TP4 : traitement et synthèse modulaire du signal audio
Chap. 1 : introduction et première mise en place

notions : toutes les notions du cours

Ce TP s'intéresse à la mise en oeuvre en POO/Java d'un système modulaire de synthèse et traitement du signal audio numérique. Il sera réalisé en groupe de deux étudiants, sur 3 séances + du travail hors séance. **Pensez à amener un casque audio doté d'une prise minijack en séance !**

Table des matières

Première partie

Introduction

De nombreuses approches existent pour synthétiser un signal audio ou traiter un signal audio préexistant. L'une d'entre elles consiste à construire pas à pas le signal final au moyen d'un ensemble de blocs de traitement du signal élémentaires interconnectés : des générateurs de signaux élémentaires sinusoidaux, carrés, etc ; des gains ; des délais ; des additionneurs ; des filtres ; des lecteurs de fichier audio ; des générateurs d'enveloppe ; etc.

La très grande modularité de cette approche la rend particulièrement pertinente pour la créativité : on construit un système de synthèse ou de traitement potentiellement, appelé *patch* en bon Franglais, potentiellement très complexe, au moyen de l'assemblage de modules très simples.

Le premier logiciel de synthèse/traitement audio de ce type fut le logiciel "Musiv-I", écrit dès 1957 aux Bell-Labs par Max Mathews , l'un des fondateurs du domaine de l'informatique musical, et son équipe.

Les principes mis en oeuvre par Max Mathews ont eu une très, très longue descendance : toute la série des programmes "Music-N", dont le célèbre "Music-V" ; mais aussi plus récemment des logiciels de création sonore et musicale tels que PureData, Max/MSP, SuperCollider, Chuck, Reaktor, etc. Ces principes sont en fait à la base de la plupart des synthétiseurs audio numériques, aussi bien *hardware* que *software*. Plus encore, au delà de l'informatique musicale, d'aucuns considèrent que les propositions de Max Mathews furent une graine pour tous les systèmes de traitement modulaire du signal. Ainsi, d'une certaine manière, "Music-I" entretient un lien de parenté conceptuel avec par exemple Matlab/Simulink. Un bel exemple de cas où une perspective artistique (ici : musicale) fut le ferment d'inventions scientifiques et technologiques majeures, ayant ensuite un impact large sur de nombreux autres domaines [1].

Ce sujet est (très) librement adapté des concepts introduits par la série des programmes "Music-N". Il les simplifie toutefois grandement. Par exemple :

- un seul flux de donnée, à fréquence audio, sera considéré, là où d'habitude un second flux à plus faible bande passante est mis en oeuvre pour le contrôle des paramètres des modules.
- nous ne considèrerons pas la notion de partition ou "score" ; on se limitera à "jouer" une seule fois le patch.
- nous n'ouvrirons pas la question relativement épineuse de l'ordre d'exécution des modules ; l'ordre d'exécution sera explicitement déterminé lors de la création du patch.
- nous ne considérerons que des signaux mono (une seule voie)
- etc.

Le sujet devrait toutefois permettre de saisir le "sel" de l'approche et d'étudier une architecture logiciel objet pas trop bête pour sa mise en oeuvre.

1. Pour votre gouverne, un autre exemple remarquable est celui du convertisseur CNA, également inventé au Bell Labs dans le contexte de travaux de recherche sur le signal sonore. Eh oui !

Deuxième partie

Notions de module et de patch

1 Module

Un **module** est une instance d'un **type de modules**. Un module a, pour l'essentiel :
— un certain nombre de **ports d'entrée**, par lesquels des échantillons entrent, en provenance d'un autre module
— un certain nombre de **ports de sortie**, par lesquels des échantillons sortent, à destination d'un port d'entrée autre module
— un certain nombre de **paramètres**, qui déterminent le comportement du module
— un **algorithme élémentaire**, qui calcule le ou les échantillon(s) en sortie à partir du ou des échantillon(s) en entrée et des paramètres.
Pour faciliter l'usage des modules, chaque module aura également un nom choisi par l'utilisateur.

Une **connexion** relie un port de sortie d'un module A et un port d'entrée d'un autre module B. Lorsqu'une connexion est établie, à chaque pas, l'échantillon produit en sortie par A sera envoyé sur l'entrée de B. Un port ne peut être connecté qu'une seule et unique fois.

Le tableau 1 liste certain des **types de modules** que l'on considérera. Les types de modules sont présentés plus en détail une annexe qui sera placée sur le site du cours d'ici peu.

2 Patch

Un patch est essentiellement une collection indexée (numérotée de 1 à N) de modules. Exécuter un pas d'un patch c'est exécuter une fois chacun de ses modules dans l'ordre. Ainsi, pour générer un signal de 2 secondes, il faut d'exécuter 44100*2 fois le patch.

Pour produire un signal audio, tout patch doit avoir au moins un module instance de `AudioDataReceiver` .

3 Quelques remarques

3.1 Fréquence d'échantillonnage des signaux

On considérera une fréquence d'échantillonnage unique, toujours fixée à 44100 Hz pour tous les signaux. Tous les fichiers audios chargés pour être traités, et tous les fichiers audio générés, seront à cette fréquence. Tous les modules et patchs fonctionneront à cette fréquence. Pour simplifier, le système réalisé ne sera pas "temps réel" : les calculs ne seront pas cadencés sur une horloge.

3.2 Type des échantillons

On travaillera avec des échantillons de type `double`. Pour votre information, sachez que les fichiers audios courants (par exemple les fichiers *.wav* que vous créerez) utilisent des échantillons de type `short` (entiers signés sur 16 bits). Lors de l'enregistrement des fichiers audio, une conversion de `double` vers `short` sera donc effectuée ; les classes fournies s'en occupent.

TABLE 1 – Quelques types de modules que l'on pourra considérer

Type	**Ports in**	**Ports out**	**Paramètres**	**Commentaires**
GenSineBasic	∅	1	f : fréquence a : amplitude	Générateur sinusoïdal basique. Génère et présente sur sa sortie un signal sinusoïdal de fréquence f et d'amplitude a
GenSine	2	1	f : fréquence a : amplitude	Générateur sinusoïdal contrôlable. Sa fréquence est prise sur son premier port d'entrée si il est connecté (vaut f sinon) ; son amplitude sur son second port d'entrée si il est connecté (vaut a sinon).
GenSquare	2	1	f : fréquence a : amplitude	Générateur carré. Sa fréquence est prise sur le premier port d'entrée si il est connecté (vaut f sinon) ; son amplitude sur le second port d'entrée si il est connecté (vaut a sinon).
Constant	0	1	c : constante	Présente en sortie des échantillons identiques de valeur c
Additionner	N	1	∅	Présente en sortie la somme de ses N ports d'entrée
Gain	1	1	g : gain	Présente en sortie son entrée multipliée par g
Splitter	1	N	∅	Présente sur ses N ports de sortie une copie de son port d'entrée
Delay	1	1	d : délais (nb ech)	Présente sur son port de sortie le signal reçu en entrée retardé de d échantillons
BandPassFIR	3	1	N : ordre f_r : fréq. de résonance $width$: largeur	Filtre FIR passe bande d'ordre N, contrôlable. Ses $N+1$ coefficients sont calculés en fonction de la fréquence de résonance f_r et de la largeur de bande $width$. Si son port d'entrée 2 (resp. 3) est connecté, il contrôle la fréquence de résonance (resp. la largeur de bande).
AudioDataProvider	0	1	∅	Présente en sortie les échantillons contenus dans un conteneur d'échantillons (instance de la classe `AudioData` fournie). Le conteneur d'échantillon sera chargé depuis un fichier audio.
AudioDataReceiver	1	1	∅	Inscrit dans un conteneur d'échantillons (instance de la classe `AudioData` fournie) les échantillons qu'il reçoit. Les transmet également sur sa sortie sans modification. Après le calcul du patch, les échantillons peuvent être écrit dans un fichier audio, ou envoyés sur les haut-parleurs. Tout patch doit avoir au moins un module AudioDataReceiver, sans quoi il ne produit rien.

3.3 Normalisation des signaux audio

La norme voudrait que les échantillons audio de type `double` générés soient compris entre $-1.$ et 1 : une valeur d'échantillon `double` 1.0 correspond, suivant cette norme, à l'amplitude maximale. Toutefois, il est relativement délicat d'étalonner un patch de telle sorte qu'il génère des échantillons à une amplitude correcte (inférieure à 1.0 et suffisamment grande pour être audible). Dans ce TP, pour faciliter, on décide donc que les signaux audio seront normés juste avant de les enregistrer dans un fichier ou de les écouter, en ramenant leur amplitude à 1.0. Les classes fournies réalisent cette normation pour vous.

Troisième partie

Premier galop

L'objectif de cette partie est de mettre en place 2 types de modules afin de générer un tout premier son : le module `GenSineBasic` et le module `AudioDataReceiver`. On réalisera ensuite une première version de la classe `Patch`. En matière de génie logiciel, il s'agit de pouvoir tester au plus tôt une première mise en oeuvre du problème, qu'on raffinera par la suite.

4 Première version des classes `ModuleAbstract`, `CommunicationPort` et `Connexion`

Vous l'aurez compris, tous les modules, quel que soit leur type, partagent un grand nombre de propriétés et de comportements. Bien évidemment, on va regrouper ces propriétés et comportements communs dans une super-classe abstraite : `ModuleAbstract`. Cela permettra en outre d'avoir recours au polymorphisme dans notre système. La classe `ModuleAbstract` s'appuiera sur deux petites classes : `CommunicationPort` et `Connexion`. Ces trois classes sont décrites ci-après. La figure 1 présente un diagramme UML de ces classes.

4.1 La classe `CommunicationPort`

Une instance de la classe `CommunicationPort` représente un port de communication (indifféremment en entrée ou sortie) d'un module.

Ses attributs sont :
— Une référence vers le `ModuleAbstract` qui possède ce port.
— Un entier : le numéro du port du module.
— Une référence vers la `Connexion` (voir ci-dessous) auquel ce port est connecté. Elle est `null` si le port n'est pas actuellement connecté.
— Un `double` : la valeur actuelle de l'échantillon présent dans le port.
Ses méthodes sont :
— Un constructeur, bien sûr. Le port n'est pas connecté lors de sa création.
— Une méthode `void setValue(double v)`.
— Une méthode `double getValue()`.
— Un accesseur `boolean isConnected()` - no comment !
— Un accesseur et un modifieur sur la `Connexion`.
— D'autres accesseurs si cela vous semble pertinent.

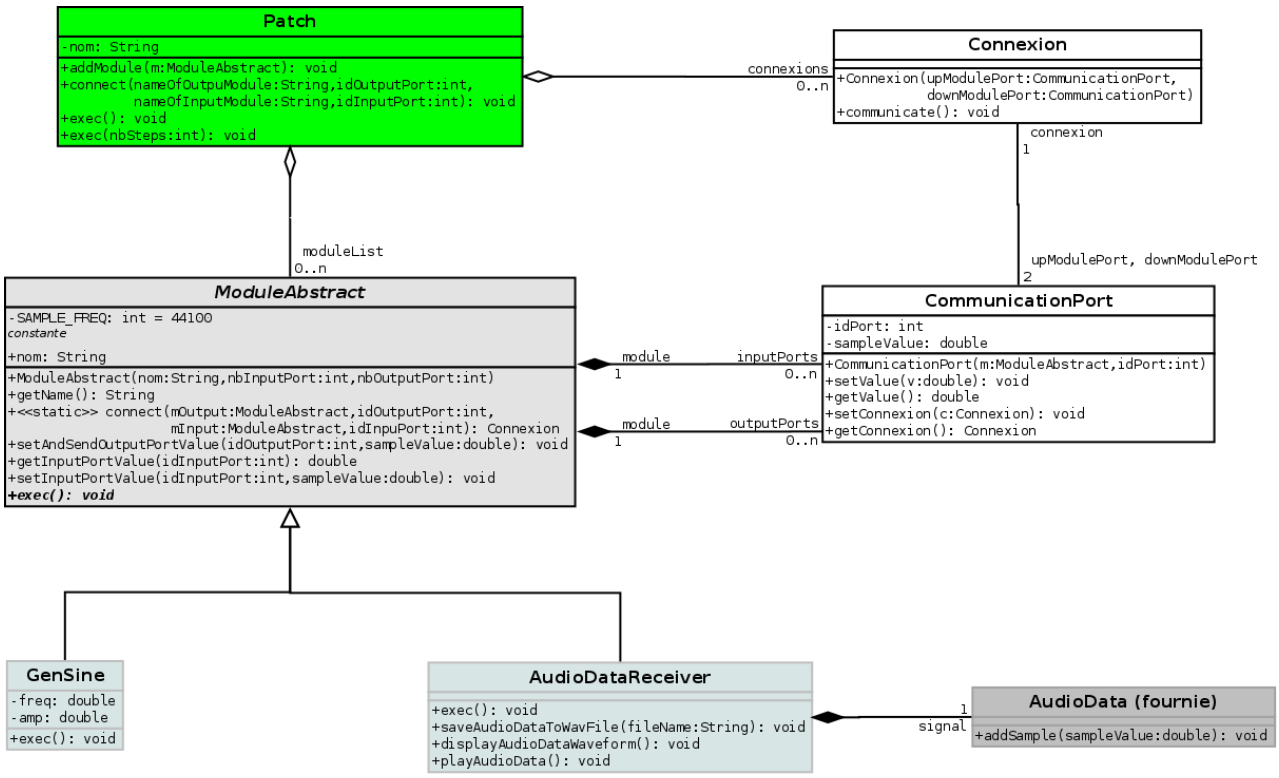

FIGURE 1 – Diagramme UML des premières classes

4.2 La classe `Connexion`

La classe `Connexion` permet de représenter une connexion (un fil, si vous voulez) entre un port de communication en sortie (amont) et un port de communication en entrée (aval).

La classe `Connexion` est simple :
- Deux attributs `CommunicationPort upModulePort; CommunicationPort downModulePort;` qui référencent les ports des modules reliés en amont et en aval par la connexion.
- Un constructeur, qui initialise tous les attributs.
- Une méthode `void communicate()` qui envoie la valeur de l'échantillon présent dans le port de sortie du module amont `upModulePort` vers le port d'entrée du module aval `downModulePort`.
- Des accesseurs sur les ports connectés, si cela vous semble pertinent.

4.3 Première version de la classe abstraite `ModuleAbstract`

Propriétés :
- une constante `SAMPLE_FREQ`, fixée à 44100 : la fréquence d'échantillonnage est fixée une fois pour toute à 44100 Hz pour tous les modules (... et tous les patchs...) de notre système [2].
- `nom` : le nom du module, choisi à la création
- `inputPorts` : un conteneur de `CommunicationPort`, dont la taille est déterminée à la création (c'est le nombre de port d'entrée du module) et qui est initialisé à la création.
- `outputPorts` : un conteneur de `CommunicationPort`, dont la taille est déterminée à la création (c'est le nombre de port de sortie du module) et qui est initialisé à la création.

Vous choisirez le type des attributs `outputPorts` et `inputPorts`. Faut-il plutôt opter pour un tableau

2. Pour se rappeler comment déclarer une constante en Java, revoir le paragraphe "static" de la fiche pédagogique 2.

Java de `CommunicationPort` ou pour une collection ? Si c'est une collection, faut-il plutôt une `List<>` ou un `Set<>` ? Si c'est une `List<>`, vaut-il mieux une `ArrayList<>` ou une `LinkedList<>` ? A vous de voir - en considérant le fait que les ports sont numérotés, et qu'il faudra pouvoir retrouver rapidement le *iieme* port d'entrée ou de sortie du module...

Opérations :

- **Constructeur** : détermine le nom du module, le nombre de ses ports d'entrée et de sortie et initialise tous ces ports. A la création du module, le module n'est pas connecté : la `Connexion` de ses ports de communication vaut `null`.
- `String getName()` : no comment !
- `static Connexion connect(ModuleAbstract mOutput, int idOutpoutPort, ModuleAbstract mInput, int idInputPort);` : crée un objet `Connexion`,et le monte entre le port de sortie numéro `idOutputPort` du module `mOutput` et le port d'entrée numéro `idInputPort` du module `mInput`. Cette méthode retourne la `Connexion` créée. Pourquoi est-il adéquat que cette méthode soit statique ?
- `void setAndSendOutputPortValue(int idOutputPort, double sample)` : enregistre le nouvel échantillon `sample` dans le `CommuncationPort` de sortie numéro `idOutputPort` du module et, si ce port est connecté, envoie cet échantillon vers le port d'entrée du module aval au moyen de la `Connexion` connectée à ce port. Pour envoyer l'échantillon, on utilisera bien sûr la méthode `communicate` de la classe `Connexion`.
- `double getInputPortValue(int idInputPort)` : retourne la valeur courante du `CommunicationPort` d'entrée numéro `idInputPort`
- `void setInputPortValue(int idInputPort, double value)` : modifie la valeur courante du `CommunicationPort` d'entrée numéro `idInputPort`
- `void exec()` : la fonction d'exécution du module.

5 Votre premier son ! Modules `GenSineBasic` et `AudioDataReceiver`

On implante maintenant deux premières classes concrètes de modules `GenSineBasic` et `AudioDataReceiver` afin de générer un premier signal sonore.

5.1 Le module `GenSineBasic`

Un module `GenSineBasic` est un générateur de signaux sinusoïdaux. Il n'a aucun port d'entrée, et un unique port de sortie. Il a deux paramètres : la fréquence f (en Hz) et l'amplitude amp du signal sinusoïdal à générer, tous deux initialisés à la construction et jamais modifiés.

La fonction d'exécution de `GenSineBasic` réalise les actions suivantes :

- Elle élabore le nouvel échantillon e suivant la formule $e = amp{*}sin(2{*}PI{*}f{*}n/SAMPLE\_FREQ)$, avec n l'indice du pas de temps - c'est à dire le nombre d'appels de la fonction d'exécution du module depuis le début. Il sera sans doute nécessaire de stocker n dans l'objet...
- Affecte e au port de sortie (unique) du module et, si ce port de sortie est connecté, envoie cette valeur vers le port du module aval, à l'autre bout de la `Connexion`.

5.2 Le module `AudioDataReceiver`

Un module `AudioDataReceiver` ne fait pas de traitement proprement dit : son rôle est de stocker les échantillons reçus dans son port d'entée dans un conteneur d'échantillons (une instance de la classe `AudioData`, fournie). Plus tard, le module peut enregistrer ces échantillons dans un fichier audio, les jouer (envoi sur la carte son) ou encore visualiser la forme d'onde.

Pour qu'un patch soit pertinent, c'est à dire pour qu'on puisse exploiter (écouter, enregistrer, afficher...) le signal qu'il produit, il faut qu'il possède au moins une instance de la classe `AudioDataReceiver`.

5.2.1 La classe `AudioData` fournie.

La classe `AudioData`, fournie dans la librairie Java `phelmaaudio.jar`, est essentiellement un conteneur d'échantillons audio. Elle permet notamment de :
- Ajouter un échantillon à la fin de la liste d'échantillons
- Récupérer un échantillon donné, par son index dans le signal
- Charger les échantillons depuis un fichier audio (format *wav*, 44100 Hz uniquement)
- Enregistrer les échantillons dans un fichier audio (format *wav*, 44100 Hz uniquement)
- Ecouter le signal (envoyer les échantillons sur la carte son de la machine)
- Visualiser le signal à l'écran (visualisation de la forme d'onde)

Pour savoir comment utiliser la classe `AudioData`, allez voir sa documentation qui a été mise en ligne sur le site du cours.

Récupérez ensuite sur le site du cours la librairie Java `phelmaaudio.jar` (qui contient la classe `AudioData` et d'autres choses). Ajoutez cette librairie à votre projet Eclipse (sans oublier de l'ajouter au "build path" du projet).

5.2.2 La classe `AudioDataReceiver`.

On implante maintenant la classe `AudioDataReceiver`.

Un module `AudioDataReceiver` a un unique port d'entrée et un unique port de sortie. Il possède en attribut un conteneur d'échantillons, instance de la classe `AudioData`, qui est instancié à la création de l'objet.

La fonction d'exécution de `AudioDataReceiver` réalise les actions suivantes :
- Ajout de l'échantillon présent dans son unique port d'entrée à son conteneur d'échantillons `AudioData`.
- Copie de cet échantillon dans le port de sortie (unique). Si ce port de sortie est connecté, la valeur de l'échantillon sera ainsi envoyé vers le module aval, sans modification. Ainsi, la classe permet de réaliser un *bypass* entre entrée et sortie. Elle peut donc être utilisée aussi bien "tout en bas" d'un patch pour récupérer ce qu'il génère, ou "au mileu" d'un patch pour "écouter" ce qui s'y passe.

Votre classe `AudioDataReceiver` possèdera également les méthodes suivantes, qui seront toutes réalisées par délégation à l'attribut `AudioData` :
- `void saveAudioDataToWavFile(String audioFileName)` : enregistre le signal contenu dans l'`AudioData` dans un fichier son nommé `audioFileName)`.
- `void displayAudioDataWaveform()` : affiche la forme d'onde à l'écran.
- `void playAudioData()` : envoie le signal sur la carte son.

5.3 Testons !

Ecrire un programme de test qui :
— Crée un module `GenSineBasic` (fréquence 442 Hz ; amplitude 1.0) nommé *gen* et un module `AudioDataReceiver` nommé *output*.
— Connecte le port de sortie du `GenSineBasic` au port d'entrée du `AudioDataReceiver`
— Génère 5 secondes de son (calcul de $44100 * 5$ pas)
— Affiche la forme d'onde du signal résultant, le sauve dans un fichier audio *test*1.*wav*, écoute le signal (oui, vous avez raison, ce n'est pas très joli pour le moment !)

Pour tester le fonctionnement du *bypass* du module `AudioDataReceiver`, modifier ce programme pour ajouter un second module `AudioDataReceiver` en aval du premier et vérifier que le même son est bien généré.

6 Première version de la classe `Patch`

En première approche, un *patch* est un conteneur (une liste) de modules interconnectés ; *exécuter* un patch, c'est exécuter chacun de ses modules. L'ordre d'exécution des modules sera, simplement, l'ordre d'ajout des modules au patch [3].

6.1 La classe Patch

Votre classe `Patch` comportera pour le moment *au minimum* les attributs :
— `String nom` : le nom du patch, choisi par l'utilisateur.
— La liste de modules du patch.
— Un ensemble contenant les `Connexion` du patch (dont on se servira ultérieurement).
La classe `Patch` comportera pour le moment *au minimum* les méthodes :
— Constructeur. Le patch est initialement vide (aucun module).
— Ajout d'un module : `void addModule(ModuleAbstract m)`.
— Connexion de deux modules : `void connect(String nameOfOutputModule, int idOutputPort, String nameOfInputModule, int idInputPort)`. Cette méthode crée une `Connexion` entre le port de sortie numéro `idOutputPort` du module de nom `nameOfOutputModule` et le port d'entrée numéro `idInputPort` du module de nom `nameOfInputModule`. La connexion créée est ajoutée à l'nsemble des connexions. On utilisera bien sur la méthode `connecter()` de la classe `Module`.
— Exécution d'un pas de calcul du patch : `void exec()`.
— Exécution d'un plusieurs pas de calcul du patch : `void exec(int nbStep)`.

Vous l'aurez remarqué, la méthode `connect()` prend en paramètre des *noms de modules*, plutôt que des références de type `ModuleAbstract`. Ce choix permettra que, pour connecter les modules, on utilise des noms (signifiants) plutôt que des références. Plus généralement, dans votre programme principal, les modules seront manipulés par nom après avoir été créés. Bien évidemment, ce choix implique que deux modules ne peuvent avoir le même nom dans un patch...

6.2 Modification du programme de test

Modifiez le programme de test de telle sorte que les modules soient maintenant ajoutés à un patch, et que ce soit le patch qui soit exécuté.

3. Comme indiqué en introduction, ce choix simpliste n'est pas conforme aux usages des "vrais" programmes de traitement et synthèse modulaire du signal audio. Il sera toutefois suffisant pour ce TP...