SynthLab
Simulateur de Synthétiseur de son analogique

Cyrille FOLLIOT : Responsable documentation,
Julien NÉVO : Responsable projet,
Julien RICHARD-FOY : Responsable conception,
Maxime SIMON : Responsable qualité.

24 février 2011

Table des matières

1 Introduction

Ce projet, baptisé **SynthPro**, consiste à réaliser une application permettant de simuler numériquement les sons issus des premiers synthétiseurs analogiques, dits à synthèse soustractive.

Le principe de l'application est basé sur la modularité : l'utilisateur doit pouvoir assembler des modules sonores entre eux et entendre le résultat du montage en temps-réel. Il n'y a, *a priori*, pas de limitation quant au nombre de modules pouvant être ajoutés, et la manière dont ils sont connectés doit rester aussi permissive que possible.

1.1 L'équipe

1.1.1 Présentation de l'équipe

Notre équipe, portant le nom de **BackSynth Boys** (en l'honneur d'un certain *boys band*) est composée de quatre personnes, chacune ayant un rôle particulier à remplir dans la gestion du projet en lui-même, mais bien sûr également dans la conception et la production de l'application :

- Julien Richard-Foy : **responsable Conception**. Chargé de la bonne conception et structuration du projet. S'est occupé de la partie Métier, de la communication entre celle-ci et l'interface graphique, et l'interface graphique en elle-même ;
- Maxime Simon : **reponsable Qualité**. Chargé de la bonne formation des programmes écrits, aussi bien structurellement que syntaxiquement, et de la pertinence des tests effectués. S'est également occupé de l'interface et de la communication avec la partie Métier ;
- Julien Névo : **responsable Projet**. Chargé de faire les rapports d'avancement du travail auprès de M. Plouzeau. S'est occupé du moteur audio, de la génération sonore des VCO et VCF, et de divers modules ;
- Cyrille Folliot : **responsable Documentation**. Chargé de la qualité de la documentation produite. S'est également attelé au développement des modules et de la partie Métier.

1.1.2 Organisation du temps de travail

Quatre semaines nous étaient imparties pour mener à bien ce projet. Les trois premiers jours furent utilisés pour mettre en place l'architecture du projet et la conception de la partie Métier. Une fois cela fait, nous nous sommes répartis les rôles en fonction de nos affinités pour le travail à effectuer. Le travail de chacun a pu être fait de manière indépendante très rapidement.

Nous avons mis en place des **itérations**, très courtes, dans lesquelles étaient consignées les tâches que chacun devait accomplir. Le but étant qu'à la fin de chaque itération, le projet fonctionne toujours et soit à chaque fois agrémenté de nouvelles fonctionnalités. Une itération peut durer de 1 à 3 jours.

1.1.3 Logiciels et outils de développement

Outils de développement Notre projet est basé sur le *framework* Qt, dans le langage C++. Nous avons utilisé l'environnement fourni, QtCreator, qui fonctionne aussi bien sous Linux et Windows. Notre utilisateur de Mac OS X a utilisé Xcode.

Versionnement Nous avons choisi d'utiliser Git pour le versionnement de notre application, et ce pour trois raisons :

- SVN est connu de tous, et nous désirions utiliser un nouveau logiciel pour les comparer ;
- Git s'avère plus efficace à gérer les conflits entre les différents *commits* que SVN. Ce projet étant mené par 4 personnes, il était nécessaire de prévoir ce genre de situations ;
- GitHub, une interface *Web* de Git et liée au dépôt de fichiers, fut très pratique pour gérer le projet. Elle permet notamment de voir l'historique de chaque fichier, du projet en général, et du travail effectué par chaque personne.

Documentation Les schémas UML ont été mis en oeuvre sous BouML. La plupart des documents ont été écrits dans le wiki de GitHub, puis convertis au format LaTeX grâce à Pandoc.

2 Modèle Métier

2.1 Le synthétiseur

L'application que nous développons consiste à simuler numériquement le comportement des synthétiseurs analogiques, dits à synthèse soustractive, ainsi que l'aspect modulaire de leur utilisation. Chaque synthétiseur est composé de **Modules** que l'on pourra lier entre eux, tout en écoutant le résultat du montage en temps-réel.

2.2 L'interface

L'application possède un environnement graphique qui permet à l'utilisateur de visualiser l'intégralité des modules disponibles. Il aura également à sa disposition une surface qu'il pourra remplir à loisir en déposant des modules du type désiré. Les différents paramètres des modules peuvent également être modifiables en temps-réel.

2.3 Les modules

Le module est la partie atomique du montage. Il n'existe pas *a priori* de limite au nombre de modules que l'on peut créer. Un module dispose d'au moins une entrée et/ou une sortie. Certains modules, sources de flux, ne disposent pas d'entrée, comme les oscillateurs dont le but est de générer des ondes. D'autres, appelés *puits*, n'ont pas de sortie et sont utiles en fin de montage pour, par exemple, diriger le son vers une sortie audio ou afficher le résultat à la manière d'un oscilloscope.

2.4 Les entrées/sorties

2.4.1 Généralités

Un module possède au minimum une entrée et/ou une sortie (appelés également **ports**). La conception de base des synthétiseurs préconisait l'utilisation de démultiplexeurs et multiplexeurs afin de pouvoir, respectivement, multiplier une entrée en plusieurs sorties pour leur faire subir un traitement différent, ou au contraire mixer plusieurs entrées en une sortie.

Ce concept nous a paru quelque peu lourd et peu intuitif, et l'avantage de la simulation est qu'elle permet de ne pas se limiter au monde physique tel qu'il existe. Ainsi, nous avons choisi de nous passer de ces étapes intermédiaires.

Notre concept est le suivant : chaque entrée et sortie d'un module se duplique à partir du moment où l'une d'elle est utilisée. En conséquence, les modules apparaissent initialement dotés d'une seule entrée et/ou sortie, mais il suffit de la connecter pour en voir apparaître une nouvelle, libre. L'avantage est double : cela simplifie le montage qui n'est plus pollué par des multiplexeurs et démultiplexeurs, et chaque module ne dispose plus que du strict nécessaire, simplifiant encore le montage.

2.4.2 Les Gates

Les ports peuvent éventuellement être de type **Gate** (**Gate in** pour les entrées, **Gate out** pour les sorties).

Qu'ils soient de type Gate ou non n'apporte pas de contrainte particulière, mais implique une sémantique légèrement différente par rapport aux ports conventionnels. Les Gates sont utilisées principalement pour réagir à un front montant ou descendant et déclencher une action en conséquence. Ainsi, le Gate Out du module Keyboard (clavier virtuel) produira un front montant à chaque pression de touche et un front descendant à son relâchement. Branché à un module ADSR, cela lui permet de détecter chaque nouvelle note et déclencher la modulation du volume.

2.4.3 Les *buffers*

Le simulateur code les signaux analogiques par échantillonnage à 44100 Hz. Les modules traitent ensuite ces valeurs par lots. Chaque port d'entrée ou de sortie possède un *buffer* contenant le lot de valeurs à traiter.

2.5 Les câbles

Pour relier un port à un autre, l'utilisateur peut créer des câbles virtuels simplement en désignant le port source vers le port de destination. La seule contrainte imposée est qu'un port d'entrée doit forcément être relié à une sortie. Il est également tout à fait possible de créer des boucles dans le montage en « nourissant » l'entrée d'un module avec la sortie de ce même module, ou la sortie d'un module suivant dans le montage.

3 Conception — PIM

La figure 1 présente la majeure partie de notre modèle de la partie métier du système. La classe `SynthPro` représente le synthétiseur, il contient un ensemble de modules lesquels contiennent des ports permettant de créer des connections entre eux.

3.1 Moteur

La classe `Sequencer` gère le *dataflow* : l'opération `scheduleModules` trie le graphe des modules contenus dans le synthétiseur en fonction de leurs dépendances, et l'opération `process` appelle l'opération du même nom sur chaque élément de la liste de modules triée, de telle sorte que chacun réalise son traitement. Le tri des modules est un simple tri topologique, réalisé par un parcours en profondeur du graphe de modules : une première passe recherche les modules « puits » (ceux qui ne possèdent pas de port de sortie), puis *via* une opération `requirements` (de la classe `Module`) qui retourne la liste des modules connectés aux ports d'entrée d'un module, le séquenceur détermine un ordre d'exécution possible pour satisfaire toutes les dépendances. L'algorithme ignore les cycles (il ne visite pas deux fois un même nœud). Une classe `Clock` (non représentée sur le diagramme) appelle périodiquement, en tâche de fond, l'opération `process` du séquenceur.

3.2 Modules

Le *package* `module` contient l'ensemble des modules (ne sont représentés sur le schéma que le VCO et le VCF). La classe mère, `Module`, contient l'opération `process` dont l'implémentation appelle `fetchInput` (qui récupère les données des ports connectés aux ports d'entrée du module) puis `ownProcess`. Cette dernière est abstraite (patron de conception *Template Method*), laissant le soin à chaque module de définir sont traitement spécifique (p. ex. appliquer un filtre sur le signal).

3.3 Composants

Le *package* `components` définit les classes représentant les éléments constitutifs des modules, en particulier les ports. Les ports sont les composants qui permettent de relier les modules entre eux, et il est fréquent de vouloir relier une même sortie aux entrées de plusieurs modules (démultiplexage) ou connecter les sorties de plusieurs modules à l'entrée d'un même module (mixage). Pour cela nous avons introduit la notion de port « virtuel » : un tel port peut se répliquer à l'infini, permettant d'être connecté à plusieurs autres ports et réalisant automatiquement le travail de démultiplexage ou de mixage (cf. explication en section 2.4.1). Il est défini par la classe `VirtualPort`. L'attribut `replicable` indique si un port peut effectivement se répliquer et les opérations `connect` et `disconnect` créent et suppriment des connexions. Nous avons également introduit un attribut `gate` permettant de différencier les port ayant une fonction de *gate* (déclencheur) des autres. Les ports de sortie de type *gate* sont plus susceptibles d'être connectés à des ports d'entrée de type *gate*, mais le système n'interdit pas de les connecter sur les autres ports.

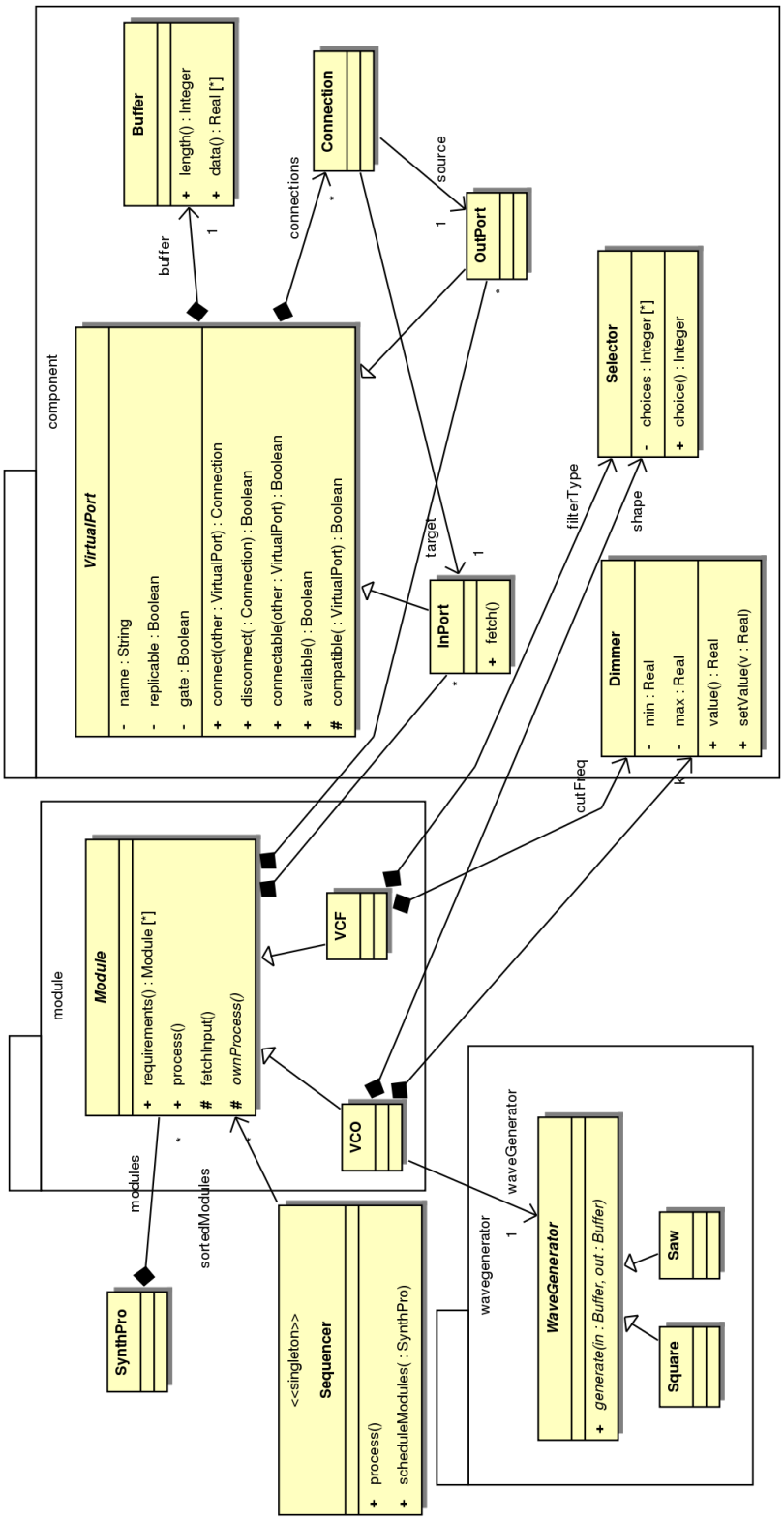

FIGURE 1 – Classes principales de la partie métier

La classe `InPort` hérite de `VirtualPort`, elle définit la notion de port d'entrée et contient une opération supplémentaire, `fetch`, qui copie les données des ports de sortie auxquels elle est reliée dans son propre *buffer* (elle réalise le mixage en faisant la somme de toutes ces données).

Enfin, les classes `Dimmer` et `Selector` définissent des composants permettant d'ajouter des réglages aux modules, sous la forme d'un potentiomètre (permettant d'indiquer une valeur dans un intervalle donné) ou d'un « sélecteur » (permettant de choisir une valeur dans un ensemble fini donné), respectivement.

Le *package* `wavegenerator` contient des classes définissant des générateurs de signaux de formes différentes : carrée, en dents de scie, etc. (patron de conception *Strategy*). Un *package* analogue, `filter` (non représenté sur le schéma) contient les différentes stratégies de filtres (passe-bas, passe-haut, etc.) utilisables par les modules.

3.4 Interface utilisateur

3.4.1 Modèle PAC

Le modèle PAC est utilisé pour rendre le logiciel interactif : pour chaque classe de la partie métier devant être représentée dans l'interface utilisateur, une classe de contrôle et une classe de présentation sont créées. La figure 2 montre un tel découpage pour les potentiomètres (classe `Dimmer`).

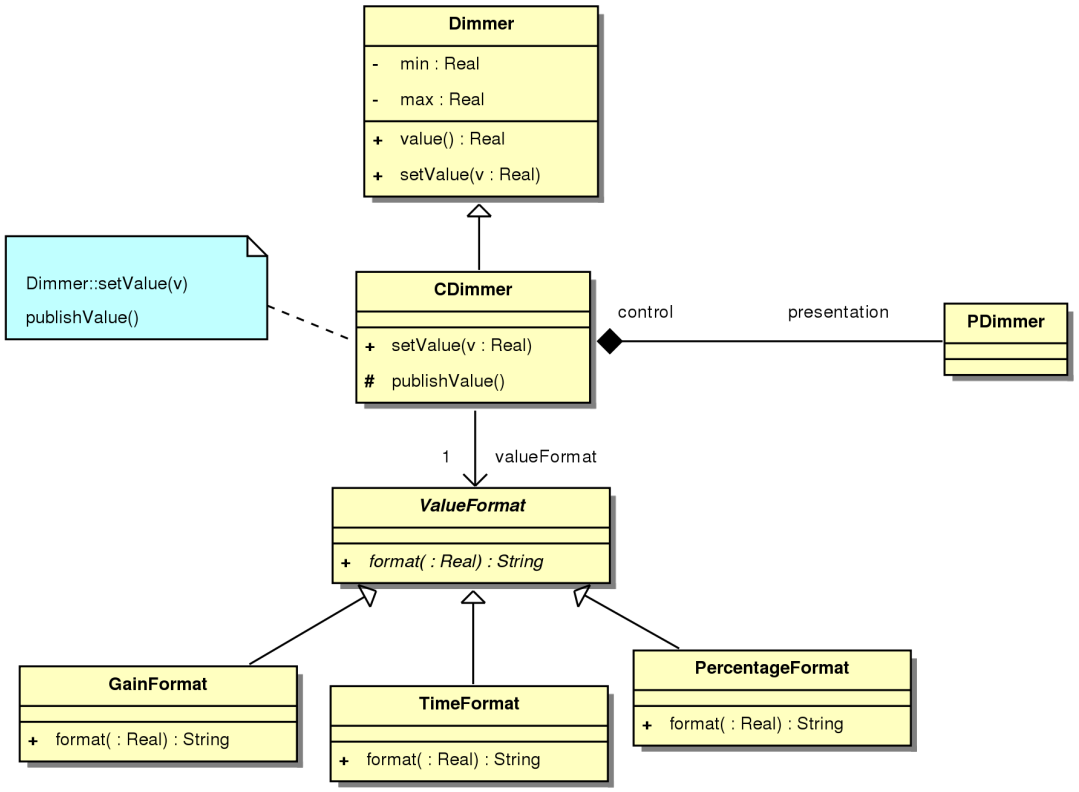

FIGURE 2 – Modèle PAC appliqué à la classe Dimmer

Nous avons choisi de réaliser un *Proxy* par héritage (plutôt que par délégation) pour définir le contrôle, d'une part parce que c'est plus pratique à écrire (nous ne redéfinissons que les opérations nécessaires) et d'autre part parce que cela permet d'intercepter également les appels internes de l'abstraction (avec une solution par délégation, si l'abstraction appelle une de ses propres opérations elle n'est pas interceptée par le proxy), en dépit du fort couplage que cela introduit entre les composants d'abstraction et de contrôle.

Dans le cas de la classe `Dimmer`, nous voulons pouvoir personnaliser l'affichage en fonction de l'interprétation de la valeur du potentiomètre. Pour certains modules cette valeur représente une fréquence, pour d'autres une amplification, ou encore un délai. Pour cela, le contrôle est associé à une fonction de mise en forme (réifiée par la classe `ValueFormat`), chargée de *mapper* une valeur numérique vers une représentation sous forme de chaîne de caractères. Ainsi le contrôle du module `VCO` configure le contrôle du réglage `k` de la fréquence du signal à générer en lui fournissant une fonction de *mapping* affichant la valeur en Hertz de la fréquence. Chaque contrôle de module fait de même avec chacun de ses potentiomètre pour personnaliser leur affichage. De cette manière, les informations sur la façon de présenter ces valeurs ne polluent pas l'abstraction.

3.4.2 *Drag and drop*

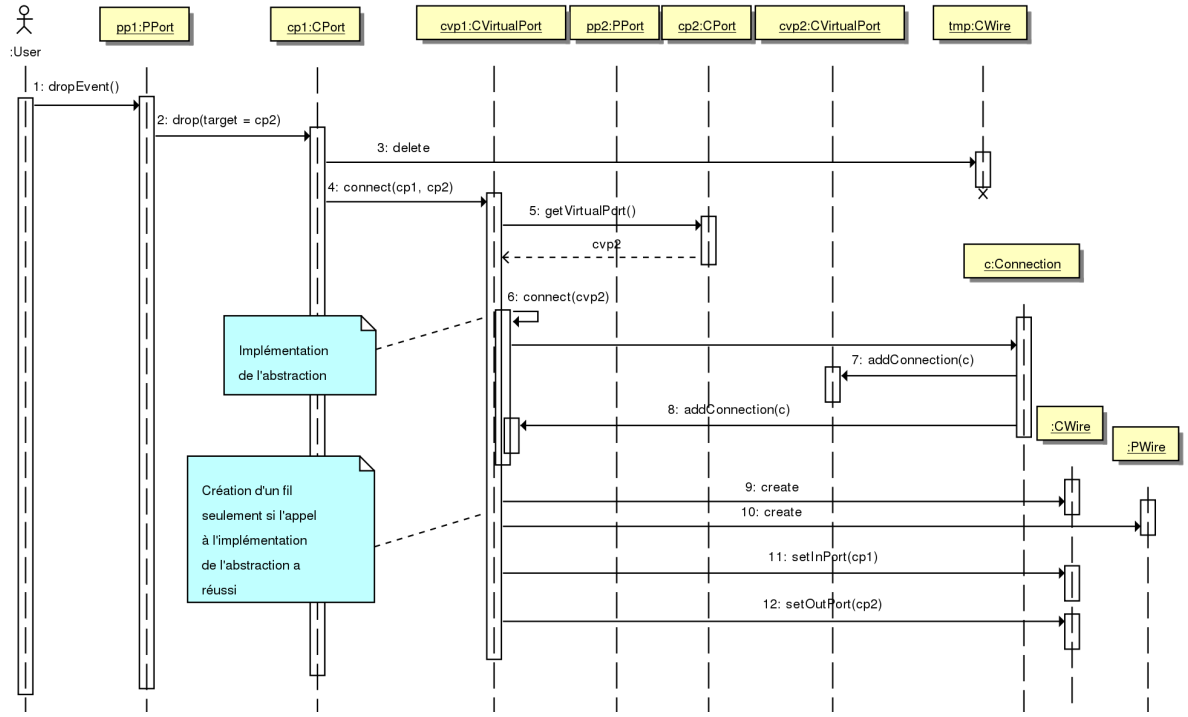

FIGURE 3 – Diagramme de séquence du drop

La figure 3 illustre les échanges entre les composants lorsque l'utilisateur termine une opération de *drag and drop* reliant un port d'un module à un autre pour les connecter.

Les classes `CPort` et `PPort` ont été ajoutées uniquement pour les besoins de l'interface utilisateur et ne correspondent pas à des classes dans l'abstraction, elles représentent la notion de port où l'utilisateur peut brancher des connexions (un `VirtualPort` peut contenir plusieurs ports). De même les classes `CWire` et `PWire` représentent la notion (graphique) de fil et n'ont pas d'équivalent côté abstraction.

(1) L'événement *drop* est capté par la présentation du port source (objet `pp1`) qui (2) transmet l'information à son contrôle, `cp1`, en lui indiquant le contrôle du port cible `cp2` (celui au-dessus duquel l'utilisateur a relâché la souris, quand il ne la relâche pas dans le vide). (3) Le contrôle `cp1` supprime le fil qui était créé temporairement durant le *drag and drop* puis (4) transmet l'ordre au contrôle de son port virtuel `cvp1` d'établir une connexion entre les deux ports. (6) La connexion est réalisée en appelant l'opération `connect` de l'abstraction (7, 8), et, selon son résultat (succès ou échec), un fil est créé entre les deux ports (9).

4 Conception — PSM

4.1 Choix technologiques

L'expérience croisée des *BackSynth Boys* nous a conduit à nous tourner vers le *framework* Qt, dont chacun apprécie l'API élégante et performante. Ce framework fournit surtout des outils pour la construction d'interfaces homme-machine, mais il est très riche et possède également des APIs pour la manipulation de collections, le traitement de documents XML, ou des fonctions donnant un accès simplifié à la carte son.

La meilleure façon de tirer parti de Qt est de programmer en C++, c'est donc le langage qui a été choisi pour le développement de SynthPro.

4.2 Conséquences sur l'architecture

4.2.1 Diagramme de classes de niveau PSM

La figure 4 représente les classes métier au niveau PSM. Il n'y a pas vraiment de changement profond par rapport au niveau PIM, les types de données de Qt sont utilisés (`QList`, `QString`, etc.), et des *signals* et *slots* apparaissent. Un *signal* est une notification qu'un composant émet pour informer d'un événement, un *slot* est une opération (*handler*) qui peut être appelée pour gérer cette notification. Le système de *signals* et *slots* est une implémentation proposée par Qt du patron de conception *Observer*.

Par exemple, la classe `VirtualPort` émet le *signal* `connectionsChanged` à chaque fois qu'une connexion ou une déconnexion a lieu. Ce signal est intercepté par le `SynthPro` qui demande au `Sequencer` de recalculer l'ordonnancement de ses modules.

Enfin, la classe `AudioDeviceProvider` fait son apparition, elle gère l'accès à la carte son en utilisant l'API QtMultimedia.

4.2.2 Héritage en diamant

Notre choix de faire hériter les contrôles des composants de leur abstraction a conduit à un problème d'héritage en diamant (cf. figure 5). Nous avons donc pu goûter aux joies de l'héritage virtuel en C++.

4.3 Conséquences sur l'interface utilisateur

Nous avons utilisé le *framework* GraphicsView de Qt pour gérer l'affichage des modules. Ce *framework* est conçu pour afficher des objets 2D dans une scène très efficacement.

4.4 Données, mesures et ordre de grandeur

4.4.1 Type des données

Afin de garder un maximum de précision lors du traitement des données, nous avons choisi d'utiliser des `double`. Une autre alternative aurait été d'utiliser des entiers, mais cela ne correspondait pas aux ordres de grandeur que nous souhaitions représenter (voir section suivante), à moins de normaliser et dénormaliser nos signaux selon le traitement, source de plus de complexité et peut-être même de pertes de performances.

4.4.2 Signaux entre les modules

Nos modules étant supposés simuler du matériel analogique manipulant des tensions, nous avons généré et traité des signaux de même dimension. Ainsi, un VCO produira un signal allant de -5V à +5V et une sortie Gate Out de 0V à 1V. De même, notre note musicale de référence est un DO de l'octave 4, correspondant à 0V. Une octave représente 1V, donc un DO–5 correspond à 1V, et un DO-3 à -1V.

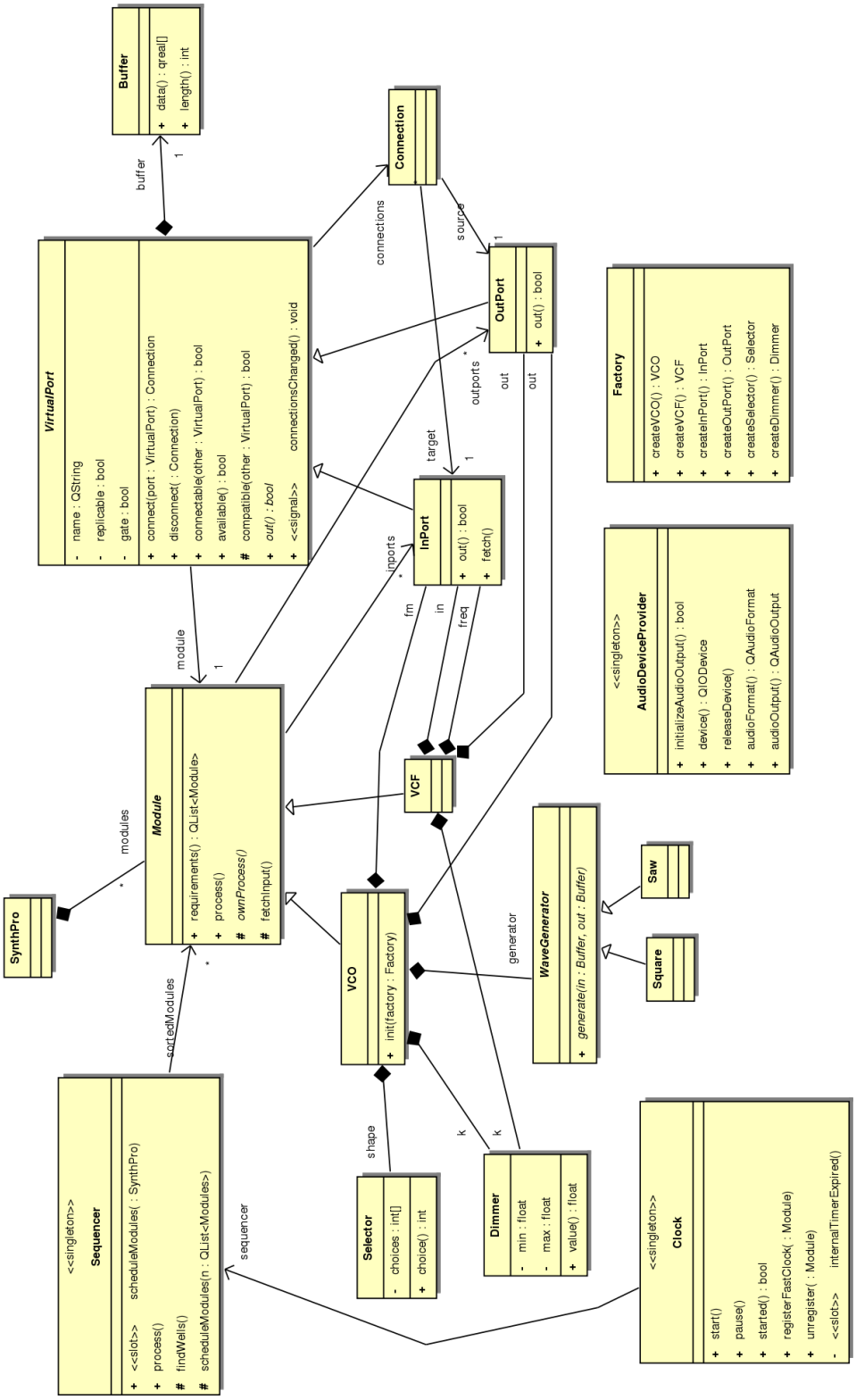

FIGURE 4 – Classes métier au niveau PSM

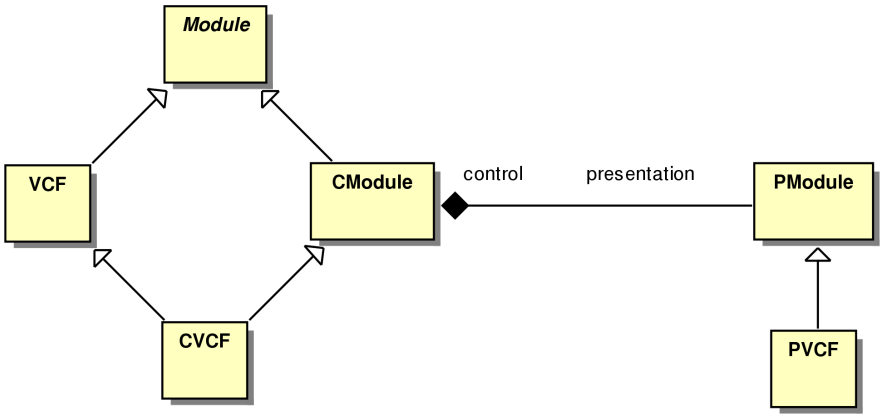

FIGURE 5 – Héritage en diamant sur le contrôle du module VCF

4.4.3 Signaux pour la carte son.

La sortie son utilisant du 16 bits, nous normalisons le signal au maximum de ce que la carte son peut supporter, soit de -32767 à 32767.

4.4.4 Gestion du débordement

Notre logiciel simulant du matériel, mais pas leur comportement électronique exact, il nous est tout à fait possible de générer des tensions que les synthétiseurs seraient (heureusement !) incapables de produire. Pour éviter tout débordement, chaque puits (qu'il représente une sortie audio ou non) dispose d'un test aux limites afin de saturer plutôt que de déborder. De plus, le VCO dispose lui aussi d'un limiteur en sortie, évitant le débordement des opérations arithmétiques, ce qui peut notamment se produire si les modules sont mis en boucle.

4.5 Le moteur audio

4.5.1 Présentation des différentes méthodes offertes

L'une des raisons pour lesquelles nous avons choisi Qt est la richesse de ses librairies. Deux d'entre elles permettent de gérer le son : Phonon et QtMultimedia. Phonon a été rapidement abandonnée car elle est de très haut niveau et plutôt adaptée pour jouer des fichiers audio et vidéo compressés.

En revanche, QtMultimedia semblait idéale car elle permet de fournir des buffers directement à la carte son de la même manière qu'on adresserait un fichier. QtMultimedia propose deux modes de fonctionnement : en *Pull* ou *Push*.

Nous avons d'abord testé le mode *Pull*, car il semblait bien adapté à notre séquenceur qui travaille lui aussi en *Pull*. Malheureusement, nos essais ont montré que si l'implémentation est remarquablement simple à mettre en place, la gestion de la latence est mauvaise : celle-ci est instable, le nombre d'octets à fournir varie de très faible (une centaine d'octets) à très important (16k). Il est bien sûr possible de stabiliser le tout en passant par un buffer intermédiaire, mais cela aurait augmenté la latence de l'application de manière insupportable pour l'utilisateur.

Nous avons alors testé le mode *Push*. À l'inverse, il faut demander régulièrement à la carte son si elle a besoin de données, à fournir au travers d'un QIODevice, composant d'entrée/sortie générique qui les achemine à la carte son. Le principal inconvénient de cette technique est qu'il faut constamment demander l'état du buffer de la carte son.

4.5.2 Base du moteur audio

Nous avons créé une classe `AudioDeviceProvider`, singleton qui se charge d'initialiser un QAudioOutput. Ce dernier permet de trouver un driver audio compatible avec le format désiré, indiqué dans un `QAudioFormat` (dans notre cas, du 44100Hz, 16 bits, mono). `QAudioOutput` fournit en plus, en mode *Push*, le `QIODevice` qui servira à envoyer les données à la carte son par l'intermédiaire de la méthode `write`. Enfin, le `QAudioOutput` nous renseigne sur le nombre d'octets dont a besoin la carte son par la méthode `bytesfree`.

Une classe `Clock`, également singleton et créée par nos soins, se charge d'appeler le séquenceur à intervalle régulier (toutes les 30 ms). Cela permet à tous les modules qui ont besoin d'être rafraîchis d'être appelés, même si aucun module `Speaker` n'est présent. L'oscilloscope, le File Writer, le WAV Looper et le Sampler notamment sont ainsi actualisés.

En revanche, si un module dit « *time critical* » (typiquement, un module `Speaker`) est placé sur l'interface, il s'enregistre auprès de la `Clock` dans une liste de timers rapides (*Fast Timers*), appelés à intervalle très rapide (5 ms). Une fois qu'un module est enregistré dans cette liste, le séquenceur n'est plus appelé par le thread non critique. Ce sera de nouveau le cas si les modules inscrits dans les timers rapides se désinscrivent.

Le traitement des *Fast Timers* est alors différent, comme expliqué plus loin.

4.5.3 *Push* : technique non utilisée

La technique de base, qui s'avérait être la plus propre d'un point de vue architecture, n'a pas été retenue dans l'implémentation finale, car nous n'avons jamais pu obtenir un résultat correct en sortie (son qui « saute »), malgré les traces que nous avons implanté et montrant un acheminement correct des informations.

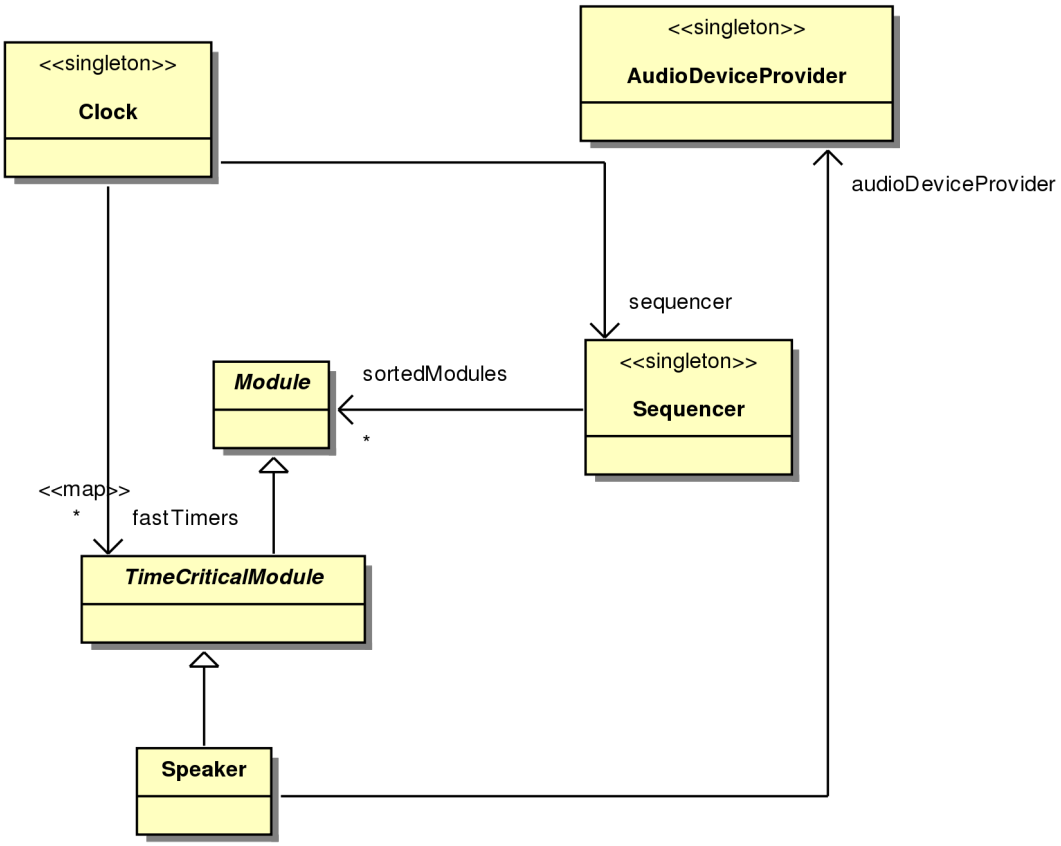

FIGURE 6 – Diagramme de séquence du moteur audio inutilisé car non fonctionnel

La figure 6 illustre cette technique. Lorsque le *Fast Timer* est écoulé, la `Clock` vérifie si la carte son a besoin d'être alimentée. Si c'est le cas, le séquenceur est appelé une fois. On recommence l'appel au séquenceur tant que la carte son a besoin de données (avec une limite arbitraire de 10 itérations). Lors de l'appel au séquenceur, celui-ci appelle la fonction `process()` du module `Speaker`.

Celui-ci dispose d'un buffer circulaire qui contient des données formatées pour être envoyées à la carte son (format char, little endian). L'appel à `process()` ayant au préalable mis à jour les entrées du module `Speaker`, il est important de les ajouter directement au buffer circulaire. On alimente ensuite la carte son avec les données possédées et dont elle a besoin. Si la quantité est insuffisante, au retour de l'appel la boucle de `Clock` rappellera de nouveau le séquenceur qui appellera de nouveau le `process()` de chaque module, dont `Speaker`.

La théorie et les traces fonctionnent parfaitement. Pour des raisons que nous n'avons pas pu déterminer, la pratique laisse à désirer, de nombreuses coupures venant s'intercaler dans le flux audio généré. Nous avons donc tenté une autre approche, moins propre d'un point de vue architecture car non symétrique, mais cependant fonctionnelle.

QtMultimedia étant une librairie très récente, il ne nous a pas été possible de trouver une explication à nos divers problèmes (mode *Pull* à latence instable, mode *Push* avec des pertes selon la technique) sur les divers forums visités.

4.5.4 *Push* : technique utilisée

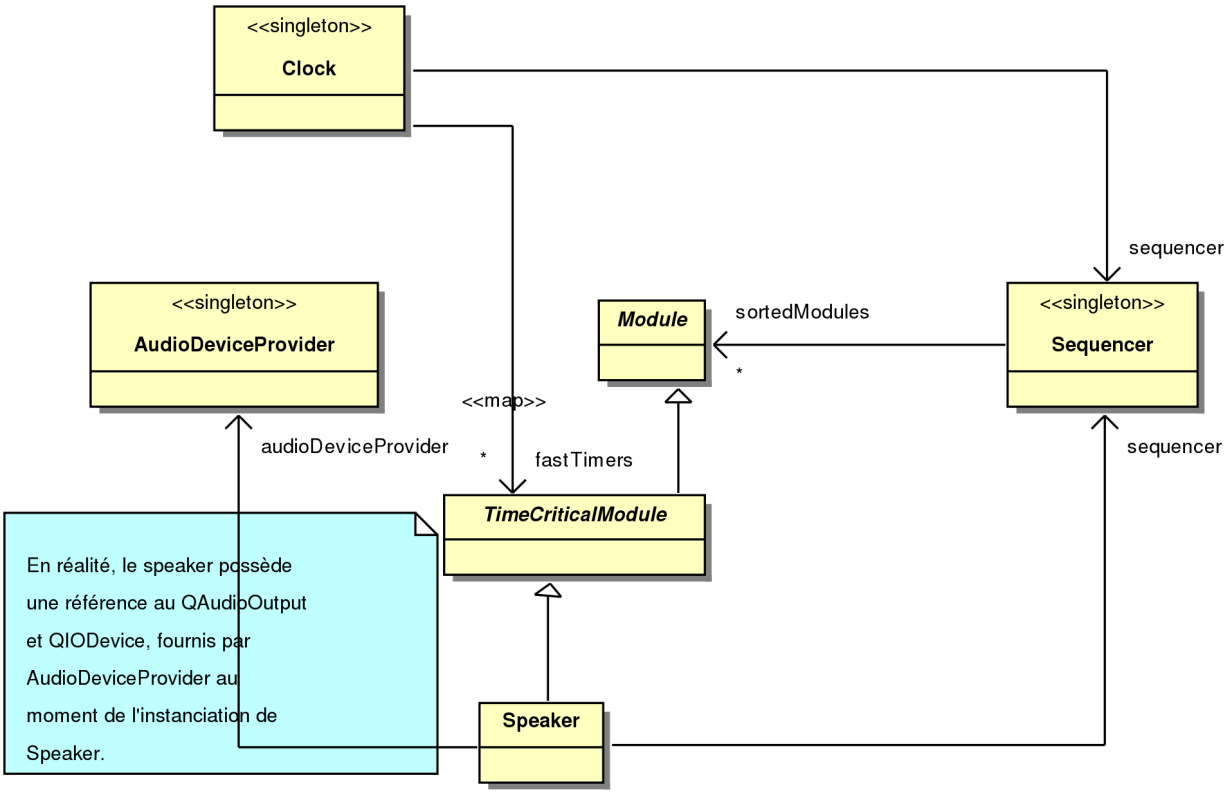

FIGURE 7 – Moteur Audio utilisé par l'application

La figure 7 illustre la technique utilisée, elle consiste à déléguer la gestion complète de la carte son au module `Speaker`. La `Clock` dispose toujours du même mécanisme de timers normaux/critiques, mais le timer critique appelle le *slot* `fastTimerExpired()` de chaque module. Cela permet de découpler l'appelant de l'appelé.

Une fois le slot `fastTimerExpired()` du `Speaker` appelé, on vérifie dans une boucle si la carte son a besoin de données. Si oui, on appelle le séquenceur à partir du `Speaker` et on envoie directement le buffer d'entrée du

module vers la carte son. Si celle-ci a encore besoin de données, on réitère la boucle, avec une limite arbitraire de 10 itérations. La figure 8 montre les ces échanges.

L'inconvénient de cette méthode est que, sans module critique, c'est la `Clock` qui appelle le séquenceur, mais avec un module critique, c'est le module lui-même qui appelle le séquenceur en fonction de ses besoins. L'appel au séquenceur est donc asymétrique. Afin que seuls les modules critiques puissent être appelés par `Clock`, et pour éviter que les modules aient à implémenter des méthodes qu'ils n'utiliseront jamais, nous avons dérivé `Module` en `TimeCriticalModule`, dont seules les instances implémentent le *slot* `fastTimerExpired`. En conséquence, les méthodes `registerFastTimer` et `unregisterFastTimer` de `Clock` n'accepteront que des `TimeCriticalModule`.

L'avantage de cette méthode est indéniable : elle fonctionne en pratique. La latence est très faible et le son stable.

5 Tests et Validation

5.1 Introduction

Afin d'assurer une stabilité et une fiabilité optimales, nous avons décidé d'effectuer des tests sur les principales classes de l'abstraction. Il s'agit de tests unitaires pour la plupart des composants (les `Ports` et l'`AudioDeviceProvider`) et le `Sequencer`, qui ont été possibles grâce à la mise en place de bouchons ; ainsi que des tests fonctionnels pour les modules. Pour cela, nous avons utilisé le *framework* de tests unitaires de Qt.

Au vu de la complexité de l'application et afin de pouvoir développer parallèlement les différentes sous parties du système, les tests ont également servi d'environnement de mise au point, évitant ainsi de démarrer un application pas forcément fonctionnelle afin de tester les nouveaux composants.

5.2 Tests unitaires et tests d'intégration

Notre choix d'un développement itératif nous a conduit à une intégration progressive des modules. La dépendance étant forte entre certains d'entre eux (par exemple entre un module et ses ports) un plan d'intégration s'est naturellement dégagé, il est explicité dans la figure 9

5.3 Moteur

5.3.1 Ports et connexions

Une batterie de tests a été écrite pour s'assurer que les classes élémentaires du système étaient valides, ces tests vérifient entre autres choses que :
- Les valeurs du *buffer* d'un port d'entrée connecté à rien sont toutes nulles ;
- Il est impossible de connecter deux ports d'entrée ensemble ;
- Il est possible de connecter un port d'entrée à un port de sortie ;
- Il est possible de déconnecter deux ports connectés ;
- Il est possible de connecter un même port *replicable* à plusieurs ports ;
- Pour les ports d'entrée, leur opération `fetch` copie effectivement (en les sommant) les données des ports auxquels ils sont connectés.

Ces tests nous ont également permis de détecter les éventuelles régressions introduites par les évolutions du code du moteur.

5.3.2 Séquenceur

Les tests du séquenceur ont consisté à vérifier que l'ordre d'exécution de l'opération `process` de modules assemblés de façon plus ou moins complexes satisfaisait leur dépendances.

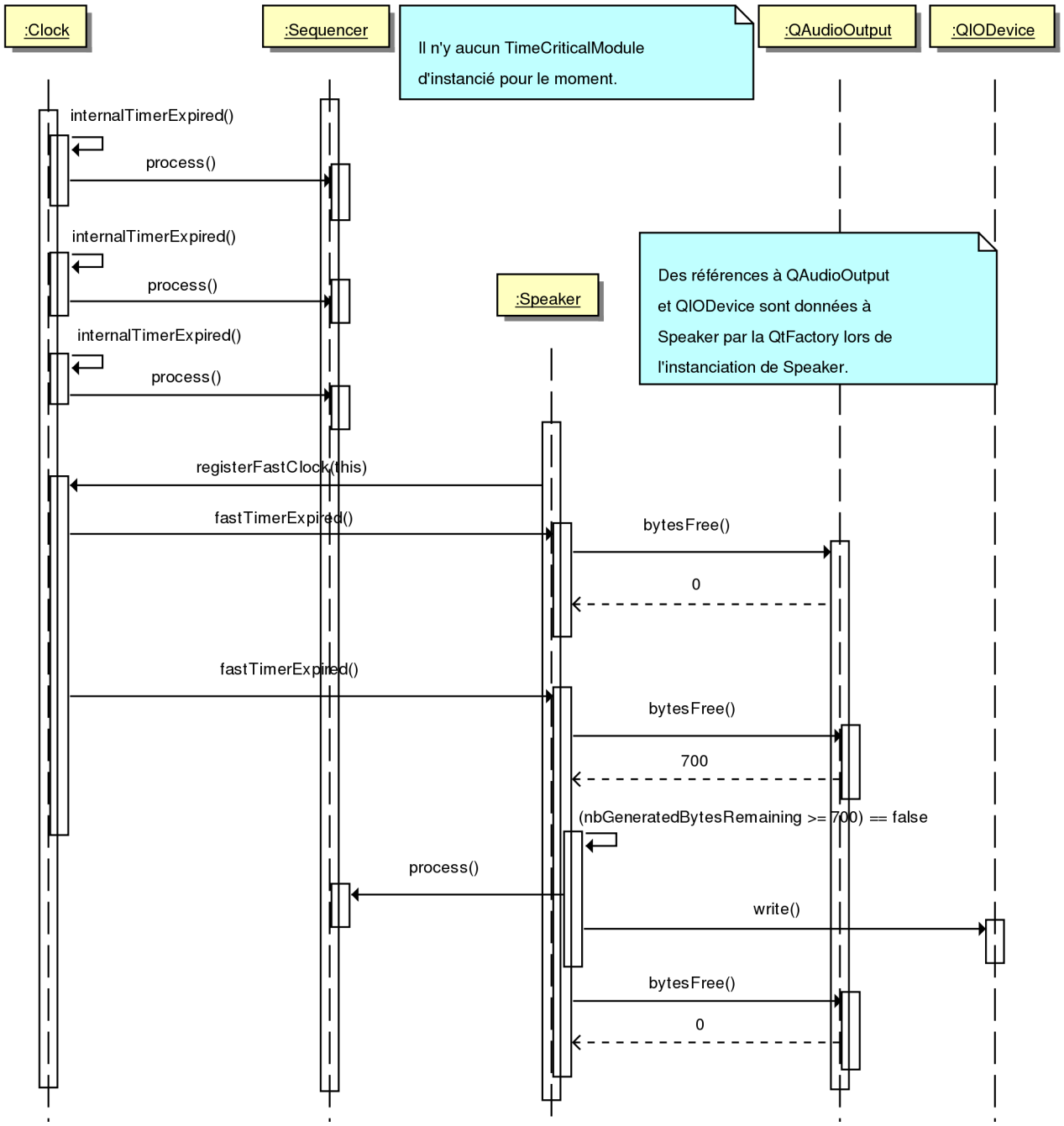

FIGURE 8 – Diagramme de séquence du moteur audio utilisé

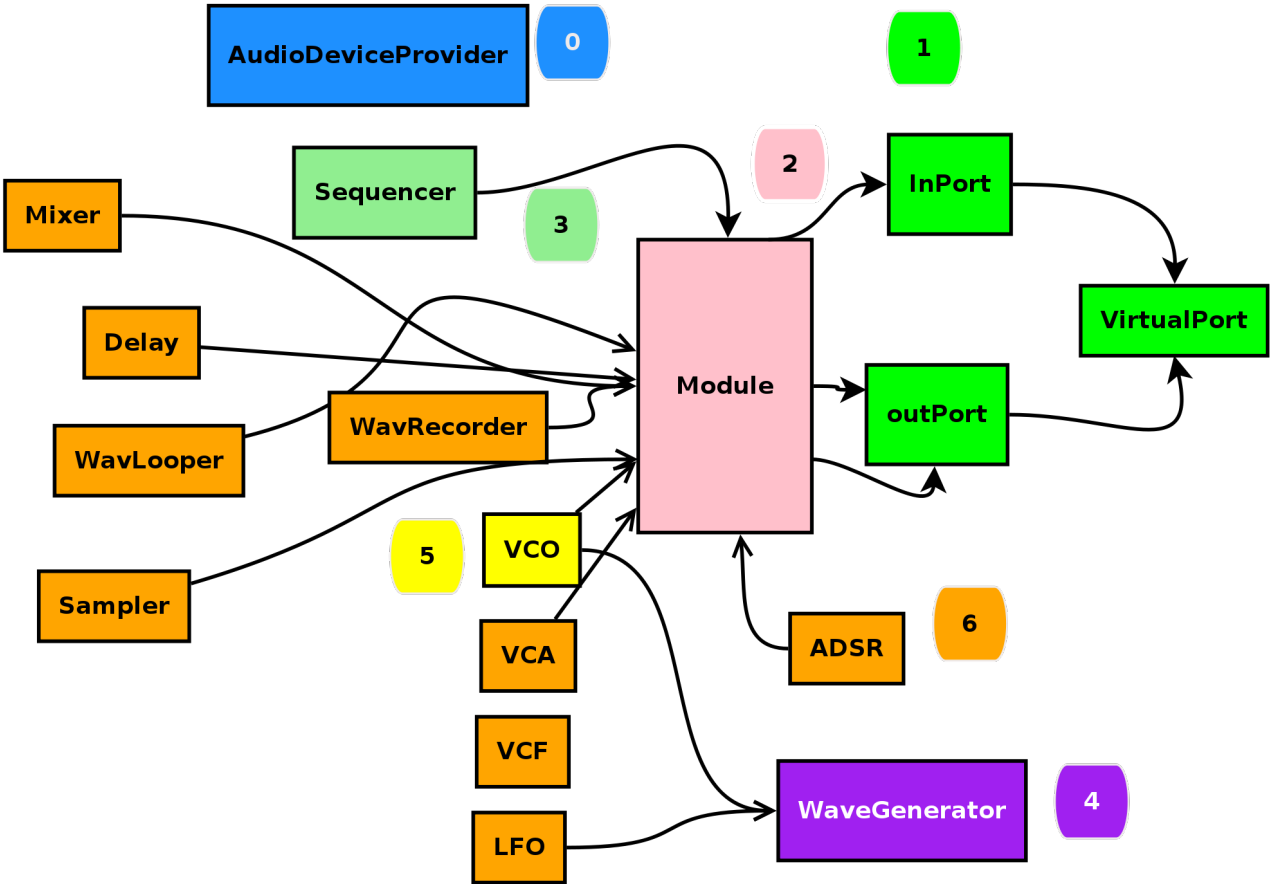

FIGURE 9 – Graphe d'intégration des composants testés

Nous avons écrits des *mocks* de modules à cet effet : leur traitement consistait simplement à écrire une trace dans un flux. Après l'exécution du traitement de chaque module par le séquenceur, le contenu du flux nous permet de retrouver l'ordre d'exécution.

La figure 5.3.2 illustre les types de montages de test réalisés et les sorties possibles attendues.

Par la suite des tests utilisant les vrais modules ont été ajoutés à ceux décrits ci-dessus.

5.4 Modules

5.4.1 VCO

Test VCO Pour ce module, nous avons testé :
- qu'un onde carrée demandée au `WaveGenerator` se retrouvait bien dans le port d'entrée d'un module *mock*. Ce test est effectué en mettant toutes les valeurs du buffer du mock dans un `QSet` et en constatant qu'au final, il ne contient que deux valeurs et que celles-ci sont opposées ;
- que la variation du `K` d'une unité faisait doubler la fréquence de l'onde obtenue ;
- que le sélecteur de forme d'onde sélectionnait effectivement l'onde demandée.

TestWaveGeneratorEmpty Avant de tester la génération des ondes sonores, nous avons dû vérifier qu'une génération simple fonctionnait correctement. Nous avons donc créé une `WaveGeneratorEmpty`, implémentation de `WaveGenerator`, qui ne sait que remplir un buffer nul. Ce test instancie donc ce générateur, lance la génération et vérifie que buffer de sortie contient bien uniquement des 0.

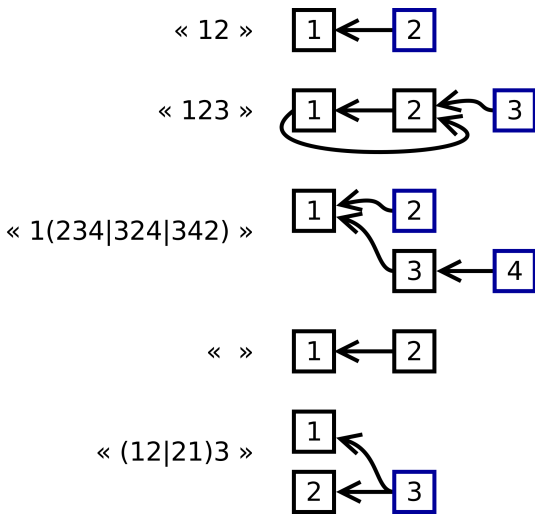

FIGURE 10 – Montages de tests pour le séquenceur. Les modules bleus sont des puits et les flèches représentent la relation de dépendance entre deux modules (dans le sens contraire du flot de données). Les expressions entre guillemets représentent les ordres d'exécution possibles.

TestWaveGeneratorSinus, TestWaveGeneratorSquare, TestWaveGeneratorSaw, TestWaveGeneratorTriangle
Tester que la production d'ondes est correcte n'est pas aisé. On peut passer par l'étude des signaux générés, mais cela demande un travail de traitement du signal important qui ne nous paraissait pas forcément utile, ni très pertinent : l'étude des signaux peut être bien plus sujette aux erreurs que la génération d'ondes elle-même.

Notre méthode de test a donc consisté à générer une onde et à la sauvegarder dans un fichier WAV grâce au module `WavRecorder`. L'écoute et la visualisation du signal sous un logiciel tel que Audacity ou Soundforge nous indique si le résultat paraît conforme.

Nous avons poussé l'étude plus loin en passant les échantillons sous un accordeur virtuel : la fréquence générée est alors affichée et il nous est alors possible de savoir si le signal a la fréquence désirée (figure 11).

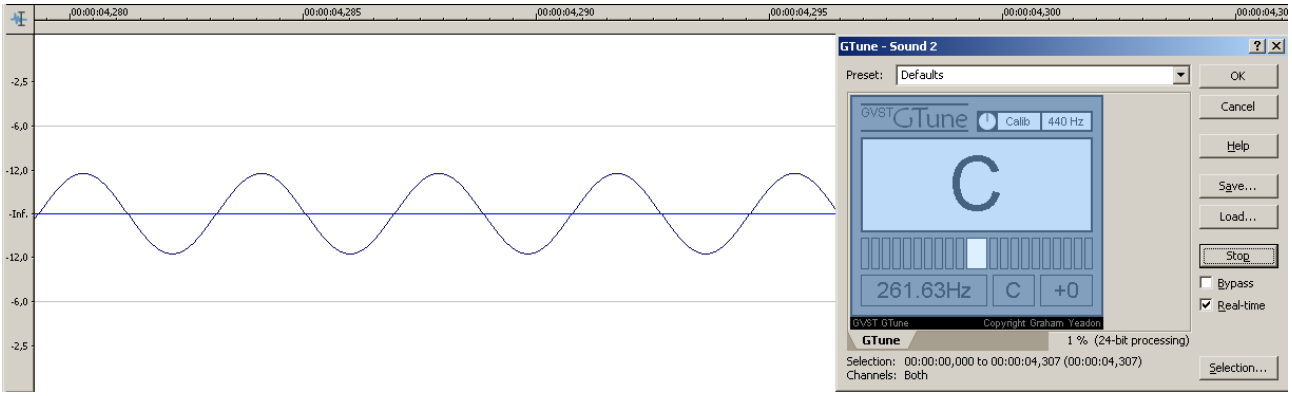

FIGURE 11 – Vérification des ondes générées grâce à d'autres logiciels

5.4.2 VCA

Le rôle du `VCA` étant de multiplier chaque valeur de son entrée par un coefficient, son test a consisté à le faire fonctionner avec un gain donné et à vérifier que les valeurs produites dans un *mock* étaient conformes à nos attentes.

5.4.3 VCF

Test VCF Le test du `VCF` consiste à lier une instance de `VCO` à un `VCF`. Le VCO ne produira qu'une onde de type `Empty`, donc ne produisant que des 0. Nous utilisons le filtre `FilterIncrement` qui se contente d'ajouter 1 à toute valeur du buffer d'entrée si elle est positive, ou –1 si elle est négative. Le test s'assure que le buffer de sortie du `VCF` ne contient que des 1.

Les filtres Il n'est pas facile de tester simplement les filtres. Tout comme pour les `WaveGenerators`, il faudrait décomposer le signal afin de tester son intégrité. Cependant, là où la génération fournie par les `WaveGenerators` pouvait être étudiée par la suite sous des logiciels tels que Soundforge ou Audacity, il n'en est pas de même pour les filtres. Nous avons donc décidé de nous fier à l'écoute du signal et à l'affichage produit par le module Oscilloscope.

5.4.4 ADSR

Nous avons testé ce module grâce au fait que son comportement est lié à ce qu'il reçoit dans son port `Gate`. Sur une « durée » de trois buffers, nous lui envoyons donc des valeurs 1, puis des zeros, puis des 1, ce qui correspond à un cycle ADSR complet. Les résultats des appels à `process` sont stockés dans un buffer puis analysés par parcours du dit *buffer* à la recherche de chaque segment du cycle. Les quatre valeurs obtenues sont ensuite comparées à celles données en entrée.

5.4.5 Mixer

Le test du Mixer est réalisé en plaçant des buffers constants en entrée de chaque…entrée et en constatant que le *buffer* de sortie est également constant avec pour valeur la moyenne des entrées.

5.4.6 WavRecorder

Le test du `WavRecorder` consiste à relier un `VCO` utilisant le `WaveGeneratorSquare`, produisant une forme d'onde carrée à période fixe, à l'entrée du `WavRecorder`, dans un fichier donné, en précisant un nombre défini d'itérations au `WavRecorder` afin qu'il ferme le fichier de sortie à l'issue de ces itérations.

Une fois l'enregistrement terminé, on ouvre le fichier en lecture, étudie le header du fichier WAV, et l'on s'assure que l'on ne trouve bien que deux valeurs différentes (front montant/descendant) jusqu'à la fin du fichier.

5.4.7 WavLooper

Le test du `WavLooper` consiste à charger un fichier `WAV` ne comportant que deux valeurs différentes consécutives, lui faire lire ce fichier, puis vérifier que la sortie du *buffer* ne comporte qu'une suite de ces deux valeurs.

5.4.8 Sampler

Pour effectuer et automatiser le test de ce module, nous utilisons sa capacité à être commandé par un port `Gate`. Le test consiste donc à faire fonctionner le module un nombre arbitraire de fois en lui donnant :
– sur le port Gate : alternativement 0, 1 ou 2 ;
– sur le port d'entrée, le buffer rempli d'entiers, chacun d'eux correspondant à sa place dans le buffer.
Le résultat attendu est alors :
– une période où le sampler est inactif ($Gate = 0$) ;
– une période d'enregistrement de l'entrée ;
– puis alternativement des périodes de lecture et de pause.
Le buffer de sortie est analysé à chaque pas du test, il doit être conforme aux attentes pour que le résultat du test reste valide.