TP4 : traitement et synthèse modulaire du signal audio
Chap. 2 : évolutions

notions : toutes les notions du cours

Ce document est la suite du TP4 "synthèse et traitement modulaire du signal audio". **Pensez à amener un casque audio doté d'une prise minijack en séance !**

Table des matières

Première partie

Evolution

Maintenant qu'une première base est en place, nous pouvons faire évoluer le programme vers un code à la fois plus correct et plus complet.

Chacune des sections qui suivent sont (pour l'essentiel) indépendantes ; vous pouvez donc les réaliser dans un ordre (plus ou moins) quelconque.

1 Les types de modules

Notre variété de types de modules n'est pour le moment pas suffisante pour faire des patchs intéressant. Il faut aussi d'autres générateurs (dont, pour le traitement audio, un générateur capable d'envoyer dans le patch des échantillons chargés depuis un fichier audio préexistant), des gains, des délais, des filtres, etc.

Un "vrai" programme de synthèse et traitement du signal audio propose en général une bonne centaine de types de modules. Sans aller aussi loin, vous implanterez pas à pas les modules spécifiés à l'annexe **??**.

Conseil : Inutile de courir ! Pour choisir les modules à implanter, choisissez l'un des patch exemples proposés à l'annexe B "Exemples de patch", puis implantez les modules nécessaires à ce patch.

2 Complétude de l'API des classes `ModuleAbstract` et `Patch`

En POO, lorsqu'on conçoit et implante une classe, une phase importante consiste à viser la "complétude de son API" (Application Programming Interface). Il s'agit de viser à ce que la liste de ses méthodes représente bien tous les savoir-faire de la classe, et rende la classe facile à utiliser.

Dans cette partie, vous allez compléter les classes déjà implantées en ce sens. Nous proposons les méthodes suivantes - mais vous pouvez, si cela vous semble pertinent, en ajouter.

Pour la classe `ModuleAbstract` :
- `void reset()` : méthode abstraite qui "remet à zéro" le module. Exemples : Remettre à zéro un module quelconque, c'est mettre à 0. les valeurs des échantillons dans tous ses ports. Remettre à zéro un `AudioDataReceiver`, c'est *en plus* effacer les échantillons stockés dans son `AudioData`.
- `int getNbInputPorts()` : no comment !
- `int getNbOutputPorts()` : no comment !
- `boolean isConnectedInputPort(int inputPortId)` : no comment !
- `boolean isConnectedOutputPort(int inputPortId)` : no comment !
- `List<ModulePort> getUnconnectedInputPorts()` : no comment !
- `List<ModulePort> getUnconnectedOutputPorts()` : no comment !

Pour la classe `Patch` :
- ajouter en attribut un tableau associatif (ou "dictionnaire") `Map<String, ModuleAbstract>` qui associe un nom de module à un module. Ce tableau optimisera les accès aux modules par nom, par exemple dans la méthode `connect` de la classe `Patch`.
- `void reset()` : *reset* tous les modules du patch.
- `List<ModulePort> getUnconnectedInputPorts()` : retourne une liste de tous les ports d'entrée non connectés des modules du patch. Les ports seront dans l'ordre des modules, puis pour chacun d'eux dans l'autre des ports du module.
- `List<ModulePort> getUnconnectedOutputPorts()` : même chose pour les ports de sortie.

3 Gestion des erreurs

Pour le moment, vous vous êtes (probablement ?) peu préoccupés de la gestion des erreurs. Il est temps d'améliorer cette gestion, au moyen d'exceptions.

- S'assurer que les cas d'erreur sont traités en déclenchant un mécanisme d'exception. Par exemple : création d'une `Connexion` invalide entre des ports qui n'existent pas ; affectation d'une valeur à un port qui n'existe pas ; dans un patch, connexion d'un module qui n'existe pas ; etc.
- Choisir les types (classes) des exceptions levées.
- Pour les cas d'erreurs les plus importants, typiquement ceux des méthodes de la classe `Patch`, il peut être pertinent de mettre en place une sémantique propre en introduisant une ou plusieurs classes d'exception, controlées ou non (voir fiche pédagogique 4).

4 Encapsulation, paquetages et visibilité

Vous savez que le *principe d'encapsulation* est une des forces de la POO, en ce qu'il permet de garantir que les données en mémoire sont toujours cohérentes et consistantes entre elles. Vous savez aussi comment mettre en oeuvre ce principe au niveau d'une classe, au moyen des qualificateurs de visibilité (`public`, `private`, `protected`...) des membres, pour garantir les *invariants de classe*.

En fait, le principe d'encapsulation est plus général. Par exemple, on peut avoir besoin de garantir des invariants entre l'état de *plusieurs* objets instances de *plusieurs* classes. A titre d'exemple, dans ce TP, les `CommunicationPort` et les `Connexion` doivent être cohérents entre eux : il faudrait que, lorsqu'une `Connexion` est connectée à un `CommunicationPort`, on soit sûr que ce port ait bien en attribut cette même `Connexion`, et jamais une autre...

Par ailleurs, en Java, il est possible de régler finement les visibilités de chaque classe au sein d'un paquetage, et de chaque membre de chaque classe, dans et hors du paquetage où ils sont définis. A titre d'exemple, il conviendrait que, dans le programme principal, on ait accès uniquement :

- aux constructeurs des classes concrète de module - mais pas aux autres méthodes de la hiérarchie des modules, qui ne devraient pouvoir être appelées que depuis les sous classes de modules, ou la classe Patch.
- aux méthodes publiques de la classe `Patch`.

Tout le reste relève de la "cuisine" interne du programme.

Pour le moment, vous aurez remarqué que le sujet n'a jamais évoqué comment régler les visibilités des classes et des méthodes... Le temps est venu de le faire. Pour cela, vous pouvez avoir recours :

- à des paquetages java, pour répartir vos classes dans des paquetages appropriés.
- aux qualificateurs de visibilité des classes `public` ou ∅ (visibilité par défaut : uniquement dans le paquetage. Voir fiche pédagogique 2, paragraphe "Visibilité").
- aux quatre qualificateurs de visibilité des membres des classes : `public`, ∅ ("par défaut"), `protected` ou `private`. Relire la section "Visibilité et encapsulation" de la fiche pédagogique 1 peut être une bonne idée.

A vous de jouer !

5 Méthode `setPatch()` de la classe `ModuleAbstract` et interface `ModulePlayable`

5.1 Position du problème

Dans cette section, nous nous intéressons à deux axes :

— Il serait pertinent qu'un module "sache" si il a été ajouté à un patch et à quel patch il appartient. Cela permettrai qu'on s'assure qu'un module déjà présent dans un patch ne soit pas par inadvertance ajouté à un au second patch - puisque, bien sûr, un module ne peut appartenir à deux patchs à la fois.
— En plus de la liste de ses modules, il faudrait qu'un patch puisse organiser les modules dans d'autres collections. Par exemple, il serait utile qu'un patch connaisse les modules contenant des données audio, c'est à dire de type `AudioDataProvider` et `AudioDataReceiver` : cela permettra qu'on puisse "écouter" (play()) tous les signaux sonores présents dans un patch, ou encore observer (display()) les formes d'onde de tous les signaux sonores présents dans un patch.

Pour le second point, une solution serait d'interroger le type dynamique du module lorsqu'il est ajouté au patch, avec par exemple quelque chose comme :

```
class Patch{
    Collection<AudioDataReceiver> lesModuleAudioDataReceiver;
    ...

    void addModule(ModuleAbstract m) {
        if(m instanceof AudioDataReceiver} {
            lesModuleAudioDataReceiver.add(m);
        } else if (m instanceof AudioDataProvider) {
            // ajout à une collection de AudioDataProvider...
        }
        // etc.

        // Puis enfin, comme avant, ajout de m à la collection de tous les
            modules
        ...
    }

    public void display() {
        //affiche les AudioData de tous les modules de la collection
            lesModuleAudioDataReceiver
    }
}
```

Cette approche n'est pas satisfaisante en POO, ne serait-ce que parce qu'elle réalise un test sur le type dynamique d'un objet, ce qu'on évite autant que possible (cf. la fiche pédagogique 5 "Polymorphisme"). Heureusement, une autre approche bien plus puissante est possible...

5.2 Travail à réaliser

Phase 1 :

— Ajouter un attribut de type `Patch` à la classe `ModuleAbstract`, qui conserve une référence sur le patch dans le module a été ajouté. Il vaut `null` tant que le module n'a pas été ajouté dans un patch.
— Toujours dans la classe `ModuleAbstract`, ajouter une méthode `void setPatch(Patch p)` et une méthode `boolean isInPatch();` qui retourne `true` si le module est déjà dans un patch.
— Modifier la méthode `void addModule(ModuleAbstract m);` de la classe `Patch` pour que : 1/ elle génère une erreur si le module `m` est déjà dans un patch ; et 2/ elle informe le module du patch dans lequel on est en train de l'ajouter.

Phase 2 :

— Créez une interface `ModulePlayable` qui déclare les méthodes :
  — `void playAudioData();` : écoute d'un signal audio
  — `void displayAudioDataWaveform()` : affichage l'écran de la forme d'onde
  — `void saveAudioDataToWavFile(String fileName)` : sauvegarde dans un fichier son

  - et `AudioData getAudioData()` : retourne une référence sur l'`AudioData`
- Assurez vous que les classes `AudioDataProvider` et `AudioDataReceiver` réalisent cette interface.
- Dans la classe `ModuleAbstract`, ajoutez en attribut une liste de `ModulePlayable`.
- Dans la classe `Patch`, écrire une méthode (non publique...) `void registerToPatch(ModulePlayable m)` qui ajoute `m` à liste des `ModulePlayable` du patch.
- Dans les classes `AudioDataProvider` et `AudioDataReceiver`, redéfinissez la méthode `void setPatch(Patch p)` de telle sorte qu'elle appelle `p.registerToPatch(this)`, afin que le module "s'enregistre" dans le patch p "en tant que ModulePlayable". N'oubliez pas d'appeler aussi la méthode `void setPatch(Patch p)` de la super classe !

Enfin, dans la classe `ModuleAbstract`, ajoutez une méthode `void displayAudioDatas()` qui affiche à l'écran, dans une seule fenêtre, les formes d'onde de tous les signaux du patch ; il suffit d'appeler la méthode `static void displayMultipleAudioData(List<AudioData> listOfAudioData)` de la classe `AudioData`.

5.3 Pour votre information : le pattern Visiteur

Dans le code précédent, notez qu'on a aussi utilisé le polymorphisme : un `AudioDataProvider` "est" un `ModuleAbstract` ; mais il "est" aussi un `ModulePlayable` puisque sa classe réalise cette interface.

On a aussi utilisé le mécanisme POO de liaison dynamique : lorsqu'un module est ajouté au patch, la "bonne méthode" `void setPatch(Patch p)` est exécutée, de telle sorte que le module soit à même de s'enregistrer dans le patch en fonction de son type dynamique.

Le petit bout d'architecture logicielle qui a été ainsi réalisée dans cette partie répond en fait à un problème assez courant. Il est suffisamment courant pour être généralisé dans un "patron de conception", nommé *Visiteur*, qui permet de le résoudre de façon générique. Ceux qui veulent en savoir plus (et qui en ont le temps...) pourront se renseigner (sur Internet par exemple) sur le "patron de conception Visiteur".

6 Clonage d'un module et clonage d'un patch

6.1 Position du problème

Maintenant que nous avons des patchs fonctionnels, il serait très utile de pouvoir en réaliser des "copies conformes". Par exemple, cela permettrait de créer un patch déjà assez complexe, puis de le copier plusieurs fois pour en réaliser des variantes (par exemple en ajoutant des modules aux copies).

Bien sûr, "copier un patch" nécessite de faire une copie de chacun de ses modules. Vous connaissez déjà la notion de "constructeur de copie" ; mais, dans le cas d'une hiérarchie de classe (comme celle de nos classes de modules), elle est insuffisante. Dans cette partie, nous allons voir pas à pas comment réaliser proprement en POO une copie de patch et de modules.

6.2 Copie d'un module

Doter toutes la classe `ModuleAbstract`, puis toutes vos classes de module, d'un constructeur de copie.

Le constructeur de copie construira une copie conforme de l'objet, si ce n'est que :
- Les connexions ne seront pas copiées : les ports de la copie seront laissés sans connexion.
- La copie n'appartient à aucun patch : son attribut `Patch p` sera laissé à `null`.
- La méthode `reset()` est appelée sur la copie.

Malgré ces constructeurs de copie, copier un module quelconque en utilisant le constructeur de copie n'est pas immédiat... Considérez par exemple le code suivant :

```
1       /** retourne une copie conforme du module other */
2       ModuleAbstract copierModule(ModuleAbstract other) {
3           return new ....
4           // on est bien embété, car on ne sait pas quel constructeur appeler !!
5           // En effet, on ne connait que le type statique de other,
6           // c'est à dire ModuleAbstract,
7           // mais on ne sait pas de quel type (dynamique) de module il s'agit !
8           // Faut il créer par copie un GenSine ? Un AudioDataReceiver ? Un autre
                module ?
9       }
```

Pour résoudre ce problème :
— Doter la classe `ModuleAbstract` d'une méthode abstraite `ModuleAbstract copier()` sans paramètre.
— Redéfinir cette méthode dans les sous classes, pour appeler le bon constructeur de copie.
Par exemple, dans la classe `GenSineBasic`, la méthode copier sera :

```
1   class GenSineBasic{
2       ...
3       ModuleAbstract copier() {
4           return new GenSineBasic(this);
5       }
6   }
```

Et voila, le tour est joué : pour copier un module, il suffit désormais d'appeler la méthode `copier()` :

```
1       ModuleAbstract m = ... ; // on crée un module d'un type quelconque
2       ModuleAbtract copie = m.copier() ;
3       // et copie est une copie de m, quelque soit le type dynamique de m !
```

Vous remarquerez qu'on a utilisé le mécanisme de liaison dynamique pour résoudre notre problème. La solution apportée est en fait très courante en POO lorsqu'on veut pouvoir copier des objets instances d'une classe placée dans une hiérarchie. Elle est tellement courante qu'en Java une interface lui est dédiés : l'interface `Cloneable`, qui définit la méthode `Object clone()`. Pour ceux qui voudraient en savoir plus (et qui en ont le temps...), nous conseillons l'excellent cours en ligne de JM Doudoux.

6.3 Copie d'un patch

Ajouter un constructeur de copie à la classe `Patch`.

Copier un patch, c'est essentiellement :
— Copier chacun de ses modules (appel de la méthode `copier()` de la classe `ModuleAbtract`.
— Ajouter toutes ces copies au nouveau patch.
— Recréer les connexions entre les modules du nouveau patch. Pour cela, on pourra parcourir la collection des `Connexion` de l'ancien patch.

6.4 Test

Créer un patch, créer une copie, exécuter la copie, vérifier !

7 Un module ... qui est en fait un patch ! ?

7.1 Position du problème

Nous avons une classe `Patch` qui regroupe plusieurs modules. Pour peu que certains des modules d'un patch aient des ports non connectés, un patch peut être vu comme un système de traitement du signal, qui réalise un traitement déjà complexe entre entrée et sortie, au moyen de ses modules.

Il serait donc fort intéressant de permettre qu'un patch existant puisse être ajouté à un autre patch, pour construire de proche en proche des patchs de plus en plus complexes.

En d'autre terme, il faudrait pouvoir considérer un patch comme un macro-module, c'est-à-dire comme un module composé de plusieurs modules, que l'on peut exécuter, connecter, etc. comme n'importe quel module.

Dans cette section, on se propose de créer un nouveau type de module : le `MacroModule`, qui contient un patch.

7.2 Travail à réaliser

Créer une classe `MacroModule` qui dérive de `ModuleAbstract` qui possède en attribut un patch, de telle sorte que "exécuter un macro-module" ce soit "exécuter" le module qui le constitue.

Voici des pistes de solutions à ce problème (mais vous pouvez trouver les vôtres...) :
- La classe a en attribut un `Patch patchInterne`. Cet attribut est initialisé par le constructeur, qui prend donc un `Patch p` en paramètre. Si vous avez traité la section précédente (copie de modules et de patchs), il serait avisé de stocker dans le macro-module une *copie* du patch p plutôt qu'une simple référence sur p.
- Le nombre de ports d'entrée du macro-module est le nombre de ports d'entrée libre (sans connexion) du patch contenu.
- Le nombre de ports de sortie du macro-module est le nombre de ports de sortie libre (sans connexion) du patch contenu.
- La classe déclare un attribut `ArrayList<ModulePort> unconnectedInputPorts`. Le constructeur du macro-module parcoure tous les modules du patch contenu, et ajoute un à un les ports d'entrée non connectés de ces modules à la liste `unconnectedInputPorts`.
- La classe déclare un attribut `ArrayList<ModulePort> unconnectedOutputPorts`. Le constructeur du macro-module parcoure tous les modules du patch contenu, et ajoute un à un les ports de sortie non connectés de ces modules à la liste `unconnectedOutputPorts`.
- Remettre à zéro un macro-module (méthode `reset()`), c'est remettre à zéro le patch contenu
- Exécuter un macro-module, c'est :
  - Copier les échantillons présents dans les ports d'entrée du macro-module dans les ports de la liste `unconnectedInputPorts`.
  - Exécuter le patch contenu.
  - copier les échantillons présents dans les ports de la liste `unconnectedOutputPorts` dans les ports de sortie du macro-module.

7.3 Test

Créer un patch avec au moins un module qui a un port de sortie non connecté. Créer deux `MacroModule` à partir de ce patch. Ajouter ces macro-module à un second patch, dans lequel on ajoutera aussi un `Mixer` et un `AudioDataReceiver`. Exécuter ce patch et vérifier que tout se passe bien.

7.4 Pour votre information : le pattern Composite

L'architecture logicielle réalisée dans cette section est en fait assez courante dans les hiérarchie de classe : elle intervient dès qu'on souhaite implanter une situation où un objet est composé d'objets similaires, ou dès qu'on veut manipuler un groupe de choses comme une chose. Un "patron de conception" (une technique d'architecture logicielle objet courante) lui est dédié : le pattern Composite.

8 Vers un *scheduleur* de modules - ou "partition de patchs"

Le système mis en place jusqu'ici est déjà pertinent... mais on est tout de même encore un peu loin de pouvoir l'utiliser pour faire de la musique.

L'un des problème essentiel restant est qu'un patch n'est "joué" qu'une fois, au début du calcul et avec des paramètres prédéterminés et non modifiables.

Imaginons, par exemple, qu'on ait écrit un très beau patch capable de synthétiser une note (par exemple en synthèse additive, voir l'Annexe B). Pour le moment :

— il ne sait jouer qu'une seule note, dont la fréquence fondamentale et la durée sont déterminées une fois pour toute à la création du patch
— la note est jouée à t=0... pas facile de commencer la note plus tard, ou de jouer plusieurs notes à la suite !

Plusieurs évolutions sont donc souhaitables :

— Il faudrait par exemple qu'on puisse créer une sorte de "partition de patchs" : une série de patchs, chacun étant "démarré" à un instant bien choisi, et "s'arrêtant" tout seul lorsqu'il a fini de faire son travail. A un instant t, un certain nombre de patchs s'exécuteraient donc en parallèle... Pour cela, introduire une nouvelle classe `Scheduler` serait peut-être appropriée...
— Il serait utile qu'on puisse modifier certains paramètres des modules d'un patch une fois le patch construit.

Ainsi, pour jouer plusieurs notes à la suite, il deviendrait possible de :

— créer un patch qu'on va utiliser comme canevas pour jouer chacune des notes
— à chaque fois qu'on veut jouer une note avec ce patch : dupliquer le patch, modifier les paramètres adéquats de la copie à la bonne valeur (bonne fréquence, bonne durée...), ajouter la copie au `Scheduler` pour que le calcule de ce patch démarre au moment voulu.

Ces idées étant proposées... à vous de "jouer" !