Université Pierre et Marie Curie

Master Informatique, Spécialité STL

Semestre 2, UE Projet STL

Un analyseur syntaxique pour MusicXML

Auteurs :
Sébastien DUCHENNE
Alexandre GASPARD CILIA

Encadrant :
Pr. Carlos AGON

Soutenue le 22 mai 2017

Table des matières

1 Remerciements

Nous souhaitons remercier notre encadrant Carlos Agon qui nous a beaucoup aidé lors de ce projet. Ses cours de musique et une petite visite de l'IRCAM nous ont permis de découvrir ce domaine que nous connaissions peu. Ses idées et ses conseils nous ont aussi beaucoup apporté pour la réalisation du programme.

Nous remercions également Karim Haddad pour ses explications de quelques éléments de la musique.

Table des figures

2 Introduction

L'IRCAM [1], Institut de Recherche et Coordination en Acoustique/Musique, est un centre de création et de recherche scientifique sur la musique. Il a été fondé en 1969 par Pierre Boulez à la demande du président Georges Pompidou.

En 1995, le CNRS et le ministère de la Culture et de la Communication s'associe et créée l'UMR 9912 STMS. Cette unité mixte de recherche, hébergée à l'IRCAM, s'intéressent aux sciences et aux technologies de la musique et du son. En 2010, elle est rejoint par l'UPMC.

Cette UMR est composé de nombreuses équipes de recherche. L'une d'elles s'intitule "Représentations Musicales", et réalise des outils de compositions musicales. Elle a notamment créée OpenMusic [2], un environnement de composition musicale assisté par ordinateur.

Les chercheurs de l'IRCAM développent actuellement un logiciel d'analyse musical se basant sur des partitions. Pour faciliter les échangent, ils souhaitent disposer de ressources leur permettant de lire des fichiers au format MusicXML.

Ce programme sera écrit en langage Java. Il permettra donc d'éditer graphiquement des structures de données musicales à partir d'un fichier. Son implémentation comportera différents modules, dont celui consistant à construire les arbres rythmiques à partir d'un fichier au format MusicXML. C'est ce module que notre tuteur de projet Carlos Agon, enseignant-chercheur à l'UPMC et membre de l'équipe de représentations musicales, nous a demandé de réaliser.

Le module que nous avons développé est déposé sur un compte GitHub [3] public. Il est donc open source. Il a été réalisé en anglais pour favoriser son internationalisation.

3 Technologies utilisées

3.1 Le langage Java

Java [4] est un langage de programmation orienté objet fortement typé développé par Sun Microsystems à partir de 1995. La société sera plus tard rachetée par Oracle en 2009 qui possède et maintient Java encore aujourd'hui.

Java se détache de la masse des autres langages de programmation notamment grâce à sa portabilité et sa facilité d'utilisation.

Listing 1 – Hello world en java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Ci-dessus, un classique "Hello world" en Java. Nous pouvons y voir la définition de la classe *HelloWorld* ainsi que la méthode principale du programme nommé *main* effectuant un affichage sur la sortie standard.

3.2 Le langage XML

Le langage XML [5, 6], acronyme de e**X**tensible **M**arkup **L**angage, est langage de balisage générique spécifié par le W3C. Il permet de définir différents espaces de noms, c'est à dire des langages avec leur propre grammaire et vocabulaire. Il permet l'échange d'information entre des programmes très différents à condition d'utiliser la même grammaire.

Il a l'avantage de pouvoir être compris par les êtres humains et les machines. Cependant, c'est un langage verbeux et qui peut donc prendre beaucoup de place s'il contient beaucoup d'information.

Un document XML est constitué de balises pouvant contenir d'autres balises ou une valeur simple. Une balise peut aussi contenir des attributs donnant des informations supplémentaires sur le contenu.

Listing 2 – Exemple d'un document XML

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
    <balise attribut="valeur">Contenu</balise>
    <baliseunique />
</racine>
```

Ci-dessus un exemple de XML simpliste mais qui met en avant les bases du langage. La première ligne annonce le type de document et la version dans lequel le document va être rédigé.

<racine> est le nœud racine du document, celui qui en somme va contenir tout le document. On peut ici, facilement remarquer que le document peut être représenté sous la forme d'un arbre.

XML permet à l'utilisateur de définir lui même la grammaire de son document grâce notamment aux DTD et au Schéma XML. Ces outils nous permettent de disposer de format d'échange de données tel que MusicXML.

3.3 Le format MusicXML

MusicXML [7] est un format de fichier permettant de représenter la notation musicale occidentale (notation classique, accords en notation anglo-saxonne, tablatures et percussions). Basé sur le langage XML, il est propriétaire mais il peut librement être utilisé avec une licence publique.

Dans ce projet, nous nous intéresserons aux différents symboles de la notation classique, à savoir les éléments de base, les ornements, les articulations et les répétitions. Nous nous sommes documentés sur cette notation [8] afin de bien comprendre les différentes balises du format.

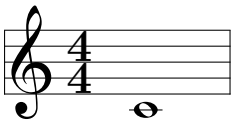
Il y a plus de 20 ans, le format MIDI était très utilisé. Cependant, il n'est pas très adapté pour représenter toutes les caractéristiques de la musique, on perd donc en informations avec ce format. Pour pallier à cela, les formats SMDL et NIFF ont été créés. Cependant, le format SMDL était complexe et donc peu compréhensible. Il était donc très peu utilisé. Le format NIFF était un format peu pratique à utiliser et n'a donc pas été adopté par certains logiciels. Ces formats n'ont donc pas eu le succès souhaité.

En 2004, la société Recordare LLC s'inspire des 2 formats universitaires MuseData et Humdrum pour créer la première version du format MusicXML. Ses avantages sont qu'il est facile à manipuler. Il permet le transfert de morceaux de musique d'une application à une autre. Il peut représenter beaucoup de caractéristiques de la musique. Cependant, il est verbeux, puisqu'il utilise le format XML.

Il est de plus en plus utilisé puisque plus de 200 logiciels de musique l'ont adopté. Il est donc possible de travailler finement sur un morceau de musique en utilisant différents programmes.

Comme le format XML est verbeux, le fichier prend de la place. La version 2.0, sortie en 2007, apporte donc la compression du fichier au format xml en un fichier au format mxl, et permet de diviser sa taille de façon importante. La version 3.0, sortie en 2011, permet le support des instruments virtuels.

On voit, dans le code correspondant à la partition suivante, que les informations sur la partition sont placées dans la balise "measure" et celle concernant la ronde sont contenues dans la balise "note".



Listing 3 – Document XML d'un Hello World en MusicXML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 3.0 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="3.0">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

3.4 Le langage Relax NG

Relax NG (**Re**gular **La**nguage for **X**ML **N**ext **G**eneration) [9] est né de la fusion de TreX de James Clark et de Relax de Murata Makoto. C'est un langage qui permet de définir la grammaire d'un document XML. Relax NG ne s'intéresse qu'à la structure du document et non à sa valeur.

Relax NG s'utilise assez facilement. Il faut tout d'abord écrire la grammaire du document *XML* que nous souhaitons définir. Pour ce programme nous utilisons une grammaire écrite par

[16]. Nous devons ensuite la compiler et l'utiliser pour valider des documents grâce aux outils fourni par [9].

C'est ce que nous utiliserons afin de s'assurer de la validité du document à traiter.

3.5 Les librairies

Dans cette section, nous aborderons les librairies utilisées pour réaliser ce projet. Cela ira de la validation du XML en passant par le parsing de ce dernier, jusqu'à la récupération des information stockées dans le fichier MusicXML.

3.5.1 Trang et Jing

Trang [10] et **Jing** [11] sont deux librairies développées par Thai Open Source. Elles permettent de générer des grammaires Relax NG et de valider des documents XML à partir de cette même grammaire.

Trang est une librairie qui permet de traduire un fichier de description grammaticale en fichier Relax NG. En effet XML n'est pas facilement lisible pour un esprit humain, c'est pour cela que Trang nous permet de créer notre grammaire dans un langage plus compréhensible. Une fois la grammaire écrite dans un fichier en *.rnc*, nous pouvons générer notre fichier Relax NG en *.rng*.

Jing, quant à lui, est une librairie Java qui permet de valider un document XML à l'aide d'un fichier Relax NG.

Listing 4 – Code java permettant de vérifier la validation d'un document XML

```
final ValidationDriver vd = new ValidationDriver();
vd.loadSchema(rngFile);

if (!vd.validate(inputTextStream)) {
        throw new ParseException("Invalid xml :(");
} else {
    System.out.println("Valid xml :)");
}
```

Le code ci-dessus est une utilisation simplifiée de Jing. Nous commençons tout d'abord par créer un objet *ValidationDriver* de Jing dans lequel nous chargeons notre fichier Relax NG. Nous n'avons ensuite plus qu'à lancer la méthode *validate(InputSource in) : boolean* qui nous indiquera si le document est valide.

3.5.2 L'API SAX

SAX [12, 13] est une API créée par David Megginson en 1998 et est l'acronyme de **S**imple **A**PI for **X**ML. Elle permet de manipuler des documents XML en utilisant des événements envoyées à chaque rencontre d'un élément.

3.5.3 Le DOM

Le DOM [14], ou **D**ocument **O**bject **M**odel, est une interface de programmation normalisée par le W3C et est indépendant de tout plateforme et langage. Il voit les documents à balises comme des arbres dont le contenu et la structure peuvent être accédés et mis à jour dynamiquement.

La figure suivante montre un exemple de DOM.

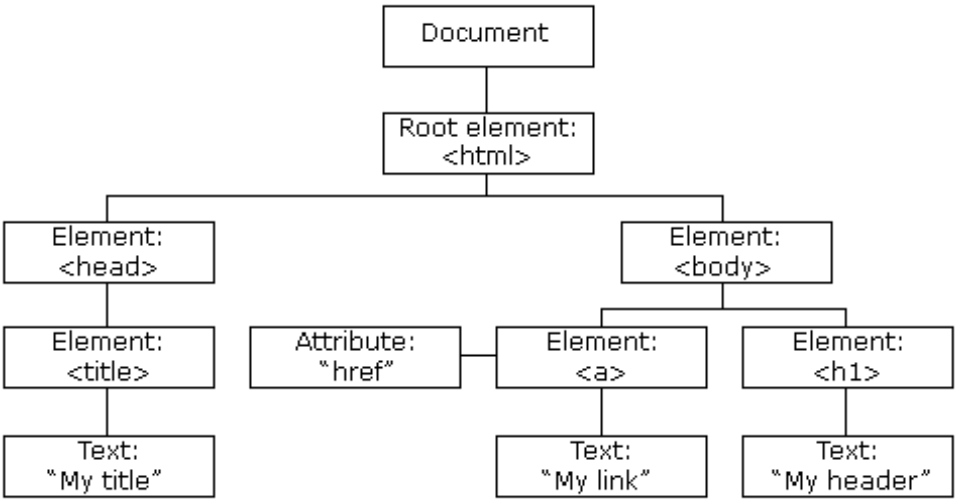


FIGURE 1 – Exemple d'un DOM d'une page HTML

4 La notation musicale

4.1 La partition

Une partition est un document contenant toutes les informations jugées nécessaires par l'auteur pour pouvoir interpréter une musique. Elle est composée de portées qui contiennent elles-même des mesures. Une mesure dispose d'une clé, d'une armure et d'une signature si elle est la première de sa portée et une ensemble de notes, accords ou silences.

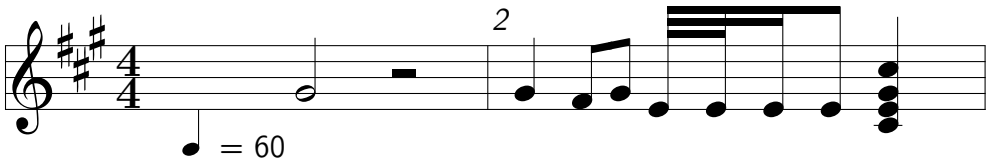


FIGURE 2 – Exemple d'une partition

Une partition peut comporter plusieurs instruments ou voix. Une voix peut être formée de une (chant) ou deux portées (piano).

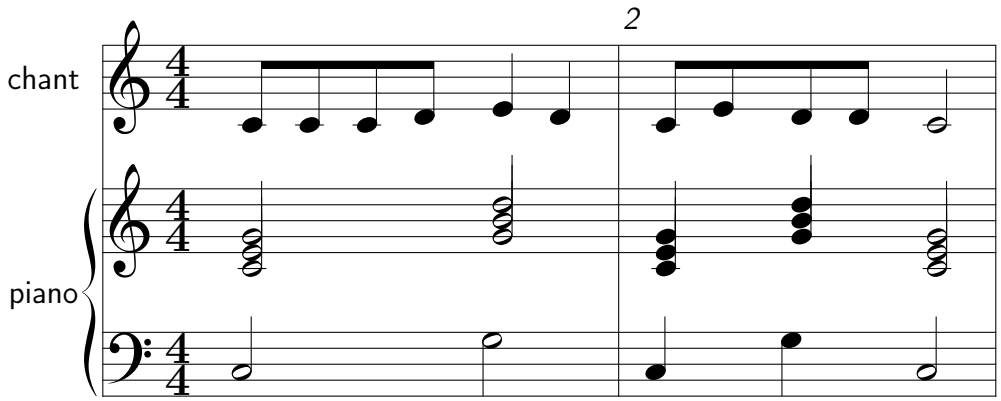


FIGURE 3 – Plusieurs portées et voix

4.2 La durée

La durée est le temps pendant laquelle la note est jouée. La durée n'est pas exprimée sous forme de valeur fixe mais comme une proportion relative à la mesure. En effet, la durée de la mesure dépend de sa signature et du tempo (expliqués plus tard).

4.3 La signature

La signature symbolise le référentiel de la durée de la mesure. Elle peut être représentée sous forme d'une fraction. Le numérateur permet de définir la durée totale de la mesure et le dénominateur la durée d'un temps dans la mesure.

4.4 La note

Une note est définie par sa hauteur et sa durée. La hauteur caractérise la fréquence du son d'une note. Cette fréquence est divisée en 8 intervalles appelés octave. Cette octave est divisée en 7 hauteurs : do, do#, ré, ré#, mi, fa, fa#, sol, la, la#, si et si#. En ce qui concerne la durée de la note, elle est indiquée par la tête de la note et la présence ou non de symboles de croches. Si l'on ne prend pas en compte les croches, il existe 4 types de durées pour une note : carrée, ronde, blanche et noire. Une note carrée vaut 2 rondes, 4 blanches et 8 noires. Afin d'exprimer une durée plus courte que celle de la noire, on peut lui rajouter un ou plusieurs symboles de croches. La durée de la note s'en retrouve divisée par deux à chaque ajout.

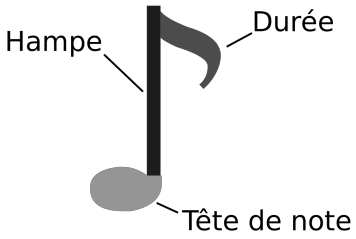


FIGURE 4 – Symbole d'une note

Dans notre programme, tout comme dans MusicXML, nous utilisons la notation anglo-saxonne. Elle diffère de la notation française par la façon dont est exprimée la hauteur et la durée de la note. Do, ré, mi, fa, sol, la et si sont remplacées respectivement par C, D, E, F, G, A et B. Concernant la durée, ce sont simplement des fractions. Par exemple, la blanche est un 1/2 (half) et la noire 1/4 (quarter).

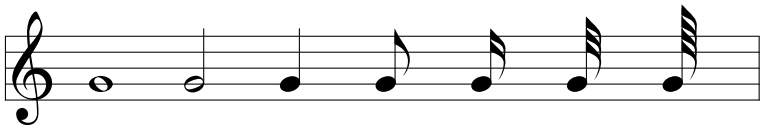


FIGURE 5 – Les différents figures de notes

4.5 Le silence

Un silence est un arrêt momentané dans l'exécution d'une œuvre. Tout comme les notes, il existe différentes figures de silence variant en fonction de leur durée. La pause a la même durée qu'une ronde et se situe sous la quatrième ligne de la portée. La demi-pause, d'une durée égale à la blanche, se place sur la troisième ligne. Les autres figures sont le soupir, le demi-soupir, le quart de soupir, huitième de soupir, seizième de soupir, etc. La figure suivante montre ces figures de silence.

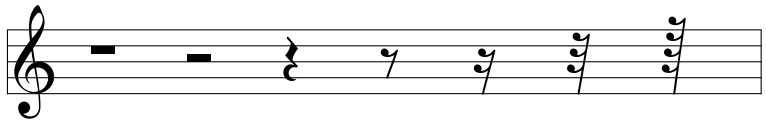


FIGURE 6 – Les différents figures de silence

4.6 Le tempo

Le tempo correspond à la vitesse d'exécution d'un morceau de musique. Il contient un type (une noire par exemple) et un nombre par minute. Ainsi, le tempo *60 à la noire* signifie qu'il y a, dans une minute, 60 noires ou bien 30 blanches. Il se place en début de mesure mais peut parfois être absent. L'interprète peut dans ce cas jouer le morceau à la vitesse souhaité.

4.7 Les symboles musicaux

Nous désignons par symbole musical, chaque artefact influençant la manière de jouer une note ou un groupe de notes.

Certains symboles ne s'appliquent que sur une figure de note. Ils sont dits unaires. La figure suivante montre quelques exemples de symboles unaires.

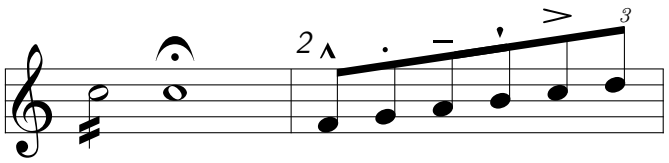


FIGURE 7 – Symboles unaires

Les symboles influençant un groupe de notes sont appelés symboles binaires. Les 4 symboles suivant sont une liaison, un triolet, une appoggiature et un groupe de croches. Ce dernier est

constitué de plusieurs croches reliées entre elles par une ou plusieurs barres selon la durée des notes. Une telle barre est appelée *beam* en anglais.

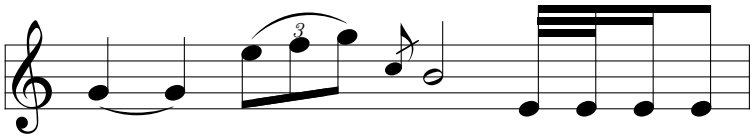


FIGURE 8 – Symboles binaires

Des symboles servent à répéter les notes d'une ou de plusieurs mesures.

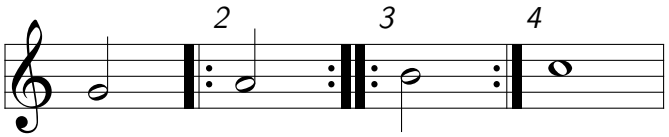


FIGURE 9 – Symboles de répétitions

5 Architecture du programme

Dans cette section nous aborderons l'architecture du programme. C'est-à-dire la façon dont est organisé notre code et notamment les choix d'implémentation que nous avons effectués.

Notre objectif ici est de concevoir un programme nous permettant de lire et de traduire un fichier au format *MusicXML* et en une liste de *voix*. Plus précisement, chaque *voix* doit comporter entre autres son *tempo*, sa liste d'*accord* et de *symboles* et enfin son *arbre rythmique*. Seulement, l'*arbre rythmique* étant une structure qui n'est pas standart dans le monde de la musique, il nous a donc fallut le générer à partir des informations contenus dans le fichier *MusicXML*.

5.1 Analyse du problème

Le format *MusicXML*, et plus généralement une partition musicale, peut comporter énormément de symboles différents. C'est pour cela que nous avons du nous adapter à la structure très "personnalisable" et penser le code pour qu'il soit très malléable. Pour ce faire, nous avons gardé un niveau d'abstraction assez élevé pour pouvoir implémenter chaque symbole en modifiant le minimum de code. Lors de la réflexion qui a découlé de ce constat, nous avons pu observer deux types de symbole : les symboles unaires et les symboles binaires.

Les symboles unaires sont assez simple à implémenter dans le sens où ils n'influence, pour la plupart du temps, qu'une seule note. Il nous suffit donc de les stocker dans ladite note.

En ce qui concerne les symboles binaires, l'implémentation est beaucoup plus contraignante. Certains symboles, comme les groupes de croches, ont des marqueurs de début, de continuation et de fin, alors que d'autres, comme les groupes de liaison, n'ont qu'un début et une fin. Cela nous oblige donc à faire du sur-mesure pour certains symboles.

Il demeure cependant des symboles qui peuvent concerner une mesure comme la partition. La balise *<metronome>* par exemple est renseignée dans la mesure de la même façon qu'une note. Il faut donc prendre en compte ces exceptions.

5.2 Architecture générale

Le programme se divise en trois parties. La première, le parser, permet au programme de disposer d'un *DOM* à partir d'un fichier *MusicXML*. La deuxième partie range les données obtenues grâce à la première dans des structures de données afin de pouvoir les traiter plus aisément. Et enfin le code ayant pour but de générer l'*arbre rythmique* constitue la troisième partie.

5.3 Parsing du fichier MusicXML

Avant d'entreprendre la manipulation de données il faut bien entendu disposer des dites données. C'est là où le code contenu dans le package *pstl.musicxml.parsing* entre en action. Nous avons choisi d'encapsuler toutes les fonctionnalités utiles au parser dans la classe *XMLParser*.

L'encapsulation de cette classe nous permet d'utiliser le parser de façon très simple. Ce n'était pas le cas lors des premières versions à cause notament de *Relax NG*. En effet comme nous ne faisons appel ni aux fichiers *DTD* ni aux fichiers *XML Schema*, il nous faut effectuer un prétraitement pour éliminer les références aux *DTD* du fichier à parser. Vous pourriez vous demander pourquoi ne pas simplement utiliser les *DTD* pour valider le document ? La raison est simple, la plupart des fichiers font référence aux *DTD* en ligne fournis par MusicXML. Or ces derniers n'autorisent que les navigateurs web à accéder à de tels fichiers. Les possibilités qui s'offraient à nous étaient les suivantes : se faire passer pour un navigateur en modifiant quelques variables d'environnement. Cela aurait eu pour désavantage tout d'abord de ne pas être très rapide, l'accès à aux *DTD* par réseau n'est pas très rapide comparé à un accès local. Nous jugions d'autre part la méthode peu honnête. En effet si l'organisation derrière MusicXML ne permet pas cela pour des raisons que nous imaginons financières (par cela j'entends le coût engendré par la maintenance des serveurs), il n'aurait pas été juste d'outrepasser leurs instructions. Et enfin le dernier choix non retenu était celui de faire appel à des *DTD* stockées localement. Cette méthode a été écartée bien que conseillée dans la documentation du format MusicXML car elle impliquait des redirections d'URI ce qui aurait fortement complexifié la création du parser.

C'est donc pour toutes ces raisons que nous avons choisi d'utiliser *Relax NG*. De plus, les fichiers décrivant la grammaire de MusicXML sont disponibles en ligne et l'auteur, qui les a déposés sur un projet GitHub [16], a fait preuve d'une certaine exhaustivité lors de la rédaction de ces derniers.

Ce parser nous permet de disposer d'un *DOM* (qui a pour nom de classe *Document* avec Java) qui pourra être parcouru plus tard. Le schéma suivant récapitule les étapes pour passer du document XML au DOM.

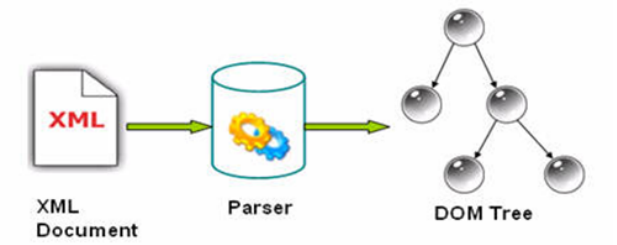


FIGURE 10 – Parsing d'un document XML en DOM

5.4 Représentation objet de la partition

Nous aurions pu nous satisfaire du *Document* retourné par le parser pour générer notre arbre rythmique mais cela aurait posé plusieurs problèmes. Tout d'abord la manipulation du *DOM* n'est pas aisée. Cette structure de données basé sur des nœuds qui contiennent des fils, attributs et leur contenu textuel, doit être exploré d'une façon assez lourde à cause en partie du fait qu'un nœud n'est pas nativement un objet itérable comme les *Collections* de Java par exemple. D'autre part, toutes les données contenues dans ces nœuds sont considérées comme des chaînes de caractère qui nécessitent un parsing et donc une manipulation assez verbeuse. De ce fait, il est plus simple pour nous de parcourir ce *Document* une seule fois et d'extraire les données qu'il contient dans une structure de données plus aisément manipulable dans Java. Cela permet par exemple d'éviter des erreurs lors du développement des autres fonctionnalités de l'application qui se basent sur ces informations. Nous utilisons donc un ensemble de méthodes contenus dans la classe *ScoreUtils* pour convertir notre *Document* en instance de *Score*.

ScoreUtils intègre aussi une partie très importante : la récupération des symboles contenus dans la partition. En effet, une partition n'est pas seulement un ensemble de notes, il contient aussi un grand nombre de symboles ayant tous des significations très différentes pouvant aussi bien influencer la tonalité de la note que sa durée.

La structure de données que nous avons élaborée se compose de la façon suivante : l'élément qui va contenir toutes les informations est la classe *Score*. Une*Score* contient une liste de *Parts* qui l'on peut qualifier de *Voix* en français. Chaque *Part* contient un nom, un identifiant ainsi qu'une liste de *Measures* (mesure) qui contient elle-même un numéro, une *Signature*, une liste de *IMusicalItem* et un *Metronome* correspondant au tempo.

IMusicalItem est la super classe de bon nombre d'éléments que nous pouvons qualifier de bout de chaîne comme les *Note*, *Rest* (silence) ou encore les *Tie* (liaison). Mais ce n'est pas tout, les groupes de *IMusicalItem* sont également des *IMusicalItem*. Cela nous permet entre autre de pouvoir imbriquer des groupes de *IMusicalItem*. Dans les premières versions de cette partie du code, il n'y avait pas de liens entre les mesures et les groupes. Nous nous sommes rapidement rendu compte que la mesure et le groupe étaient à peu de chose près la même chose. Nous avons donc décidé qu'il serait mieux que le mesure soit aussi un groupe pour limiter la duplication de code. Un groupe peut représenter des croches liées ou encore un *triolet*.

Pour chaque symbole musical, une classe lui est associée. Le symbole peut être unaire ou binaire. Les symboles unaires sont liés à une seule note, tel que les points d'orgues. Les symboles binaires sont liés à 2 notes, comme les liaisons. Pour chaque note, on récupère tous les symboles qui lui sont associés, on créé les objets intermédiaires et on les ajoute à l'objet *Note*.

La création de l'objet *Score* se fait en deux étapes. Tout d'abord, nous récupérons les données brutes dans le *Document* que nous mettons dans l'objet *Score* final. Une fois toutes ces données à disposition, nous pouvons créer les groupes en fonction des symboles que nous rencontrons lors du parcours des mesures. Prenons comme exemple la balise *<beam>*, elle

FIGURE 11 – Structure d'une partition sous forme d'un diagramme de classe UML

peut être présente dans une balise *<note>* et peut contenir les informations nécessaires pour représenter un groupe de croches dans *MusicXML*. Comme le montre le code ci-dessous, la balise possède un attribut *number*. Il nous permet de connaître le numéro du groupe actuel dans le cas où il y a plusieurs groupes de croches dans la même mesure. Et enfin la balise contient un énumération *begin, end, continue* qui nous permet de connaître la position de la note dans le groupe. Lors de notre premier passage nous stockons ces informations dans un objet *Beam*. Lors de notre second passage nous utilisons ces données pour créer nos groupes. Cette

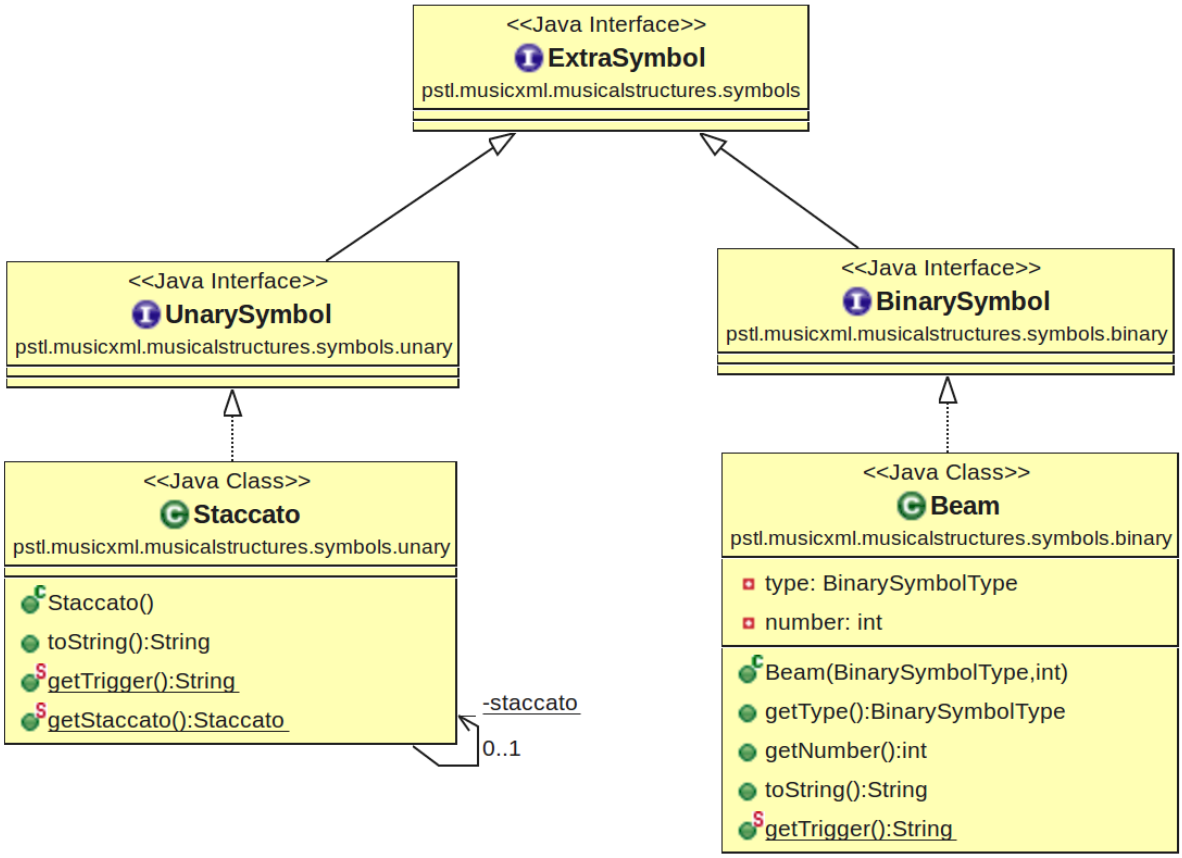


FIGURE 12 – Structure des symboles sous forme d'un diagramme de classe UML

étape est essentielle car elle precalcule les groupes qui seront formés dans l'*arbre rythmique*. En effet, ces groupes influence directement la durée des notes et accords et donc les sous-arbres d'un *arbre rythmique*.

Listing 5 – Exemple d'une balise beam de MusicXML

```
<note>
 [...]
<beam number="1">begin</beam>
</note>
```

On pourrait penser que ce modèle de données est lourd, mais cela est largement compensé entre autres par l'aisance d'utilisation qu'il procure ainsi que la possibilité notamment de déduire les "coordonnées" des éléments qui le composent. En effet rien de plus facile que de dire qu'une note se trouve dans le *Chord* 4 de la *Measure* 2 de la *Part* 2 de telle *Score*. Ainsi nous pouvons par exemple utiliser ces coordonnées pour placer certains symboles à des endroits précis.

5.5 Construction des arbres rythmiques

Notre implémentation de l'*arbre rythmique* est structurellement assez simple. Un *RhythmicThree* contient une *Signature*, une *Fraction*, un *ItemType*, une liste d'*ExtraSymbols* et une liste de *RhythmicTree* correspondant aux fils.

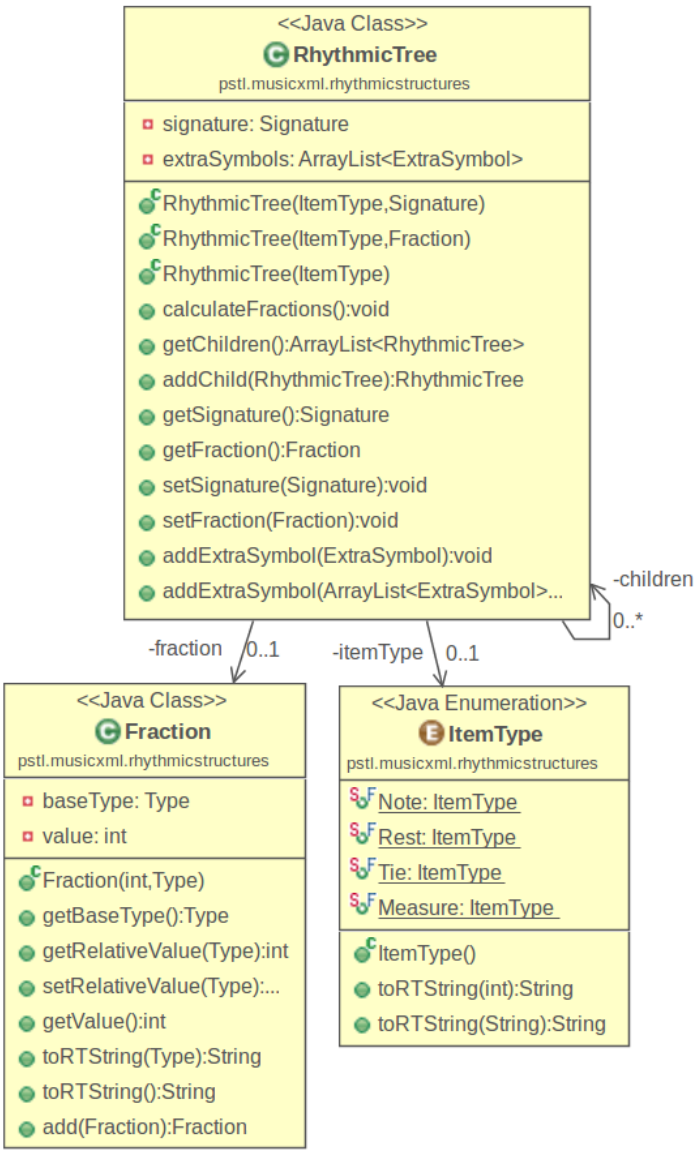


FIGURE 13 – Structure de l'implémentation de l'arbre rythmique

La *Signature* contient simplement deux entiers. La *Fraction* quant à elle est un peu plus complexe. Elle contient un *Type* qui est simplement un énumération des durées possibles d'une

note et qui est utilisé comme le dénominateur de la fraction. Le rôle du numérateur est assuré par un simple entier. Les valeurs de la *Fraction* ne change jamais. Les méthodes de cette classe permettent de calculer la valeur relative à un *Type* de *Fraction*. Plus concrètement, prenons la *Fraction* $a = 1/2$, en utilisant la méthode $a.getRelativeValue(Type.QUARTER)$ soit 4 en dénominateur nous obtiendrons 2 et donc $a = 2/4$. Et enfin l'énumération *ItemType* nous permet de savoir si l'arbre courant est une note, un silence, une liaison ou une mesure.

Une fois la *Score* créé, les arbres rythmiques pour chaque mesure sont construits. Cette étape est réalisée par les classes du package *rhythmicstructures*. La classe factory *RhythmicTreeFactory* parcours tout l'arbre des objets intermédiaires, et pour chaque *Measure* rencontrée, un nouvel objet de la classe *RhythmicTree* est créé. La seule étape que nous pouvons qualifier d'épineuse ici est celle qui consiste à traiter les groupes. Si une mesure contient des groupes imbriqués, il faut créer les arbre rythmiques correspondant et les remplir. L'algorithme est donc récursif, manipulation d'arbres oblige. Ayant suivi au premier semestre le module d'algorithmique avancé qui traitait entre autres de manipulation d'arbres, nous avons pu réaliser cet algorithme sans trop de soucis.

Listing 6 – Code de la méthode buidRTFromMeasure

```
public static RhythmicTree buidRTFromMeasure(Measure measure) {
        RhythmicTree result = new RhythmicTree(ItemType.Measure, measure.getSignature());

        result.setSignature(measure.getSignature());

        measure.getItems().forEach(item -> {
                if (item instanceof Chord) {
                        result.addChild(buildRTFromChord((Chord) item));
                } else if (item instanceof Group) {
                        result.addChild(buildRTFromGroup((Group) item));
                } else if (item instanceof Rest) {
                        Rest r = (Rest) item;
                        if (r.getType() == Type.UNKNOWN) {
                                result.addChild(new RhythmicTree(ItemType.Rest, new
                                    Fraction(1, Type.WHOLE)));
                        } else {
                                result.addChild(new RhythmicTree(ItemType.Rest, new
                                    Fraction(1, item.getType())));
                        }
                } else if (item instanceof Tie) {
                        result.addChild(new RhythmicTree(ItemType.Tie, new Fraction(1,
                            item.getType())));
                }
        });

        result.calculateFractions();

        return result;
}
```

Le code ci-dessus est la dernière partie de la génération de l'*arbre rythmique*. Nous commençons par définir la signature de l'arbre. Son absence ne pose pas de problème au reste du code. Ensuite nous parcourons l'ensemble des éléments qui constituent la mesure. En fonction de leur type nous créons la structure adéquate. On peut remarquer qu'une fois arrivé à cette

étape du code, les groupes ont déjà été formé et que nous n'avons plus qu'à effectuer une simple traduction. La constitution des groupes de la partie précédente est donc fondamentale.

5.6 Test du programme

Afin de vérifier que notre programme fonctionne correctement, une base de test a été créée dans le dossier test-data. Elle est constituée de nombreux fichiers au format MusicXML.

Chaque fichier du dossier simple contient quelques notes avec un symbole musical (signe crescendo, staccato, etc.). Chaque classe associée à un symbole peut ainsi être testée. Par exemple, lorsqu'on exécute le programme sur le fichier «test_chord», constitué d'une noire et d'un accord de trois noires, on obtient l'arbre rythmique suivant :

(2/4 (1 1))

La signature et le nombre d'accords sont corrects. On voit bien que l'arbre rythmique voit la mesure comme une suite d'accords.

Plusieurs œuvres complètes de Bach, nommées BWV, sont présentes dans le dossier «chorales.all.musicxml» pour tester sur un grand nombre de mesure.

Pour calculer le taux de couverture des tests, on peut calculer le nombre de fonctions testées par rapport au nombre de fonctions dans le programme. Le nombre total de fonctions est de 213. En fonction du fichier à tester, le nombre de fonctions appelées ne sera pas le même. Ce dernier va de 136 à 213. Le taux de couverture va donc de 64% à 100% si le fichier contient tous les symboles musicaux.

6 Conclusion

Ce projet nous a permis de découvrir le domaine de la musique qui était obscur pour nous. De plus, nous ne pensions pas que l'informatique pouvait être utile à cet art.

Comme nous l'avons exprimé précédemment, il existe des centaines de symboles musicaux. Tous les traiter dans le temps qui nous était imparti n'était pas raisonnable. Le programme a cependant été pensé pour permettre d'ajouter ces symboles en ajoutant seulement un condition à un *If*. Nous aurions pu automatiser cela en parcourant une liste de symboles compatibles mais cela nous aurait obligé à faire des choix d'implémentation potentiellement regrettables.

D'autre part, l'algorithme permettant de créer les groupes peut sans doute être simplifié. Nous pensons que certains cas traités sont peut être superflus. Mais jusqu'à présent il fonctionne, il n'était pas dans nos priorités. Nous aurions aussi pu lier les groupes et mesures de façon plus profonde mais nous ne pensons pas que cela aurait été d'une grande utilité car cette partie est assez généralisable pour pouvoir être rendu générique.

Références

[1] IRCAM, "Accueil | ircam," 2017. [Online]. Available : https://www.ircam.fr

[2] C. Agon, G. Assayag, and J. Bresson, 2016. [Online]. Available : http://repmus.ircam.fr/openmusic/home

[3] S. Duchenne and A. Gaspard, "Github - screachfr/pstl_musicxml," 2017. [Online]. Available : https://github.com/ScreachFr/pstl_musicxml

[4] Oracle, "The java tutorials," 2015. [Online]. Available : https://docs.oracle.com/javase/tutorial/

[5] W3C, "Extensible markup language (xml)," 2016. [Online]. Available : https://www.w3.org/XML/

[6] L. Roland, "Structurez vos données avec xml," 2017. [Online]. Available : https://openclassrooms.com/courses/structurez-vos-donnees-avec-xml

[7] MakeMusic, "Musicxml for exchanging digital sheet music," 2017. [Online]. Available : https://www.musicxml.com/

[8] C. Loup, "Apprendre le solfège - apprendre la musique," 2017. [Online]. Available : http://www.apprendrelesolfege.com/

[9] J. Clark and M. Makoto, "Relax ng home page," 2014. [Online]. Available : http://relaxng.org/

[10] T. O. Source, "Trang," 2008. [Online]. Available : http://www.thaiopensource.com/relaxng/trang.html

[11] ——, "Jing," 2008. [Online]. Available : http://www.thaiopensource.com/relaxng/jing.html

[12] D. Megginson, "Sax," 2004. [Online]. Available : http://www.saxproject.org/

[13] Oracle, "Lesson : Simple api for xml," 2015. [Online]. Available : https://docs.oracle.com/javase/tutorial/jaxp/sax/index.html

[14] W3C, "Document object model (dom)," 2005. [Online]. Available : https://www.w3.org/DOM/

[15] C. Agon, K. Haddad, and G. Assayag, "Representation and Rendering of Rhythmic Structures," in *WedelMusic 2002*, NA, France, 2002, pp. –, cote interne IRCAM : Agon02b. [Online]. Available : https://hal.archives-ouvertes.fr/hal-01106194

[16] V. Biragnet, "Relaxng-for-music-xml," 2012. [Online]. Available : https://github.com/VincentBiragnet/RelaxNG-for-Music-XML

[17] T. G. Team, "Graphstream - a dynamic graph library," 2015. [Online]. Available : http://graphstream-project.org/