisateur veut **programmer une activité.**

La classe **Activité** est instanciée, les attributs propres, *nom* et *type*, sont renseigné ainsi que les attributs hérités, *fin* et *début*. L'activité est ensuite programmée (et stockée) dans la classe **Programmation**.

2. L'utilisateur veut **créer un projet.**

La classe **Projet** est instanciée, les attributs *nom*, *disponibilité* et *échéance* sont renseignés. Le projet est ensuite stocké dans la classe **ProjectManager**.

3. L'utilisateur veut **ajouter des tâches à un projet.**

Il commence par créer les tâches unitaires du projet. La classe **TacheUnitaire** est donc instanciée, renseignée et stocké dans la classe **Projet** correspondante.

S'il veut créer une tache composite, l'utilisateur devra premièrement créer les sous tâches puis créer un tâche composite les contenant. La classe **TacheComposite** sera alors instanciée et les tâches unitaires y seront stockées.

4. L'utilisateur veut **programmer une tâche.**

Une fois la tâche crée, il suffit de renseigner ses attributs hérités *fin* et *durée* puis de la stocker dans la classe **Programmation**. Notons que, seule une tache unitaire peut-être programmée.

II. Évolutions

A. Design patterns

Concernant les designs patterns, nous avons remarqué la possibilité de créer un design pattern composite pour tache. En effet, il est pratique d'abstraire le concept de tache, d'autant plus que Tache Composite et Tache Unitaire ont globalement le même comportement.
Le design pattern visitor est également intéressant car il lest capable de retrouver le type exact d'une tache.

Nous avons essayé de prendre le meilleurs de ces deux design patterns pour créer notre classes taches. On retrouve la composition de composite vers composant tout en implémentant des méthodes virtuelles. Enfin, l'attribut 'is_unitaire' est une manière peut-être simpliste de réaliser en partie le design pattern visitor, mais devant le faible nombre de type de taches (2), il nous a paru judicieux de l'implémenter comme ceci.

Aussi, ProjetManager, ActiviteManager et Programmation sont des classes singleton. Nous ne voyons vraiment pas comment faire sans ! Ce design pattern est extrêmement pratique et permet d'utiliser la même instance d'une classe n'importe où dans l'application, y compris dans une toute nouvelle partie. Il suffit d'inclure l'entête de ces classes.

Enfin, ExportManager s'inspire fortement du design pattern stratégie. Néanmoins, les exports XML, Texte ou autres sont fondamentalement trop différents pour appliquer ce design pattern sans créer une classe très lourde et compliquée.

B. Code et Doxygen

Toutes les évolutions partent avant tout d'une bonne compréhension de l'ensemble du programme. C'est pourquoi nous n'avons pas négligé la documentation. Celle-ci a été réalisée avec Doxygen.

Pour plus de clarté et d'attrait vers ces pages nous avons décidé de remplacer le thème par défaut des pages HTML par quelque chose de plus moderne : twitter bootrap.

De plus, le projet, à notre avis, est réalisé dans un C++ clair et intuitif. Si quelque chose ne l'est pas, nous avons positionné des commentaires.

C. Ouverture

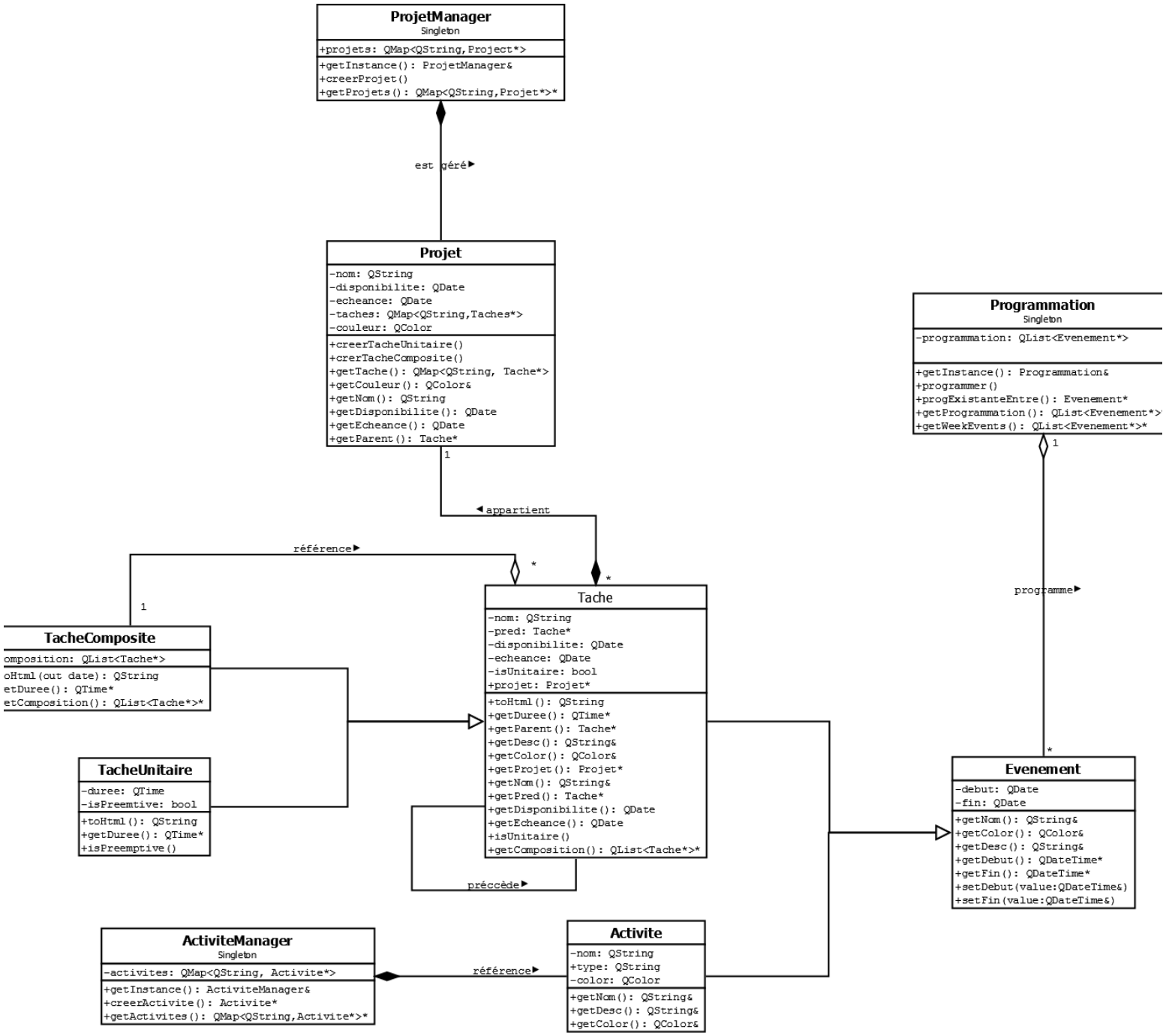
Enfin, les évolutions partent d'un besoin, d'une idée ! Voici quelques fonctionnalités non demandées dans le sujet qui nous parraissent intéressantes :

- Éditer/Supprimer les tâches, projets et activité
- Importer un projet
- Enregistrer les informations rentrées entre deux lancements de ce programme avec une base SQLite par exemple
- Ajouter une notion de personnes appartenant à un projet et leur laisser la possibilité de synchroniser leurs projets communs par le réseau

Conclusion

Il a été très intéressant et même parfois amusant (sic !) de réaliser cette application. Notre niveau de C++ s'est envolé et la découverte de Qt ne nous a pas laissé indifférents. Nous admirons les créateurs qui ont très bien pensés leur architecture et nous réalisons que nous avons encore beaucoup de choses à apprendre dans ce domaine !

ANNEXE 1


ProjetManager
Singleton
+projets: QMap<QString,Project*>
+getInstance(): ProjetManager&
+creerProjet()
+getProjets(): QMap<QString,Projet*>*
est géré
Projet
-nom: QString
-disponibilite: QDate
-echeance: QDate
-taches: QMap<QString,Taches*>
-couleur: QColor
+creerTacheUnitaire()
+crerTacheComposite()
+getTache(): QMap<QString, Tache*>
+getCouleur(): QColor&
+getNom(): QString
+getDisponibilite(): QDate
+getEcheance(): QDate
+getParent(): Tache*
1
appartient
*
référence
1
Programmation
Singleton
-programmation: QList<Evenement*>
+getInstance(): Programmation&
+programmer()
+progExistanteEntre(): Evenement*
+getProgrammation(): QList<Evenement*>
+getWeekEvents(): QList<Evenement*>*
1
programme
*
Tache
-nom: QString
-pred: Tache*
-disponibilite: QDate
-echeance: QDate
-isUnitaire: bool
+projet: Projet*
+toHtml(): QString
+getDuree(): QTime*
+getParent(): Tache*
+getDesc(): QString&
+getColor(): QColor&
+getProjet(): Projet*
+getNom(): QString&
+getPred(): Tache*
+getDisponibilite(): QDate
+getEcheance(): QDate
+isUnitaire()
+getComposition(): QList<Tache*>*
TacheComposite
omposition: QList<Tache*>
oHtml(out date): QString
etDuree(): QTime*
etComposition(): QList<Tache*>*
TacheUnitaire
-duree: QTime
-isPreemtive: bool
+toHtml(): QString
+getDuree(): QTime*
+isPreemptive()
Evenement
-debut: QDate
-fin: QDate
+getNom(): QString&
+getColor(): QColor&
+getDesc(): QString&
+getDebut(): QDateTime*
+getFin(): QDateTime*
+setDebut(value:QDateTime&)
+setFin(value:QDateTime&)
précède
ActiviteManager
Singleton
-activites: QMap<QString, Activite*>
+getInstance(): ActiviteManager&
+creerActivite(): Activite*
+getActivites(): QMap<QString,Activite*>*
référence
Activite
-nom: QString
+type: QString
-color: QColor
+getNom(): QString&
+getDesc(): QString&
+getColor(): QColor&