

Documentation technique

Projet Medley



Résumé

Ce document s'adresse aux membres du labEIP et à toute personne qui voudrait prendre connaissance des caractéristiques techniques de Medley (développeur, architecte projet).

Il donne d'abord un aperçu global de l'architecture de notre projet et de son fonctionnement. Puis il permet d'avoir une vision un peu plus précise des différentes parties qui le composent : le logiciel, les plugins et le site internet.

Nous expliquerons plus en détails le fonctionnement du logiciel, qui fournit une interface graphique ainsi qu'une API permettant d'utiliser des plugins. Une API ouverte qui permet donc à des développeurs externes de créer leurs propres outils.

Suit une description technique des plugins que nous réalisons afin de lancer le projet. Quatre plugins sont réalisés, un accordeur, un métronome, un enregistreur/lecteur de son et un lecteur de partition.

Enfin nous détaillerons l'implémentation du site internet qui nous permet de communiquer autour de notre projet, en plus de proposer en téléchargement le logiciel, les plugins et les différentes versions de l'API.

Description du document

Titre	2015_TD4_FR_Medley
Date	23/12/2014
Responsable	Julie Nicolas
Email	medley_2015@labeip.epitech.eu
Sujet	Documentation technique du projet Medley
Version	4.0

Révisions

Date	Auteur	Section(s) modifiée(s)	Commentaires
23/12/2014	Julie Nicolas	Toutes	Première version du document
27/12/2014	Joris Mathieu	Description globale/Description Site web	
03/01/2015	Thomas Favre	Plugins : lecteur/éditeur partition	Ajout description technique

SOMMAIRE

I.	Description globale du projet	1
a)	But du projet.....	1
b)	Présentation Globale	1
1.	Logiciel	1
2.	Plugins.....	2
3.	Web	2
c)	Choix technologiques	3
1.	IDE.....	3
2.	Langage.....	3
3.	Librairies	3
d)	Architecture générale	5
1.	Logiciel	5
2.	Web	7
3.	Plugins.....	7
II.	Description détaillée.....	8
a)	Logiciel et API.....	8
1.	Core	8
2.	Liste de Plugins	12
3.	Workspace	12
4.	Navigation.....	12
5.	Création d'un Plugin	13
b)	Plugins.....	14
1.	Métronome.....	14
2.	Accordeur	15
3.	Enregistreur/Lecteur de son	16
4.	Lecteur de partition	17
5.	Editeur de partition	17
c)	Site	18
1.	Vue globale	18
2.	Couche applicative	19
3.	Webservices.....	22
4.	Base de données.....	22

I. Description globale du projet

a) But du projet

Medley est un projet destiné aux musiciens. Beaucoup de logiciels de musique existent à l'heure actuelle, mais ils sont pour la plupart payants, ou assez compliqués à prendre en main, surtout pour un débutant en la matière. Pour répondre à ce problème, nous avons choisi de créer un outil simple et ergonomique qui peut s'adapter aux musiciens amateurs comme aux plus avancés.

Le projet est composé de plusieurs parties : un logiciel, des plugins qui sont intégrés à ce logiciel, une API qui permet cette intégration, et un site web.

Le but du projet est d'utiliser les plugins (des outils musicaux : un métronome, un lecteur de partition, ...) au sein du logiciel. Ainsi, on peut personnaliser son logiciel en y ajoutant seulement les fonctionnalités dont on a besoin. On peut aussi faire communiquer les plugins entre eux, ce qui permet d'étendre les possibilités si besoin et de s'amuser avec son logiciel Medley de manière légèrement plus complexe.

Le site web quant à lui permet de télécharger le logiciel, les plugins qu'on souhaite utiliser et également l'API qui permet de créer ses propres plugins. Il est en outre le support communautaire du projet.

b) Présentation Globale

1. Logiciel

Notre logiciel est composé de 4 parties principales :

- Core : Partie principale du projet, c'est dans cette partie que le logiciel se lance.
- Liste des plugins : Ce module permet d'ajouter ou de supprimer des plugins. Il permet également de les afficher dans une liste.
- Workspace : Gestion des plugins lancés. Il permet de séparer l'espace de travail jusqu'à 4 partie et gère la communication inter-plugin.
- Navigation : Cette partie est un listing de dossier filtrant les tablatures, les musiques et les partitions de musique.

Chaque partie est un projet indépendant, ont leur propre contrôleur et leur propre vue. La partie principale « Core » charge les autres parties exportées en .jar et récupère les différentes vues de ces modules.

2. Plugins

Métronome

Dans notre projet, le plugin Métronome va permettre de produire un son audible permettant d'indiquer un tempo, vitesse à laquelle doit être jouée une musique.

Accordeur

L'accordeur permet à l'utilisateur d'ajuster son instrument, son but étant de l'informer de la justesse de la note qu'il est en train de jouer.

Premièrement il doit sélectionner une note. Puis l'accordeur va écouter les sons entrant, les analyser grâce à l'algorithme de la transformation rapide de Fourier (FFT).

Enregistreur/Lecteur de son

Le SoundRecorder permet d'enregistrer et/ou lire des sons.

L'utilisateur peut enregistrer un son vers un fichier en appuyant sur un bouton puis en sélectionnant le fichier de destination de l'enregistrement. Il peut aussi lire un fichier audio de son choix en effectuant un glissé-déposé du fichier audio vers le plugin. L'utilisateur peut également contrôler la progression de sa musique, le volume, la pause et la reprise de la musique jouée. Le tout en utilisant les technologies fournies par JavaFX.

Lecteur de partition

Le lecteur de partition permet de lire une partition exportée au format musicXML.

L'utilisateur peut écouter et/ou stopper la lecture d'une partition. Un bouton 'Loop' permet à l'utilisateur d'écouter la partition en boucle.

Editeur de partition

L'éditeur de partition permet à l'utilisateur de créer une partition et de l'exporter au format musicXML.

3. Web

Le site internet a trois fonctions principales :

- Communiquer autour de notre projet (même rôle que notre site vitrine)
- Proposer les différentes versions de notre logiciel, de nos plugins et de notre API au téléchargement
- Permettre aux différents utilisateurs, qu'ils soient musiciens et/ou développeurs de se retrouver

Certaines pages sont donc très simples à réaliser puisqu'il ne s'agit que de pages statiques sans actions utilisateurs, ou accès en base de données. D'autres

demandent un peu plus de traitements et de vérifications pour être opérationnelles.

c) Choix technologiques

1. IDE

Comme nous partageons les sources en java de notre projet grâce à Subversion, certains membres travaillent sous Eclipse et d'autres sous IntelliJ selon leur préférence.

2. Langage

Pour faciliter le travail d'équipe sur ces différentes parties, et parce qu'un membre de l'équipe peut être amené à contribuer à une partie ou une autre durant le développement du projet, nous avons décidé d'utiliser le même langage de programmation pour toutes les parties. Le logiciel, les plugins et l'API sont donc développés avec la dernière version de Java, et le site en J2EE.

Nous avons choisi Java parce que nous étions tous familiers avec ce langage de programmation, et parce qu'il permet de porter notre logiciel sur différents OS de façon très simple.

3. Bibliothèques

- Logiciel et Plugins

Au niveau de l'interface graphique, nous utilisons Javafx, pour ses fonctionnalités et ses composants facilement personnalisables et parce que c'est une bibliothèque qui va être améliorée par Sun dans l'avenir. En effet, java 8 est sortie dans le courant de l'année 2014, et une version améliorée de Javafx lui est intégrée.

Comme nous avons décidé d'utiliser Javafx, un problème potentiel s'est présenté au niveau de l'API de notre projet. Quand Sun a sorti Javafx, beaucoup de développeurs utilisaient (et peut être utilisent toujours) la bibliothèque Swing. Nous aurions donc voulu laisser la possibilité aux utilisateurs de développer leurs plugins en Swing, ce qui s'est avéré compliqué. Les nouvelles versions de Javafx résolvent ce problème puisqu'il est désormais possible d'utiliser des composants Swing au sein de JavaFx.

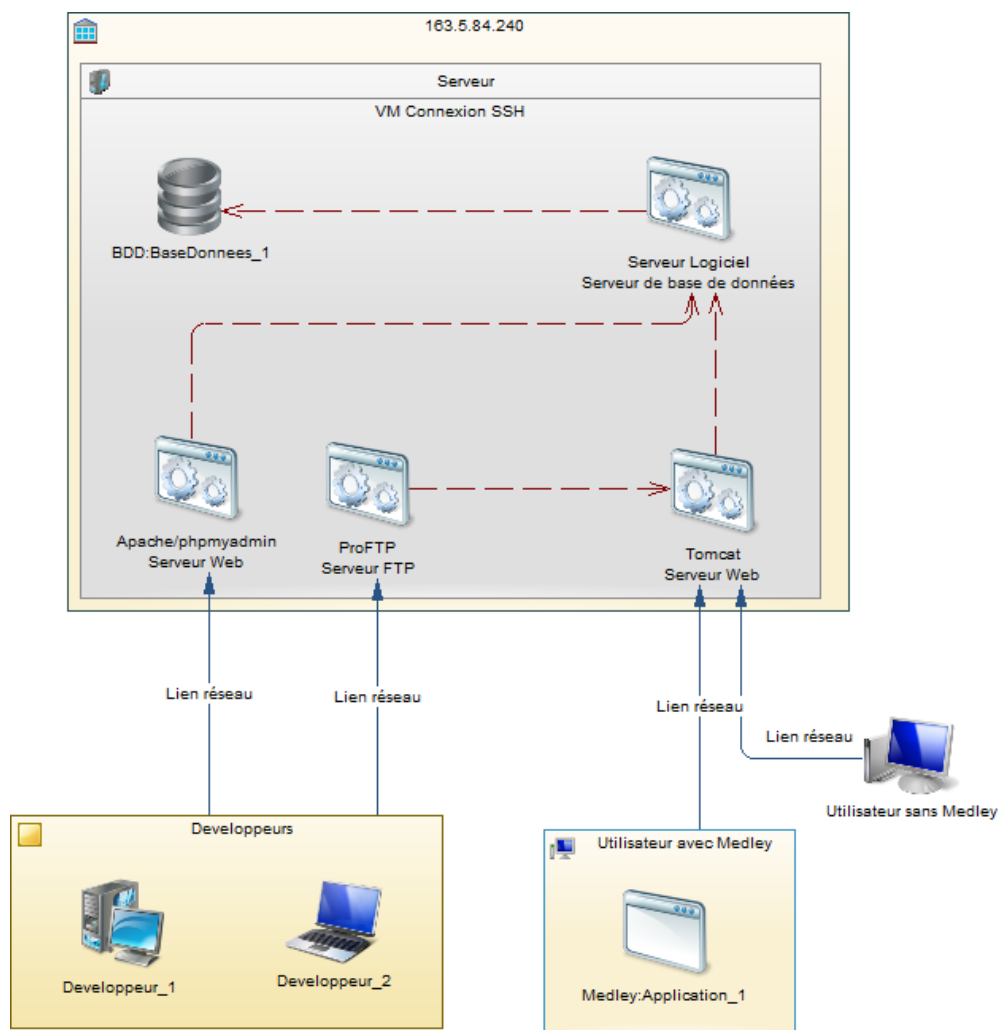
- Web

En plus des outils de base du J2EE, nous utilisons un certain nombre de frameworks et librairies. Toutes ces librairies sont ajoutées à notre projet grâce à Maven. Maven est un outil, très simple à configurer, qui permet de télécharger et mettre à jour les librairies compatibles J2EE de notre projet.

Voici une liste des composants que nous utilisons :

- **JSF (Java Server Faces)** : JSF est un framework MVC (Modèle Vue Contrôleur) qui aide la construction des vues grâce à une bibliothèque de balises et de composants. En outre il nous simplifie le développement et évite la copie de code.
- **Hibernate** : Hibernate est une bibliothèque ORM (Object Relational Mapping). Son rôle est d'assurer le lien entre les objets Java que nous manipulons et la base de données.
- **JDBC** : JDBC est une API (Application Programming Interface) utilisée par Hibernate. Elle permet, grâce à un système de driver, de changer de type de base de données rapidement.
- **Spring core/security** : Nous nous servons de deux modules du framework Spring en parallèle de JSF pour deux tâches bien séparées. Spring core nous permet de contrôler la durée de vie de nos objets tandis que Spring security va gérer l'accès aux pages de notre site internet via un système de rôle.
- **Primefaces** : Primefaces est une bibliothèque de composants. Elle enrichie la collection déjà disponible avec JSF et facilite le développement des interfaces.
- **AspectJ** : Petit module utilisé en parallèle de Spring core et qui nous est utile pour éviter de mélanger les codes de debug et le code de nos classes métiers.
- **JUnit** : C'est un framework de test que nous utilisons pour nos tests unitaires.

d) Architecture générale



Vue globale du projet Medley

1. Logiciel

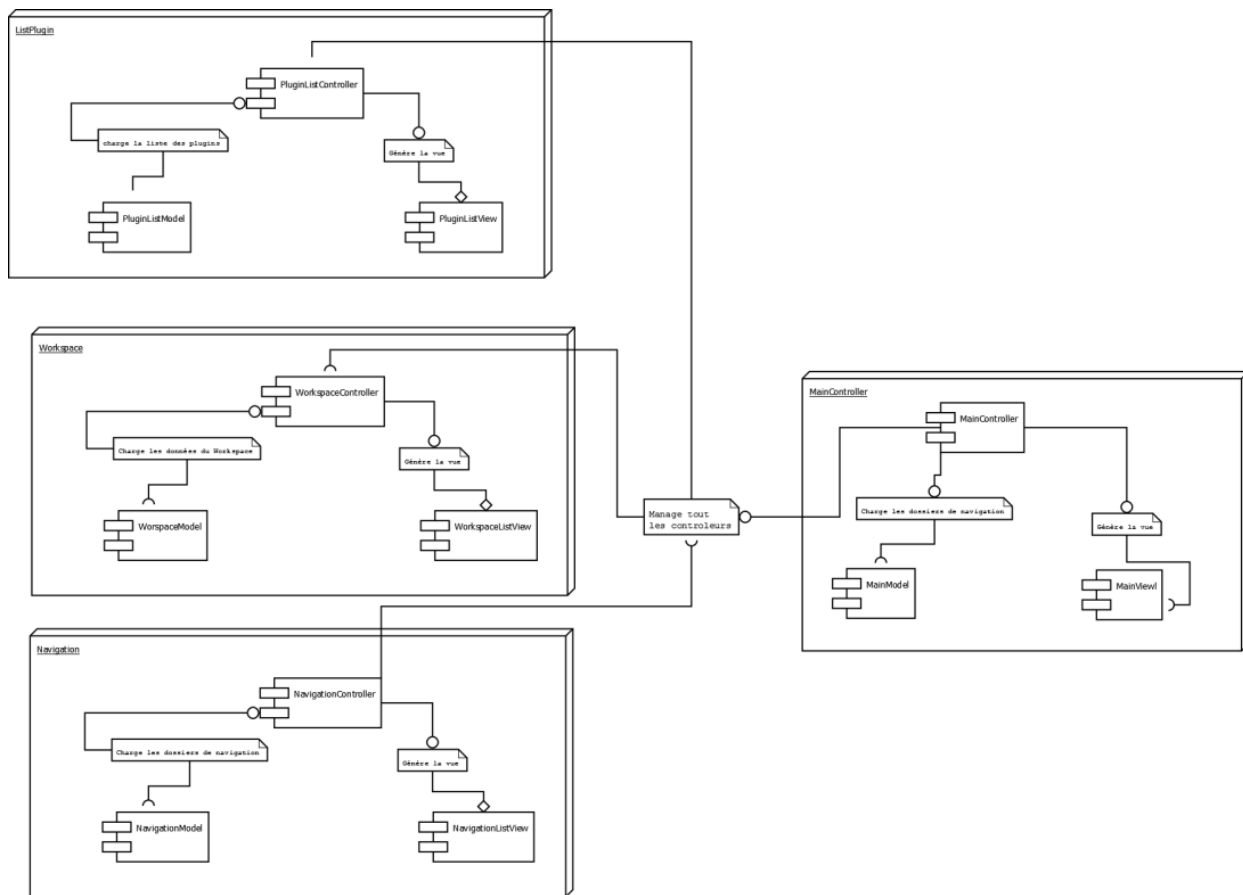
- Organisation :

Pour développer notre logiciel, nous utilisons le patron de conception MVC. Cela nous permet de séparer les différentes parties de notre application, et d'en avoir une vision simplifiée.

Notre logiciel est composé de 4 parties principales.

Chacune de ces parties correspond donc à une vue, dont les données à afficher seront contenues dans un modèle, lui-même mis à jour et fourni par un contrôleur, et est contenue dans un fichier .jar (Java Archive). En découpant ainsi notre logiciel en plus petites parties, nous le rendons versionnable plus aisément. Le système de chargement des fichiers .jar permettra de télécharger les mises à jour sans avoir à télécharger à nouveau tout le logiciel, ni à le relancer.

Pour permettre de comprendre plus facilement l'architecture de notre logiciel, nous avons fait un diagramme de composants plus détaillé qui résume ce qui a été dit plus haut :



- Mise à jour du logiciel :

La mise à jour du logiciel est effectuée avant le lancement du logiciel. L'exécutable "launcher" est le programme qui effectuera cette mise à jour.

Toutes les parties (core, navigation, liste des plugins et workspace) sont mises à jour séparément.

Chaque partie contient un dossier "version" avec un fichier nommé "version" dans lequel est définie sa version courante.

Le programme "launcher" utilise un web service qui renvoie les différents éléments essentiels aux mises à jour :

- les numéros des dernières versions
- les liens vers les téléchargements
- les tailles des téléchargements

Les fichiers sont téléchargés dans le dossier "tmp" puis appliqués au logiciel si les téléchargements se sont bien déroulés.

Lorsque les différentes mises à jour sont terminées, le logiciel se lance.

- Stockage de données :

Au niveau du logiciel, nous stockons les données de configuration dans des fichiers XML qui seront parsés grâce à la librairie JDOM (Java Document Object Model).

2. Web

Afin de faire fonctionner le site internet, nous avons eu besoin d'installer :

- Un serveur **Tomcat** : c'est le serveur qui va compiler et interpréter notre code Java et retourner une page html/CSS/JS aux internautes.
- Une base de données **Mysql** : c'est elle qui contient toutes les données de notre site internet et qui est utilisée par notre code Java.
- Un serveur **proFTP** : ce serveur nous permet d'accéder au dossier « webapps » du serveur Tomcat et d'y envoyer nos fichiers plus simplement.
- Un serveur **Apache + phpmyadmin** : pour permettre de gérer la base de donnée plus facilement.
- Une base **postgreSQL** pour tester le multi-db.

Le nom de domaine medley-community.com a également été acheté pour permettre un accès au public.

3. Plugins

Le logiciel Medley est composé des plugins suivants :

- Un métronome
- Un accordeur
- Un enregistreur/lecteur de son
- Un lecteur de partition
- Un éditeur de partition

II. Description détaillée

a) Logiciel et API

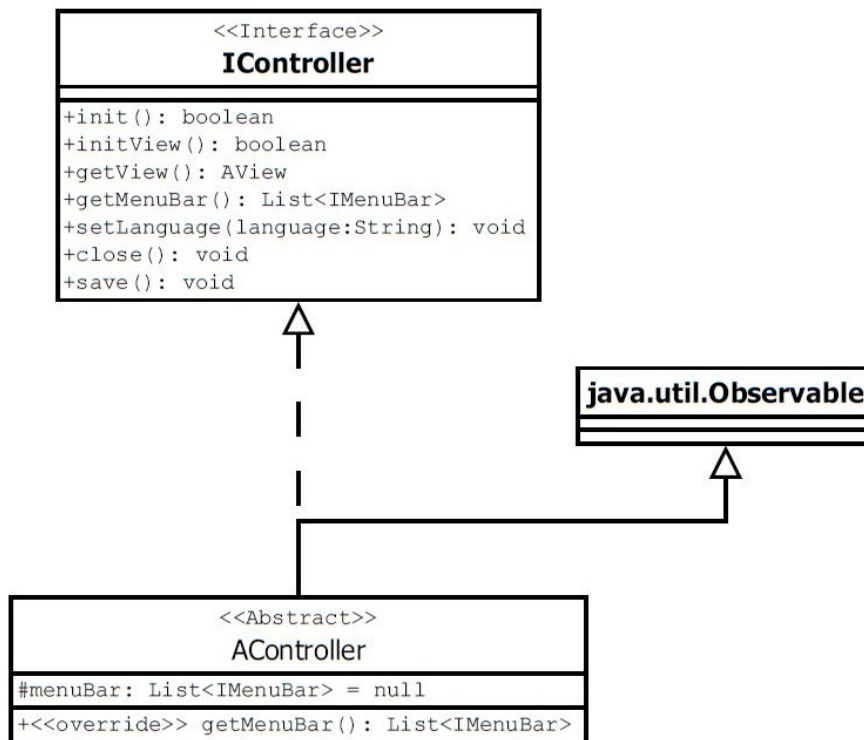
1. Core

- Chargement et mise en place des différentes parties.

Comme vu dans la partie générale, le logiciel Medley est composé de plusieurs modules (ou parties). Ces modules sont représentés par des bibliothèques dynamiques (fichier .jar en Java). Le module "core" comprenant le main du projet a pour but de charger et de placer les différentes vues du projet.

Ces modules sont complètement indépendants entre eux. Chaque module a un contrôleur principal qui est chargé par le core grâce à une interface commune qu'il doit implémenter. Cette interface est contenue dans le package Utils. Ce dernier est mis à disposition des autres modules et contient un ensemble de classes utilitaires pour chaque module.

L'interface IController :

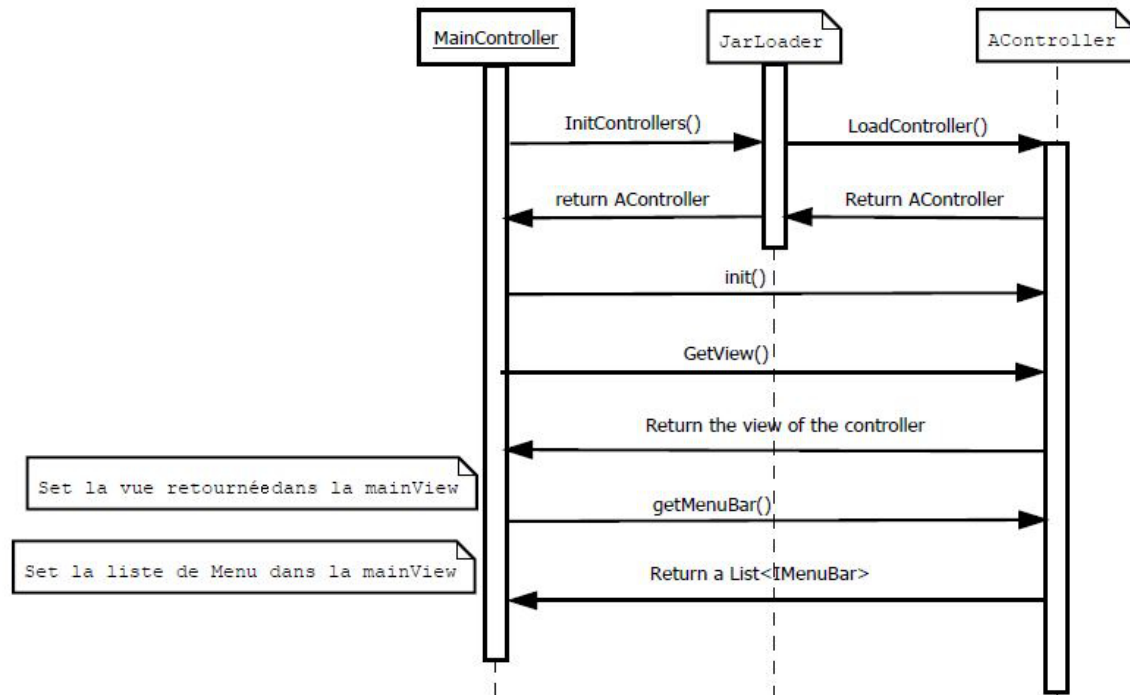


Cette interface comprend des méthodes basiques pour le bon fonctionnement d'un module (initialisation, initialisation de la vue, sauvegarde, fermeture, récupération de son principal élément graphique et de son numéro de version).

Nous avons créé une classe abstraite AController (module Utils dans le package Abstraction) héritant de IController pour pouvoir charger les différentes parties.

La méthode "Control `getMainComponent()`" est la méthode pour récupérer l'élément graphique du module. Nous avons choisi de retourner la classe Control de javafx car elle est le parent de tous les conteneurs graphiques de la bibliothèque. Cela permet de pouvoir rattacher n'importe quel élément graphique à la vue principale du projet.

Ci-dessous, un diagramme de séquence du logiciel, pour mieux s'en représenter le fonctionnement :



Ce diagramme décrit l'initialisation des sous-contrôleurs et des sous-vues.

Tout d'abord le MainController va charger la bibliothèque dynamique correspondant à chaque partie (navigation, liste des plugins et espace de travail).

Chaque partie est représentée par l'objet AController dans le diagramme. Après le chargement des parties, le MainController les initialise grâce à la fonction `init()`. Les sous contrôleurs vont donc s'initialiser mais également initialiser leur vue. Le MainController va récupérer ces vues grâce à la méthode `getView()` de AController et les rajouter à la mainView (la vue principale). La barre des menus est générée dynamiquement en fonctions des parties.

Chaque partie crée ses menus et le mainController va les récupérer grâce à la méthode `getMenuBar()`. Enfin ce dernier met à jour la barre des menus de la mainView.

- Communication inter-modules

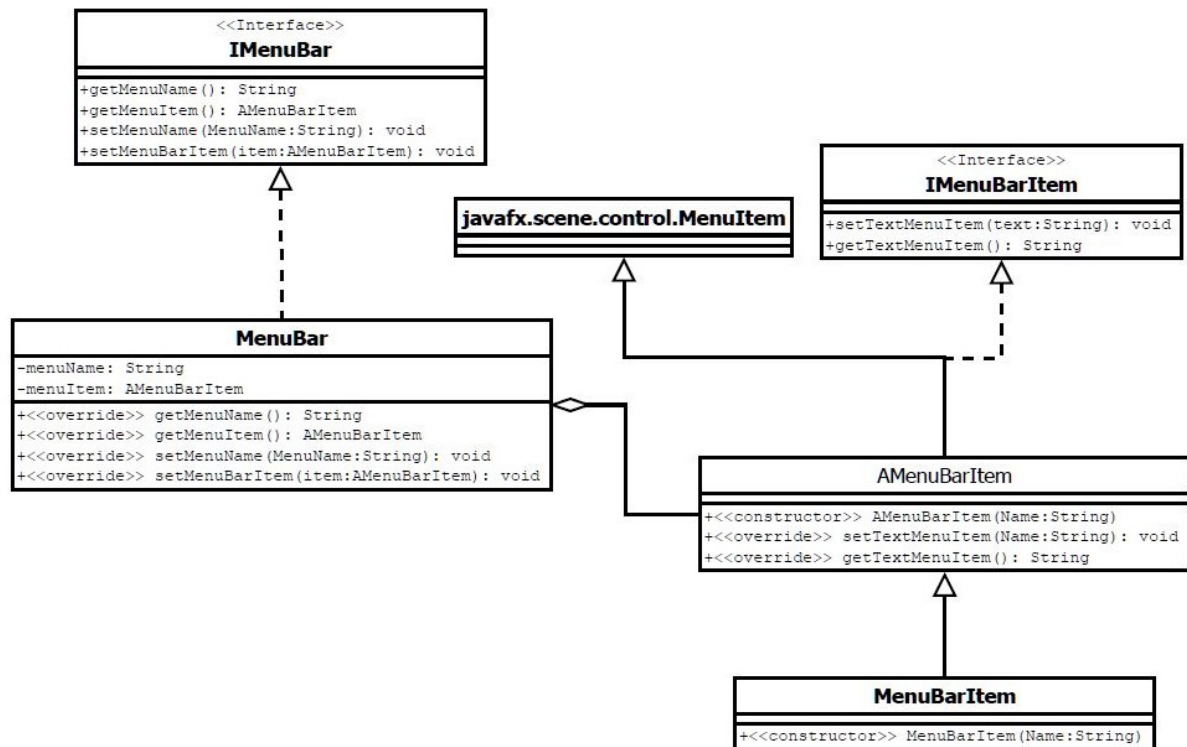
Pour le bon fonctionnement du projet les différents modules ont besoin de communiquer. Pour cela nous utilisons le design pattern Observable. En effet chaque principal contrôleur hérite de la classe Observable. Cette classe permet de pouvoir communiquer directement du module vers le core du projet. Grâce à la méthode "`init(Observer ob)`" nous assignons l'observer à chaque module.

La communication se fait principalement par chaînes de caractère mais des objets peuvent également être envoyés.

- Barre des menus

La conception de la barre de menu est un peu particulière : elle est construite par les parties du logiciel qui correspondent aux actions. Par exemple, l'action "lancer un plugin" correspond à la partie liste des plugins, et sera donc gérée par le contrôleur de cette partie.

Les composants graphiques utilisés pour la construction de la barre de menus seront abstraits, afin de pouvoir les utiliser dans des bibliothèques séparées.



L'interface **IMenuBar** contient tous les éléments pour pouvoir construire cette barre de menu indépendamment.

La méthode `getMenuName()` retourne le nom du Menu associé.

La méthode `getMenuItem()` retourne l'élément graphique correspondant à l'item à placer dans la barre des menus. Cet élément aura au préalable ses listeners déjà assignés pour que les actions associées soient gérées à l'intérieur de son module.

- Système de langue

Chaque partie du logiciel, exportée au sein du projet Medley sous forme d'un module .jar, a son propre dossier `Ressources/Languages/` qui contient un fichier xml par langue.

Dans chaque fichier on trouve la clé qui correspond à la langue du fichier et le nom de la langue.

Suit la liste des traductions. Chaque traduction a une clé qui correspond à l'élément graphique (label, titre, popup) où elle doit être affichée.

Ainsi, lors du chargement des fichiers de langue par chaque partie du logiciel, les traductions dans chaque langue sont stockées par texte à afficher.

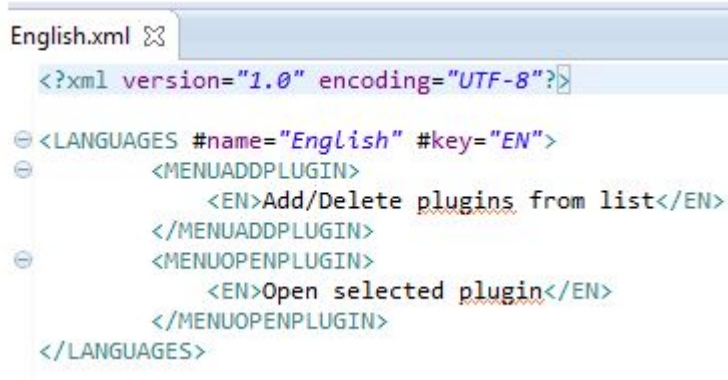
C'est la classe `LanguageLoader` contenue dans le module d'utilitaires du projet (`MedleyUtils.jar`) qui se charge de charger les fichiers de langue. La méthode `load()` de cette classe récupère le contenu

des fichiers .xml dans le dossier Ressources/Languages du module .jar qui sont ensuite parsés grâce à la librairie jdom2.

Lorsque l'utilisateur veut changer de langue, la méthode `changeLanguage(String newLanguage)` de la classe `MainController` est appelée.

Elle va envoyer la clé de la langue choisie aux sous-contrôleurs en appelant leurs méthodes `changeLanguage` respectives qui vont elles-mêmes passer cette clef à leur classe de vue via la méthode `setTexts(String language)` afin que cette dernière puisse modifier ses textes dans la bonne langue.

Exemple de fichier de langue :



```
<?xml version="1.0" encoding="UTF-8"?>
<LANGUAGES #name="English" #key="EN">
  <MENUADDPLUGIN>
    <EN>Add/Delete plugins from list</EN>
  </MENUADDPLUGIN>
  <MENUOPENPLUGIN>
    <EN>Open selected plugin</EN>
  </MENUOPENPLUGIN>
</LANGUAGES>
```

2. Liste de Plugins

- Ajout/Suppression d'un plugin

Lors de l'ajout d'un plugin, la méthode `SaveAndApplyPluginList()` est appelée. Si un plugin est ajouté `addPluginsConf` est appelée et permet d'ajouter le plugin au fichier de configuration `plugins.xml`. S'il est supprimé, `removePluginsConf` l'enlève du fichier. Enfin, la vue (et donc la liste des plugins) est mise à jour grâce au fichier ainsi modifié.

- Lancement d'un plugin

Lors d'un double-click sur un plugin (ou d'un cliqué-déposé), la méthode `openPluginClick(APlugin plugin)` est appelée. Elle fait transiter le plugin par le core qui le transmet au contrôleur de la partie `Workspace`.

3. Workspace

- Lancement d'un plugin

La méthode `launchPlugin(APlugin plugin)` permet d'exécuter un plugin dans la partie `Workspace`. Elle ajoute le plugin à la liste des plugins en cours d'exécution, puis l'envoie à la méthode `addPlugin(PluginsRunningStatus newPlug)` de la vue qui lance le contenu graphique du plugin dans un nouvel onglet.

4. Navigation

- Gestion des onglets de navigation

La gestion des onglets se fait grâce à la classe `NavigationController`. Elle possède toutes les méthodes d'ajout et de suppression d'onglet.

Un onglet correspond à la classe `NavigationTab` qui hérite de la classe « `Tab` » et contient toutes les informations à afficher lors de la sélection d'un dossier.

- Modification des filtres

La modification des extensions de fichiers se fait à partir d'un fichier de configuration situé dans `Ressources/config/navigation.xml`.

5. Création d'un Plugin

Une des classes du `.jar` doit implémenter les méthodes de l'interface `IPlugin` fournie dans la bibliothèque "`plugincontract.jar`" et doit hériter de la classe abstraite `Aplugin`. Et c'est cette classe qui sera appelé par le logiciel pour pouvoir utiliser votre plugin.

Cette classe abstraite hérite de la classe `java.util.Observable` et implémente les méthodes de `IPlugin` et de `java.util.Observer`.

C'est grâce à l'utilisation du design pattern `Observable` que les plugins peuvent communiquer entre eux.

L'objet graphique qui sera afficher lors de l'utilisation du plugin est un `Pane`. C'est un layout d'une scène de `JavaFX`, le plugin sera affiché alors dans l'espace de travail ou dans une fenêtre séparée.

A l'ajout du plugin, un test sera effectué pour vérifier si au moins une classe du `.jar` implémente bien notre contrat. Ce sont les méthodes de cette classe qui seront appelées par `Medley`.

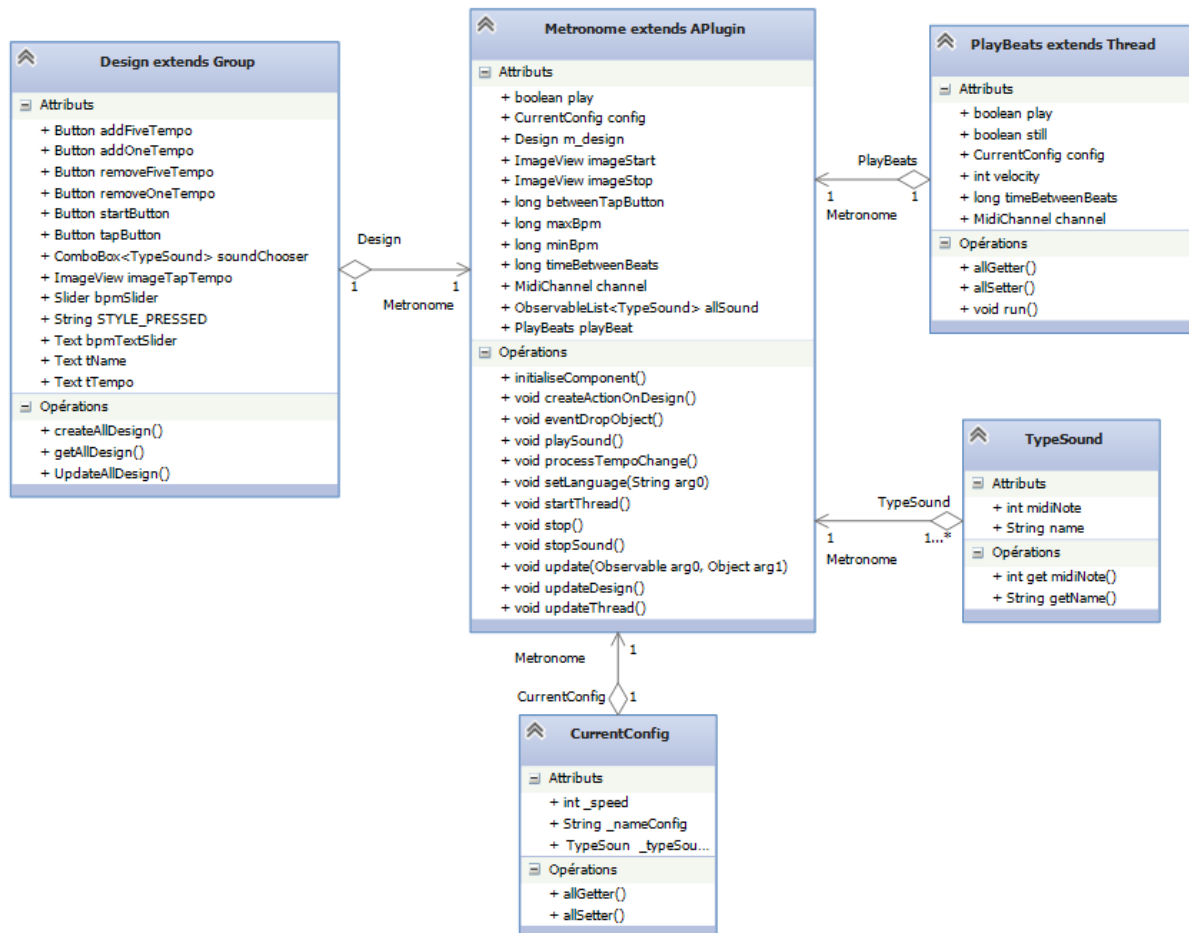
Méthodes à implémenter

L'API `Medley` est composée de 3 méthodes obligatoires et 2 optionnelles :

initialiseComponent	<i>il est nécessaire de la définir</i>	Utilisée pour initialiser tous composants utiles au bon fonctionnement du plugin.
stop	<i>il est nécessaire de la définir</i>	Utilisée pour arrêter le plugin en cours d'exécution.
update(Observable arg0, Object arg1)	<i>il est nécessaire de la définir</i>	Réception des données des autres plugins. <code>arg0</code> étant le plugin envoyant des données et <code>arg1</code> les données reçu.
eventDropObject()	<i>optionnelle</i>	Utilisée pour effectuer un traitement sur les arguments passés en paramètre via l'event <code>inputArgs</code> .
SetLanguage(String key)	<i>optionnelle</i>	Utilisée pour appliquer le changement de langue du logiciel par le plugin. <code>Key</code> étant la nouvelle langue du logiciel.

b) Plugins

1. Métronome



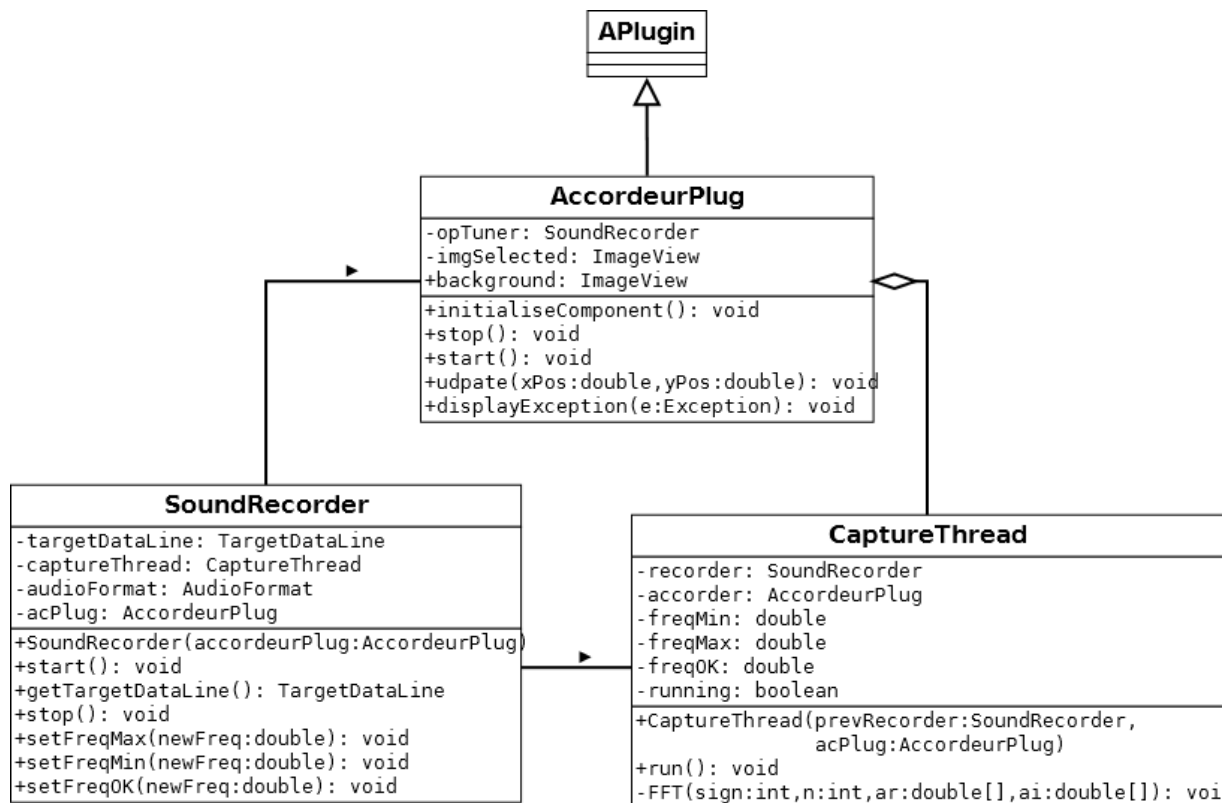
PluginMetronome implémente les besoins de l'API ce qui permet que ce plugin soit en règle avec le contrat.

LoadConfig permet de sauvegarder/charger un tempo.

Config est la config actuelle que va pouvoir jouer le Métronome grâce à la classe MetronomeSound.

TypeSound sont les différents type de son possible.

2. Accordeur



La classe AccordeurPlug hérite de APlugin, c'est donc cette classe qui implémente l'API.

La méthode initialiseComponent() permet d'initialiser tout ce qui sera nécessaire au fonctionnement de notre accordeur.

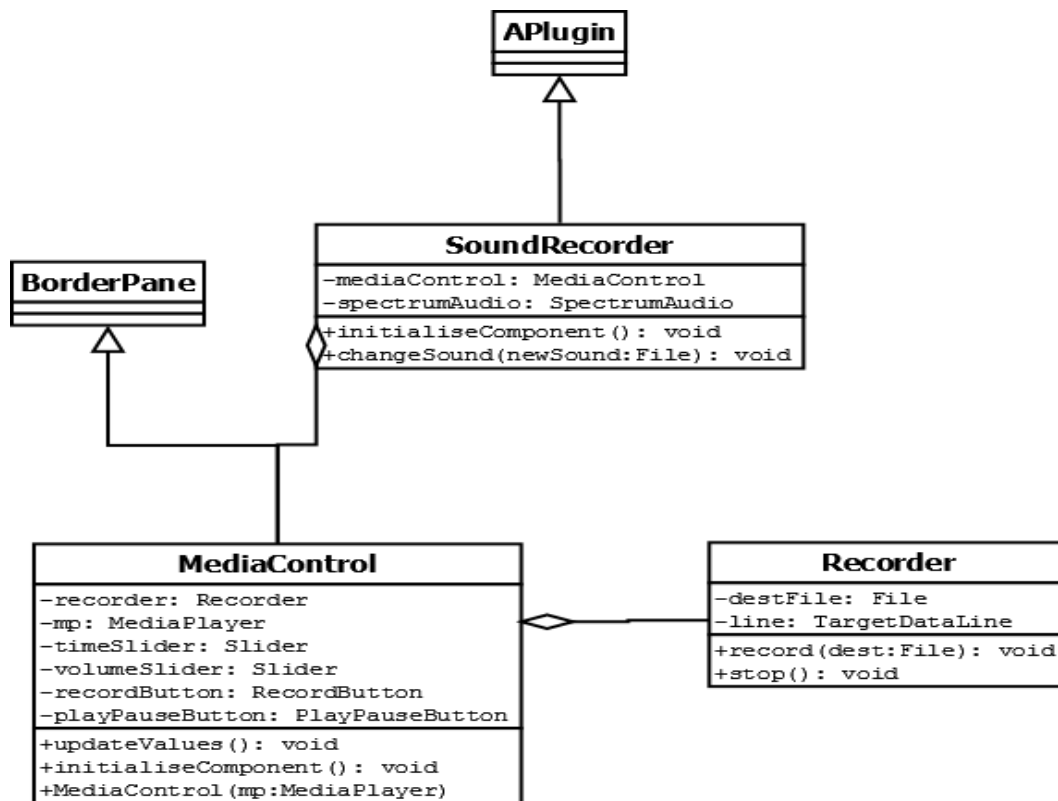
La classe SoundRecorder va permettre de paramétrer l'enregistrement de son en initialisant des objets AudioFormat et TargetDataLine. AudioFormat permet de manipuler le format d'enregistrement et la TargetDataLine sera le buffer sur lequel sera enregistré le son.

La méthode start() du SoundRecorder va initialiser un CaptureThread qui va enregistrer et analyser le son.

Une fois le CaptureThread créé on appelle la méthode start() de Thread qui appellera le run() de capture Thread. La méthode run() de CaptureThread() enregistre du son pendant 1 seconde puis appelle la méthode FFT().

FFT() utilise l'algorithme « Fast Fourier Transform », son but est de lisser le son enregistré en se basant sur les attributs freqMin, freqMax et freqOK. Une fois l'algorithme passé il appelle la méthode update() de AccordeurPlug en lui passant les nouvelles coordonnées de l'aiguille.

3. Enregistreur/Lecteur de son



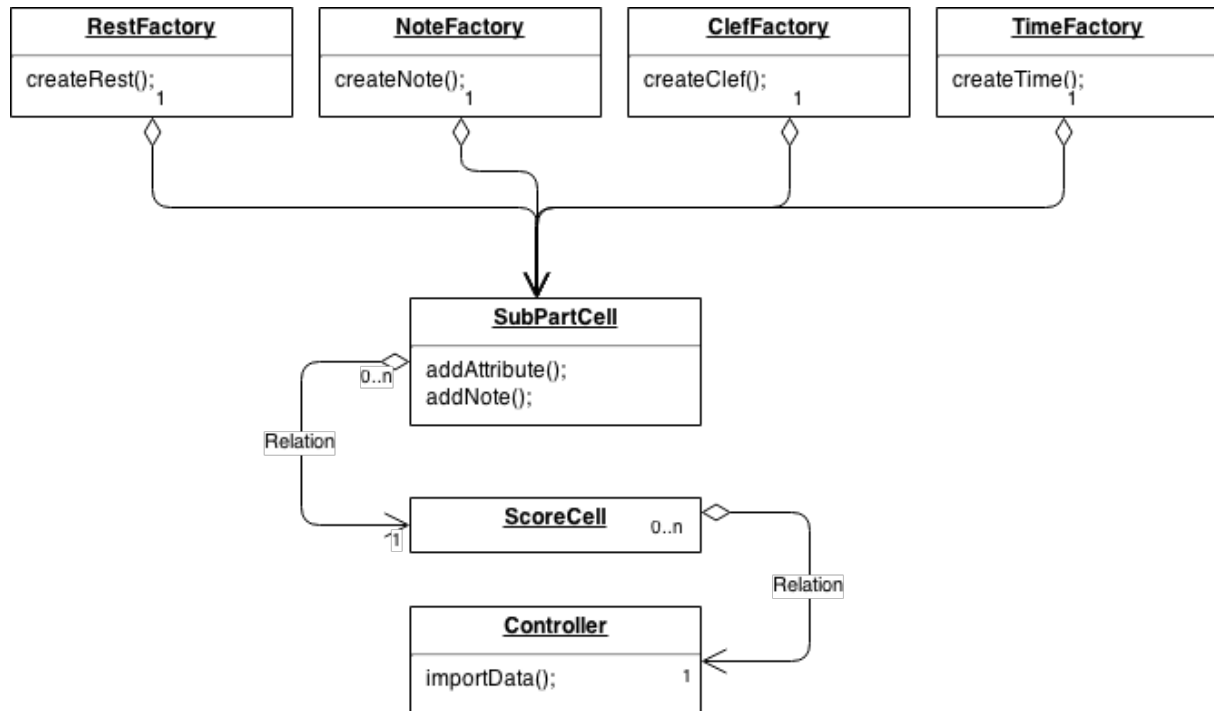
Le **SoundRecorder** initialise un **MediaPlayer** à l'aide d'un fichier de son que nous lui passons en paramètre, ce **MediaPlayer** est envoyé au constructeur de **MediaControl**, il contiendra le média à lire. La classe **MediaControl** va permettre de contrôler les médias que l'on souhaite lire. Les contrôleurs `volumeSlider`, `playPauseButton` et `recordButton` permettent à l'utilisateur de contrôler le flux audio ou d'enregistrer un son. Quand un `Slider` est modifié, la méthode `updateValues()` est appelée pour mettre à jour les informations de lecture.

La classe **Recorder** permet d'enregistrer un son en appelant sa méthode `record`, cette méthode prend en paramètre le fichier de destination.

Lorsque l'on appelle la fonction `record()` un thread écoutant les entrée du microphone va être ouvert. Une fois que la méthode `stop()` est appelée, l'enregistrement va s'arrêter.

SoundRecorder hérite de **APlugin** et implémente toute ses méthodes, `initialiseComponent()` va initialiser les éléments graphiques et le contrôleur, `eventInputArgs()` va permettre de récupérer les fichiers envoyés, cette méthode appelle `changeSound()` qui va changer le fichier pris en charge par le contrôleur.

4. Lecteur de partition



Le lecteur de partition a été réalisé avec JavaFX.

Ce plugin utilise la librairie JFugue pour importer un fichier au format MusicXML, le parsing de ce fichier se fait grâce à la librairie ProxyMusic. Cette librairie permet de transformer les données du fichier en objet. Les objets générés permettent de peupler la vue du lecteur via une Factory.

La Factory détermine le type de l'objet, puis, suivant le type de l'objet (une clé, une note, etc...) un objet visuel est créé et fourni à la vue qui l'affiche.

La lecture du fichier importé se fait à l'aide de JFugue.

5. Editeur de partition

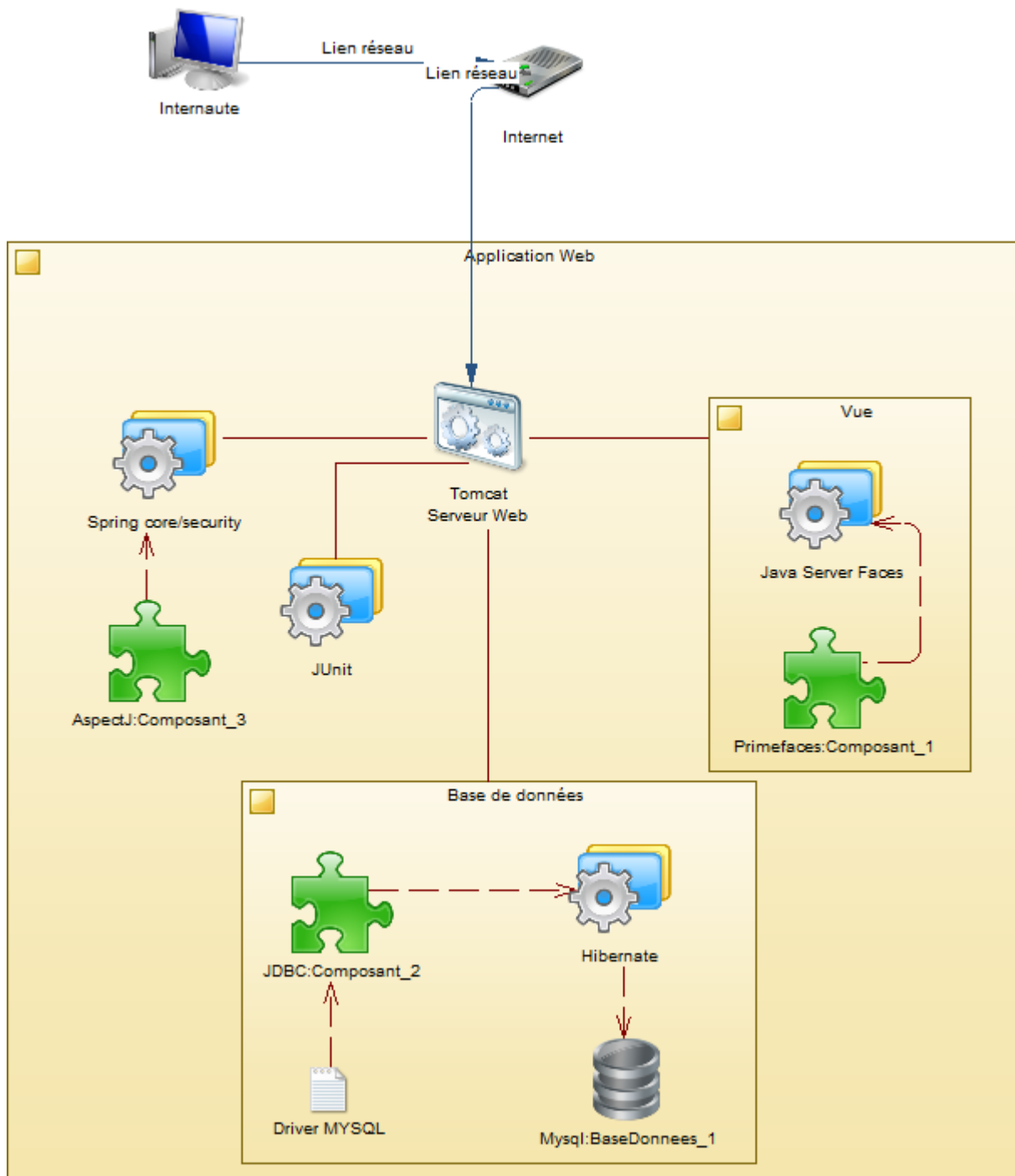
L'éditeur de partition fonctionne exactement de la même manière que le lecteur de partition à l'exception que celui-ci n'importe pas de fichier mais en export au format MusicXML.

Les Factory permettant de peupler la vue sont appelées via des actions utilisateurs et non par un parsing d'un fichier.

c) Site

1. Vue globale

Si nous reprenons les frameworks/librairies précédents dans un diagramme de composants nous obtenons ceci :



Vue globale des composants

2. Couche applicative

Pour expliquer le fonctionnement de notre site internet au niveau objet, nous avons réalisé le diagramme de classe suivant. Pour plus de clarté, il ne représente que les classes impliquées lors de l'affichage de la page de téléchargement de plugins. Néanmoins, toutes les pages du site internet suivent le même principe de fonctionnement.

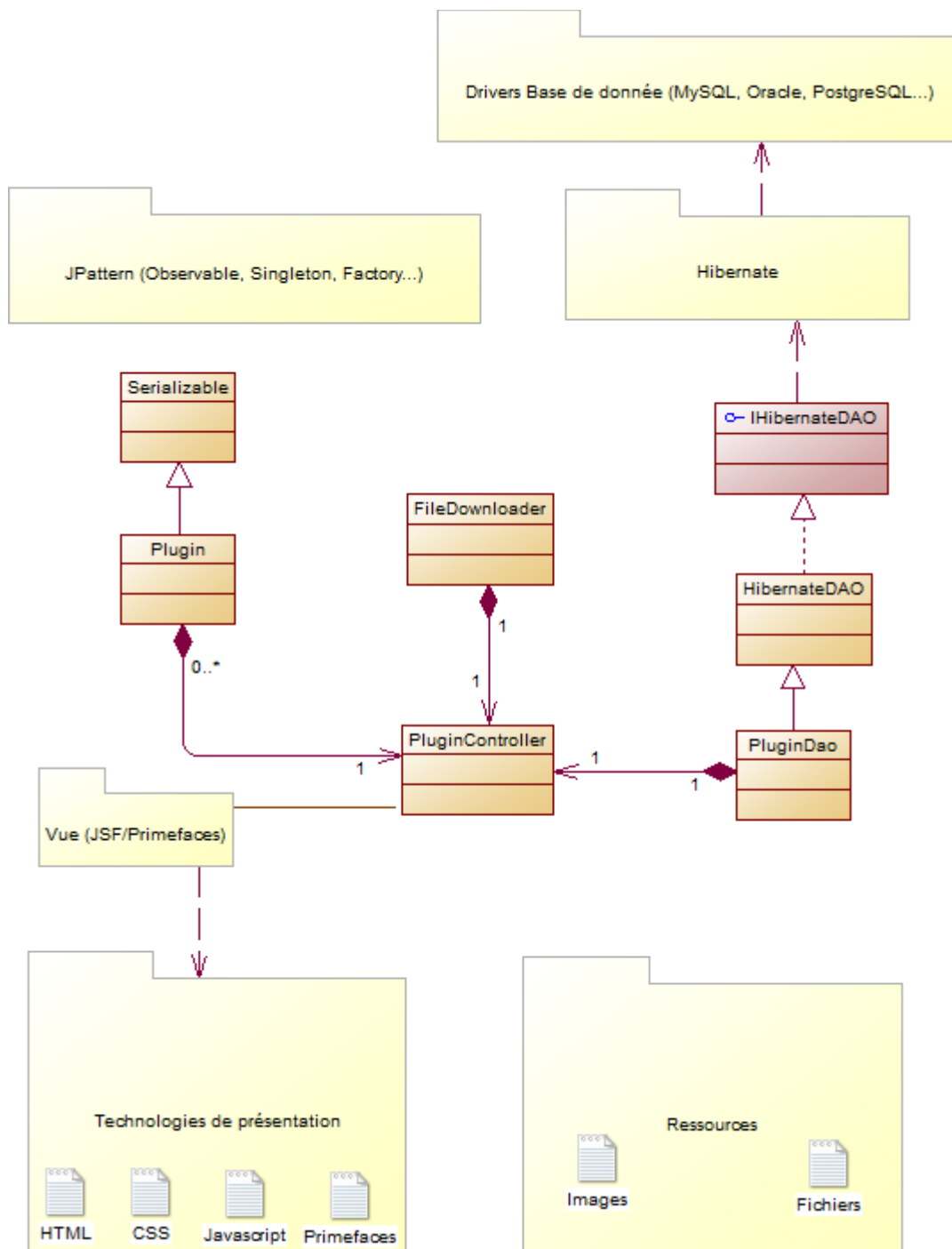


Diagramme de classe générique

Les classes les plus importantes de ce diagramme sont :

- **PluginController** (package **controllers**) : C'est uniquement cette classe que nous appelons lors de la création de notre vue de téléchargement de plugin. À sa création, elle va récupérer la liste des plugins contenus dans la base de données et peut effectuer des appels aux classes de traitements si besoin (télécharger un plugin par exemple). C'est-à-dire que **PluginController** n'effectuera jamais de traitements directs sur nos données.
- **Plugin** (package **models.bl.bean**) : C'est une classe de stockage uniquement, elle contient les informations d'un plugin et elle va nous permettre d'utiliser les données récupérées de la base.
- **PluginDao** (package **models.dal.dao**) : C'est notre Data Access Object, qui va communiquer avec la base de données et appliquer les opérations CRUD (Create Read Update Delete). Elle hérite d'une classe HibernateDAO qui va, en fonction du type passé au constructeur, récupérer notre fichier de description de la table et faire le mapping entre celle-ci et notre bean (ici la classe **Plugin**).
- **FileDownloader** (package **models.bl.managedBean**) : Le package **managedBean** contient les classes de traitement dont nous avons besoin et que vont utiliser les controllers. Ici cette classe va être utile pour le téléchargement de nos fichiers.

Ces classes sont les seules que nous devons développer pour chaque nouvelle page demandant des traitements. De cette manière, on garde une architecture claire et chaque type de classe a son propre rôle. Le fonctionnement d'Hibernate nous permet également d'avoir un code facilement adaptable à notre base de données. Par exemple, si nous souhaitons rajouter un champ dans notre table des plugins, nous n'aurons par la suite qu'à modifier sa description dans un fichier de config et rajouter ce nouveau champ dans notre classe Plugin.

Tous les fichiers de configuration se trouvent dans le dossier **Resources** du projet. Vous pourrez ici modifier la description de la base de données (pour Hibernate), gérer la sécurité (pour Spring Security) et gérer le multi-db (avec JDBC).

Comme librairies, nous utilisons notre **package JPattern** qui contient toutes les interfaces et classes abstraites de nos design patterns ainsi que le **driver JDBC** de connexion à la base de donnée choisie. La base de données peut ainsi être changée très rapidement en téléchargeant le nouveau pilote correspondant et en modifiant quelques lignes de code (URL, identifiants). On trouve aussi dans ce diagramme nos **technologies de présentation** (HTML, CSS, Javascript et les composants Primefaces) ainsi que les **ressources** dont nous auront besoin, notamment les images, vidéos, logiciel, plugins et documents.

Diagramme de séquence :

Nous avons également réalisé un diagramme de séquence qui permet d'expliquer de manière temporelle les liens entre ces classes. De la même manière que notre diagramme de classe,

il représente l'action d'aller sur la page de téléchargement des plugins, mais les étapes sont les mêmes quel que soit la page demandée.

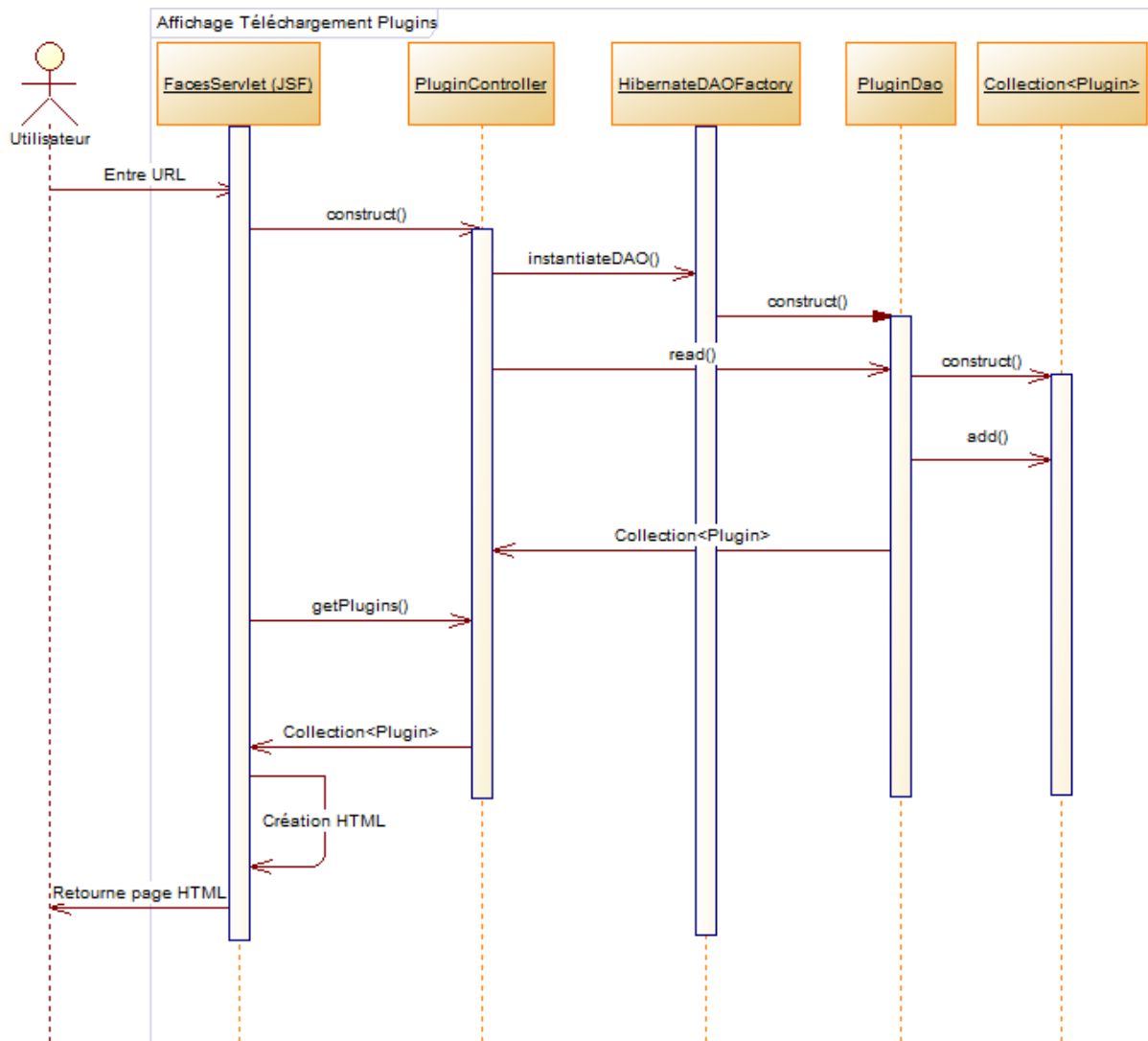


Diagramme de séquence générique

Lorsqu'un internaute va cliquer sur la page de téléchargement des plugins, le framework **JSF** va automatiquement le rediriger vers une page HTML. Cette page va avoir besoin de la liste des plugins actuellement dans la base.

Grâce aux Expressions Language (EL) de JSF, une instance de la classe **PluginController** va être créée.

Celle-ci va demander à la classe **HibernateDAOFactory** de lui retourner le DAO correspondant, **PluginDao**, tout en créant la connexion à la base de données. Une fois le DAO récupéré,

PluginController va appeler sa méthode **read** (par exemple), avec ou sans critère, pour récupérer les plugins qui l'intéressent.

Une fois la liste de **Plugin** récupérée, la page HTML et les composants Primefaces, formateront les informations et l'internaute recevra donc la page HTML qu'il a demandée.
La durée de vie de ces classes est directement gérée par le framework Spring.

Documentation Doxygen :

La documentation des sources a été mise à jour à l'adresse suivante :

<http://eip.epitech.eu/2015/medley/Doxygen/website>

Les fichiers de configurations ainsi que les fichiers de présentation (html/css/Javascript/images) ne sont pas inclus dans cette Doxygen. Les quatre packages les plus importants sont ceux cités ci-dessus, à savoir : **Bean**, **ManagedBean**, **Controllers** ainsi que **Dao**.

3. Webservices

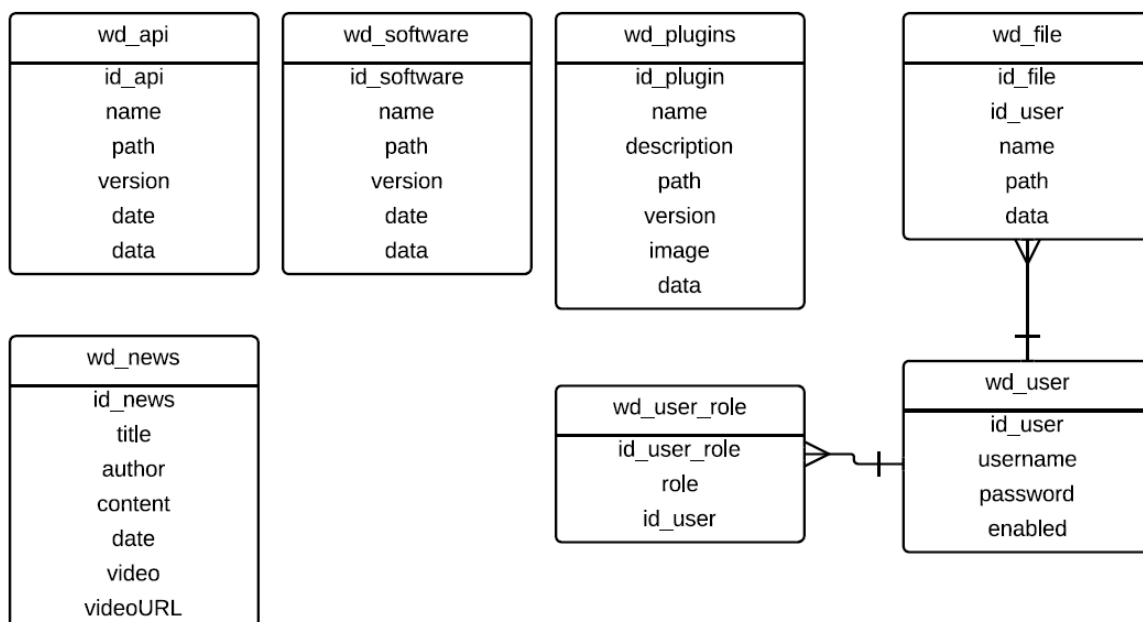
Nous avons également mis en place deux webservices sur notre serveur étant destinés aux opérations suivantes :

- La mise à jour du logiciel
- La mise à jour des plugins au sein du logiciel

Ces deux webservices ont été réalisés à l'aide de **JAX-WS** (Java API for XML Web Services).

4. Base de données

Voici les tables actuellement créées pour nos bases de données (mysql et postgresQL) :



Les tables **wd_api**, **wd_software** et **wd_plugins** sont les tables utilisées pour stocker respectivement nos différentes versions d'API, de logiciel et nos plugins. **Wd_news** permet de stocker les différentes news et **wd_file** les fichiers utilisateurs.