Synthlab : Rapport de conception

Auteurs :
ANNE Aurélien,
COISNARD-SIMON Marie,
FORTUN Nicolas,
LAURENT Julien,
NOMANE Ahmed
SALMON Kevin

7 mars 2018

Table des matières

1 Choix techniques

Notre synthétiseur numérique Synthlab permet de manipuler des flux audio via différents modules. Le product owner (PO) nous a fortement conseillé d'utiliser la librairie JSyn en Java. Cette librairie permet de créer et manipuler un signal audio. Elle embarque la gestion des signaux, des filtres et certains modules que nous avons réutilisés. Nous avons réalisé l'affichage avec JavaFX. Il nous a permis de créer librement une interface graphique avec Gluon.

2 Architecture de l'application

2.1 MVC

Notre application se découpe en 3 parties, la partie graphique avec les fichier fxml, le modèle comportant nos classes module et des contrôleurs. Nous avons un contrôleur pour chaque module. Ceux-ci ont pour tâche de transmettre à leur module associé les valeurs saisies par l'utilisateur. Nous avons également un contrôleur pour le board et un autre principal. Le contrôleur du board, IHMController, s'occupe principalement du Drag&Drop des modules. Il s'occupe également de récupérer les évènements sur l'IHM tels que l'ajout de module, le changement de skin, la sauvegarde ou le chargement d'une sauvegarde. Il appelle ensuite le controlleur principal, Controller, pour réaliser ces tâches.

2.2 Découpage des packages

L'application est divisée en huit packages :
— module : modules audio utilisant JSyn
— controller : gestion de l'IHM
— utils : classes utilitaires
— ihm : classes faisant l'interaction entre l'IHM et les modules
— example : programmes principaux utilisés au début du développement pour tester l'application.
— exceptions : gestion d'erreurs possibles de l'application
— filter : filtres JSyn
— sauvegarde : sauvegarde de l'espace de travail

2.2.1 Package module

Pour les modules, nous avons créé une classe abstraite Module qui hérite de Circuit, une classe venant de JSyn. Celle-ci comporte les méthodes concrètes connect et disconnect, et une méthode abstraite getPort qui, depuis le nom d'un port, doit retourner un tuple contenant le UnitPort et le PortType de celui-ci. UnitPort venant de JSyn et PortType étant une énumération recensant les types des ports.

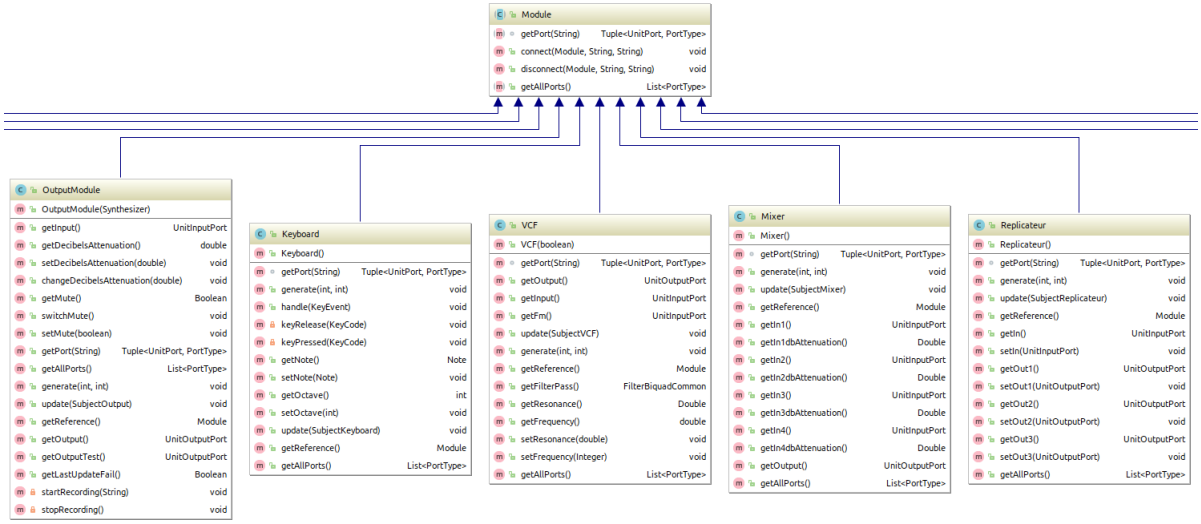

FIGURE 2 – Extrait du package Module

2.2.2 Package controller

Une classe Controller (fig. 3) permet de faire le lien entre les Controllers du package *ihm* et JavaFX. Par exemple, cette classe ajoute les modules au workspace, modifie le skin, sauvegarde et recharge le workspace etc.

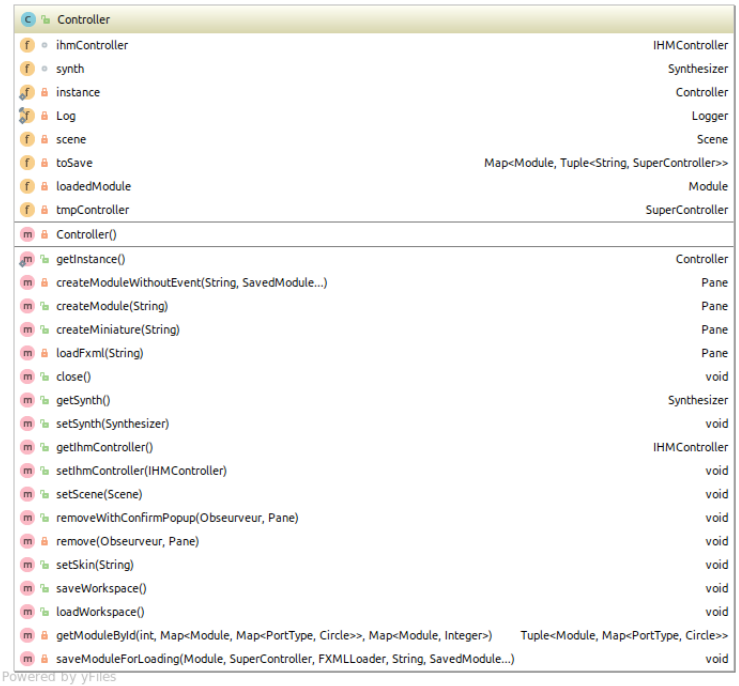

FIGURE 3 – Classe Controller du package controller

2.2.3 Package utils

Le package *utils* (fig. 4) regroupe les classes utilitaires nécessaires au bon fonctionnement de l'application. Il contient les classes suivantes :

— FxmlFilesNames : Énumération qui contient le chemin du fxml de chaque module ;

— OscillatorType : Énumération qui définit les différents types d'oscillateur ;
— PortType : Énumération qui définit les différents types de port ;
— SkinNames : Énumération qui contient le chemin des différents thèmes ;
— Tuple : Classe qui définit un couple (A ;B) ;
— Cable : Pojo servant à modéliser un câble ;
— CableManager : Singleton servant à gérer la connexion entre les câbles ;
— OscillatorFactory : Classe qui génère un oscillateur suivant un type précis.

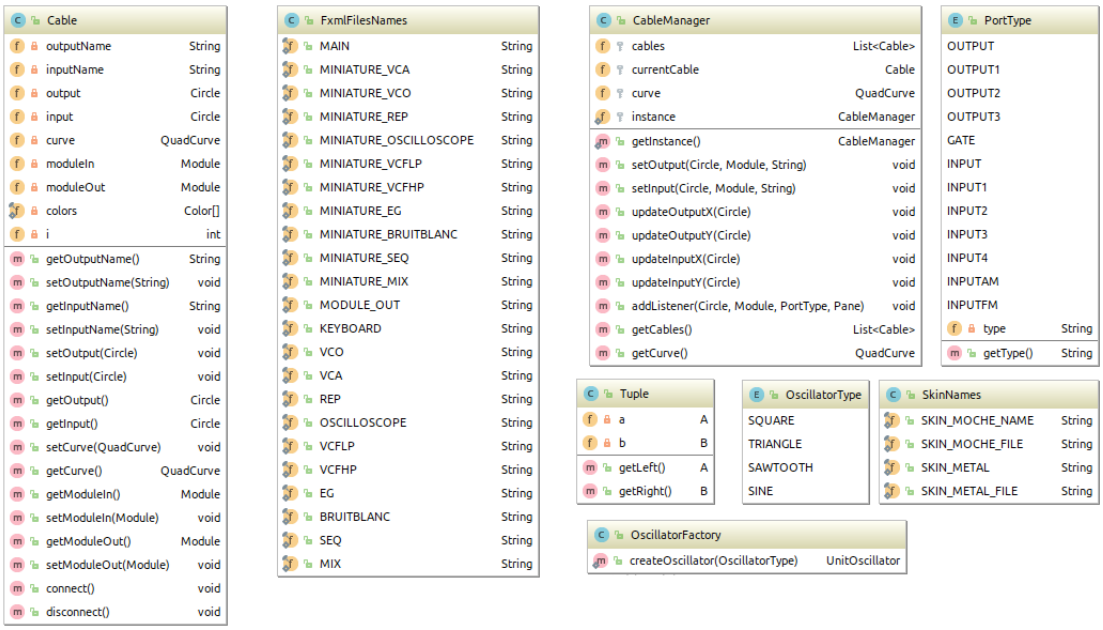

FIGURE 4 – Package Utils

Câble Manager

Le CableManager est un singleton qui comprend les méthodes permettant de relier deux modules depuis l'IHM. Celui-ci ajoute un listener sur les ports graphiques grâce à la méthode addListener. Ce listener va permettre, quand le port reçoit un clic, de récupérer ce port et de le lier ensuite. Le CableManager comporte une liste de Cable. La classe Cable est un POJO qui contient les informations sur les ports d'entrées et sorties liés. La gestion de l'effet "pendouillant" est géré dans cette classe, par une courbe de bézier. Il y a un point de contrôle, calculé comme suit : $x = (x_{input} + x_{output})/2; y = y_{input} + y_{output}$.

Pour faire communiquer le controller graphique d'un module et le module concret, nous avons mis en place un pattern Observer. Dans celui-ci, ce sont les modules qui observent leurs controllers graphiques. Ainsi, dès qu'une modification des paramètres est effectuée sur un module, nous pouvons mettre à jour le module concret, le tout en ayant un couplage faible entre IHM et modèle. Nous avons pour cela une interface Obseurveur<T> avec une méthode update et une méthode getReference. Cette dernière est appelée uniquement pour passer cette référence au CableManager, elle n'est en aucun cas gardée par les controlleurs. L'interface Subject, quant à elle, propose des méthodes permettant à un Obseurveur de s'abonner, se désabonner et d'être notifié. Cependant, il était important dans cette communication de passer les nouvelles valeurs saisies par l'utilisateur via la partie graphique. Le problème est que chaque module a des propriétés en quantité différente, et ayant des noms, types et intervalles de valeurs différents. Pour résoudre cela, nous avons donc créé pour chaque module une sous-interface de Subject. Ainsi, lorsque le

module est notifié d'un changement, il peut récupérer très facilement toutes les valeurs voulues tout en ne connaissant pas le type concret de ce Subject.

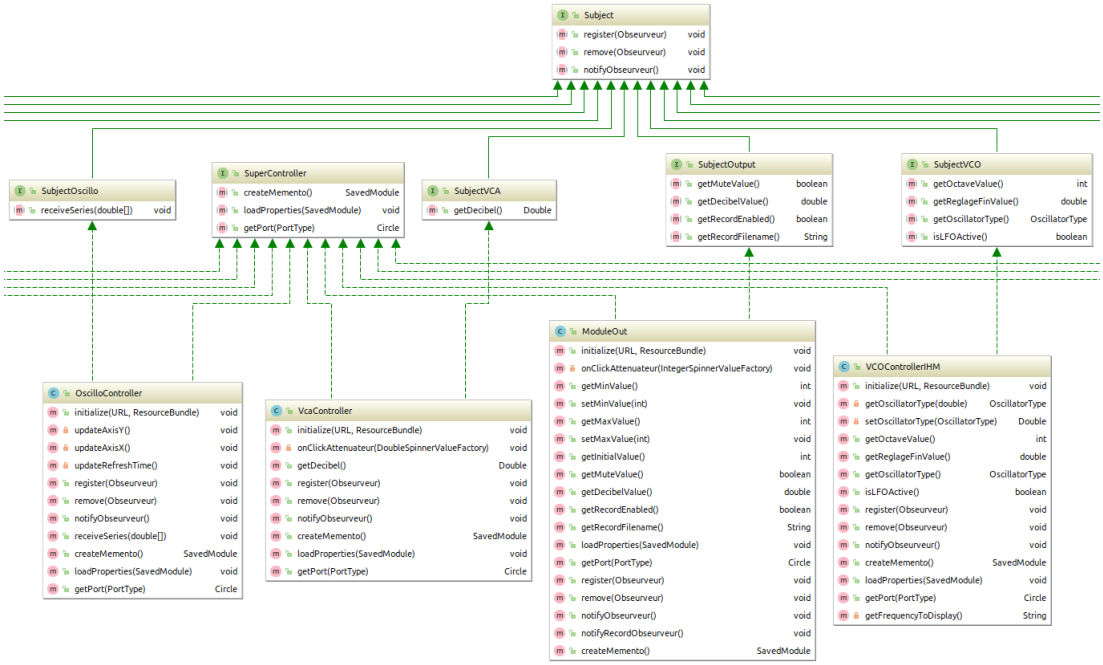

FIGURE 5 – Extrait des classes du package ihm

2.2.4 Package example

Le package *example* (fig. 6) a été utilisé lors du développement afin de principalement tester les premiers modules lorsque l'IHM ne supportait pas encore les connexions par câbles. Les paramètres du VCO, VCA et du module de sortie peuvent être ajustés au clavier.

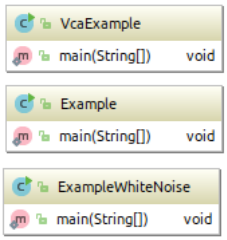

FIGURE 6 – Diagramme des classes du package example

2.2.5 Package exceptions

Le package *exceptions* (fig. 7) contient trois exceptions dédiées à Synthlab. Elles sont utilisées pour détecter les erreurs potentielles de l'application.

2.2.6 Package filter

Le package *filter* (fig 8) contient une unique classe *AttenuationFilter*. Elle hérite de la classe JSyn *UnitFilter* et permet d'appliquer un gain en décibels à un signal audio. Cette abstraction de JSyn permet de réutiliser ce filtre dans plusieurs modules.

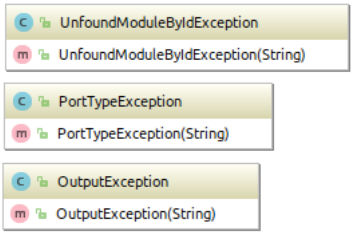

FIGURE 7 – Diagramme des classes du package exceptions

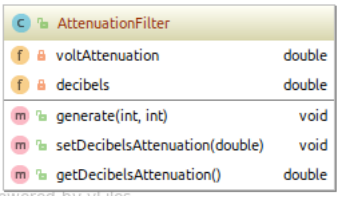

FIGURE 8 – Diagramme de classe du package filter

2.2.7 Package sauvegarde

Le package *sauvegarde* est utilisé par le pattern Memento pour sauvegarder l'état des modules lors de la sauvegarde du workspace (fig 9). Une classe mère *SavedModule* sauvegarde l'id, l'adresse du fichier fxml et la position en X et en Y du module dans le workspace. Tous les modules sauvegardables ont une classe qui hérite de *SavedModule*. Les classes filles sont des POJO qui mémorisent les informations propres au module concerné.

Lors de la sauvegarde, ces classes sont sérialisées dans un fichier JSON grâce à Jackson. XML n'a pas été retenu car il est plus verbeux et moins lisible. Ce fichier JSON est divisé en deux objets : *SavedModules* qui est la liste des classes filles et *SavedCables* qui est une liste de câbles avec les identifiants et les noms des ports auxquels le câble est connecté. Lors du chargement, ce fichier est lu et parsé par Jackson afin de recréer les objets. Ils sont ensuite ajoutés à l'IHM puis les câbles sont ajoutés.

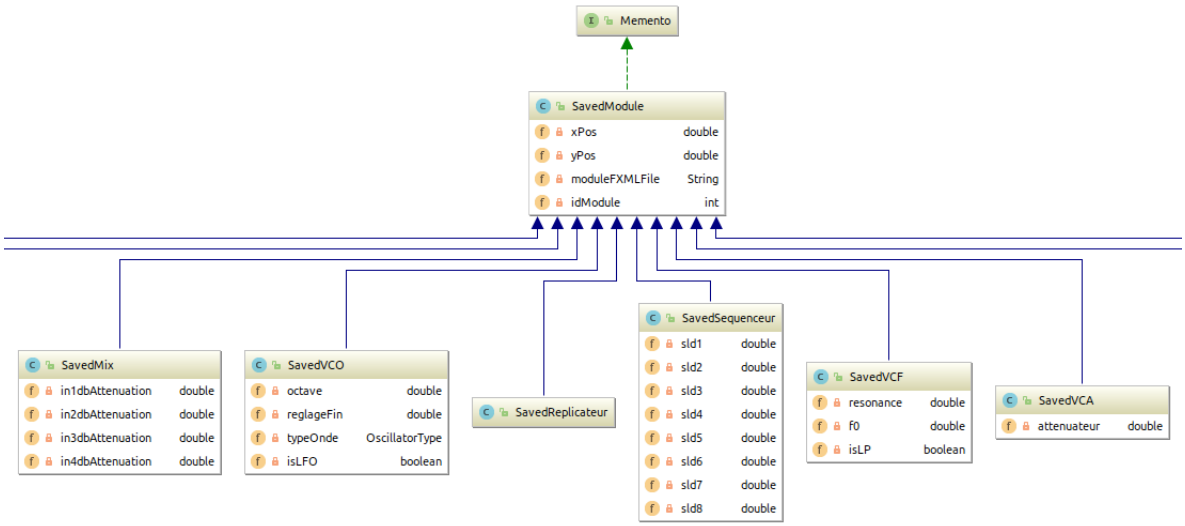

FIGURE 9 – Extrait du package sauvegarde

3 Retours critiques

Synthlab a été réalisé en Java avec JSyn pour la partie audio et JavaFX pour la partie IHM. Le langage n'a pas été un problème car l'équipe connaît le langage et ses possibilités. Il ne nous a pas handicapé.

La bibliothèque JSyn à été difficile à appréhender car sa documentation est succincte et peu évidente. Il nous a par exemple fallu du temps pour comprendre comment fonctionne les différents getters et setters sur les ports, et certains détails nous échappent encore. De plus, plusieurs cas particuliers nous ont ralentis. Par exemple, la gestion des connexions se fait de manière asynchrone, sauf que JSyn ne donne pas la possibilité de récupérer des informations sur le thread impliqué. Il est donc impossible de savoir si les connexions sont encore en cours, et donc qu'il suffit d'attendre, ou si un problème a eu lieu.

JavaFX nous a également posé problèmes. Plusieurs propriétés JavaFX ne sont pas mises à jour et donnent des résultats erronés, notamment sur la position dans l'espace des éléments, ou la taille de ceux-ci a certains moments donnés. Il arrive qu'aucun message d'erreur ne s'affiche lorsque JavaFX bugge.

L'architecture du projet s'est avérée très bonne durant le développement. Une fois qu'elle a été établie, nous avons pu rapidement développer chaque module de façon indépendante. Avec l'ajout progressif de nouvelles fonctionnalités, nous n'avons pas été prisonniers de notre architecture. Elle est restée flexible, simple et intuitive.

Table des figures