Université de Nantes
UFR des Sciences et Techniques
Master ALMA 1

TP Génie Logiciel - MyCalendar

Guillaume Charon
Jerôme Pages

Nantes, le 5 décembre 2012

Sommaire

Introduction

Ce document a pour objectif de présenter la conception de l'application MyCalendar. Cette application, écrite en C++ avec le framework Qt4, présente une interface graphique correspondant à un emploi du temps semaine par semaine modifiable localement. L'utilisateur peut s'il le souhaite créer, modifier, ou supprimer des évènements.

Autour de ces fonctionnalités basiques s'attachent deux autres fonctionnalités permettant à l'utilisateur d'intéragir avec certains services en ligne pour importer ou exporter des évènements depuis Internet vers l'application locale et vice-versa.

Petit rappel des spécifications des exigences logicielles à partir desquelles la présente conception est basée :

- Architecture modulaire permettant une maintenance facile de l'application ;
- Interface graphique (GUI [1]) *user-friendly* par semaine ;
- Respect de la charte graphique GNU/Linux ;
- Édition locale de l'emploi du temps (ajout, modification, suppression) par clics intuitifs sur la GUI ;
- Gestion des conflits (que faire lorsque l'utilisateur essaie d'ajouter un évènement sur un créneau non libre ?) ;
- Sauvegarde locale des évènements ;
- Exporter l'emploi du temps vers un service en ligne ;
- Récupérer l'emploi du temps depuis ce même service en ligne ;
- Protection de l'emploi du temps distant avec des identifiants de connexion ;
- Utilisation du protocole REST pour la communication avec ce service en ligne ;
- Communication non bloquante et temps de réponse de l'ordre de la seconde ;

Une autre exigence est concernée par la conception de l'application puisque nous avons eu le temps de l'implémenter : la récupération d'un emploi du temps universitaire depuis le site de gestion d'emploi du temps de l'Université de Nantes. Cependant, aucun autre format d'emploi du temps n'a été supporté, cette exigence ayant été elle aussi reportée.

1. Graphical User Interface

1 Conception générale

Afin de répondre à la première exigence, c'est-à-dire concevoir l'application à partir d'une architecture modulaire, nous avons utilisé le pattern architectural MVC (pour Model-View-Controller). Cette architecture nous permet de diviser l'application en plusieurs sous-systèmes :

1. L'interface graphique utilisateur ;
2. Le modèle, contenant tous les évènements et proposant des méthodes pour agir dessus ;
3. Les interfaces de communication, pour interagir avec différents services d'emploi du temps en ligne ;

Le contrôleur joue quant à lui le rôle d'arbitre entre tous ces systèmes.
Exemples :

La vue demande la sauvegarde des évènements dans un fichier local : le contrôleur reçoit le signal correspondant et traite la demande.

La vue demande l'export des évènements vers un service en ligne : le contrôleur reçoit ce signal, vérifie la connexion à ce service (est-ce que l'utilisateur est authentifié ?) et appelle la bonne interface de communication.

Un conflit est détecté lors de l'ajout d'un nouvel évènement : le contrôleur traite ce conflit, soit en demandant à l'utilisateur que faire, soit en appliquant un comportement par défaut par exemple.

Ci-dessus, un diagramme de classe généraliste présentant ce système dans sa globalité. La conception de chaque sous-système est présentée dans les prochains chapitres.

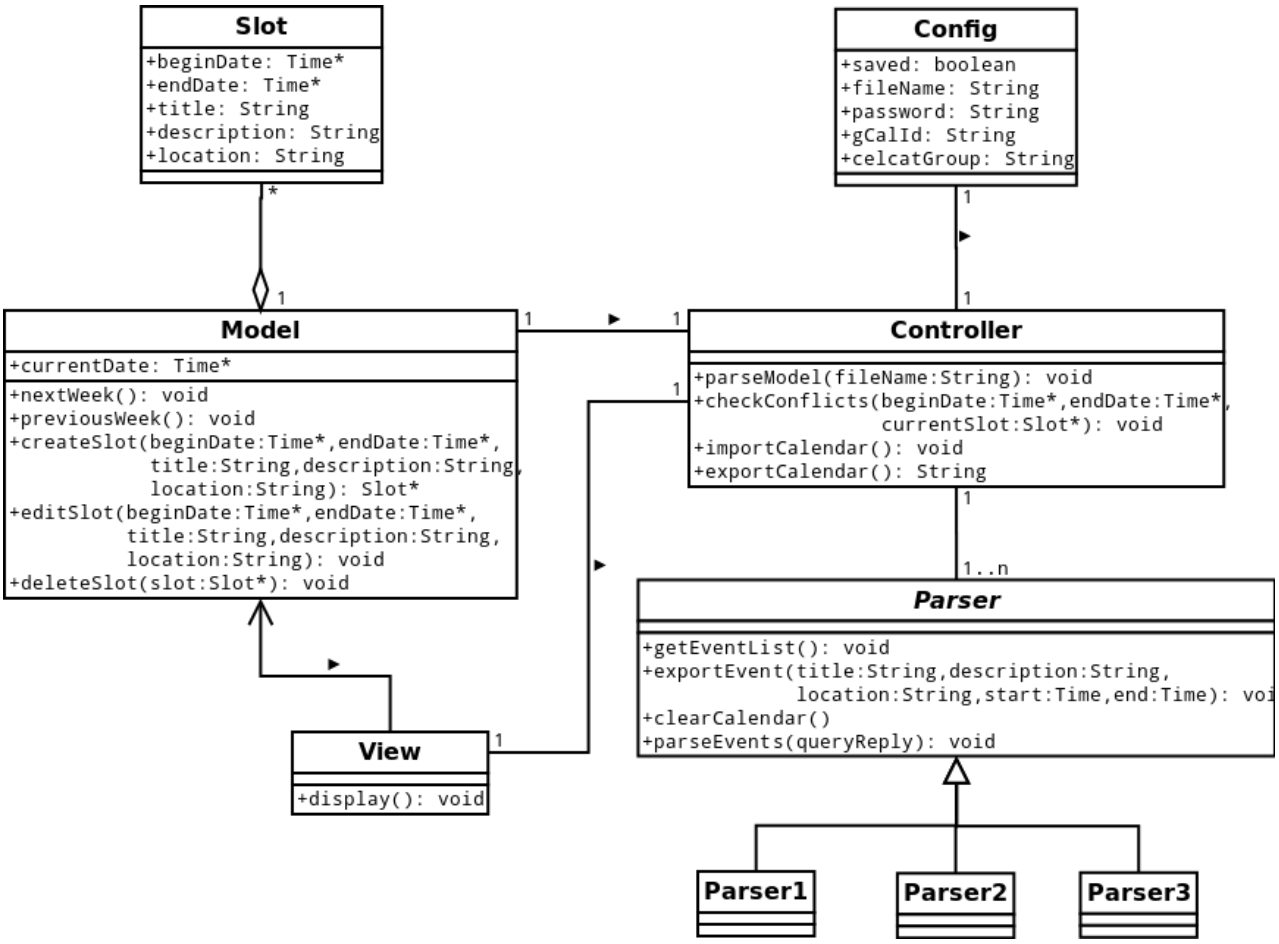

FIGURE 1.1 – Diagramme de classe général

2 Conception du modèle

Le modèle contient toutes les données de l'application, aussi bien les objets de l'emploi du temps manipulés utilisés par les différentes classes que la configuration du logiciel. Il correspond à la première partie du patron architectural MVC.

Le modèle est composé des classes suivantes :

2.1 La classe Model

La classe modèle est la classe principale de ce sous-système. Elle possède plusieurs types d'attributs :

- Time* currentDate : Un pointeur vers un objet de type Time qui contient la date courante.
- Set<Slot*> slotlist : Un ensemble de pointeurs de slot contenu dans un Set afin de trier les objets. De cette manière, les opérations de base (ajout, suppression, lecture et mise à jour) sont disponibles facilement, en parcourant la liste via un itérateur. Pour cela, il sera convenu de définir une méthode de comparaison des créneaux qui se basera sur leur date de début pour effectuer ce tri.

Voici la liste des méthode de cette classe :

nextWeek

Méthode : void nextWeek()

- Objectif : Mettre l'attribut currentDate à la date du premier jour de la semaine suivante.
- Pré-condition : /
- Post-condition : La nouvelle date correspond à sept jours après l'ancienne.

nextWeek

Méthode : void nextWeek() ;

- Objectif : Mettre l'attribut currentDate à la date du premier jour de la semaine précédente.
- Pré-condition : /
- Post-condition : La nouvelle date correspond à sept jours avant l'ancienne.

createSlot

Méthode : Slot* createSlot(Time *beginDate, Time *endDate, string title, string description, string location)

- Objectif : Création de l'objet Slot correspondant aux données en entrée et ajout dans l'ensemble de créneaux.
- Pré-condition : Les données entrées ont été vérifiées (dates biens formées, dates de début et de fin cohérentes et pas de conflits avec d'autres créneaux).

– Post-condition : Le créneau est créé et a été ajouté dans l'ensemble slotlist.

deleteSlot

Méthode : void (Slot deleteSlot *slot)

– Objectif : Suppression dans l'ensemble du créneau concerné et destruction de l'objet.
– Pré-condition : Le créneau existe et est dans la collection slotlist.
– Post-condition : L'objet n'est plus référencé dans l'ensemble et a été détruit. Il n'est plus accessible via son pointeur.

cleanList

Méthode : void cleanList()

– Objectif : Destruction de tous les créneaux de l'ensemble slotlist.
– Pré-condition : /
– Post-condition : Aucun créneau n'est accessible via les pointeurs et l'ensemble de créneau est vide.

2.2 La classe Slot

La classe Slot est la classe qui contient les données d'un créneau qui contenu dans le modèle. Elle contient toutes les informations qui les définissent : une plage horaire qui explicite les heures de début et de fin ainsi que le titre, la description et le lieu.

– Time* beginDate : Heure de début de l'événement.
– Time* endDate : Heure de fin de l'événement.
– String title : Le titre du créneau sous forme de chaine de caractère (string).
– String description : Une chaine de caractère qui contient une courte description du créneau.
– String location : L'information sur le lieu de l'événement dans une string.

Pour comparer les différents créneaux, il est nécessaire que les opérateurs de comparaison « == » et « != ». Dans le cas de l'égalité, tous les attributs devront être identiques pour renvoyer la valeur booléenne « vrai » tandis que dans l'opération d'inégalité, il suffit d'une seule différence pour renvoyer « vrai ». De cette manière, tous les traitements sur les créneaux se feront de la manière la plus claire et rapide possible pour ne pas alourdir le code.

Les attributs étants tous modifiables par l'utilisateur, il est nécessaire de prévoir des accesseurs. Ceux-ci seront aussi bien des *getters* que des *setters*, qui premettent respectivement d'accéder à la valeur et de la modifier.

Les autres méthodes de cette classe vont être décris maintenant.

areSlotsOverlapping

Méthode : bool areSlotsOverlapping(Time* beginDate, Time* endDate)

– Objectif : Vérifier si deux créneaux sont en conflit pour une même plage horaire.
– Pré-condition : /
– Post-condition : Retourne la valeur booléenne « vrai » si les créneaux ont au moins une minute en commun et faux sinon.

editSlot

Méthode : void editSlot(Time *beginDate, Time *endDate, string intitule, string description)

- Objectif : Modifier toutes les valeurs d'un créneau à la fois (date et chaines de caractère).
- Pré-condition : Les valeurs d'entrée sont au bon format.
- Post-condition : Le créneau possède maintenant les nouvelles valeurs.

toString

Méthode : string toString()

- Objectif :Retourner une chaine de caractère contenant toutes les informations sur le créneau dans une forme lisible par un humain.
- Pré-condition : /
- Post-condition : La chaine retournée indique la date de début, de fin et l'intitulé du créneau afin de facilement le distinguer des autres.

3 Conception des interfaces de communication

Ce sous-système de l'application MyCalendar constitue, derrière le modèle local, son centre névralgique. En effet, c'est toute cette partie de l'application qui va gérer les communications avec l'extérieur, que ce soit pour récupérer un calendrier distant ou pour exporter en ligne les données locales.

Il existe deux types de communications : les communications entrantes (import d'un calendrier vers les données locales) et les communications sortantes (export des données locales vers un calendrier distant).

Premièrement, l'application est sensée, à terme, pouvoir gérer plusieurs types de services en ligne. Cela signifie qu'il faut non seulement permettre aux développeur de pouvoir ajouter la prise en charge d'un nouveau service facilement, mais aussi proposer aux utilisateurs une interface lui proposant de choisir lui-même quel service il souhaite utiliser pour l'import comme pour l'export.

Deuxièmement, un service de calendrier propose l'import de données en même temps que l'export de données. Cela ne sera pas vrai pour un emploi du temps universitaire, qui est sensé ne pas pouvoir être modifiable par un client quelconque, mais cela le sera normalement pour tout autre service distant.

Partant de ces deux conclusions, nous avons décidé de partir sur la conception d'une classe abstraite *Parser* déclarant toutes les méthodes permettant l'import et l'export de données. Pour implémenter un nouveau service distant, il faut ensuite étendre cette classe abstraite et de définir les méthodes qu'il faut. Enfin, pour l'utiliser, il suffira de modifier le controller (que nous avons vu précédemment) pour instancier ce parseur et utiliser les méthodes que l'ont veut.

Dans l'application *MyCalendar*, seulement deux parseurs ont été définis.

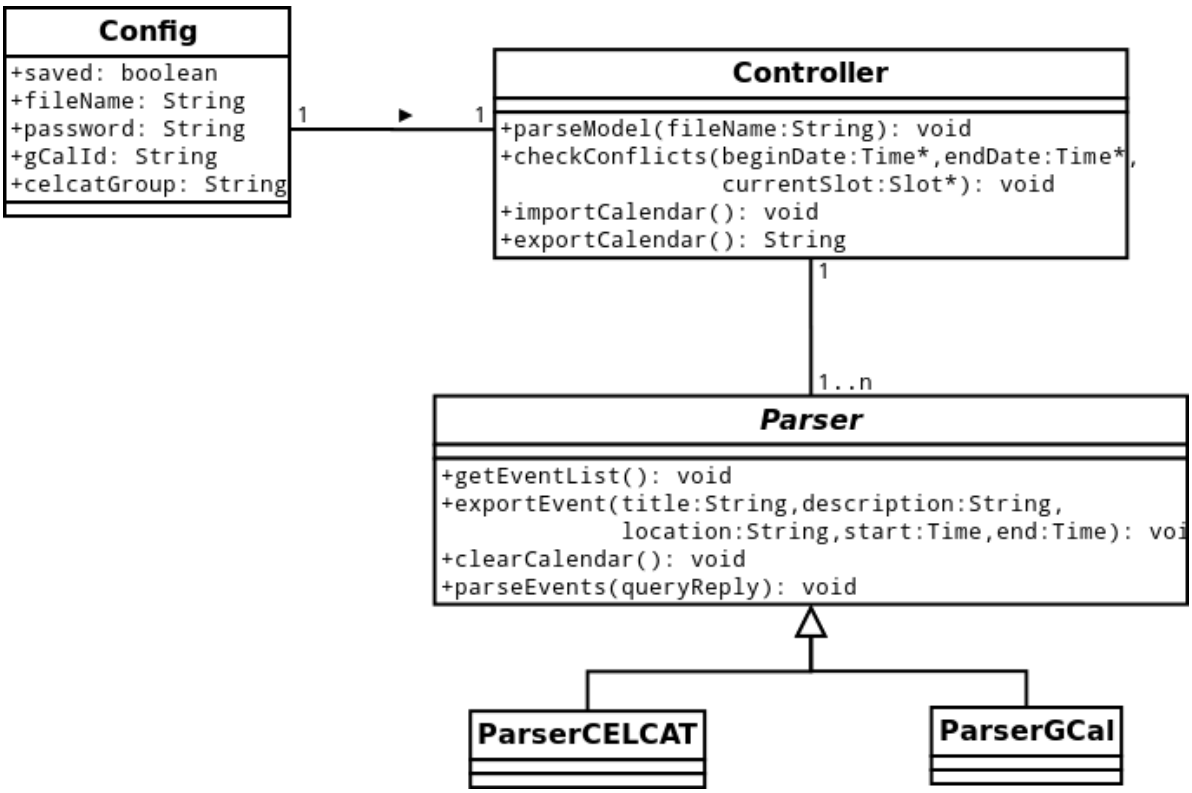

FIGURE 3.1 – Diagramme de classe des interfaces de communication

3.1 Spécification des classes

3.1.1 Classe Parser

getEventList()

- Objectif : Envoyer une requête vers le service distant pour récupérer tous les évènements
- Pré-condition : Le calendrier distant (identifiant, url, etc) a été défini
- Post-condition : Une requête GET ou POST a été envoyée

exportEvent(String title, String description, String location, Time start, Time end)

- Objectif : Envoyer une requête vers le service distant pour créer un nouvel évènement avec les données passées en argument
- Pré-condition : Le calendrier distant (identifiant, url, etc) a été défini
- Post-condition : Une requête GET ou POST a été envoyée

clearCalendar()

- Objectif : Envoyer une requête vers le service distant pour supprimer tous les évènements qui y sont liés
- Pré-condition : Le calendrier distant (identifiant, url, etc) a été défini
- Post-condition : Une requête GET ou POST a été envoyée

parseEvents()

- Objectif : Transformer les évènements du calendrier pour les inclure vers le modèle local

– Pré-condition : Une requête *getEventList()* a été envoyée et la réponse a été reçue
– Post-condition : Des évènements ont été ajoutés au modèle local

3.2 Diagrammes de séquence

3.2.1 Récupération des évènements distants

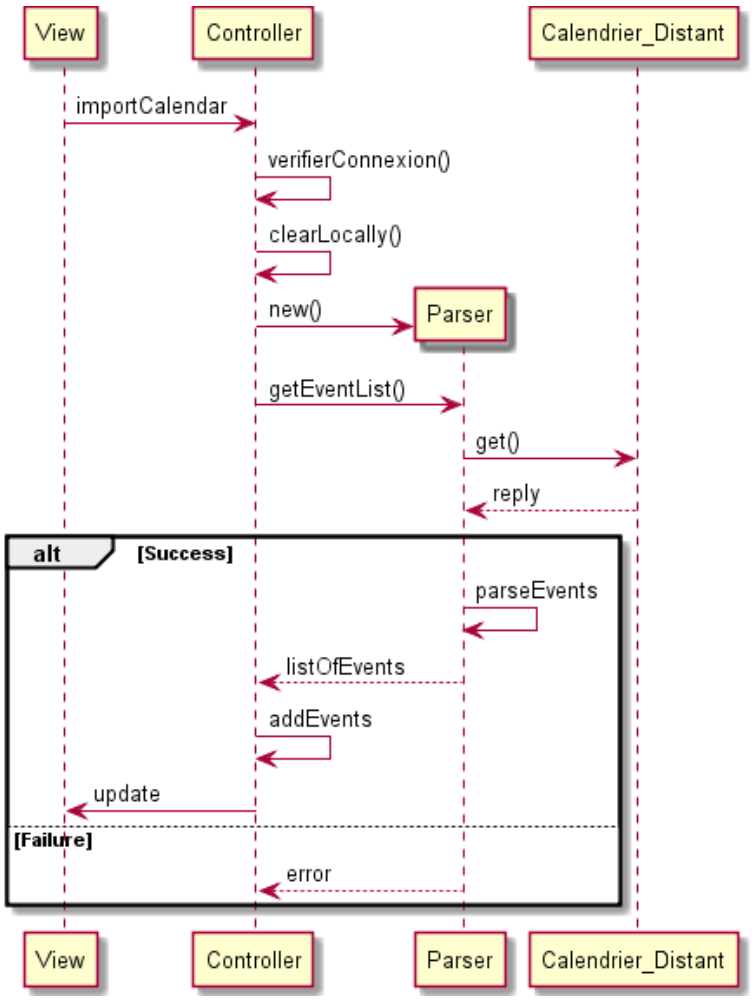

FIGURE 3.2 – Récupération des évènements distants

3.2.2 Exportation des évènements locaux

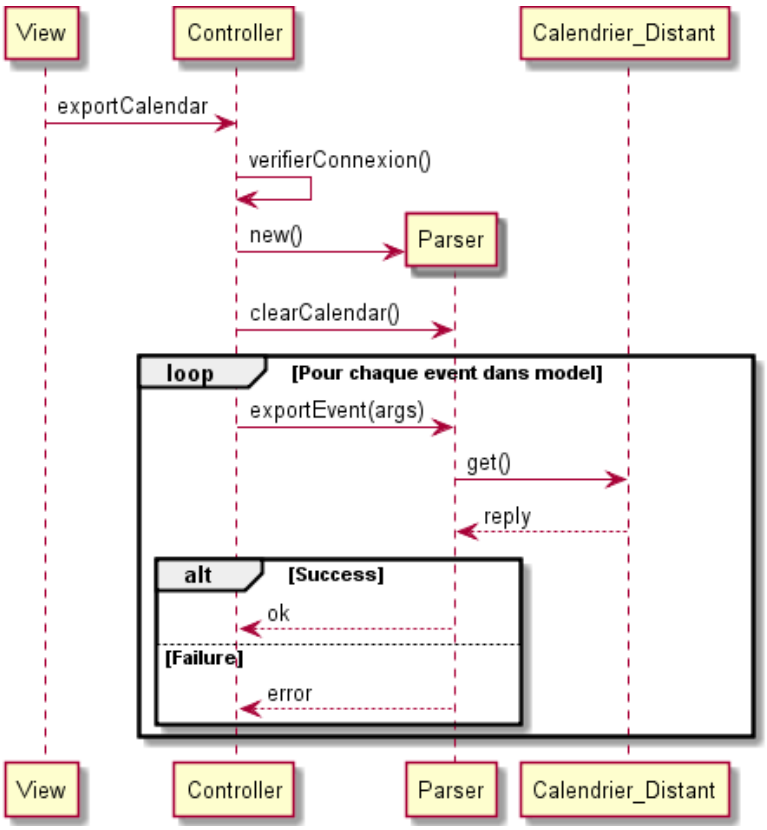

FIGURE 3.3 – Exportation des évènements locaux

4 Conception de l'interface utilisateur

5 Controller

juste dire qu'il utilise la classe Config qui lui permet juste de stocker les informations dont il peut avoir besoin pour certaines de ses actions