GOMES Killian, ESPASA Kevin groupe 501A/502C

Projet : Lecteur Multimédia

Année 2015-2016 *Objet et développement d'applications*

Table des matières

Introduction :

Nous avons choisi de réaliser un lecteur multimédia pour ce projet d'objet et de développement d'applications, qui porte sur l'implémentation et l'utilisation de 3 designs patterns vu en cours. Il est réalisé dans le langage C++. Dans ce rapport nous expliquerons comment utiliser le lecteur, ainsi que nos choix de designs patterns et leur fonctionnement.

Compilation et éxécution :

Pour compiler le programme, il faut d'abord éxécuter un petit script "modifVariable.sh", qui permet d'indiquer le chemin des librairies lors de l'éxécution du programme. *Sans cela, il y a soit des conflits entre les librairies installées a la sources, et les librairies installées localement qui sont parfois les mêmes, soit il ne trouve pas les librairies.* Nous aurions pu faire éxécuter ce script dans le make, mais cela voudrait dire qu'il serait éxécuté à chaque compilation *(et donc un processus est lancé à chaque fois)*, ce qui est mauvais d'un point de vue mémoire.
On peut aussi préférer éxécuter la commande « export LD_LIBRARY_PATH=`pwd`/lib » directement sur le terminal au lieu d'utiliser le script.
Après l'éxécution du make puis du script *(cela fonctionne dans l'ordre inverse)*, il faut lancer l'éxécutable en oubliant pas de spécifier qu'il se trouve dans le dossier bin.

Récapitulatif :

```
killian@KILLIAN-K70IO:~/Faculté/Semestre5/Projet_LecteurMultimedia$ make
killian@KILLIAN-K70IO:~/Faculté/Semestre5/Projet_LecteurMultimedia$ ./modifVariable.sh
```

OU

```
killian@KILLIAN-K70IO:~/Faculté/Semestre5/Projet_LecteurMultimedia$ export LD_LIBRARY_PATH=`pwd`/lib
killian@KILLIAN-K70IO:~/Faculté/Semestre5/Projet_LecteurMultimedia$ ./bin/lecteur
```

Utilisation du programme

- ## Panel de début :

Lorsque l'utilisateur exécute le programme, un panel de choix lui est proposé. Il peut choisir de lire soit une vidéo, soit un audio, soit une image.
Nous détaillerons donc l'utilisation de chaque média.

- Vidéo :

Si l'utilisateur a choisi de lire une vidéo, le programme lance le lecteur vidéo. Il propose à l'utilisateur de choisir la vidéo qu'il veut lancer parmi celles présentes dans le dossier Video (*Ressources/Video)*. Une fois lancée, l'utilisateur peut exécuter des actions sur la vidéo, à l'aide des boutons fournis. L'application propose a l'utilisateur de mettre la vidéo en lecture, en pause, ou en arrêt. Cette gestion d'états est gérée par le design pattern "State" que nous expliquerons par la suite. Nous permettons aussi a l'utilisateur de lire un fichier de sous-titres *(format .txt)* en parallèle a la vidéo *(ce fichier doit avoir le même nom que celui de la vidéo)* . La gestion des sous-titres est elle effectuée par le design pattern "Observer".
Nous voulions en premier lieu utiliser la librairie VLC pour gérer les vidéos, mais cette librairie avait trop de dépendances, et vu que toutes les librairies devaient être installées localement, nous avons opté pour une librairie plus accessible, telle que sfeMovie (une surcouche de SFML). Le problème avec cette librairie, c'est que les formats de vidéos que l'on peut gérer sont peu nombreux, et peu communs.

Format accepté : ogg.

Une mise à jour de la librairie SfeMovie (la dernière version requière SFML 2.3) permettrait d'augmenter le nombre de formats acceptés (WebM, MPEG4).

- Audio :

L'audio est similaire à la vidéo. Elle possède quasiment la même interface, et les mêmes actions peuvent être exécutées. La différence se trouve seulement dans le format et dans le fait qu'on ne puisse pas lire de sous-titres. *Les flux audio sont géré par la librairie SFML.*
Extension possible de l'application : ajouter une barre de défilement sur l'audio et la vidéo.

- Image :

Si l'utilisateur a choisi de lire une image, le programme propose l'ensemble des images présentes dans le dossier Image. Une fois l'image ouverte, l'application permet a l'utilisateur de changer d'image (soit l'image suivante, soit l'image précédente dans le dossier). *Nous avons pensé a implémenter un bouton Zoom mais la date butoire approchant nous nous sommes dit que le rapport temps/valeur apportée a l'application n'etait pas assez intéressant.*

Les librairies utilisées :

Les trois classes des médias sont composées d'une interface graphique, créée avec la librairie TGui, une autre surchouche de SFML.
Donc pour récapituler, nous utilisons principalement la librairie SFML, et deux de ses surcouches, TGui et sfeMovie, en plus de chacune de leur dépendances.
Si nous tenions a mentionner les librairies dans notre rapport, c'est parce que les installer localement a été notre plus gros soucis lors de ce projet, et donc la chose qui nous a fait perdre le plus de temps *(car nous ne voulions pas trop avancer le code sans être sur de la librairie utilisée)*. Nos choix se sont portés sur d'autre librairies au préalable, et après maintes erreurs d'installation, ou de compatibilité de version *(entre les librairies, avec le compilateur, ou encore même avec la version de linux)*, nous nous sommes arrêtés sur celles-la. Donc ce n'est probablement pas le meilleur choix que nous ayons pu faire, mais c'est le premier qui a *(à peu près)* fonctionné comme nous le souhaitions.

Nous utilisons également la librairie Boost afin d'avoir les fichiers contenus dans les dossiers Image, Video et Musique.

Design patterns

- Pattern Abstract Factory : (UML : Annexes p.7)

Le patter abstract factory a été le plus dur a mettre en place. Nous avions besoin d'une interface graphique pour cette application. Nous devions gérer la création de classe de librairies importées dans nos factory, et nous ne pouvions pas modifier ces classes pour nous arranger. Nous avons donc du nous débrouiller avec les fonctions proposées par ces librairies, pas forcément adaptées à une Abstract Factory.
Toutefois, ce pattern nous a semblé vraiment adapté à notre application. Il y a deux familles : les « boutons », et les « formats ». D'un coté, on peut avoir des boutons adapté pour l'audio et la vidéo *(play/pause/stop appelés boutonsVA)*, et de l'autre, des boutons adapté pour l'image *(image suivante/précédente appelés boutonsI)*. Idem pour les formats, d'une part le format « grand » *(grande fenêtre, permettant de lire agréablement des videos ou des images)* et d'autre part le format petit *(nul besoin d'une grande fenetre pour lire un audio)*.
Donc selon le média choisi, l'application appelait la factory adaptée *(nous diposions donc d'une factory par média)* à la création de l'interface du média.
Une réaction en chaine démarrait donc dès l'appel de la factory, qui elle-même appelle les fonctions createButtons() et createFormat() respectivement des boutons et du format fournit en paramètre de sa propre fonction createInterface(), afin de renvoyer un objet Interface, dont les différentes classes représentantes des médias sont composées.

Ce pattern a été le premier que nous avons essayé d'implémenter, parce que nous nous doutions que c'etait le plus chronophage. Seulement, c'est l'un des dernier que l'on travaille dans l'UE. Nous nous sommes donc rendu compte *(plusieurs fois)* au fil de la création de l'application que ce que nous avions créé n'était pas une Abstract Factory.

Quand aux extensions possibles de notre Abstract Factory, nous aimerions dans un futur proche améliorer l'esthetique de notre interface graphique *(par exemple, un seul bouton pour le play/pause, des fonds un peu plus sobres)*, mais aussi l'ergonomie de l'application *(cela nécessitera peut-etre de changer les librairies que nous utilisons)*. Enfin, pourquoi pas des fonctionnalités en plus, telles que l'avance et le retour rapide, le zoom mentionné précedemment, ou encore un bouton de gestion du son.

- Patterns State : (UML : Annexes p. 8)

Dans notre programme, nous nous sommes servi pattern State à deux reprises, avec la même fonction mais pour deux médias différents, l'audio et la vidéo.
Nous avons choisi ce pattern car nous voulions que l'utilisateur puisse effectuer des actions sur le média, mais que le résultat de ces actions dépendrait de l'état actuel du média.
Pour faire simple, considérons le média « Vidéo ». La vidéo peut être dans plusieurs états « Lecture », « Pause » et « Arret », et l'utilisateur peut décider d'utiliser les boutons « Lecture », « Pause » et « Stop » quand il le souhaite. Seulement, utiliser le bouton pause lorsque la vidéo est dans l'état « Arret » n'aura aucun effet. De même, utiliser le bouton « lecture » lorsque la vidéo est déjà dans l'état « Lecture » n'aura aucun effet. *(Dans d'autres applications du même type, le bouton Lecture/Pause est en fait le même, mais ce n'est pas encore le cas pour nous)*. C'est pour gérer ce type d'actions que nous avions besoin du pattern State.

- Pattern Observer : (UML : Annexes p.9)

Lorsque nous avons pensé aux patterns qui pourraient nous servir, l'idée de l'Observer nous a paru bonne, afin de permettre a l'utilisateur de lire un fichier de sous-titres. Comme dit précedemment, pour l'instant l'application permet juste de lire un fichier texte, et afin de simplifier la tache de l'application, l'utilisateur devra faire en sorte que la vidéo et le fichier texte contenant les sous-titres aient le même nom. *(Une extension future pourra permettre a l'application de lire des fichiers .srt)*
Le SubtitleSubject est le sujet concret, il regarde dans le fichier sous titre et grâce à la fonction *setData(string filepath),* il envoit à son observer les données que ce dernier doit afficher. On utilise le current time de la vidéo pour savoir à quel moment le sujet doit envoyer ses données à l'observer afin que celui ci puisse les afficher à l'écran.
SubtitleLineObs est l'observer concret, il renvoit à l'écran la ligne de sous titre à afficher.

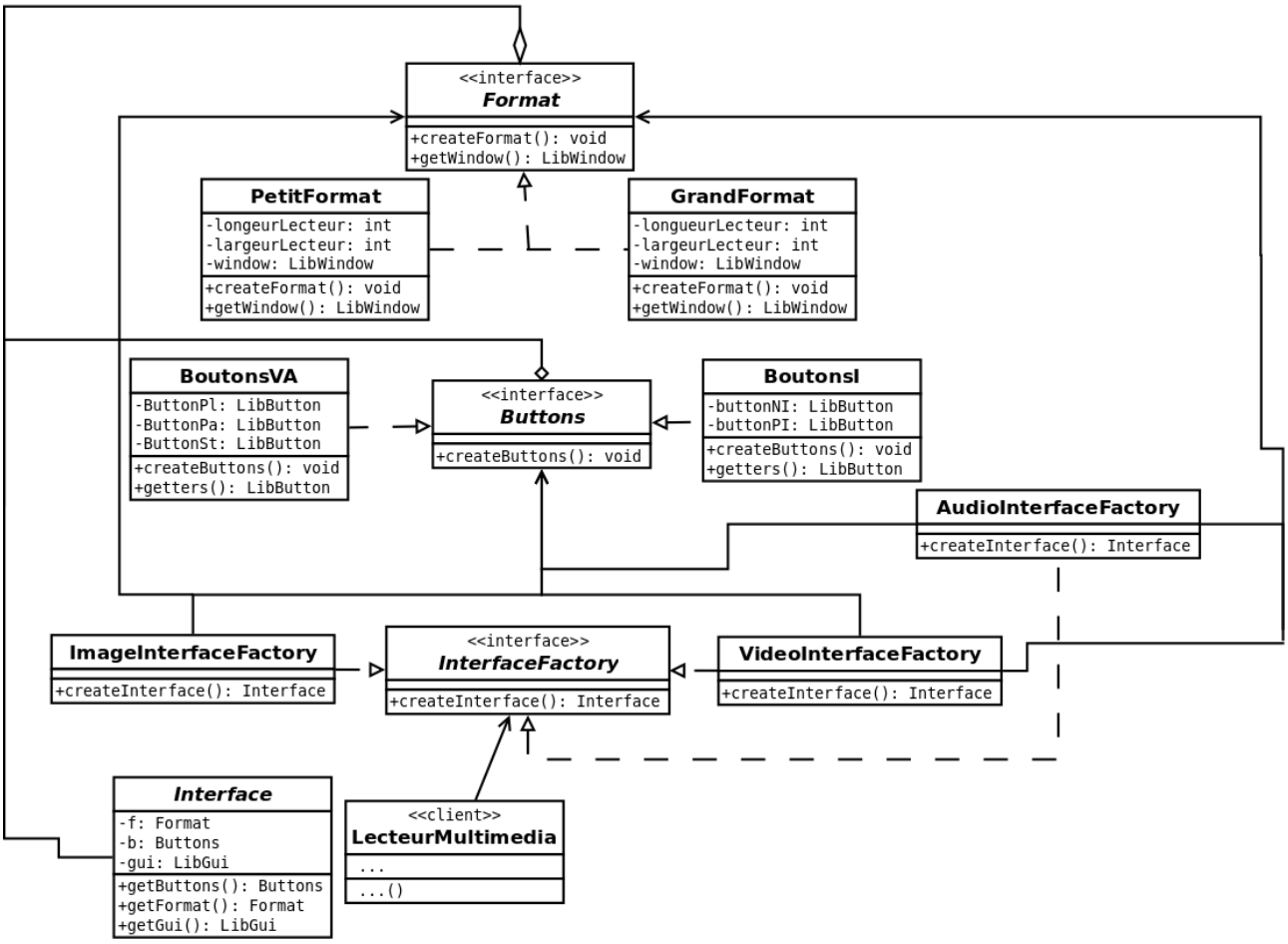
Conclusion :

Ce projet nous a permis de nous familiariser avec le concept de design pattern, des outils permettant de rendre un programme plus extensible et plus maintenable.
Ce fut une expérience enrichissante que de conduire un projet ambitieux et dans un langage que nous conaissions peu, néanmoins cette expérience fut ternie par nos problème d'installation des librairies localement, qui nous a fait commencer le projet bien plus tard que prévu.

Nous somme déçus de ne pas avoir pu ajouter toutes les fonctionnalités citées dans le rapport, mais nous considérons qu'elles pourront faire l'obet de futures extensions. Les plus impératives sont celles consistant a améliorer le pattern Abstract Factory, l'esthetique et l'ergonomie étant essentielles a ce type d'applications, ainsi que la prise en compte de plus de format de médias.

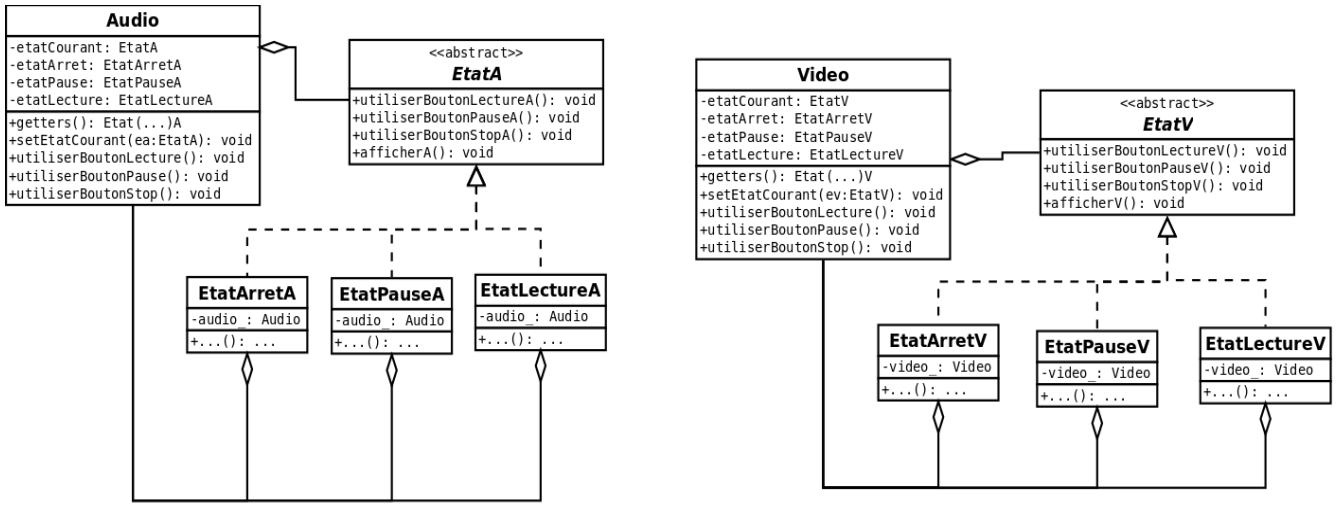
Ce fut un projet de plus grande envergure que ceux auxquels nous avions pu participer, et bien plus concret. Nous sommes donc heureux d'avoir pu nous soumettre à ce type d'exercice, d'autant plus que nous en sortons meilleurs, que ce soit en C++, dans la maitrise des design patterns, ou dans la gestion du temps d'un projet comme celui-ci.

Annexe 1 : Diagramme UML du design pattern « Abstract Factory »



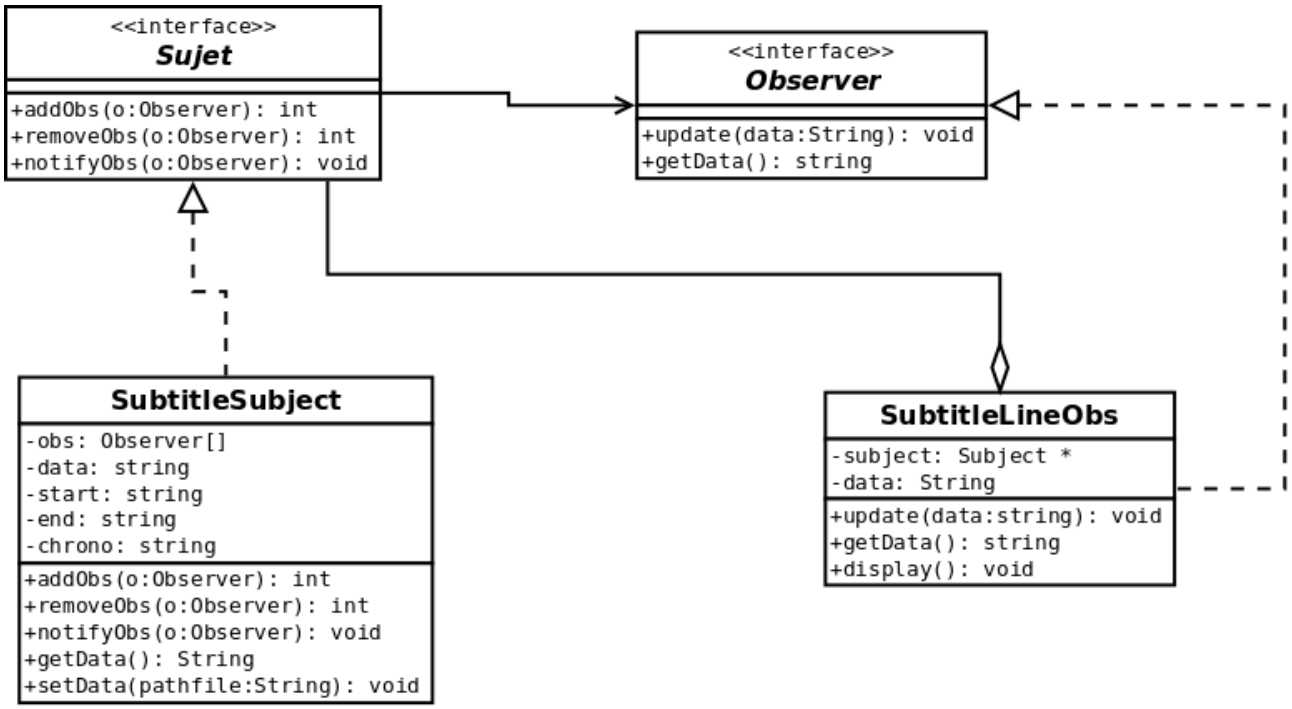
LibButton, LibWindow, et LibGui correspondent aux classes des librairies utilisées pour ce pattern.

Annexe 2 : Diagramme UML du design pattern « State »



Pour éviter de surcharger, nous n'avons pas recopié les méthodes de EtatA(ou V) que les Etats concrets implémentent.

Annexe 3 : Diagramme UML du design pattern « Observer »



Annexe 4 : Diagramme de toutes les classes