LO21 - Programmation et conception orientée objet

Université de Technologie de Compiègne

Rapport de projet - application ProjectCalendar

Auteurs :
Zineb Slam
Marie Martin

Responsables :
Antoine Jouglet
Taha Arbaoui

14 juin 2015

Table des matières

1 Introduction

Dans le cadre de l'UV LO21, nous avons développé une application possédant les fonctionnalités d'un agenda et celles d'un outil de gestion de projet. Cette application a été développée en C++ avec le framework Qt. Plusieurs fonctionnalités sont indispensables pour ce projet :

- Affichage de l'emploi du temps
  - ⋆ Hebdomadaire
  - ⋆ Avec les événements programmés
- Gestion de l'ensemble des projets
  - ⋆ Créer un projet, créer des taches
  - ⋆ Ordonnancer des taches unitaires
  - ⋆ Visualisation des dépendances (tree-view)
- Ajout d'événements à l'agenda
- Export ...
  - ⋆ De l'agenda de la semaine
  - ⋆ De la programmation d'un projet
  - ⋆ En texte brut et en XML

Nous avons donc pensé notre architecture en fonction des différentes fonctionnalités à intégrer dans le projet.

2 Description de l'architecture

2.1 Contraintes

La programmation de taches implique de nombreuses contraintes sur les différents objets. Voici les contraintes principales :

- Une tache ne peut être associée qu'à un seul projet.
- Cas d'une tache unitaire préemptée : on peut commencer la tache, l'interrompre et la reprendre plus tard et cela autant de fois que l'on veut. La durée d'une tache non-préemptée doit être inférieure à 12h.
- On ne peut pas commencer une tache si l'ensemble de ses taches précédentes ne sont pas terminées.
- L'échéance d'une projet doit être supérieure au maximum des échéances des taches qui le composent.

2.2 Modélisation

Nous avons des classes correspondant à des objets.

2.2.1 Tache

Une tache possède un identifiant (id), un titre, une durée, une date de disponibilité et une date d'échéance. Une tache peut être unitaire ou composite. Comme ces deux types de taches ont un comportement et une interface différente, nous avons créé deux taches dérivées TacheU et TacheC (voir ci-dessous). Il y a donc une relation d'héritage entre Tache et TacheU et TacheC.
Comme une tache ne doit pas être dupliquée (nous voulons éviter les doublons) et que nous voulons contrôler sa création et sa modification, nous avons créé une classe TacheManager qui sera responsable de la création, de la modification et de la suppression d'une tache. Il nous faut donc mettre dans la partie privée le constructeur de recopie et l'opérateur d'affectation, et dans la partie protégée le constructeur, le destructeur et les fonctions de modifications. Néanmoins, nous autorisons l'utilisateur a avoir l'accès en lecture aux attributs membres, donc nous définissons plusieurs fonctions de lecture dans la partie publique.

2.2.2 TacheU

Cette classe modélise une tache unitaire. Les taches unitaires ont la caractéristique de pouvoir être programmée et de pouvoir être préemptive (ce qui signifie qu'on peut commencer la tache, l'arrêter puis la reprendre plus tard). Nous ajoutons donc l'attribut booléen preemptive et l'attribut entier non-signé progression (car nous voulons savoir à quel point nous sommes avancé dans la tache, ce qui est utile à savoir lorsqu'elle est préemptive).
De plus, il existe des contraintes de précédence : une tache unitaire peut être précédée d'une ou de plusieurs taches. Nous définissons donc deux vecteurs de pointeurs de taches unitaires (un pour les taches précédentes, un pour les taches suivantes).

```
vector<TacheU*> precedence;
vector<TacheU*> suivante;
```

2.2.3 TacheC

Cette classe modélise une tache composite. Une tache composite est une tache qui est composée de plusieurs autres taches, qui peuvent être elles-mêmes composites ou unitaires. Nous avons donc une relation d'aggrégation entre TacheC et Tache, ce qui se traduit par un vecteur de pointeurs de Tache.

2.2.4 TacheManager

Cette tache permet de créer, de supprimer et de modifier des taches. Il y a donc une relation de composition entre TacheManager et Tache. La classe TacheManager a donc pour attribut un vecteur de pointeurs de Tache.
De plus, il faut que TacheManager soit une instance unique afin d'empêcher la création de taches par des plusieurs objets (il y aurait un risque d'incohérence et de doublons). Nous utilisons alors le design pattern Singleton qui permet d'assurer l'unicité de l'objet. C'est

dans cette classe que sont déclarées les fonctions d'ajout de taches, comme ajouterTacheU par exemple, ainsi que des fonctions d'accès aux taches existantes et d'affichage.

2.2.5 Event

La classe Event est une classe abstraite (elle contient des fonctions membres virtuelles pures) qui représent un événement. Un objet de la classe Event est soit une tache unitaire, soit une activité (d'où un lien d'héritage entre Event et ses classes filles TacheU et Activite). Un événement peut être programmé, nous avons donc ajouté un attribut booléen programmee qui prend la valeur true si une programmation a été liée à l'événement.

2.2.6 Activite

Une activité est un événement particulier. Elle possède un id, un titre, une durée et un lieu.

2.2.7 Rdv

Un rendez-vous est une activité particulière. Elle hérite de la classe Activite, et donc de ses propriétés. Elle a un attribut supplémentaire par rapport à sa classe mère : l'attribut personne, qui désigne les personnes participant au rendez-vous.

2.2.8 Reunionn

une réunion est une activité particulière. Elle hérite de la classe Activite, et donc de ses propriétés.

2.2.9 Programmation

Cette classe permet de programmer des événements. Il y a une association entre la classe Programmation et la classe Event, qui se traduit par la présence d'un pointeur d'Event comme attribut de la classe Programmation.
En plus, il est nécessaire pour une programmation d'avoir un attribut date et un attribut horaire afin de placer l'événement dans le calendrier. Comme pour une tache, il ne faut pas avoir de doublons. Chaque programmation est associée à un et un seul événement. Les fonctions membres de modifications et les constructeurs sont donc en privé, et nous créons des programmations avec une instance unique de ProgrammationManager (voir ci-dessous).

2.2.10 ProgrammationManager

Cette classe est responsable de la création, destruction et modification des programmations. Il y a une relation de composition entre Programmation et ProgrammationManager car cette dernière est responsable du cycle de vie de Programmation. Cela se traduit par un vecteur de pointeurs de Programmation comme attribut de la classe. Nous lui appliquons le

design pattern Singleton afin de s'assurer de l'unicité de l'instance.
Comme fonctions membres, il y a les fonctions pour ajouter des programmations, ainsi que des fonctions statiques pour récupérer et libérer l'instance unique.

2.2.11 Projet

Un projet est un ensemble de taches. Il y a donc un vecteur de pointeurs de Tache dans un objet Projet. Toutefois, il n'est pas responsable de la création d'une tache, ni de son ajout dans son vecteur (c'est le rôle de TacheManager).
Un projet a un id, un titre, une date de disponibilité, une date d'échéance et un indicateur de sa complétude (un booléen qui permet de savoir si le projet est terminé ou non). La création, suppression et modification d'un projet est également contrôlée par un manager, ProjetManager. Toutefois, plusieurs fonctions d'accès en lecture aux attributs sont disponibles.

2.2.12 ProjetManager

ProjetManager fonctionne de façon identique à TacheManager et ProgrammationManager. C'est un singleton qui est responsable du cycle de vie des projets qui le composent. La relation de composition qui existe entre Projet et ProjetManager se traduit par un vecteur de pointeurs de Projet.

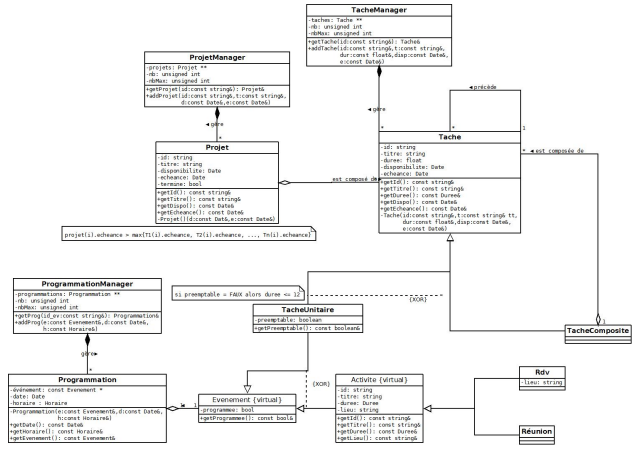

FIGURE 1 – UML de l'architecture

2.3 Interface

Nous avons divisé notre interface en deux parties : une partie affichage de l'agenda et une partie gestion des projets. Pour son implémentation, nous avons utilisé la bibliothèque de widgets QtWidgets. La partie affichage est celle toujours visible à l'écran (nous avons notre emploi du temps qui s'affiche dans la fenêtre principale). La partie gestion de projets est "cachée" : elle s'exprime à travers les différentes actions que l'utilisateur peut effectuer :

rechercher un projet, supprimer une tache, ajouter une tache à un projet, afficher le treeview... Dès la conception de notre application, nos buts principaux étaient d'avoir une interface ou l'utilisateur pourra avoir toutes les information sur ses projets , taches et évènements . Ainsi en plus de l'emploi du temps hebdomadaire , l'utilisateur dispose d'un Menu Arborescence avec un TreeView des projets avec les taches de chaque projets, un TreeView des taches avec les sous taches des taches composite, et enfin un affichage de toutes les Programmations avec la durée qui reste à programmer pour les taches préemptives . L'utilisateur pourra soit garder ses évènements après les avoir complétés ou les supprimer, de plus il sera propre maître de ses « objets » , il pourra à n'importe quel moment les ajouter puis les supprimer sans se soucier des éventuels ambiguïté , par exemple en supprimant une tache le programme se chargera de supprimer la taches de tous les projets et des programmations aussi.
L'utilisateur dispose aussi d'un menu de recherche qui lui fournir toutes les informations, par exemple il saura pour une tache unitaires les taches les taches qui la précèdent et celles qui suivent, et pour évènements le programme affichera la date et l'horaire mais aussi la progression des taches préemptives. Toujours dans l'affichage on s'est beaucoup concentré dans l'affichage de l'emploi du temps. Ce dernier présente plusieurs utilités. La première est que l'utilisateur n'aura pas à se soucier à placer son événement , en effet dès que l'utilisateur programmera un événement le programme le placera automatiquement au bon endroit dans l'emploi du temps. De plus l'un des points forts de notre agenda c'est qu'il pourra être utilisé à vie, en effet notre agenda gère les dates allant de l'année 2015 à 3000 , et grâce au numéro de semaine le programmera placera toujours l'évènement au bon endroit à n'importe quel date. Enfin , nous avons essayé de rendre notre agenda plus esthétique en mettant une couleur pour les taches et une couleur pour les activités pour que l'utilisateur se repère mieux .

2.3.1 Partie agenda

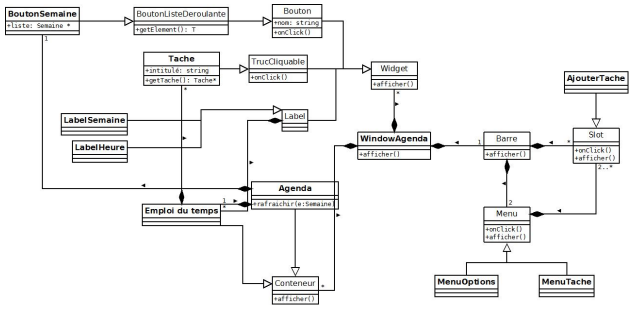

FIGURE 2 – UML de l'interface, partie agenda

2.4 Livrables

Nous avons organisé notre projet en plusieurs fichiers. Chaque fichier correspond à une classe ou un groupe de classes. Voici la liste des fichiers :

— timing.h et timing.cpp : classes Duree, Horaire et Date → gestion des dates (jour/-mois/années), des horaires(heure/minute) et des durées (nombre de minutes)
— calendar.h et calendar.cpp : classe CalendarException (reprise de TD) → gestion d'exception
— tache.h et tache.cpp : classe mère Tache, classes dérivées TacheU et TacheC et classe TacheManager → gestion de taches unitaires ou composites, gestion de la création, destruction, modification par un manager (singleton)
— evenement.h et evenement.cpp : classe mère Event, classe dérivée Activite, classes Rdv et Réunion (dérivée de Activite) → gestion des événements, gestion de différents types d'activités
— programmation.h et programmation.cpp : classe Programmation et ProgrammationManager → programmation des événements, création, suppression et gestion des programmations par un manager (singleton)
— customscene.h et customscene.cpp : classe CustomQGraphicsScene → classe personnalisée héritant de QGraphicsScene
— agendawindow.h et agendawindow.cpp : classe AgendaWindow, classes ItemActivite et ItemTache dérivées de QGraphicsItem → création de la fenêtre principale, implémentation des slots et création de QGraphicsItem personnalisés
— window.h et window.cpp : classes NewProgrammation, NewTask, newActivity → widgets personnalisés
— projet.h et projet.cpp : classes Projet et ProjetManager → gestion des projets

3 Caractère évolutif de l'architecture

Grâce à la classe abstraite Event, nous pouvons facilement ajouter de nouveaux types d'événements (autre que Tache ou Activite).
De plus, le fait qu'Activite soit elle aussi une classe abstraite permet d'ajouter de nouveaux types d'activité facilement. Il est également possible d'ajouter des spécificités à chaque type d'activité en ajoutant des attributs dans les classes qui héritent de Activite. C'est une des principales raisons pour laquelle nous avons créé des classes filles plutôt qu'un type énuméré dans la classe Activite pour représenter les différentes activités existantes.
Le design pattern Composite utilisé pour les taches composites permet une évolution facile des taches.
La classe Tache et ses relations avec ses classes filles permettent d'ajouter facilement de nouveaux types de taches par simple héritage, en créant une nouvelle classe.
Ensuite, nous avons été attentives à créer de nombreuses fonctions virtuelles dans les classes mères, afin de les redéfinir dans les classes filles pour adapter leur comportement à la classe désirée. Cela permet de faire évoluer l'application sans avoir à effectuer trop de changements dans les fonctions de toutes les classes.
De plus même en rajoutant un nouveau type de tache on aura pas à changer tout le code car seule Tache Manager gère les taches donc il n y'aura pas de conflits après. De plus si on décide de gérer nos taches ou nos programmations différemment avec des listes chainées

par exemple grâce à la composition on n'aura qu'à changer la classe Manager sans changer l'implémentation de nos objets. En plus grâce au polymorphisme la majorité code n'aura pas à connaître quel classe derivée est utilisée, et donc la selection de quel class à instancier ou quelle méthode sera fait un point précis dans le code. Le code sera donc plus facile à gérer. Enfin même en changeant les méthodes de la classe de base on aura toujours de bon comportement en appelant une méthode qui sera le comportement de la classe appropriée.

Dans le même sens, ajouter un nouveau type de tache ou d'activité ne changera en rien comment les programmations seront gérés.

4 Améliorations possibles

Il serait envisageable d'ajouter de nouvelles classes héritant d'Activite, afin d'enrichir le modèle par de nouveaux types d'activités.