Thomas PELLETIER

Paul BUZAUD

Amélie THERY

Rapport du projet

“Project calendar”

Introduction :

Dans le cadre de l’UV Lo21, nous avons été amené à élaborer la conception et implémenter un calendrier dynamique codé en c++ avec une interface graphique en qt. Nous avions un certain nombre de contraintes à respecter et avons dû adapter en conséquence le développement de notre application.

Nous verrons donc dans ce rapport quel choix d’architecture nous avons fait, et nous expliquerons aussi en quoi celle-ci permet des évolutions.

Description de l’architecture :

Le but de ce projet était de permettre à un utilisateur de gérer à la fois ses projets en cours et son emploi du temps . Il devait pouvoir programmer ses activités à venir facilement ainsi que gérer la programmation de ses projets. Il était alors nécessaire de traiter ces 2 parties d’applications de sorte qu’elles soient liées par un élément commun. Pour ce faire, nous sommes partis du principe qu’un événement était une classe abstraite commune aux activités traditionnelles et aux taches. C’est donc par ce biais que nous avons relié ces 2 parties. Il était alors plus aisé de gérer les accès concurrentiels à la programmation ainsi que l’unicité de chaque programmation d’événement.

Nous avons modélisé notre architecture sur deux axes. Le premier, gestion des projets, des taches qui le composent, le second gestion de la programmation, c’est à dire de l’emploi du temps et des évènements.

Nous avons par ailleurs décidé de créer 2 singletons permettant de bien séparer la gestion des taches et des programmations soit : ProjetManager qui regroupe et gère tous les projets et ProgrammationManager qui regroupe et gère toutes les programmations.

Ces deux classes sont crées tel quel car il est impératif pour notre implémentation qu’une seule instance de chacune d’entre elle n’existe, sans quoi deux projets pourraient avoir le même nom ou deux programmations pourraient être faites au même moment.

ProjetManager est en réalité une composition d’un ensemble de projet. Il permet donc d’accéder individuellement à chaque projet et d’éviter d’avoir des doublons en fonction de leur titre. C’est donc cette classe qui fait appel aux constructeurs et aux destructeurs de projet. Chaque projet étant composé d’un ensemble de taches, ProjetManager permet aussi , via chaque projet, d’accéder efficacement à toutes les tâches que peut ajouter l’utilisateur à tout moment dans le code.

Les projets gèrent leurs propres tâches via leur identifiant interne au projet. Ce choix a été fait afin de permettre à l’utilisateur d’ajouter des tâches avec le même nom mais qui ne sont pas dans le même projet. En effet, on retrouve souvent les mêmes étapes lors de la réalisation d’un projet comme la modélisation, la conception, l’implémentation et il fallait permettre à l’utilisateur de pouvoir réutiliser ces mêmes étapes à condition que ce soit dans différents projets. Dans un second temps, cela permettait une importation xml des projets plus aisée qu’avec une id globale.

Concernant la classe tâche, nous avons décidé de l’implémenter abstraite pour différencier les types de taches, tout en conservant les mêmes propriétés (une date de disponibilité et une date d’échéance). Sachant qu’ une tâche est soit unitaire soit composite, et que ces classes n’ont pas les mêmes attributs ni les mêmes fonctions que les tâches composites, cette implémentation était nécessaire.

Les taches composites regroupent quant à elles un ensemble de tâches qui peuvent être de type unitaire ou encore composite. Elles peuvent aussi exister indépendemment des autres tâches dans un projet mais ne peuvent pas être programmées, puisque ce sont ses sous-taches qui le sont. Il nous a alors semblé judicieux de les faire correspondre à des agrégats de taches et non des compositions.

Comme la seule différence entre une tâche unitaire préemptée et une non préemptée est que l’une peut-être programmée en plusieurs fois et que l’autre non, nous avons décidé de rajouter l’attribut durée effectuée à la classe tache préemptée qui hérite d’une tache unitaire ( donc non préemptée). Ainsi dès que l’on programme une tâche préemptée, on ajoute la durée programmée à la durée effectuée et on vérifie que celle-là ne dépasse la valeur de la durée (attribut de la classe mère : tache unitaire). Ces 2 types de tâches ainsi que tache composite hérite bien sûr de la classe tâche héritant elle-même de la classe événement.

ProgrammationManager gère donc les programmations, de sorte qu’une programmation ne soit pas au même moment qu’une autre. Une programmation est en fait une date et un horaire lié à un évènement. Autrement dit cela permet de définir quand à lieu notre évènement dans notre emploi du temps. Un évènement est une tache, un rendez-vous ou une réunion. Evenement est donc une classe abstraite. C’est ici que se lient donc nos deux axes, et que nos taches déjà créées sont programmées.

Les classes Rendez-vous et réunion ne dépendent pas de ProjetManager car d’une part elles n’entrent pas dans le cadre d’un projet, ce sont des évènements à part, et de plus il peut y en avoir plusieurs de même titre avec les mêmes interlocuteurs. La seule contrainte les concernant est la programmation. Ces deux types d’évènements sont donc créées en même temps que leur programmation, ils ne peuvent pas être créés sans et c’est ProgrammationManager qui s’occupe d’appeler leurs destructeurs.

Vous trouverez ci après notre UML.

Pourquoi notre architecture permet-elle des évolutions?

Le choix de notre architecture en deux axes nous permet facilement des évolutions. En effet, si nous voulons rajouter un type de taches ou d'évènement ce ne serait pas un problème puisque nos classes héritent de classes mère abstraites, contenant toutes les méthodes nécessaire au fonctionnement de notre calendrier. Il faudra bien sur dans certains cas surcharger certaines fonctions si la nouvelle classe contient des spécificités. Le design pattern Singleton protège la base de notre implémentation : L’unicité des taches dans un même projet et l’unicité d’une programmation dans une même plage horaire. Ajouter des classes filles ne devrait donc pas mettre en danger ces concepts.

Nous pouvons ensuite envisager la gestion de plusieurs emplois du temps, correspondant à plusieurs utilisateurs par exemple. Dans ce cas précis, il faudrait ajouter une classe ProgrammationManagerPartagé et une classe ProjetManagerPartagé qui permettrait de partager un meme projet et de le programmer sur tous les emplois du temps par exemple.

Concernant l’importation xml, le fait que chaque projet possède sa propre numération de tâche permet une plus grande liberté au niveau de leur importation; En effet, il n’est pas nécessaire d’importer à chaque fois tout un emploi du temps avec les projets et leurs programmation mais simplement des fichiers xml de projet. Cela permet de pouvoir changer rapidement ses projets et de les échanger avec d’autres utilisateurs. C’est aussi un moyen d’être plus adaptable par rapport aux calendriers en ligne.

**Conclusion :**

Ce projet nous a été très enrichissant car il nous a permis d’avoir une première approche concrète de la programmation objet. Par son biais, nous avons pu nous familiariser avec un ensemble de design pattern tel que : le singleton, la factory ou encore les itérators … et de développer notre propre interface graphique. Nous nous sommes rapidement aperçu que notre première conception UML n’était pas complète et nous avons dû revenir sur certain choix de conception en fonction d’interprétations erronées ou encore de difficultés d’implémentation. Cela nous a donc appris à nous adapter selon les situations pour trouver les solutions les plus judicieuses possibles. Etant novice en c++ au début du semestre, ce projet nous a aussi poussé à être autonomes, créatifs et à acquérir une base solide en langage orienté objet. Ces qualités nous semblent par ailleurs primordiales dans la formation d’ingénieur informaticien.