Université de Technologie de Compiègne

LO21

Rapport de projet

ProjectCalendar

Printemps 2015

Romain PELLERIN - Lila SINTES
Groupe du mardi après-midi
15 juin 2015

Table des matières

1 Introduction

1.1 Préambule : compilation

Pour compiler le projet, vous devez avoir installé le framework Qt (nous avons testé sur la version 5.4.1) et vous devez avoir un compilateur C++11 (nous avons testé sur GCC version 4.6.1). Vous pouvez ensuite importer le projet dedans ou compiler dans un terminal sur Linux, comme ceci :

Shell

```
QMAKE=‘find / -path "*Qt/*gcc*/bin/qmake" -print0 2>/dev/null‘ # Peut être un
    peu long
mkdir -p build && cd build/
$QMAKE ../ProjectCalendar/ProjectCalendar.pro -r -spec linux-g++ CONFIG+=debug
make clean && make
```

1.2 Modélisation

Après avoir étudié le sujet, nous avons commencé le projet par une étape essentielle de modélisation. Nous avons représenté toutes les classes « métiers » dans un schéma UML. Nous avons pour cela réutilisé des *Design Patterns*, tels que **Singleton** pour les managers (qui sont en fait des *wrappers* de `std::vector`). ou **Composite** (une `TacheComposite` contient d'autres `Tache`, qui peuvent être de plusieurs types : composite, unitaire ou unitaire et préemptable).

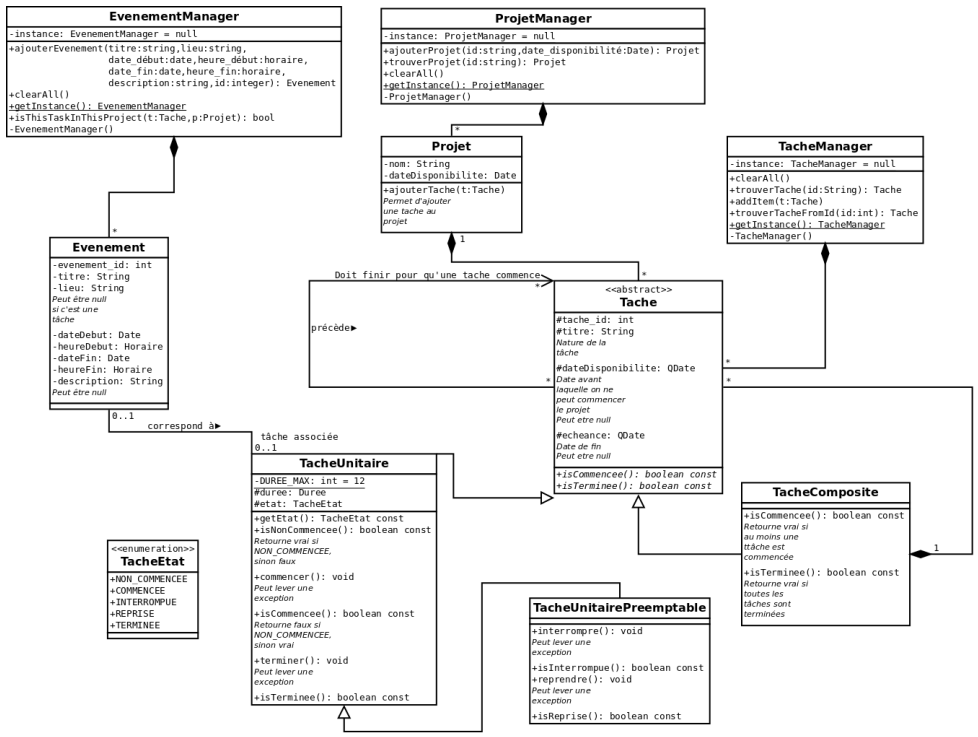

FIGURE 1.1 – UML des classes métiers

1.3 Choix techniques

1.3.1 Qt ou STL

Le *framework* Qt offre de nombreuses classes qui sont des améliorations des classes standards du C++, comme par exemple `QString`, qui n'est finalement qu'un `sdt::string` amélioré. Ainsi, pour pouvoir plus facilement manipuler les interfaces graphiques, nous avons décidé de n'utiliser que des classes propres à Qt lorsque cela était possible. En voici une liste non exhaustive :

— `QString`

— `QTime`

— `QDate`

— `QDebug` (pour afficher sur la sortie standard)

— ...

Seul `std::vector` a été utilisé à la place de `QVector`, sans raison particulière.

1.3.2 Sauvegarde de l'application

Pour sauvgarder les programmations, projets et tâches de notre application, nous avions le choix entre le XML, le JSON et une base de données SQLite. **Notre choix s'est porté sur SQLite**. En effet, utiliser un SGBD [1] offre de nombreux avantages :

— Possibilité d'assurer une forte cohérence des données grâce à des contraintes (*check*)

— Possibilité d'empêcher l'utilisateur de modifier la base de donnée dans un éditeur de texte (cela reste possible en ligne de commande ou avec un logiciel, mais beaucoup moins simple)

— Pas de *parsing* à faire comme avec XML et JSON. *Parser* un XML est relativement compliqué à mettre en place (bien qu'il existe des classes pour manipuler le XML), tandis que SQLite est très bien intégré dans Qt, avec des classes pour les requêtes, la gestion d'erreur, etc.

1.3.3 Interface graphique

Pour créer l'interface graphique, nous avions le choix entre tout coder « à la main » ou utiliser Qt Designer. Après avoir commencé à tout faire à la main, nous nous sommes rendus que la moindre modification ou l'ajout d'un élément graphique entraînait la nécessité de modifier beaucoup de code. **Nous avons donc finalement décidé d'utiliser Qt Designer**, qui permet de créer des interfaces graphiques facilement et rapidement.

Par la suite, nous avons modélisé au papier nos interfaces graphiques, pour nous mettre d'accord et pouvoir par la suite implémenter chacun de notre côté l'interface.

1. Système de Gestion de Base de Données

2 Implémentation

2.1 Classes métiers

Ayant un UML à disposition, nous avons par la suite commencé le projet en créant les classes métiers, qui sont en fait les implémentations des entités UML. Par rapport à l'UML, nous avons ajouté les constructeurs, les *getters* et *setters*, les attributs implicites sur l'UML (par exemple, un `std::vector` pour les compositions), etc. Voici un extrait de la classe `Projet` :

projet.h

```
class Projet {
    QString nom;
    QDate dateDisponibilite;
    vector<Tache*> taches;

public:
    Projet(QString m_nom, QDate m_dispo): nom(m_nom), dateDisponibilite(m_dispo)
        {}

    void ajouterTache(Tache *t);

    /**
     * @brief getEcheance permet de connaitre l'échéance du projet
     * @return la date d'échéance la plus éloignée, de l'ensemble des tâches du
         projet
     */
    QDate getEcheance() const;

    // Getters
    QString getNom() const {return nom;}
    QDate getDispo() const {return dateDisponibilite;}

    // Setters
    void setNom(const QString& m_nom) {nom = m_nom;}
    void setDispo(const QDate& m_dateDispo) {dateDisponibilite = m_dateDispo;}

    ~Projet();
};
```

Nous avons choisi d'utiliser **trois singletons** pour les tâches, les projets et les événements (programmations) de l'agenda. Ces trois types d'entités seront respectivement contenu dans des `TacheManage`, `ProjetManager` et `EvenementManager`. Cela permet d'accéder aux même objets, depuis n'importe quelle classe de l'application.

2.2 Classes des interfaces graphiques

Lors de la création des interfaces graphiques avec Qt, les classes minimales sont générées (constructeur et destructeur). Nous avons donc rajouté des méthodes pour effectuer certaines

actions, comme par exemple :

— Des signaux et slots pour réagir à des clics sur des éléments graphiques

— Des méthodes qui rafraîchissent l'ensemble de l'interface graphique

2.3 Fonction `main()`

Notre fonction `main()` joue un rôle très important dans l'application. Voici dans l'ordre ce qu'elle fait :

1. **Instanciation des trois singletons.**
2. **Création (si non présent) du fichier de la base SQLite dans un sous-répertoire propre à notre application, dans le répertoire de l'utilisateur.**
3. **Connexion à la base et vérification que les tables nécessaires existent, sinon la table est « re-initiliasée ».** Pour pouvoir faire évoluer l'application et la base facilement, il conviendrait d'avoir une table qui contient le numéro de version de l'application. Ainsi, lors des mises à jour de l'application, il serait aisé de savoir s'il faut recréer la base de données ou non.
4. **Chargement dans les trois singletons des projets, tâches et événements contenus en base de données.**
5. **Affichage de la fenêtre graphique.**

Les différentes exceptions qui peuvent survenir sont gérées par des blocs `try/catch` avec affichage des messages d'erreurs à l'utilisateur via une *pop-up*. Par exemple :

main.cpp

```
try {
    em->setDatabase(&sdb); // em = EvenementManager::getInstance()
    em->loadEvents();
} catch (EvenementManagerException& e) {
    fenetre.showError("Project Calendar", e.getInfo());
    app.exit(0);
    return 0;
}
```

mainwindow/h

```
void showError(QString titre, QString description) {
    QMessageBox::warning(this, titre, description);
}
```

3 Fonctionnalités

Voici le rendu visuel de l'application, composé de deux contenus principaux, accessible via des onglets :

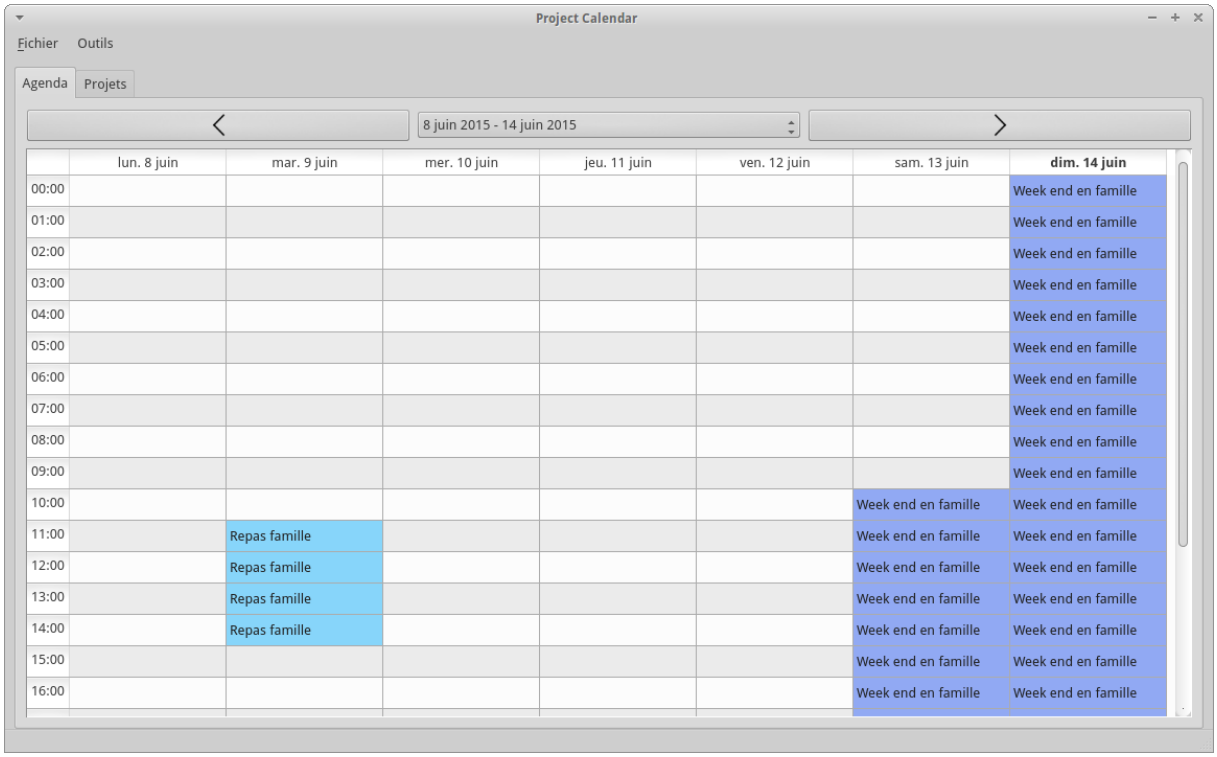

FIGURE 3.1 – Partie de l'application dédiée à l'agenda

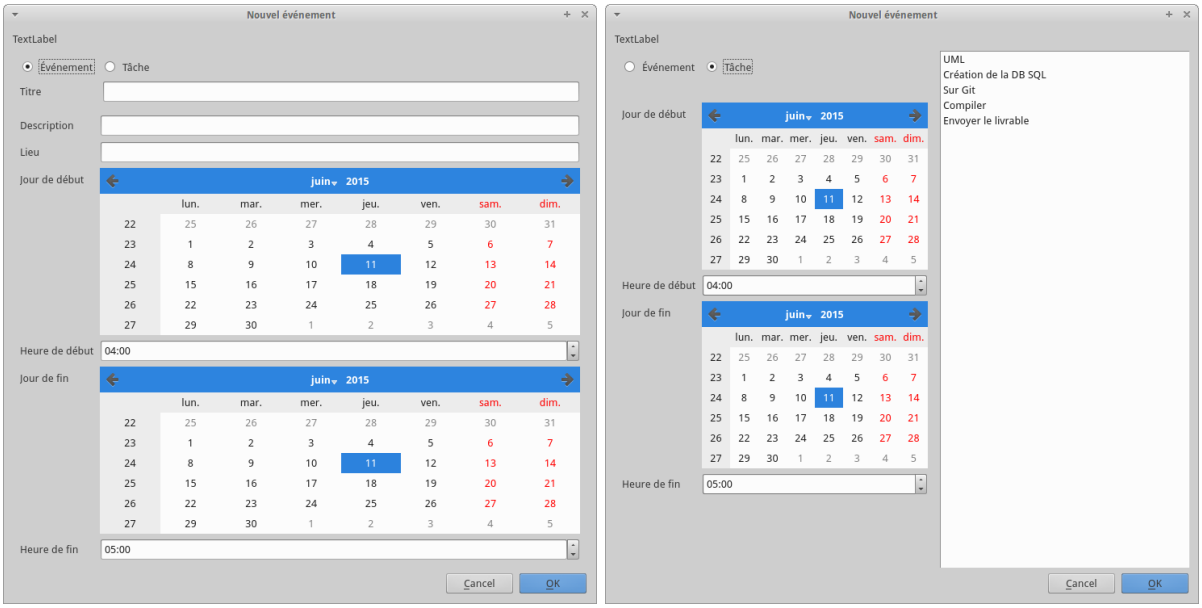

FIGURE 3.2 – Création d'un nouvel événement

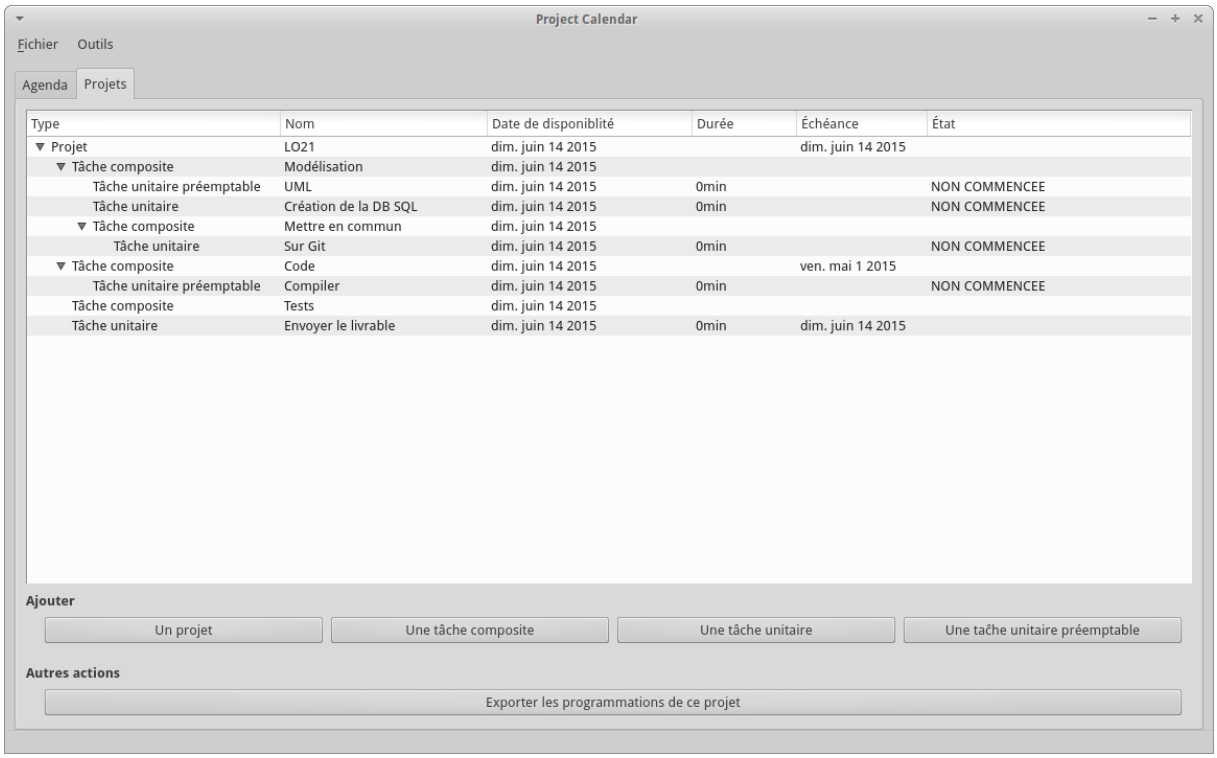

FIGURE 3.3 – Partie de l'application dédiée aux projets et tâches

Nous avons implémenté les fonctionnalités suivantes.

3.1 Choix de la semaine à afficher

Par défaut, la semaine courante est affichée. Le jour courant est en gras. L'utilisateur peut changer la semaine affichée à l'aide d'une liste déroulante (au milieu de l'écran) ou à l'aide des boutons « Suivant » et « Précédent ».

L'agenda affiche les événements programmée pour la semaine affichée.

3.2 Création

3.2.1 D'un événement (une programmation)

Un clic sur un case de l'agenda permet de programmer un événement (les champs dates et heures sont pré-remplis en fonction de la case cliquée). Des vérifications sont faites (par exemple, un événement doit finir après sa date et heure de début)

3.2.2 D'un projet

Via l'onglet « Projets », l'utilisateur peut créer un nouveau projet.

3.2.3 D'une tache composite, unitaire ou unitaire et préemptable

De la même façon qu'un projet, l'utilisateur peut créer les trois type de tâches. Néanmoins, il doit au préalable avoir sélectionné le projet ou la tâche composite qui va accueillir la nouvelle tâche. Les tâches composites ne peuvent être contenus que dans des projets ou des tâches composites. Les éléments de « *top-level* » sont donc les projets.

3.3 Modification

Les attributs des projets et tâches peuvent être modifiés (la durée d'une tâche unitaire par exemple).

3.4 Export XML

3.4.1 De la semaine actuellement affichée

Il est possible, via le menu, d'exporter l'ensemble des programmations de la semaine couramment affichée dans un fichier XML.

3.4.2 Des programmations relatives aux tâches d'un projet donné

On peut également exporter uniquement les événements (tâches programmées) d'un projet selectionné, via un bouton en bas, sur l'interface des projets.

3.5 Suppression

Il est possible via le menu de supprimer soit l'ensemble des événements, soit l'ensemble des tâches et projets. La suppressions d'une tâche entraîne la suppression de sa programmation, s'il y en a une.

4 Code

4.1 Design Pattern

Nous avons utilisé les *patterns* suivants dans notre projet :

— Iterator (sur des `std::vector`)
— Singleton (pour `Tache`, `Projet` et `Evenement`)
— Composite (une tâche composite contient d'autre tâches, composites ou unitaires)
— Factory (seul `TacheManager` peut créer des `Tache`, les constructeurs sont privés)
— Adapter (autour de `std::vector`)

4.2 Évolutivité

Nous n'avons pas trouvé l'usage des *templates*, étant donné que nous utilisons la classe abstraite `Tache`, qui contient des méthodes virtuelles pures, et dont toutes les tâches héritent. En cela, l'application peut être difficilement maintenable sur certains aspects.

Nous aurions pu utiliser des *templates* pour une classe-mère `Manager<T>`, dont nos trois managers auraient hérités. Cette classe auraient contenus les méthodes abstraites `load()` et `saveToDB()` (que nous utilisons déjà dans nos trois managers). Le fait d'être un Singleton aurait alors été géré par cette classe-mère.

En revanche, nous avons bien segmenté nos classes et notre code de manière générale, regroupant d'un côté les classes métiers, de l'autres les classes applicatives (gérant l'interface utilisateur par exemple).

5 Conclusion

De manière générale, le projet était très long mais nous a permis de bien découvrir et assimiler les notions apprises pendant le semestre. Qt est un framework avec lequel il est très agréable de développer, offrant un réel plus par rapport à la STL. Le seul bémol de ce projet est sa complexité : en effet, d'autres UV ont des projets et celui de LO21 demandait beaucoup de temps, au risque de négliger les autres projets.