Rapport de conception

SynthLab (Équipe 2)

Corentin Beaucé, Antoine Brioy, Johan Champion, Anthony Cobac, Pol Le Tue,
Salim Nadour & Jérémy Yziquel

Master 2 Génie Logiciel
Année 2015/2016

istic Informatique Électronique

Université de Rennes 1, Campus de Beaulieu
UFR Informatique et Électronique
263, avenue du Général Leclerc
CS 74205, 35 042 Rennes Cedex

Table des matières

Introduction

Ce rapport a pour objectif de présenter et de justifier les choix de conception que nous avons été amenés à effectuer au cours de la réalisation du projet Synthlab.

La conception, en accord avec la méthodologie Agile que nous avons choisi d'utiliser, s'est faite tout au long du projet selon un processus itératif.

Notre architecture se présente sous deux aspects principaux : le Domaine métier et la Vue. Ils sont indépendants les uns des autres.

1 Domaine métier

1.1 Architecture

Compte tenu de la durée relativement courte du projet et du nombre restreint de fonctionnalités à implémenter, nous avons jugé pertinent de concevoir une architecture minimaliste.

En analysant le domaine métier, on remarque que le synthétiseur se compose de modules totalement indépendants les uns des autres, qui communiquent au travers de câbles. Avec cela en tête, nous avons créé notre premier modèle d'architecture, qui est l'interface du modèle métier (voir figure 1).

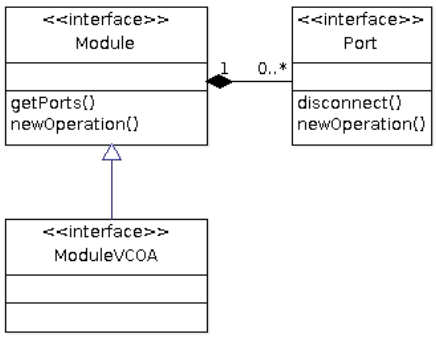

Figure 1 – Diagramme de classes simplifié des interfaces du domaine métier

Ce modèle, très simple, peut être répliqué pour chaque module de façon parallèle. Chaque implémentation d'un module métier aura juste à répondre à l'interface `Module`. Les `Port`s seront ensuite utilisés par l'application pour relier les modules qui fonctionnent de façon autonomes avec les données qu'ils reçoivent.

1.2 Implémentation

L'implémentation du modèle métier repose sur JSyn, une librairie Java permettant de réaliser des applications interactives de musique. C'est une librairie généralement utilisée pour générer des effets sonores, des environnement audio ou de la musique.

Les filtres La majorité des modules utilisent la librairie par le biais de filtres héritant des interfaces JSyn `UnitGenerator` et `UnitFilter`. Ils permettent la gestion des ports des modules ainsi que les traitements que doivent effectuer ces mêmes modules. Les modules, eux, gèrent le traitement des informations que l'on va diffuser aux bons filtres.

Les modules La librairie JSyn implémente déjà nos modules. Oscillateur, Mixeur, Enveloppe, ... : la plupart de ces modules sont déjà présents. Nous avons simplement eu besoin de les adapter à nos besoins. Nos modules résultent de la combinaison des filtres et modules JSyn.

Les ports Les classes des ports, `InputPort` et `OutputPort`, sont des surcouches de leurs homologues définie par JSyn. Ils savent en permanence s'il est connecté ou non. Lors d'une connexion, il vérifie le type de l'autre port. Si le nouveau port est d'un type opposé, il dit à JSyn de se connecter dessus et il récupère d'autres informations. Dans le cas où le port n'est pas connectable, le port garde seulement les informations annexes. Ce système permet de connecter n'importe quel type de ports ensemble.

1.3 Problèmes rencontrés

Le principal problème qui s'est posé à nous, au-delà de la compréhension du domaine métier, est la compréhension de la librairie JSyn. Nous avons décidé d'encapsuler le fonctionnement de JSyn dans notre propre modèle métier et d'en modifier : de cette façon, ce dernier respecte le fonctionnement d'un synthétiseur modulaire tel que défini par cette librairie.

Nous avons cependant dû ajouter des comportements supplémentaires (i.e. non permis par JSyn) dans notre application. Par exemple, dans JSyn, il n'est pas possible de connecter une entrée avec une autre entrée alors que notre implémentation de l'interface laisse l'utilisateur interagir avec notre synthétiseur de la même façon qu'avec un modèle physique, c'est-à-dire qu'il peut commettre des erreurs dans ses branchements sans que cela ne n'entre en conflit avec notre implémentation de JSyn.

2 Vue

2.1 Conception

2.1.1 Technologie

Compte tenu des connaissances communes du groupe et du choix de la librairie JSyn (qui nous impose un modèle écrit en Java) ; JavaFX s'est imposé comme un choix évident pour réaliser l'interface graphique. Nous avons décidé de définir l'interface de nos interfaces JavaFx à l'aide de fichiers .fxml pour avoir un format plus léger et intuitif de la description des modules.

2.1.2 Intégration avec le modèle métier

L'architecture de la vue repose sur la même philosophie que celle du domaine métier : elle est composée de modules indépendants. En comparant la figure 1 à la figure figure 2, on s'aperçoit que la classe abstraite servant de base à la représentation visuelle de tous les modules s'intègre naturellement à la suite du domaine métier.

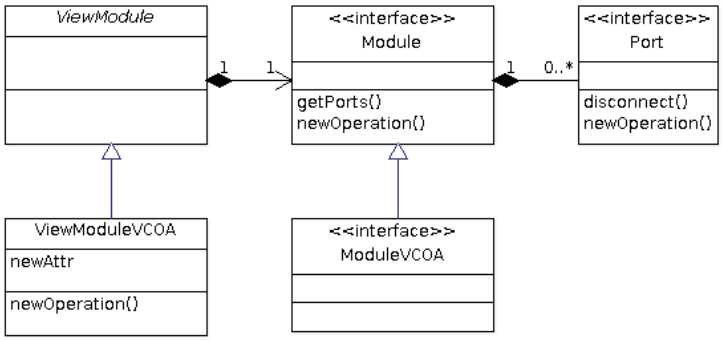

FIGURE 2 – Diagramme de classes simplifié des interfaces du domaine métier et de la vue

Communication avec le modèle métier Pour garder l'architecture la plus simple et claire possible nous sommes partis du principe que la vue et le modèle ne communiquent que dans un sens. En effet, la vue ne permet de changer que des paramètres du modèle et n'affiche aucune information (à la manière des synthétiseurs modulaires physiques). Ceci est correct pour le cas général, mais dans le cas du module représentant un oscilloscope, on a eu besoin d'afficher des données du modèle métier ; cela nous a causé plusieurs problèmes qui sont détaillés dans la partie 2.3.

2.1.3 Architecture

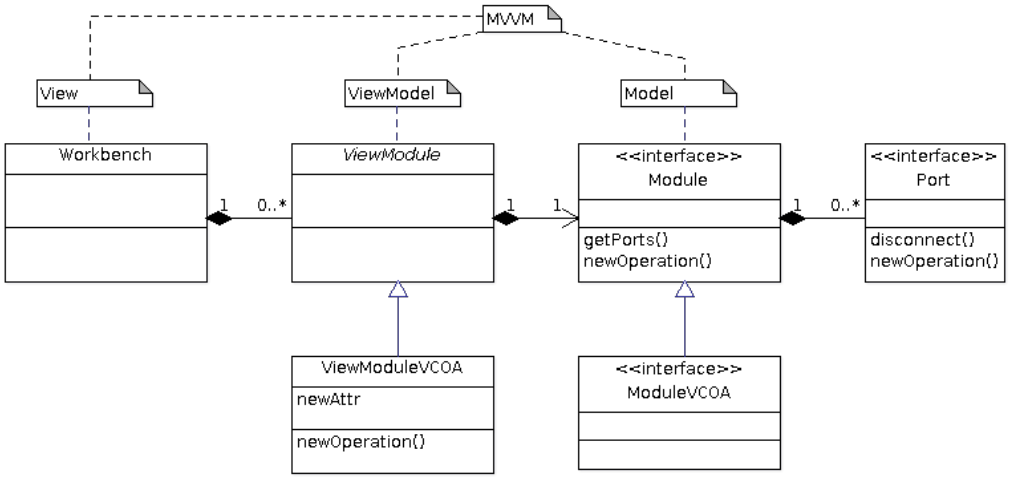

FIGURE 3 – Diagramme de classes simplifié d'un module de la vue

Patrons de conception Cette architecture suit le patron de conception Model-View-ViewModel (*MVVM*), qui est une variante du très populaire Model-View-Controler (MVC). Ce pattern se caractérise par une synchronisation du ViewModel sur le Model au travers de commandes (le Model n'a pas conscience du ViewModel mais il est manipulé par ce dernier), et une communication bidirectionnelle entre la View et le ViewModel.

Comme on peut le voir sur la figure 3, nos Models sont les instances de `Module` du modèle métier, les ViewModels sont des instances `ViewModule` et la View est la classe `Workbench` qui à l'aide du data-binding de

JavaFX communique avec les `ViewModule`s. Cette classe contrôle la position et l'affichage des modules (gestion du drag and drop), ainsi que celui des câbles.

Communication entre les modules Toute la communication entre les modules (transmission des information sonores et des tensions) s'effectue au niveau du modèle métier (voir partie 1.2). Pour que deux modules communiquent entre eux, il faut les connecter ensemble. La méthode `connect(Port)` est appelée pour connecter deux modules, et ils peuvent alors commencer à communiquer. L'aspect visuel des câbles est géré par les classes `Workbench` et `CableManager`.

Organisation des modules Chaque module du synthétiseur est composé de deux éléments :

— Une classe implémentant `Module.java` (comme décrit partie 1.2) qui est le modèle.

— Une classe qui étend `ViewModule.java` qui charge un fichier FXML et lui sert de contrôleur.

Il suffira ensuite de rajouter une entrée dans l'énumération `ModuleType` et du créer des méthodes dans les factory `ModuleFactory` et `ViewModuleFactory` pour initialiser le modèle la vue et les commandes du module. Cette organisation permet une création très rapide de nouveaux modules. Pour la vue, la plupart des composants standards (potentiomètres, afficheurs, boutons, etc.) sont déjà mis en place et ont juste besoin d'être ajoutés au FXML. Le modèle quand à lui peut être plus ou moins difficile à implémenter selon les possibilité de la librairie Jsyn.

2.2 Aspect visuel de l'application

Pour l'aspect visuel du synthétiseur nous avons utilisé en grande partie des images libres de droit ou des créations par des membres de l'équipe. L'apparence des composants est directement inspirée du design des Synthétiseur modulaire physique, ainsi nous avons gardé des couleurs sombres et métalliques. Il est possible de changer de thème, mais cela s'applique seulement a la Workbench.

Composants des modules JavaFX nous permet de rendre notre interface visuellement cohérente grâce à des composants réutilisables (tels que les `Knob` et les `Plug`), ces composants sont utilisés et paramètres dans les FXML de chaque module. À la manière des composants JavaFX prédéfinis, il est possible de changer leur apparence et leur fonctionnement au moyen de paramètres directement dans le FXML.

2.3 Problèmes rencontrés

La classe du plan de travail (`Workbench.java`) fut l'une des plus problématiques, car une des plus importantes. Au cours du développement cette classe a accueilli le code gérant la position des modules ainsi que celui gérant les câbles. Ceci a posé un problème pour localiser les dysfonctionnements des comportements des câbles et d'enregistrement de ces derniers. Le code gérant les câbles a été déplacé dans une autre classe durant les derniers jours du projet, ce qui a nécessité de gros efforts pour garder le code existant compatible.

Pour une question d'optimisation (qui s'est avérée être prématurée), l'état des connexions entre les `ViewModule`s n'est pas directement synchronisé avec celui des `Module`s. Chaque connexion est stockée à la fois dans les `Plug` des `Module` et les `Cable` du `CableManager`. Cela permet de ne pas avoir à parcourir tout les modules pour obtenir la liste des câbles à tracer, mais cela nous a causé à plusieurs reprises des problèmes de synchronisation entre les états de la vue et du modèle.

Communication modèle vers vue La communication du modèle vers la vue ne se fait que dans pour deux modules : l'oscilloscope et le séquenceur.

Pour l'oscilloscope, le modèle crée un objet Swing servant d'affichage et le transmet à la vue à l'initialisation du module.

Dans le cas du séquenceur, la communication Modèle → Vue est nécessaire pour récupérer le pas courant du séquenceur. On a mis en place, pour gérer cela, un pattern Observer. La vue observe le modèle, qui a notifie de chaque changement du pas courant.

Composants de l'interface Les potentiomètres ont également posé quelques problèmes, car nous avons décidé de les rendre paramétrables afin de les générer et modifier plus facilement. La classe (`Knob.java`) est vite devenu assez conséquente et compliquée à explorer.

Création d'un module depuis la boite à outils La boite à outils et le workbench ont eu un problème de détection du glisser-déposer entre eux, lors de la création de nouveaux modules. Ce problème vient de JavaFX et il a fallut utiliser des techniques qui ne devrait pas avoir lieu d'être et seront certainement obsolètes dans les prochaines versions de JavaFX.

Oscilloscope La réalisation de la partie graphique de l'oscilloscope n'a pas été sans nous poser quelques problèmes. Le principal d'entre eux est dû au fait que nous sommes partis d'un Panel Swing fourni avec la librairie JSyn, que nous avons encapsulés et inclus dans l'interface JavaFx à l'aide de `SwingNode`. Or, l'exécution des éléments inclus dans dans `SwingNode` se fait dans un Thread différent de celui de l'interface JavaFx. Aussi, nous nous sommes retrouvés confrontés à des problèmes de concurrence (en ce sens, ils furent difficiles à détecter).

3 Conclusion

Nous avons préféré, au vu de la courte durée du projet, de réaliser une architecture minimale et rapide à mettre en oeuvre. L'architecture que nous avons proposés a pour avantage de rendre très simple l'ajout de nouveaux modules.

En effet, avec les composants JavaFx et les ports, il nous suffisait de créer deux ou trois fichiers courts, indépendants du reste de l'application, pour réaliser un module complet. Et les bugs d'un module n'auront aucun impact sur le fonctionnement des autres modules.

Néanmoins, notre conception ne permet pas une modification globale de certain aspects de nos modules du fait de leur indépendance. Cela n'est cependant pas un problème majeur, étant donné la portée du projet ainsi que le faible nombre de types de modules composant notre synthétiseur.