Rapport de conception

Chakib Benkebir
June Benvegnu-Sallou
Antoine Ferey
Emmanuel Loisance
Christophe Planchais
Youssouf Roudani

https://github.com/Kwodhan/SynthLabC

L'objectif de ce projet était de réaliser une application client lourd écrite en Java. Dans ce rapport, nous allons détailler la conception et les choix faits.

I- Modèle

JSyn

Dans le cadre de ce projet, le principal avantage a été pour nous l'utilisation d'une bibliothèque telle que JSyn qui est exhaustive. Elle couvre à peu près tout ce qu'il est possible de faire pour synthétiser du son. En effet, les différents modules que l'on peut trouver dans un synthétiseur modulaire analogique sont disponibles (oscillateurs, filtres, et autres fonctions).

Modules standards

Nous avons eu la chance de trouver quelques modules déjà présents dans l'API JSyn : WhiteNoise (Bruit blanc) et l'enveloppe EG (EnvelopeDAHDSR). Nous avons encapsulé ces modules pour reproduire les fonctionnalitées demandées et les adapter à notre architecture. Cependant, la compréhension de certains modules déjà implémentés par JSyn restait difficile.

Il y a une particularité pour le module de sortie. Bien que le module de sortie était déjà implémenté, nous avons décidé de l'étendre pour ajouter nos fonctionnalités. Par

exemple, la variable pour l'atténuateur à été insérée directement dans la fonction de manipulation de signal (*generate*) pour diminuer le son en sortie.

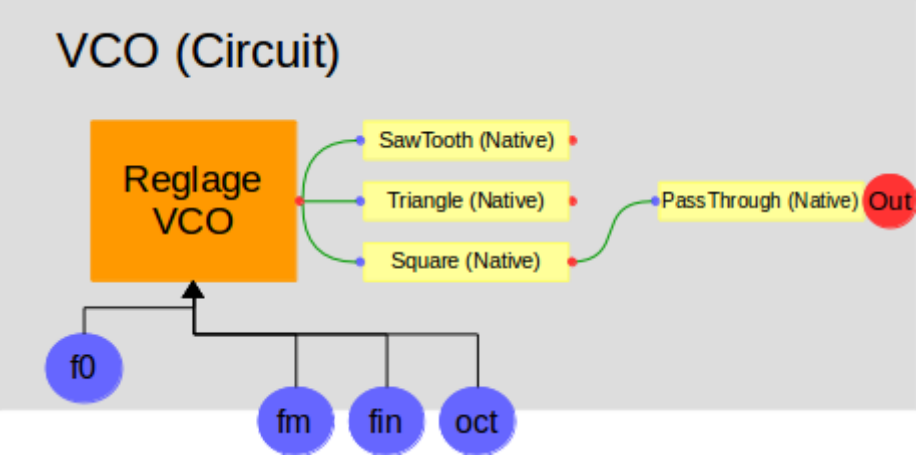
Module en Circuit

La plupart des modules demandés devaient être composés de plusieurs objets JSyn. Par exemple, le VCO est composé de 3 oscillateurs et d'un réglage pour les entrées (fm, in, octave et réglage fin). Pour réaliser ce genre de module, nous avons utilisé l'objet *Circuit* qui répond à cette exigence. Nous pouvons ainsi construire des circuits complexes, destinés au traitement du signal audio.

Nos circuits se composent en général d'un *Reglage* qui gère les entrées (signal ou réglage manuel) et d'un ou plusieurs composants JSyn. Le composant *Reglage* étend la classe *UnitGenerator*. Cette dernière nous permet de faire des traitements du signal en fonction des entrées via la méthode héritée *generate*.

Après l'implémentation du *Reglage*, nous réalisons le branchement sur les entrées des Objets JSyn pour opérer la fonctionnalité du module.

Voici ci-dessous le circuit du module VCO :



Exemple de Circuit JSyn (VCO)

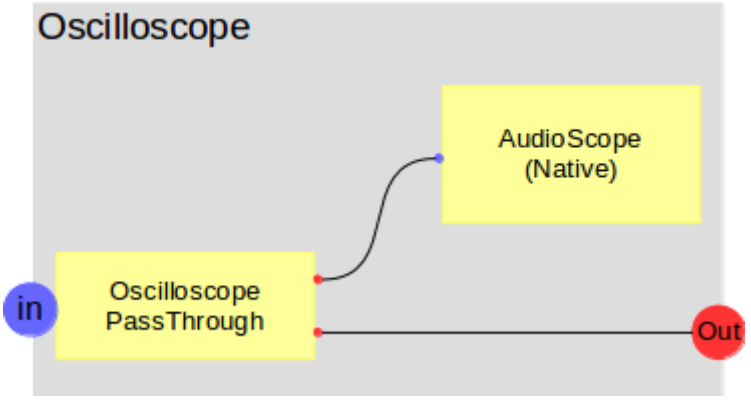
L'objet PassThrough prend un signal en entrée et le réplique en sortie sans changement. Ici pour le VCO, on l'utilise pour simplifier la déconnection du câble sortant de la sortie Out.

Module Oscilloscope

Le module oscilloscope est un peu différent des autres modules. En effet, JSyn prévoit déjà un oscilloscope et renvoie un composant Swing prêt à l'emploi. Néanmoins nous avons dû lui apporter des modifications pour qu'il corresponde à notre besoin.

Le principal problème était lors de la suppression du câble sur la sortie de l'oscilloscope. En supprimant ce câble de sortie, nous déconnectons également la liaison qui avait été établie avec l'AudioScope de JSyn. Nous avons donc dû redéfinir un PassThrough avec une entrée et deux sorties. Avec ces deux sorties, lorsque l'utilisateur supprime le câble de sortie, la connexion vers l'AudioScope n'est pas perdue et l'oscilloscope reste fonctionnel même après un re-câblage.

Pour le *PassThrough*, nous avons étendu la classe *UnitGenerator* et ajouté un *UnitInputPort* et deux *UnitOutputPort*. La fonction *generate* ne fait que dupliquer le signal d'entrée sur les deux sorties à la fois. Il ne reste plus qu'à ajouter l'une des sorties à l'oscilloscope de JSyn via la méthode *addProbe* sur l'objet *AudioScope*.



Architecture de l'oscilloscope

Classe Port

Les classes de noms : *PortX.java* sont en fait des classes qui encapsulent les classes native de JSyn : *UnitOutputPort*, *UnitInputPort* et *UnitGatePort*. Cela nous permet de définir certains comportements particuliers en fonction du type de Port (cf. ci-dessous).

Pour réaliser la connexion en fonction du type des deux ports, nous avons choisi d'utiliser le patron de conception "Visiteur". Chaque instanciation d'un Port spécifique (in, out, am, fm, gate) possède un visiteur qui définit son comportement lors d'une tentative de câblage avec un autre port. Avec cela, nous sommes sûrs d'obtenir une connexion ou un refus quelque soit le type de port.

Dans le cas d'une connexion possible entre deux ports, on réalise la connexion dans la méthode du visiteur: *portOutput.getUnitOutputPort().connect(portInput.getUnitInput())*. La méthode du visiteur renvoie "true" si la connexion est réalisée, "faux" si les deux ports sont incompatibles.

Classe Câble

Cette classe prend deux ports en argument et réalise la connexion si les deux ports sont compatibles. Elle réalise aussi la déconnexion du câble.

II- Contrôleur

Contrôleur Principal

Il s'agit du contrôleur de la fenêtre principale de l'application. C'est ce dernier qui va instancier les modules et les câbles. On demande à la fonction *addMod* l'ajout d'un node FXML correspondant au module ajouté. Si un emplacement est libre, le module est ajouté aux enfants du panneau *stackPane* et on rend possible le mode drag-and-drop pour ce noeud.

A chaque instanciation d'un module ou d'un câble, on l'ajoute à une liste du contrôleur principal. Cela va permettre différents traitements expliqués dans la suite du rapport.

Il s'occupe aussi du tracé de la ligne noire lorsqu'on débute un câblage.

Contrôleur Abstrait ModulesController

ModuleController est la classe mère de tous les contrôleurs de module. Elle gère les comportements à adopter lors des événements (clics sur les ports, Bouton, Slider).

JavaFX permet de créer et manipuler un objet Java avec l'attribut UserData dans le FXML. Cela se fait grâce à la méthode getUserData() qui permet de récupérer un objet initié dans le FXML. Dans notre cas, on se sert de cet attribut pour récupérer le contrôleur du module depuis le FXML à chaque fois que ce dernier est créé.

Connexion

Pour faire une connexion, cela se passe en deux clics :

- Premier clic sur le Port 1 d'un module A : le module A enregistre le Port 1 et la position absolue (variables "double x" et "double y") de l'ImageView correspondant au Port 1. Le module A est sauvegardé dans une variable temporaire du contrôleur principal.

- Deuxième clic sur le Port 2 d'un module B : comme lors du premier clic, le module B enregistre la position du Port 2. Comme nous avons enregistré le module A dans une variable temporaire, nous avons les deux coordonnées des imageView et les deux ports où nous voulons tenter d'établir la connexion.

Si les deux ports sont compatibles entre eux, nous réalisons la connexion et dessinons le câble.

Suppression

En plus de la suppression du module, il est aussi nécessaire de déconnecter les câbles qui lui sont branchés. C'était le point difficile de cette fonctionnalité.

Vue que ni le module, ni les ports ne connaissent les câbles qui lui sont branchés, nous devons rechercher parmi la liste de câbles dans le contrôleur principal lesquelles doivent être débranchés. C'est une faiblesse de notre architecture.

Contrôleur de cable

Cette classe possède en attribut le modèle Câble, ainsi que la ligne graphique pour la représentation sur l'IHM. Elle garde des références vers les contrôleurs du module d'origine et celui de destination pour connaître les coordonnées graphiques des ports lors du tracé et la mise à jour graphique du câble (lors d'un drag-and-drop d'un module).

III- Vue

Nous avons décidé d'utiliser JavaFX pour la création d'interface graphique. L'outil SceneBuilder nous a permis de dessiner notre IHM rapidement.

Drag & Drop

C'est dans la classe DragAndDrop que nous définissons le comportement des modules lors des processus de drag and drop. Nous rendons chacun des 12 emplacements (panes) du board droppable pour les modules. Lors de l'instanciation de n'importe quel *ModuleController*, nous le rendons "draggable".

Conversion Swing / JavaFX : Oscilloscope

Le composant oscilloscope fourni par JSyn est basé sur la bibliothèque graphique Swing. Pour notre application nous avons utilisé JavaFX la nouvelle bibliothèque officielle du langage Java. Nous avons donc dû trouver un moyen d'afficher ce composant "JPanel" de Swing dans notre interface.

La solution a été de passer par les *SwingNode* de JavaFX qui permettent d'intégrer un composant Swing à une application utilisant JavaFX. Pour cela, il suffit de créer un objet

SwingNode et de passer en paramètre le composant Swing à la méthode *setContent*.

```
1 SwingNode swingNode = new SwingNode();
2 SwingUtilities.invokeLater(() -> swingNode.setContent(this.audioScopeView));
```

Thème

Dans un premier temps, nous avons utilisé des feuilles de style CSS qui correspondaient chacune à un thème en particulier (Modena, Caspian). La feuille de style était ajoutée directement sur l'élément parent → *pane.getStylesheets().add("dark.css")*;

Par la suite, nous avons décidé de modifier la façon de procéder pour changer le thème de notre application. En effet, dans un soucis de personnalisation et de simplification, nous avons opté pour la création d'une classe *Style*. Cette classe contient dans des tableaux les différentes valeurs pour chaque thème à partir d'un indice.

Ensuite, lorsque l'utilisateur souhaite changer de thème, deux méthodes sont appelées. La première modifie l'apparence générale du plan de montage et la seconde est appelée pour mettre à jour l'apparence de chaque module déjà présent.

Pour chaque thème, nous avons besoin de trois informations qui sont regroupées dans la classe *Style* :

- colorBorder : contient les couleurs utilisées pour les bordures
- colorBackground : pour les couleurs d'arrière plan des modules
- imageBackground : pour les images d'arrière plan du plan de montage

IV- Sauvegarde

Configuration d'un plan montage

Le design de départ de SYNTHLABC n'a pas été prévu pour la sauvegarde des configurations de nos circuits audio. Ce défaut d'architecture fût la dernière difficulté à surmonter (c.f. section "Sprint3" dans le rapport d'agilité).

Les classes des *ModuleController* et de *CableController* supportent la sérialisation. Cela permet la sauvegarde et le chargement en format JSON. Chaque module et chaque câble doit définir une méthode de sérialisation. Cette méthode passe par les JsonObject qui servent à sauvegarder les attributs essentiels de chaque élément.

Après désérialisation (avec un JSONParser appliqué au fichier de configuration), on effectue la restauration de chaque module et chaque câble à l'aide du *JsonObject*. Cette deuxième partie de restauration est implémentée par les deux méthodes *openModules*() qui gèrent les modules dans le fichier de configuration, ainsi que *openCables*() qui gère les câbles.

On récupère ainsi, pour chaque module, la position dans la fenêtre et leur configuration (position des sliders, Radio bouton, etc…). Pour les câbles, on enregistre la couleur, la position du module émetteur et du module récepteur ainsi que les ports utilisés.

Sauvegarde du son

Cette fonctionnalité est attachée au module de sortie. Nous avons utilisé une classe native de JSyn : *WaveFileWrite*. Elle permet directement d'enregistrer un flux depuis une méthode *generate()*. Grâce à cela, nous avons fait en sorte que chaque module de sortie enregistre seulement son entrée.

WaveFileWrite nous permet d'enregistrer un fichier dans le format WAV. Cependant l'US nous demandait de produire des fichiers MP3. Nous avons donc utilisé la librairie "Jave" pour convertir les fichiers WAV en MP3. Nous utilisons cette librairie dans la classe *AudioFile*.

Pour la conversion en MP3, le fichier WAV est enregistré dans un fichier temporaire localisé dans le dossier */tmp/*. Puis on convertit ce dernier en MP3 dans le dossier que l'utilisateur a choisi depuis l'objet *FileChooser*.

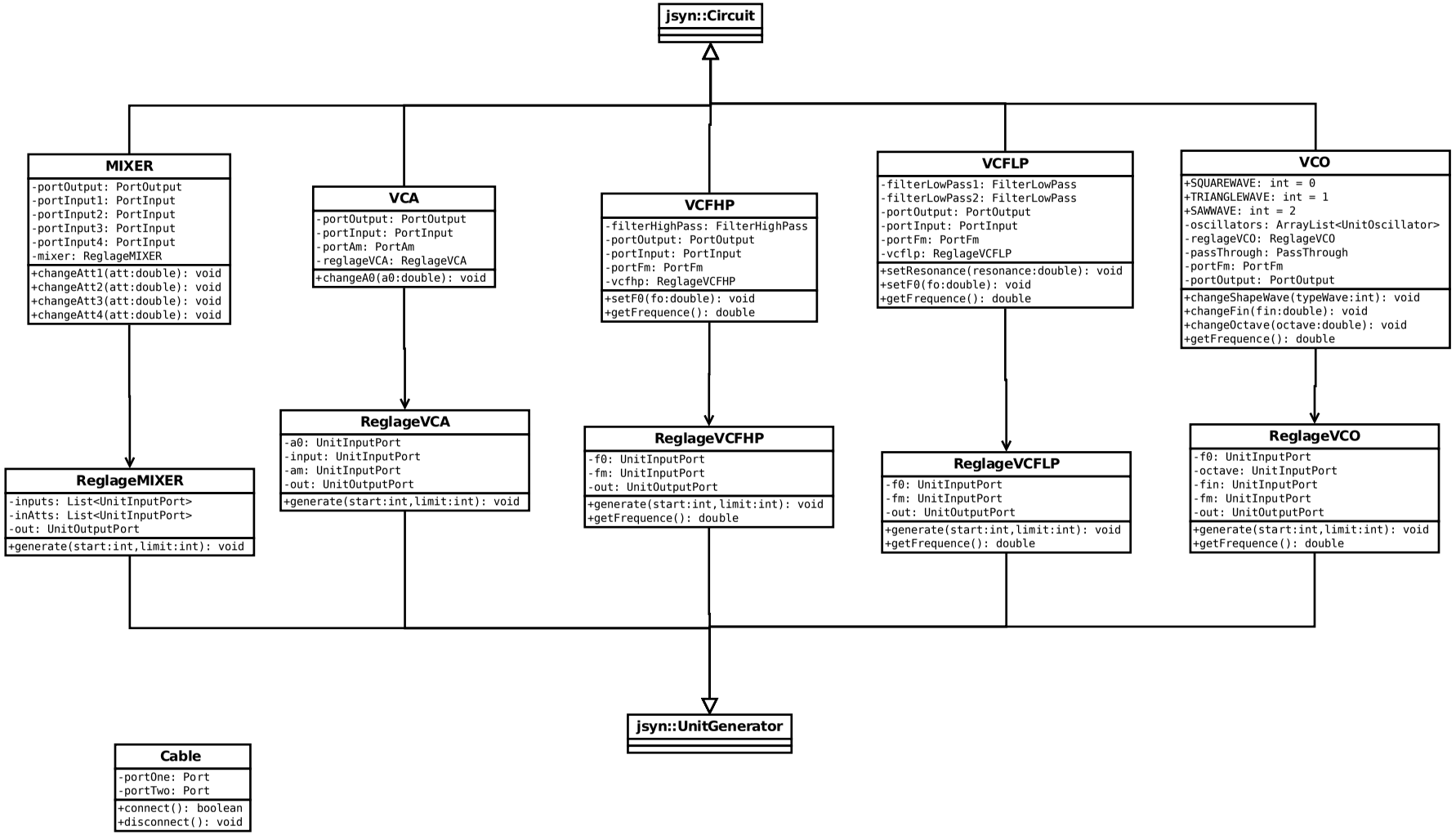
Conclusion

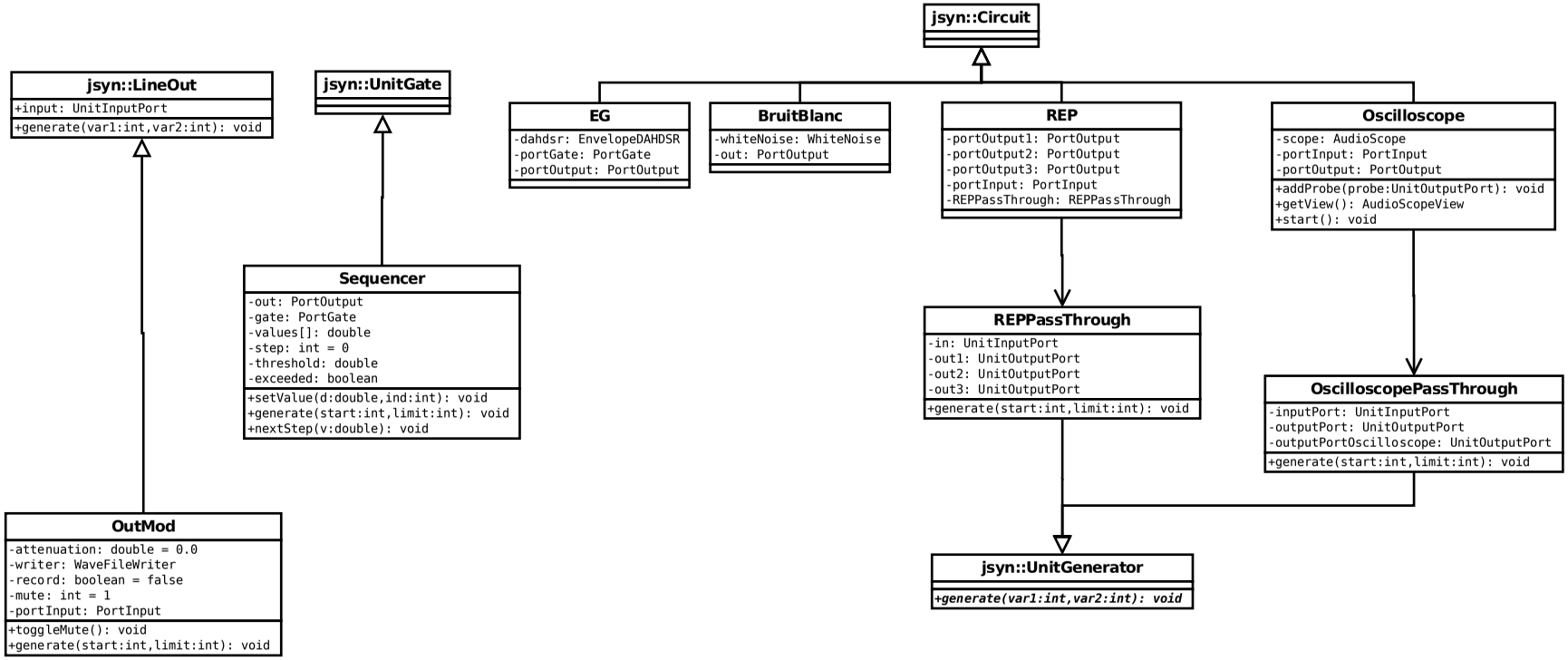
La décision de refactorer notre code lors du sprint 1 nous a fait gagner beaucoup de temps lors des sprints suivants. Grâce à notre architecture, certains modules (Model, View, Controller) furent implémentés en moins d'une heure.

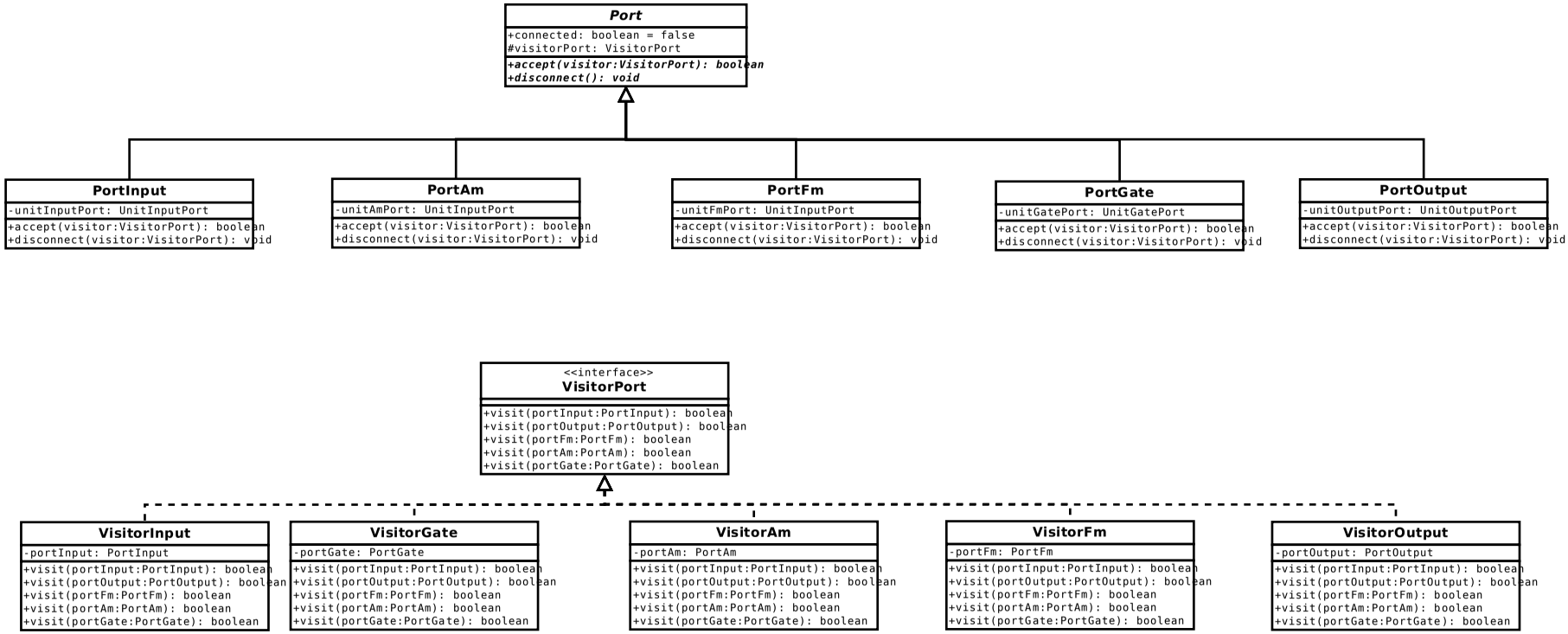
Une amélioration de l'architecture resterait à faire au niveau de la gestion du câblage IHM. Ce refactoring simplifierait également le chargement et la sauvegarde des câbles.

L'application a été développée avec les outils suivants :

- *Java : 8*
- *Junit : 4.12*
- *TestFX : 4.0.11*
- *JSyn : 20170815*
- *Jave : 1.0.2.*


jsyn::Circuit
MIXER
-portOutput: PortOutput
-portInput1: PortInput
-portInput2: PortInput
-portInput3: PortInput
-portInput4: PortInput
-mixer: ReglageMIXER
+changeAtt1(att:double): void
+changeAtt2(att:double): void
+changeAtt3(att:double): void
+changeAtt4(att:double): void
VCA
-portOutput: PortOutput
-portInput: PortInput
-portAm: PortAm
-reglageVCA: ReglageVCA
+changeA0(a0:double): void
VCFHP
-filterHighPass: FilterHighPass
-portOutput: PortOutput
-portInput: PortInput
-portFm: PortFm
-vcfhp: ReglageVCFHP
+setF0(fo:double): void
+getFrequence(): double
VCFLP
-filterLowPass1: FilterLowPass
-filterLowPass2: FilterLowPass
-portOutput: PortOutput
-portInput: PortInput
-portFm: PortFm
-vcflp: ReglageVCFLP
+setResonance(resonance:double): void
+setF0(fo:double): void
+getFrequence(): double
VCO
+SQUAREWAVE: int = 0
+TRIANGLEWAVE: int = 1
+SAWWAVE: int = 2
-oscillators: ArrayList<UnitOscillator>
-reglageVCO: ReglageVCO
-passThrough: PassThrough
-portFm: PortFm
-portOutput: PortOutput
+changeShapeWave(typeWave:int): void
+changeFin(fin:double): void
+changeOctave(octave:double): void
+getFrequence(): double
ReglageMIXER
-inputs: List<UnitInputPort>
-inAtts: List<UnitInputPort>
-out: UnitOutputPort
+generate(start:int,limit:int): void
ReglageVCA
-a0: UnitInputPort
-input: UnitInputPort
-am: UnitInputPort
-out: UnitOutputPort
+generate(start:int,limit:int): void
ReglageVCFHP
-f0: UnitInputPort
-fm: UnitInputPort
-out: UnitOutputPort
+generate(start:int,limit:int): void
+getFrequence(): double
ReglageVCFLP
-f0: UnitInputPort
-fm: UnitInputPort
-out: UnitOutputPort
+generate(start:int,limit:int): void
+getFrequence(): double
ReglageVCO
-f0: UnitInputPort
-octave: UnitInputPort
-fin: UnitInputPort
-fm: UnitInputPort
-out: UnitOutputPort
+generate(start:int,limit:int): void
+getFrequence(): double
jsyn::UnitGenerator
Cable
-portOne: Port
-portTwo: Port
+connect(): boolean
+disconnect(): void


jsyn::Circuit
jsyn::LineOut
+input: UnitInputPort
+generate(var1:int,var2:int): void
jsyn::UnitGate
EG
-dahdsr: EnvelopeDAHDSR
-portGate: PortGate
-portOutput: PortOutput
BruitBlanc
-whiteNoise: WhiteNoise
-out: PortOutput
REP
-portOutput1: PortOutput
-portOutput2: PortOutput
-portOutput3: PortOutput
-portInput: PortInput
-REPPassThrough: REPPassThrough
Oscilloscope
-scope: AudioScope
-portInput: PortInput
-portOutput: PortOutput
+addProbe(probe:UnitOutputPort): void
+getView(): AudioScopeView
+start(): void
Sequencer
-out: PortOutput
-gate: PortGate
-values[]: double
-step: int = 0
-threshold: double
-exceeded: boolean
+setValue(d:double,ind:int): void
+generate(start:int,limit:int): void
+nextStep(v:double): void
REPPassThrough
-in: UnitInputPort
-out1: UnitOutputPort
-out2: UnitOutputPort
-out3: UnitOutputPort
+generate(start:int,limit:int): void
OscilloscopePassThrough
-inputPort: UnitInputPort
-outputPort: UnitOutputPort
-outputPortOscilloscope: UnitOutputPort
+generate(start:int,limit:int): void
OutMod
-attenuation: double = 0.0
-writer: WaveFileWriter
-record: boolean = false
-mute: int = 1
-portInput: PortInput
+toggleMute(): void
+generate(start:int,limit:int): void
jsyn::UnitGenerator
+generate(var1:int,var2:int): void


Port
+connected: boolean = false
#visitorPort: VisitorPort
+accept(visitor:VisitorPort): boolean
+disconnect(): void
PortInput
-unitInputPort: UnitInputPort
+accept(visitor:VisitorPort): boolean
+disconnect(visitor:VisitorPort): void
PortAm
-unitAmPort: UnitInputPort
+accept(visitor:VisitorPort): boolean
+disconnect(visitor:VisitorPort): void
PortFm
-unitFmPort: UnitInputPort
+accept(visitor:VisitorPort): boolean
+disconnect(visitor:VisitorPort): void
PortGate
-unitGatePort: UnitGatePort
+accept(visitor:VisitorPort): boolean
+disconnect(visitor:VisitorPort): void
PortOutput
-unitOutputPort: UnitOutputPort
+accept(visitor:VisitorPort): boolean
+disconnect(visitor:VisitorPort): void
<<interface>>
VisitorPort
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean
VisitorInput
-portInput: PortInput
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean
VisitorGate
-portGate: PortGate
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean
VisitorAm
-portAm: PortAm
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean
VisitorFm
-portFm: PortFm
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean
VisitorOutput
-portOutput: PortOutput
+visit(portInput:PortInput): boolean
+visit(portOutput:PortOutput): boolean
+visit(portFm:PortFm): boolean
+visit(portAm:PortAm): boolean
+visit(portGate:PortGate): boolean