P15

Rapport de Projet LO21

PROJECT CALENDAR

LEFEBVRE Clémence | JUDAS Théo

Sommaire

I. Description de l'architecture

A) UML

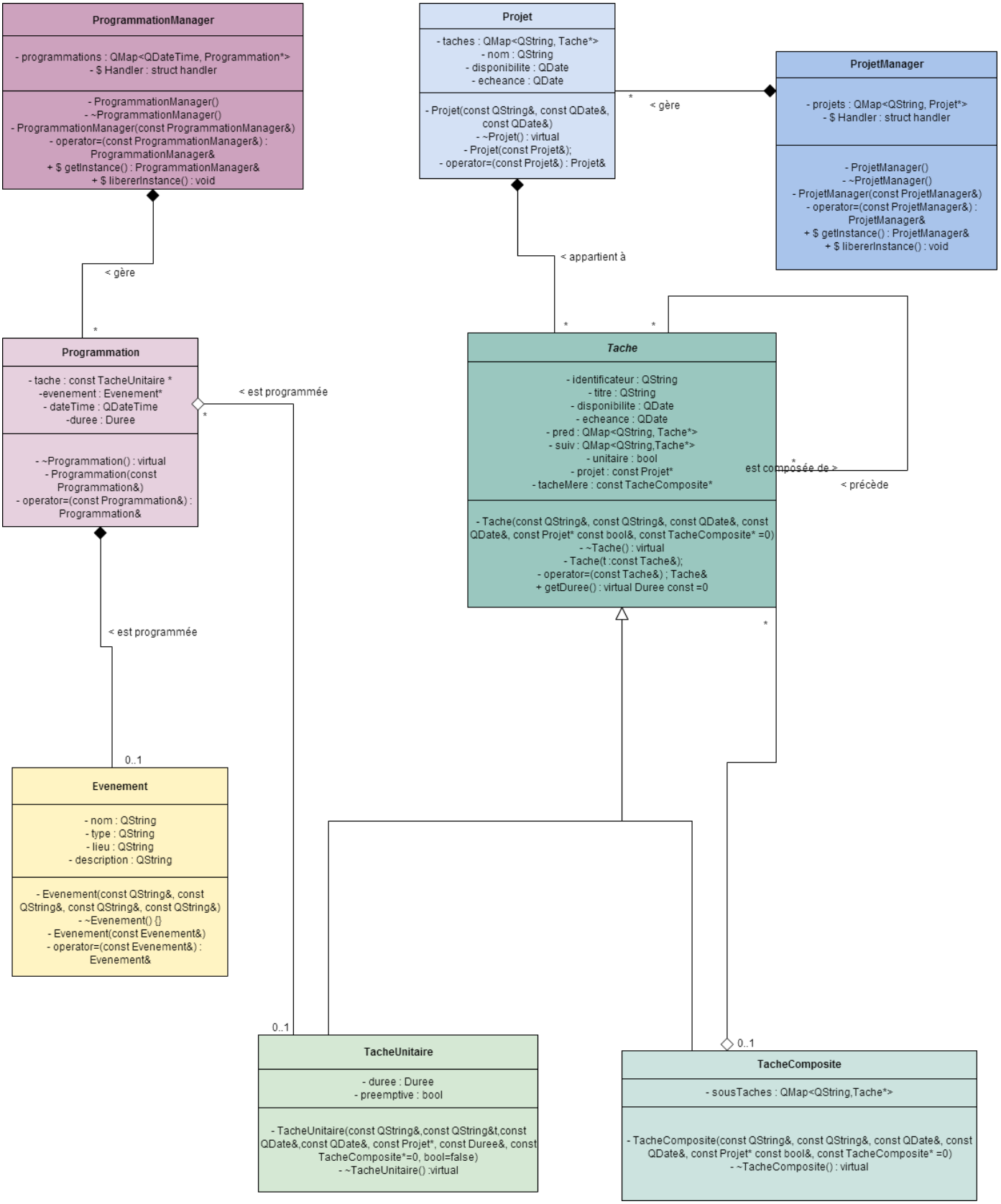
Ci-dessous est joint l'UML partiel du projet. Sont indiqués les attributs, les constructeurs et le destructeur de chacune des classes participant à l'architecture et la gestion des données du projet, la partie back-end.

On remarque qu'aucune classe n'est instanciable directement et que la recopie n'est jamais autorisée, on peut accéder seulement à *ProjetManager* et *ProgramamtionManager*, grâce au design pattern Singleton et l'attribut et les méthodes statiques.

Les projets sont tous gérés par *ProjetManager*, responsable de leur cycle de vie. Les tâches elles sont gérées chacune par un et un seul projet, aussi responsable de leur cycle de vie. Les tâches doivent être unitaire ou composite, la classe Tâche est abstraite et possède plusieurs méthodes virtuelles pures, comme *getDuree*() qui renvoie la durée d'une tâche unitaire ou la somme de la durée des sous-tâches d'une tâche composite. Des tâches se précèdent et se suivent, on stocke cette information avec des *QMap<QString,Tache*>*. On n'utilise pas comme deuxième élément un const, parce qu'une relation de précédence est dans les deux sens et donc si par exemple on supprime cette relation d'un côté, il faut le faire de l'autre côté, et par conséquent accéder à une méthode de la classe Tâche qui modifiera la *Tâche* pointée par l'élément du tableau. Les Tâches composites contiennent un tableau de sous-tâches, pointant sur leurs différentes sous-tâches.

Les programmations sont gérées de manière identique aux projets, avec *ProgramamtionManager*. De plus, dans le cadre du projet, il nous a semblé pertinent de voir qu'un *Evènement* (ou Activité) n'avait pas de raison d'exister s'il n'était pas programmé. C'est pour cela qu'une Programmation est responsable du cycle de vie de l'*Evènement* qu'elle pointe. Le pointeur sur l'Evènement est à 0 si la *Programmation* concerne une *Tâche*. Cette solution permet aussi de ne pas avoir de doublon d'informations sur la durée ou la date d'un *Evènement*, s'il n'était pas programmé.

Concernant la précédence, deux tâches sont liées par une contrainte de précédence si la première est dans le tableau des précédences de l'autre, si elle est précédente d'une tâche précédente de la seconde, si elle précède la tâche mère de la seconde tâche. Il n'y a pas de précédence entre une tâche mère et une tâche précédant une de ses sous-tâches.



UML Partiel de l'architecture back-end

B) Description du fonctionnement de l'architecture back-end

Toutes les fonctionnalités internes de l'énoncé sont couvertes.

Tache

Abstraite, la classe fournit un large panel de méthodes. Outre les accesseurs en lecture, parfois en écriture, des attributs, il y a des méthodes en lien avec les précédences et les programmations. Ainsi, à l'ajout d'une précédence, *verifAJoutPrecedence* permet de savoir si les deux tâches sont déjà liées par une relation de précédence, même au niveau de leur tâche mère, de sous-tâche ou si leurs dates sont cohérentes, ou encore que même si une tâche peut en précéder une autre, si la durée la tâche rend impossible sa fin avant la date d'échéance que la tâche qu'elle doit précéder, l'ajout de précédence est bloqué. *isSousTache* permet de savoir si une tâche est une sous-tâche directe ou indirecte d'une autre. Enfin, des méthodes virtuelles pures sont dédiées aux programmations pour savoir si une tâche est programmée, programmée avant une QDateTime précise ou encore savoir si ses précédences sont programmées (en vérifiant les précédences de sa tâche mère aussi).

TacheUnitaire

Par rapport à Tache, cette classe apporte la notion de préemption et l'attribut durée. On peut savoir si elle est programmée en regardant les programmations la concernant et si la *DureeRestanteAProgrammer* est égale à 0. Si cette durée est égale à 0 en s'intéressant en s'intéressant à une borne supérieure définie par une date, on sait si elle est programmée avant celle-ci.

TacheComposite

Une tâche composite peut avoir des sous-tâches si elles ne sont pas liées avec elle par une contrainte de précédence et si leurs dates rentrent dans la tranche de disponibilité de la tâche composite, et qu'elles sont dans le même projet. On gère également la suppression de sous-tâche, qui détruit le lien unissant une tâche composite à une sous-tâche mais ne détruit pas la sous-tâche.

Projet

A partir de Projet, on peut trouver les tâches disponibles à une certaine date, les tâches composites du projet. Savoir si une tâche fait partie du projet à partir de son identificateur, on fonctionne en O(1) grâce à la méthode *find* des *QMap*. L'ajout de tâche se fait par deux méthodes, une pour les tâches unitaires, une pour les tâches composites, vérifiant à chaque fois la cohérence des dates notamment. La suppression d'une tâche est possible, dans ce cas on veille à ce que la tâche disparaisse de toutes les relations dans lesquelles elle est impliquée, on supprime ses programmations et on la détruit. Cette méthode renvoie une itérateur sur les tâches du projet, qui facilite la suppression successives des tâches dans le cadre de la suppression d'un projet.

ProjetManager

Classe à la source des projets et des tâches. Ses méthodes permettent de trouver des tâches en cherchant dans chaque projet si la tâche existe. On peut trouver si un projet existe

à partir de son nom, supprimer un projet en supprimant successivement ses tâches puis en le supprimant.

Programmation

Une programmation concerne un évènement ou une tâche unitaire. Par conséquent, la tâche ou les attributs d'un évènement font partie des différents constructeurs de la programmation. Grâce à ses constructeurs, on assure qu'il n'est pas possible de programmer un évènement et une tâche unitaire avec la même programmation.

ProgrammationManager

Toutes les programmations sont regroupées dans cette classe. La classe permet notamment de savoir une plaire horaire est disponible, c'est-à-dire qu'aucune programmation de chevauche ou contient une plage horaire à une date précise. C'est à partir de cette classe qu'on instancie les programmations évènementielles et celles de tâches unitaires. On permet la déprogrammation totale d'une tâche, la déprogrammation partielle. La déprogrammation totale est surchargée pour les tâches composites, ce qui déprogramme totalement toutes les sous-tâches. La déprogrammation totale ou partielle déprogramme les tâches qui suivent la tâche concernée. On n'oublie pas qu'il est possible de déprogrammer des évènements, ce qui entraîne leur destruction.

C) Interface graphique

Toutes les fonctionnalités de l'énoncé sont couvertes. L'utilisateur a accès via des fenêtres dépendantes d'une fenêtre principale à toutes les fonctionnalités énoncées précédemment. De plus, on a inséré une *TreeView* qui propose une réelle arborescence, puisque les sous-tâches sont des branches des tâches composites de manière récursive. En cliquant sur un projet ou une tâche, sa description apparait à côté de l'arbre.

L'emploi du temps s'affiche pour une semaine sélectionnée par l'utilisateur grâce à *QTableWidget*, on affiche ensuite les informations sur les tâches et les évènements en cliquant sur une programmation de l'emploi du temps. Les programmations étant classées par *QDateTime* dans *ProgrammationManager*, le remplissage de l'emploi du temps est facilité.

Dès la création d'une tâche, one peut ajouter une tâche mère ou des précédences. L'application tente de faciliter la vie de l'utilisateur, en l'empêchant au maximum de rentrer des informations incohérentes, en ajoutant des bornes aux dates en fonction du projet sélectionné, en ne proposant de programmer que des tâches dont la date d'échéance n'est pas passée, en affichant des informations à la programmation, notamment pour la durée restante à programmer. Il y a beaucoup de tests et même si l'utilisateur rentre des informations incohérentes, des explications lui sont fournies pour qu'il y remédie.

Enfin, il est possible d'importer et d'exporter sous le format XML des fichiers. On peut sauvegarder u nouveau calendrier dans un fichier existant ou non, et si le chargement échoue, l'erreur est attrapée dans un catch.

II. Evolution de l'architecture

A) Possibilités

Il est possible vu que les destructeurs de *TacheUnitaire*, *TacheComposite*, *Projet*, *Programmation* et *Evenement* sont virtuels d'implémenter de nouvelles classes filles, avec des contraintes nouvelles par exemple ou des précisions, l'ajout d'attributs. Les constructeurs sont en protected, ce qui permet vraiment la naissance de classes filles. *Tache* peut bien évidemment donner des d'autres classes filles, mais il faudra se faire une place avec les tâches unitaires et composites et ajouter des méthodes à Projet pour l'ajout de tâches ni unitaires ni composites. Les méthodes *getDuree*, *isProgramee*, *isProgrammeeAvant* doivent également être définies si de nouvelles classes filles sont créées, car elles sont virtuelles pures.

Les méthodes de vérification d'ajout de précédence et de sous-tâche sont virtuelles donc on peut facilement ajouter de nouvelles contraintes si de nouvelles relations entrant en jeu avec de nouvelles classes filles de *Tache*.

B) Améliorations

Concernant la partie back-end, la classe *Programmation* aurait pu avoir deux classes filles, une pour les évènements, une pour les tâches unitaires, afin de pouvoir programmer d'autres types d'objets.

Concernant la partie interface graphique, on pourrait rajouter pas mal de restrictions pour que l'utilisateur de rentre que des données cohérentes par exemple, le guider davantage, lui permettre de charger différents types de fichiers. Nous pourrions rajouter des boutons qui font des liens entre les sous-fenêtres pour naviguer plus facilement. L'emploi du temps pourrait représenter dans le temps les programmations par rapport à l'heure, mais il aurait fallu faire des sections d'une minute par cases du tableau, c'est pour cela que cette solution n'a pas été privilégiée. Cependant, il serait possible de mettre les tâches qui durent sur plusieurs jours sur plusieurs cases correspondant à chaque jour. Nous avons délibérément laissé la possibilité de déprogrammer des tâches même si la date est passée dans le cas où l'utilisateur n'a pas effectué la tâche et souhaite donc la reprogrammer, sachant que la programmation n'est possible que dans le futur.

Conclusion

LO21 fut une UV formatrice, fondamentale et très intéressante. Ce projet fut long, complet. Mais il nous a permis de mettre en application ce que l'on avait appris, de nous confronter à la gestion back-end et front-end d'un projet, d'apprendre à se servir des bibliothèques Qt, de gérer efficacement le stockage des données avec les bibliothèques STL et leurs dérivées. Bien que plusieurs points d'améliorations aient été soulignés, nous sommes satisfaits du résultat, de notre réflexion et du travail fourni. Nous avons fait un effort sur la complexité et la non-redondance des données, ce qui était parfois contradictoire et demandait de privilégier l'un sur l'autre. Nous avons beaucoup réécrit l'architecture pour nous adapter au fur et à mesure, opter pour de meilleures solutions, nous nous sommes investis sérieusement et avons essayé de faire des choix toujours justifiés, et nous avons pris du plaisir à voir le projet prendre forme jusqu'à la version d'aujourd'hui.

Consignes

- Le projet est disponible sous le nom de *CalendarProject*, il suffit de lancer le .pro pour l'ouvrir dans Qt Creator.
- Le fichier calendrier.xml est là à titre d'exemple pour importer un fichier.
- Toute la documentation Doxygen est accessible à partir du répertoire html.