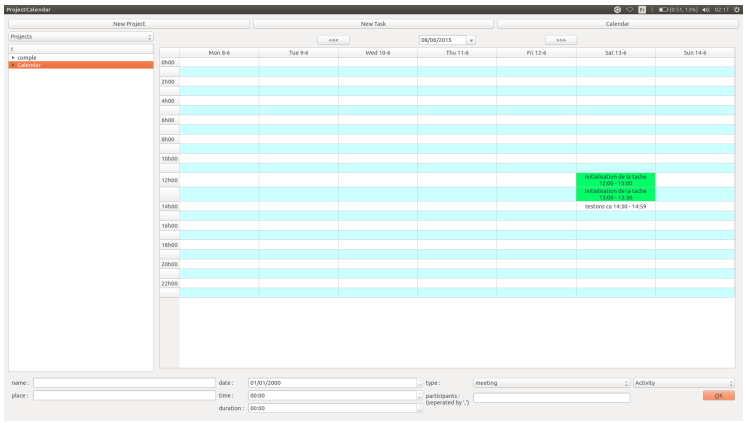
LO21
Project Calendar

Charmat Othman
Baaziz Mohamed-Lamine

15 juin 2015

1 Introduction

L'objectif de ce Projet était de réaliser une application qui lie les fonctionnalités d'un gestionnaire de projets et d'un agenda électronique, permettant ainsi de générer des projets, des tâches, et de programmer une réalisation de tâche ou un évement quelconque à une date donnée. Le langage de programmation utilisé est le c++ (+Qt). Nous allons dans une première partie présenter l'architecture globale de l'application, puis dans un second temps, démontrer l'adaptabilité du code source.



2 Architecture de l'application

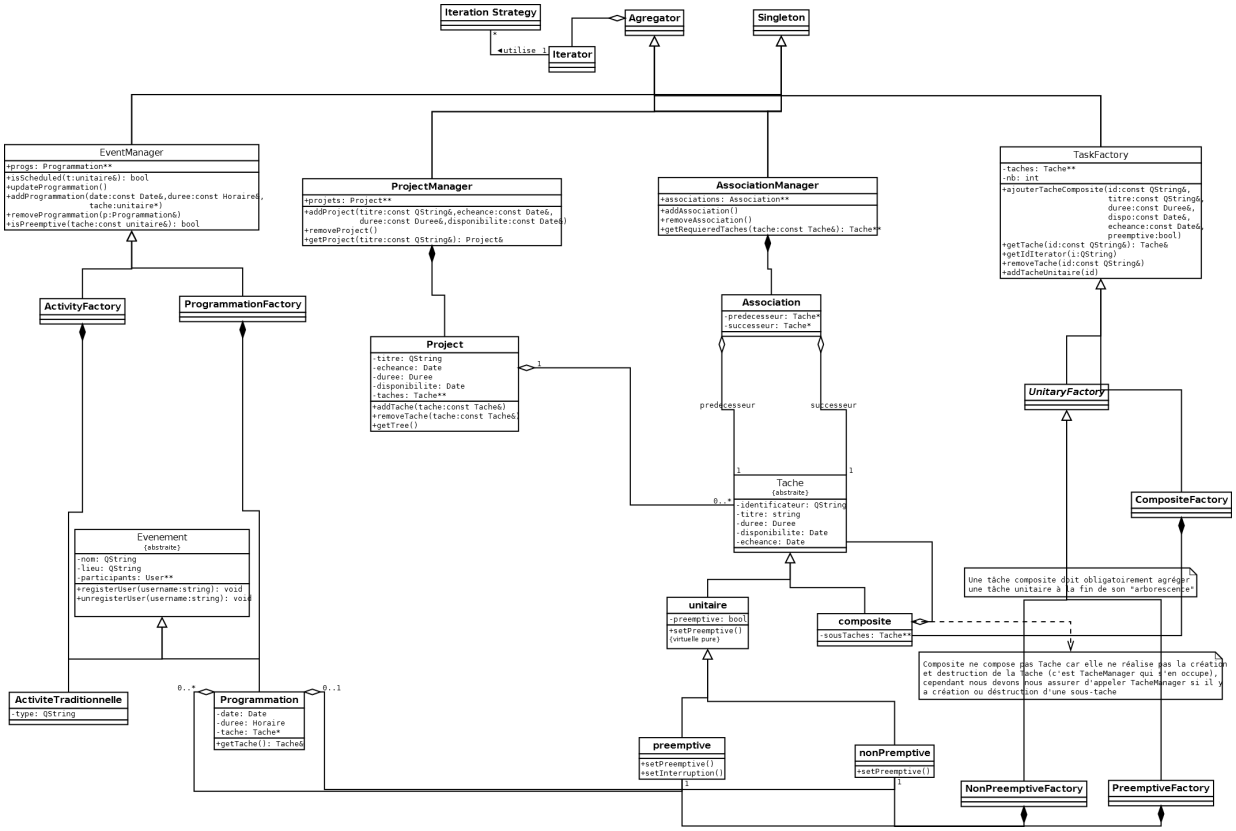


FIGURE 1 – Diagramme UML non exhaustif, à visée communicative

2.1 back-end

Les principaux élements manipulés par l'applications sont les :
- Tâches (composites, préemptives et non préemptives)
- Projets
- Evenements (Activités et Programmations et de tâches)
- Associations (contraintes de précédences et de succession)

Ce sont des objets qui doivent pouvoir être manipulables depuis plusieurs endroit, et cela de manière controllée. Nous avons alors fait le choix d'utiliser des Singletons qui auront pour fonction de gérer ces objets (TaskFactories, ProjectFactory, AsssociationManager ...). Ces "'Managers"' sont par escence des agrégateurs de leurs objets. De ce fait, il bénificient de fonctionnalités communes aux agrégateurs, tel que la possibilité d'utiliser des itérateurs (les tâches composites sont aussi des agrégateurs).

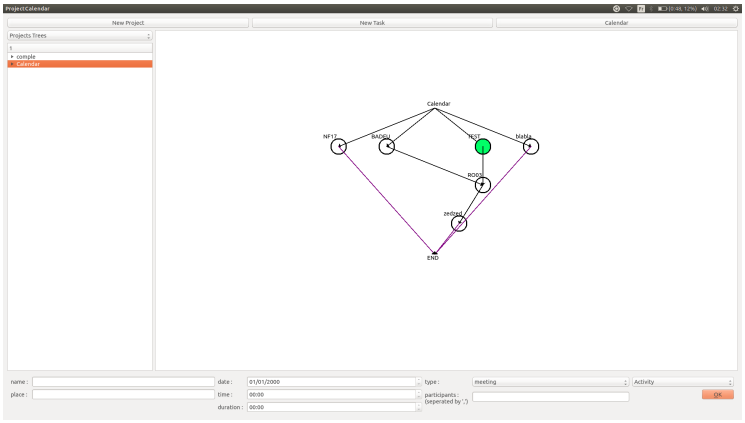
La gestion des plusieurs types de tâches est très similaire. Cependant la fabrication de tâches doit s'adapter au type de tâche voulu. Cette configuration nous a poussé à utiliser le design Pattern Abstract Factory pour implémenter les différentes "'Usines à Tâches"'. Chaque usine produisant sont type de tâche associé (exemple : PreemptiveTaskFactory produit des tâches préemptives ...). Nous avons utiliser la même stratégie pour la gestion des évenements (qui se décline en deux avec les usines à Programmations et les usines à Activités).

Pour stocker les tâches, projets, évenements et associations, nous avons choisi d'utiliser les listes chainées implémentées par la STL (std : :list) afin de favoriser les insertions et suppressions (ce qui semble fréquent dans une telle application).

2.2 front-end

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.

Dans la partie affichage de l'agenda, nous avons un tableau affichant les jours de la semaine et les heure, nous avons utilisé la bibliothèque QTableview ainsi que QCalendar pour afficher un calendrier adapté à notre tableau. Dans cette même partie, nous avons la possibilité d'afficher un arbre représentant le cycle d'évolution du projet (cela en faisant un clique droit dans la partie gestion de l'agenda, a gauche) sur le projet que l'on souhaite afficher.



Dans la partie gestion de l'agenda, nous avons des boutons avec trois différentes fonctions : New Project, New Task et Calendar. En cliquant sur New Project, une nouvelle fenêtre s'affiche nous permettant de remplir les différentes informations par rapport au nouveau projet qu'on souhaite ajouter, sur cette même fenêtre nous avons la possibilité de créer des taches. En cliquant sur New Task nous obtenons une fenêtre nous permettant de remplir les informations sur la tache ainsi que le type de la tache(composite, preemptive ou non-preemptive) ses successeurs et ses prédécesseurs. Le boutton Calendar nous permet de passer de l'affichage de l'arbre de projet à celui du calendrier.

Un clique droit sur le nom du projet ou de la tache nous donne plusieurs options. Pour la gestion d'un projet on peut : Ajouter un projet, afficher les informations, afficher l'arbre et supprimer le projet. Pour la gestion d'une tache : on peut planifier une programmation, modifier les contraintes, afficher les informations, supprimer la tache et ajouter une sous-tache pour les taches composites.
Un simple menu en haut de la fenêtre permet d'importer un fichier XML ou d'exporter le contenu courant dans un fichier XML.

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

3 Adaptabilité

3.1 Singleton généralisé (cf. Singleton.h)

Nous avons pu à travers un système de classes patrons généraliser la caractéristique Singleton. En effet, pour qu'une classe devienne un singleton, il suffit de la faire hériter de la classe Singleton, et de faire ces 3 déclarations :

- friend class Singleton<classeCliente> ;
- friend class Handler<classeCliente> ;
- template<>
  Handler<classeCliente> Singleton<classeCliente> : :handler = Handler<classeCliente>() ;

Ainsi, cette stratégie facilite grandement l'implementation des singletons (surtout dans une telle application, où les "'usines singleton"' sont nombreuses) et factorise son implémentation. Par conséquent, si une modification dans la manière d'implementer le singleton venait à changer, elle ne serait faite qu'à un seul endroit.

3.2 Agrégateur généralisé (cf. Iterator.h)

De même que pour le singleton, nous avons pu généralisé la propriété d'agrégation. Ainsi, toutes classes agrégatrices d'élements peut sous classer la classe Aggragator, et ainsi bénéficier d'iterateurs, et d'une méthode isItemeHere() qui permet de savoir si un élement est aggrégé ou non. Le principal avantage est la factorisation de l'iterateur ; cependant, cette stratégie nous permet de pouvoir ajouter des fonctionnalités propres aux aggrégateurs très simplement et qui seront accéssibles à toutes classes agrégatrices. Pour qu'une classe devienne un agrégateur, il suffit de faire hériter la classe avec Aggregator, tout en initialisant la partie Aggregator avec l'adresse du conteneur d'objets de la classe Cliente. Par ailleur, il est possible d'appeler la méthode getIterator d'une classe agrégatrice en précisant en paramètre template le type d'objets que nous voulons (ex : getIterator<CompositeTask>() n'iterera que les tâches composites). Il faut bien entendu que la classe spécific soit une classe dérivée de la classe de base agrégée.

3.3 Stratégies d'Iterations (cf. Iterator.h)

Nous avons utiliser le design pattern Strategy pour personnaliser la manière d'itérer sur les objets agrégés d'une classe agrégatrice. Il suffit de définir une classe qui hérite de IterationStrategy et qui définit la méthode "'test"'. Ensuite, il suffit de donner l'adresse de la stratégie en paramètre de la methode getIterator() de la classe agrégatrice pour n'itérer que sur les objets qui vérifient la condition test. Il est de plus possible de combiner stratégie d'itération avec précision d'un type d'item spécific lors de l'appel à la méthode getIterator() (ex : getIterator<specificClasse>(stratégie) , on n'itérera que sur les objects spécifics, vérifiant la strétégie donnée). Cette implémentation nous permet ainsi de personnaliser les itérateurs donnés à toutes classes agrégatrices, sans devoir redéfinir d'itérateurs.

3.4 classes Patrons

Lorsque les formes d'une classe générale (ex : tâches) peuvent évoluer, nous avons préféré utiliser des classes patrons pour la usines qui les gère (ex : TaskFactory). Ainsi, si nous décidons d'ajouter d'autres formes de cette classe (ex : ajouter un autre type de tâche), alors les méthodes pour controller cette nouvelle forme sera "'déjà"' implémentée dans l'usine patron, via l'utilisation de paramètres templates en s'assurant que la nouvelle forme hérite l'usine avec les bons paramètres template (ex : la façon dont CompositeFactory, PreemptiveFactory réutilise TaskFactory).

4 Conclusion

Ce projet nous a permis de mettre en œuvre les enseignements que nous avons pu suivre tout au long du semestre dans l'UV LO21 et fut donc très bénéfique. Ce projet représente une expérience remarquable dans la programmation orientée objet, ainsi que dans l'élaboration d'une intérface homme-machine. Cette application peut encore être améliorée en ajoutant de nouvelles fonctionnalités comme le calcul des tâches critiques, des dates de début au plus tôt des tâches etc ... Il est aussi envisageable d'ajouter de nouvelles classes héritant de la classe Activity , afin d'enrichir le modèle par de nouveaux types d'activités, ou plus simplement en ajoutant des "'ActivityTypes"' dans l'énumeration ActiityType et son équivalent en QString dans activityTypeTable.