CrestMuseXML Toolkitで始める音楽情報処理入門

北原鉄朗
（JST CrestMuse／関西学院大学）
t.kitahara[at]kwansei.ac.jp

2009年5月21日

CrestMuseXML Toolkitとは

開発の経緯 (1/3)

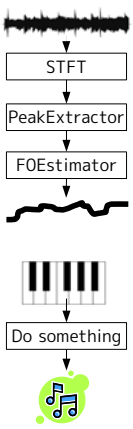
- CrestMuseで名ピアニストの演奏表現のDBを作るプロジェクト（CrestMusePEDB）開始
  - ピアノの演奏表現：各音符の打鍵・離鍵時刻、打鍵（・離鍵）の強さ
    →これらを記述するXMLフォーマットを開発着手
  - 上述のパラメータが基準（楽譜通りの機械的な演奏）からどれだけ離れているか（deviation）を、楽譜データ（MusicXML）から分離して記述する方針採用
    →MusicXML + DeviationInstanceXML = 演奏
  - 複数のXMLフォーマットを使い分ける必要性
    →複数のXMLフォーマットをシームレスに扱えるライブラリの開発開始

開発の経緯 (2/3)

- CrestMuseXML Toolkitの開発開始
  - 複数のXMLフォーマットをほぼ同じAPIで使用可能
    - MusicXML: 楽譜データ
    - DeviationInstanceXML: deviationデータ
    - MusicApexXML: 旋律の階層的なグルーピング構造
    - SCCXML: SMFを簡略化したフォーマット
    - MIDI XML: SMFと等価なXMLフォーマット
    - AmusaXML: 音響特徴時系列など
  - オープンソースソフトウェアとして配布開始
  - CrestMuseXMLは、上述の各種XMLフォーマットの集合であり、単一のXMLフォーマットの名称ではない

開発の経緯 (3/3)

- CrestMuseXML Toolkitの拡張
  - 音響信号処理のためのAPI「Amusa」を設計・実装
    - 「データフロー型」を採用
    - 「モジュール」を組み替えることで自由自在にアルゴリズムを変更可能
    - マルチスレッド化が容易
  - MIDIデータを扱えるようAPIを拡張
    - セッションシステムのようなインタラクティブシステムを容易に開発可能
  - 確率推論のためのAPIも追加

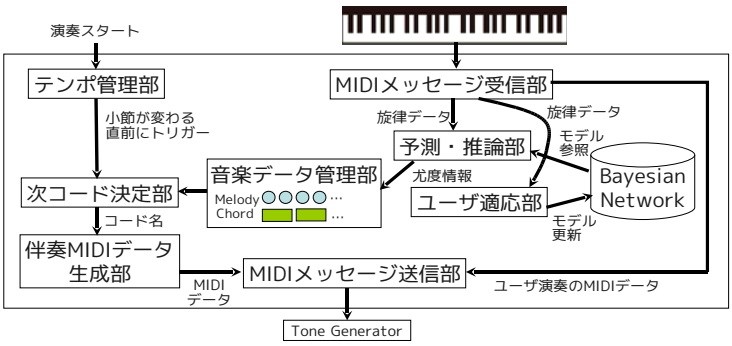


CrestMuseXML Toolkitで何ができるのか

- 各種音楽データの読み書き
  - MusicXML, DeviationInstanceXML, MusicApexXML SCCXML, AmusaXML, SMF etc.（一部未対応）
  - 「入力→処理→出力」の一連の流れの雛形を提供
  - 自作のXMLフォーマットへの対応も可能
- インタラクティブシステムの開発
  - データフロー型API「Amusa」を提供
  - マルチスレッド処理を意識せずに可能
  - 音楽要素を推論するためのデータ構造を提供

CrestMuseXML Toolkitの使用例

- BayesianBand
  - ユーザが弾く主旋律の続きを予測してコードを決定



チュートリアル

MIDIベースの簡単な
インタラクティブシステムを作ってみよう

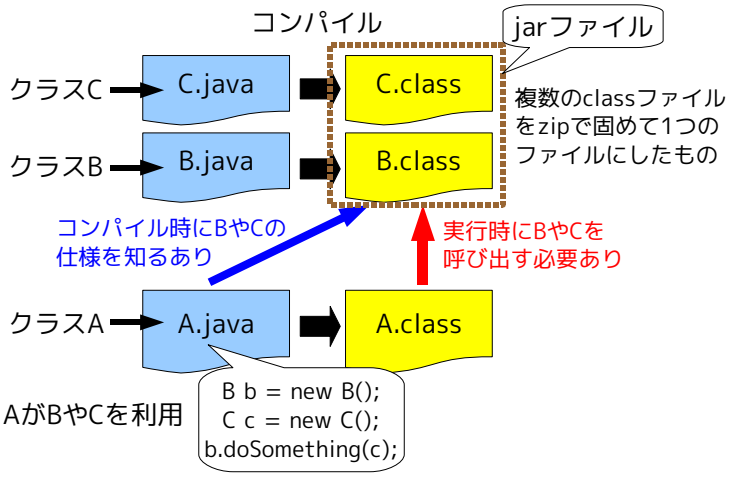
Step 0 インストール

1.http://www.crestmuse.jp/cmx/ から
cmx-0.50.zipダウンロード

2.cmx-0.50.zipを展開

3.cmx-0.50.jarとlibディレクトリにあるjarファイルをJavaが参照できるようにする

そもそもjarファイルとは？



jarファイルを使えるようにする方法

i. jarを任意の場所に置いて、CLASSPATHを通す
%setenv CLASSPATH=\
/home/tetsu/lib/cmx-0.50.jar:/home/tetsu/lib/...
（csh系の場合、bash系、winの場合は異なる）
※CLASSPATHにjarファイルを指定する場合は、置いたディレクトリではなくファイル自体を指定すること

ii.（JREのディレクトリ）/lib/ext にjarを置く
（※実行時のみ有効という話もあり？）

iii.jarを任意の場所に置いて、extdirsオプション
%javac -extdirs /home/tetsu/lib MyClass.java
※コンパイル時のみ有効

jarファイルをコピーしてコンパイルを試そう

1.jarファイルを/lib/extにコピーする
UNIX系環境（Cygwin含む）の場合は、./setup.sh
ただし、同名のjarファイルがある場合は注意

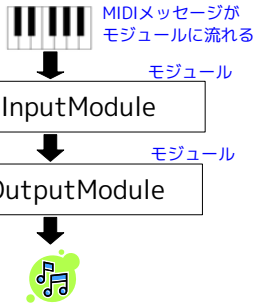
2.下記のプログラムをコンパイル・実行

```
import jp.crestmuse.cmx.commands.*;
public class CMXTest1 extends CMXCommand {
  public static void main(String[] args) {
    try {
      CMXTest1 t1 = new CMXTest1();   t1.start(args);
    } catch (Exception e) {
      e.printStackTrace();    System.exit(1);
    }
  }
}
```

Step 1 MIDI Input/Output (1/2)

Amusa APIを使って、MIDI Inputから来た情報をそのままMIDI Outputに出力するプログラムを作成

次の3つのクラスを使用
- SPExecutor
  （モジュールを登録し実行するクラス）
- MidiInputModule
- MidiOutputModule



Step 1 MIDI Input/Output (2/2)

1. 必要なクラス（SPExecutor、MidiInputModule、MidiOutputModule）のインスタンスを作成する
2. 必要なモジュール（MidiInputModule、MidiOutputModule）をSPExecutorに登録する
3. 登録したモジュールをつなぐ
   （MidiInputModuleの出力をMidiOutputModuleの入力につなぐ）
4. SPExecutorの実行を開始する

1. 必要なクラス（SPExecutor、MidiInputModule、MidiOutputModule）のインスタンスを作成する

インスタンスとは？

コンストラクタを呼び出す

クラス → インスタンス

オブジェクトの設計図　　オブジェクトの実体

クラスベースのオブジェクト指向言語では、インスタンスを作成する処理を経ないとクラスを利用できない

各クラスのコンストラクタの仕様をJavadocで調べてみよう

1. 必要なクラス（SPExecutor、MidiInputModule、MidiOutputModule）のインスタンスを作成する

SPExecutor:

```
SPExecutor exec = new SPExecutor();
```

MidiInputModule: 第1引数にMIDIデバイス指定

MidiOutputModule: 第1引数にレシーバを指定
（レシーバとは、MIDIメッセージを受信可能なデバイスを抽象化したもの）

MIDIデバイスをどうやって取得するか

Java Sound APIを直接たたく必要あり

```
static MidiDevice getDevice() throws Exception {
  MidiDevice.Info[] info=MidiSystem.getMidiDeviceInfo();
  for (int i = 0; i < info.length; i++)
    System.out.println(i+": "+info[i].getDescription());
  System.out.print("どれを使いますか。");
  BufferedReader r = new BufferedReader
    (new InputStreamReader(System.in));
  int n = Integer.parseInt(r.readLine());
  return MidiSystem.getMidiDevice(info[n]);
}
```

今回は詳細は省略します。おまじないだと思ってください。
上の例では、入力デバイスか出力デバイスか区別しません。

1. 必要なクラス（SPExecutor、MidiInputModule、MidiOutputModule）のインスタンスを作成する

SPExecutor:

```
SPExecutor exec = new SPExecutor();
```

MidiInputModule: 第1引数にMIDIデバイス指定

```
MidiInputModule mi
 = new MidiInputModule(getDevice());
```

MidiOutputModule: 第1引数にレシーバを指定

```
MidiDevice dev = getDevice();
dev.open();
Receiver r = dev.getReceiver();
MidiOutputModule mo = new MidiOutputModule(r);
```

もっと簡単にするには

MIDI入力デバイスとしてVirtualKeyboardを使う

```
VirtualKeyboard vkb = new VirtualKeyboard();
MidiInputModule mi = new MidiInputModule(vkb);
```

レシーバとしてデフォルトのものを使う
（たいていはOS付属のソフトシンセ）

```
Receiver r = MidiSystem.getReceiver();
MidiOutputModule mo = new MidiOutputModule(r);
```

2 必要なモジュール（MidiInputModule、MidiOutputModule）をSPExecutorに登録する

```
exec.addSPModule(mi);
exec.addSPModule(mo);
```

3 登録したモジュールをつなぐ
（MidiInputModuleの出力をMidiOutputModuleの入力につなぐ）

```
exec.connect(mi, 0, mo, 0);
```

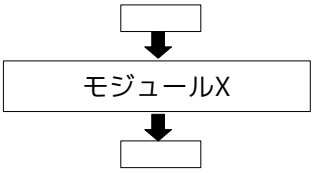
チャンネル番号（後述）

4 SPExecutorの実行を開始する

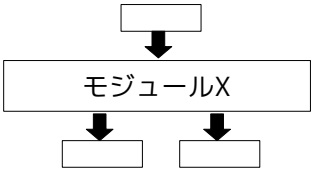
```
exec.start();
```

モジュールのチャンネルとは？

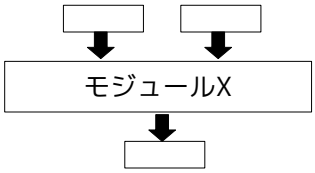
入出力ともに1チャンネル



入力が1チャンネル
出力が2チャンネル



入力が2チャンネル
出力が1チャンネル



チャンネル番号は０からスタート

```
import jp.crestmuse.cmx.amusaj.sp.*;
import jp.crestmuse.cmx.sound.*;
import javax.sound.midi.*;
import java.io.*;
public class CMXTest2 {
  // ここにgetDeviceメソッドをコピー
  public static void main(String[] args) {
    try {
      SPExecutor exec = new SPExecutor();
      MidiInputModule mi = new MidiInputModule(getDevice());
      MidiDevice dev = getDevice();
      dev.open();
      Receiver r = deg.getReceiver();
      MidiOutputModule mo = new MidiOutputModule(r);
      exec.addSPModule(mi);
      exec.addSPModule(mo);
      exec.connect(mi, 0, mo, 0);
      exec.start();
    } catch (Exception e) {
      e.printStackTrace();    System.exit(1);
    }
  }
}
```

```
import jp.crestmuse.cmx.amusaj.sp.*;
import jp.crestmuse.cmx.sound.*;
import javax.sound.midi.*;
public class CMXTest2 {
  public static void main(String[] args) {
    try {
      SPExecutor exec = new SPExecutor();
      VirtualKeyboard vkb = new VirtualKeyboard();
       MidiInputModule mi = new MidiInputModule(vkb);
      Receiver r = MidiSystem.getReceiver();
      MidiOutputModule mo = new MidiOutputModule(r);
      exec.addSPModule(mi);
      exec.addSPModule(mo);
      exec.connect(mi, 0, mo, 0);
      exec.start();
    } catch (Exception e) {
      e.printStackTrace();    System.exit(1);
    }
  }
}
```

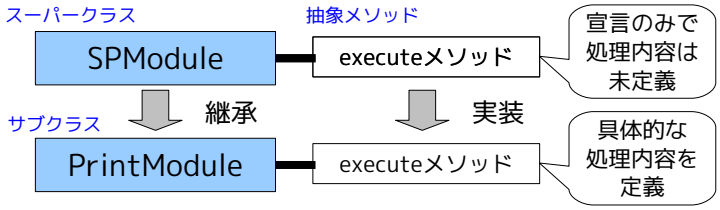
（VirtualKeyboard使用版）

Step 2 独自モジュールの作成 (1/2)

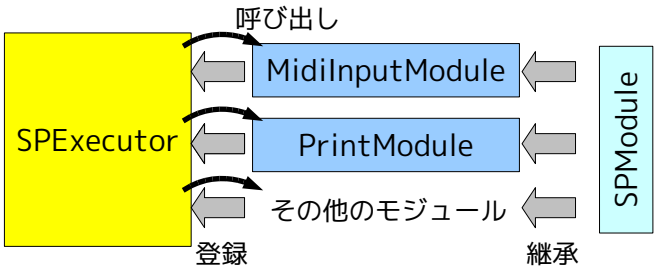
MIDIメッセージを受け取って画面表示する
モジュール「PrintModule」を作成してみよう

基本的な考え方

SPModuleクラスを継承し、必要なメソッドを実装



Step 2 独自モジュールの作成 (2/2)



・すべてのモジュールは、SPModuleオブジェクトである。
・SPExecutorは、モジュールがexecuteメソッドを持っていることは知っている。処理内容は知らない。
・SPExecutorは、データが到着し次第、各モジュールのexecuteメソッドを呼び出す。

ちなみに・・・

普通のクラス	抽象クラス	インターフェイス
抽象メソッドを持たない	抽象メソッドを持てる	抽象メソッドしか持てない
⇩	⇩	⇩
インスタンスを作れる	サブクラスを作らない限りインスタンスを作れない	実装しない限りインスタンスを作れない

なぜこんなものがあるのか

クラスを呼び出す側を共通化するため

⇩

ポリモーフィズム

ポリモーフィズム? (1/2)

MidiInputModule — executeメソッド
MIDIデバイスから得たMIDIデータを出力

PrintModule — executeメソッド
MIDIデータを受け付けて画面表示してそのまま出力

MidiOutputModule — executeメソッド
MIDIデータを受け付けてMIDIデバイスに送信

executeメソッドの処理内容は異なるが、executeメソッドを持ち、0個以上のデータを受け付けて0個以上のデータを出力することは共通

⇩

executeメソッドを持ち上記の仕様を満たすことさえ保証されれば、具体的なクラスが何かは、呼び出す側は気にする必要がない

ポリモーフィズム? (2/2)

executeメソッドの処理内容は異なるが、executeメソッドを持ち、0個以上のデータを受け付けて0個以上のデータを出力することは共通

⇩

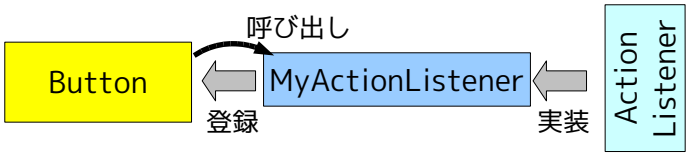
executeメソッドを持ち上記の仕様を満たすことさえ保証されれば、具体的なクラスが何かは、呼び出す側は気にする必要がない

⇩

SPModule

executeメソッドが抽象メソッドとして宣言されているので、このサブクラスのインスタンスはexecuteメソッドを持つことが保証されている

考え方はAWT/Swingの○○Listenerと一緒




```
JButton b = new JButton("XYZ");
b.addActionListener(new MyActionListener());
```

```
class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // ボタンを押されたときの処理
  }
}
```

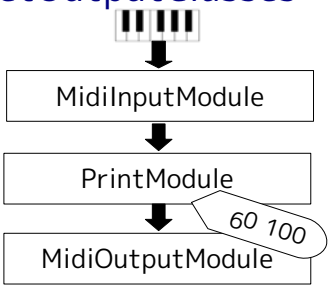
PrintModuleで実装すべきメソッド

- void execute(SPElement[] src,
  TimeSeriesCompatible<SPElement>[] dest)
  モジュールがすべき処理内容をここに記述
  各チャンネルの入力データがsrcに格納
  処理結果はdestに書き込む
- Class<SPElement>[] getInputClasses()
  各入力チャンネルが受け付けるクラスの配列
- Class<SPElement>[] getOutputClasses()
  各出力チャンネルに出力するクラスの配列

```
import jp.crestmuse.cmx.amusaj.sp.*;
import jp.crestmuse.cmx.amusaj.filewrappers.*;

class PrintModule extends SPModule {
  public void execute(SPElement[] src,
          TimeSeriesCompatible<SPElement>[] dest)
          throws InterruptedException {
    // ここに処理内容を記述
  }

  public Class<SPElement>[] getInputClasses() {
    // 各入力チャンネルが受け付けるクラスの配列をreturn
  }

  public Class<SPElement>[] getOutputClasses() {
    // 各出力チャンネルに出力するクラスの配列をreturn
  }
}
```

getInputClasses / getOutputClasses

- 入出力ともに、チャンネル数＝1 MidiEventWithTicktime オブジェクト



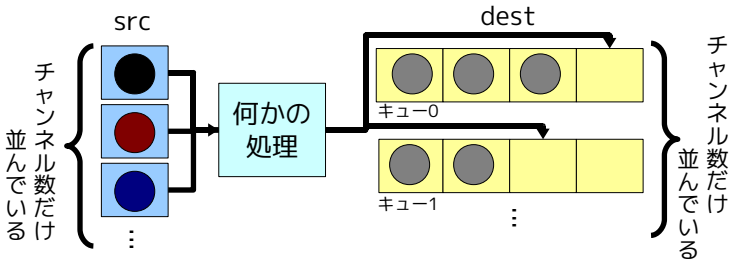

```
public Class<SPElement>[] getInputClasses() {
  return new Class[]{MidiEventWithTicktime.class};
}
public Class<SPElement>[] getOutputClasses() {
  return new Class[]{MidiEventWithTicktime.class};
}
```

補足説明

- Classクラス
  A.classとすると、Aクラスを表すClassオブジェクトが得られる。
- Class<SPElement>
  <...>内のクラスを型パラメータという。List<String>などの形でよく使われる。ここでは気にしなくてよい。
- new Class[] {...}
  配列の書き方の1つ。s=new String[]{“a”, “b”}は、s=new String[2]; s[0]=”a”; s[1]=”b”; と等価

execute (1/2)

```
void execute(SPElement[] src,
    TimeSeriesCompatible<SPElement>[] dest)
```



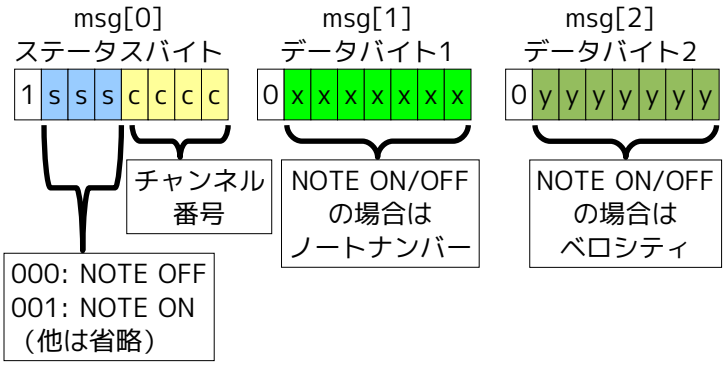
・入力チャンネル0に到着したデータは src[0] でアクセスできる
・出力チャンネル0にデータを出力するには dest[0].add(xxx)

execute (2/2)

```
public void execute(SPElement[] src,
    TimeSeriesCompatible<SPElement>[] dest) throws ... {
  MidiEventWithTicktime evt
      = (MidiEventWithTicktime)src[0]; ①
  byte[] msg = evt.getMessage().getMessage(); ②
  if (msg.length == 3)
    System.out.println(msg[0] +“ ”+ msg[1] +“ ”+ msg[2]); ③
  dest[0].add(evt); ④
}
```

① このモジュールはチャンネル0にMidiEventWithTicktimeが来る前提なので、ダウンキャストする
② このようにするとMIDIメッセージをbyte[]で受け取れる
③ MIDIメッセージの仕様は次スライド
④ 入力されたMIDIイベントをそのまま出力

MIDIメッセージ



※NOTE OFFは、ベロシティ＝0のNOTE ONで代用されることがある。
※時間に関する情報は一切含まないことに注意。

キャスト（型変換）とは

1. 基本型同士の型変換
   i) 整数から実数へ
   `int n = 10;  double x = n;` ⇨ OK
   ii) 実数から整数へ
   `double x = 1.1;  int n = (int)x;` ⇨ 情報が失われる恐れあり
2. オブジェクト同士の型変換
   i) サブクラスBからスーパークラスAへ（アップキャスト）
   `B b = ...;  A a = b;` ⇨ OK
   ii) スーパークラスAからサブクラスBへ（ダウンキャスト）
   `A a = ...;  B b = (B)a;` ⇨ aが元々Bのインスタンスでなければ失敗する

PrintModuleを使ってみる

Step 1のプログラムに以下の部分を追加しよう

PrintModuleインスタンスの生成

```
PrintModule pm = new PrintModule();
```

PrindModuleをSPExecutorに登録

```
exec.addSPModule(pm);
```

モジュールのつなぎ方を以下のように変更

```
exec.connect(mi, 0, pm, 0);
exec.connect(pm, 0, mo, 0);
```

```
import jp.crestmuse.cmx.amusaj.sp.*;
import jp.crestmuse.cmx.sound.*;
import javax.sound.midi.*;

public class CMXTest3 {
  public static void main(String[] args) {
    (中略)
    PrintModule pm = new PrintModule();
    exec.addSPModule(mi);
    exec.addSPModule(pm);
    exec.addSPModule(mo);
    exec.connect(mi, 0, pm, 0);
    exec.connect(pm, 0, mo, 0);
    exec.start();
    (中略)
}
```

Step 3 「ハモリ」モジュールの作成

PrintModuleを改造して、3度下でハモる
モジュール「HamoriModule」を作ってみよう

PrintModuleからの変更点

入力されたMIDIメッセージを出力するだけでなく、
3度下のメッセージを生成して出力する

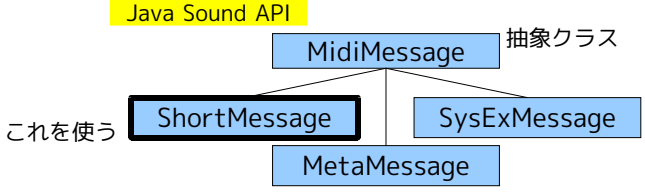
変更すべきはexecuteメソッド内のみ

MIDIメッセージの作り方

生成すべきは、MidiEventWithTicktimeオブジェクト
→コンストラクタの仕様をJavadocでチェック

MidiEventWithTicktime(MidiMessage message,
long tick, long position)

MidiMessageオブジェクトを先に作る必要あり
→MidiMessageクラスの仕様をJavadocでチェック



HamoriModuleのexecuteメソッド

```
public void execute(SPElement[] src,
    TimeSeriesCompatible<SPElement>[] dest) {
  MidiEventWithTicktime evt
      = (MidiEventWithTicktime)src[0];
  byte[] msg = evt.getMessage().getMessage();
  if (msg.length == 3) {
    byte[] msg2 = new byte[] {msg[0], msg[1]-4, msg[2]};
    ShortMessage sm = new ShortMessage(msg2);
    MidiEventWithTicktime evt2
        = new MidiEventWithTicktime(sm, 0, 0);
    dest[0].add(evt2);
  }
  dest[0].add(evt);
}
```

Step 4 伴奏(SMF)を再生しよう

SMFPlayerクラスを使えばよい

```
(import省略)

public class CMXTest4 {
  public static void main(String[] args) {
     (一部省略)
    exec.connect(mi, 0, pm, 0);
    exec.connect(pm, 0, mo, 0);
    SMFPlayer player = new SMFPlayer(r);
    player.readSMF("banso.mid");
    exec.start();
    player.play();
  }
}
```

MIDIキーボードの各打鍵時刻の伴奏SMFにおける位置を知りたい

基本的な考え方

MidiInputModuleのsetTickTimerメソッドを使う

TickTimer: 現在時刻をtick単位で返すgetTickPosition()メソッドを宣言したインターフェイス

SMF: TickTimerインターフェイスを実装
getTickPosition()を呼び出すと、現在再生中のSMFにおける現在位置をtick単位で返す

```
mi.setTickTimer(player);
```

PrintModuleのexecuteを改造する

```
public void execute(SPElement[] src,
    TimeSeriesCompatible<SPElement>[] dest) {
  MidiEventWithTicktime evt
      = (MidiEventWithTicktime)src[0];
  byte[] msg = evt.getMessage().getMessage();
  System.out.println(evt.music_position + ": "
                    + msg[0] +" "+ msg[1] +" "+ msg[2]);
  dest[0].add(evt);
}
```

MidiInputModuleにsetTickTimerされていれば、
そこから得られるMidiEventWithTicktimeオブジェクトは、
music_positionという変数から時刻を知ることができる。

Step 5 演奏を可視化する

MIDIキーボードから入力されたMIDIメッセージをピアノロールもどきで可視化しよう

基本的な考え方

これまでのテクニックを総動員して、
MIDIメッセージ受信の度に時刻とノートナンバーを取得し、それぞれをx座標、y座標として表示する

MIDIメッセージ受信の度に時刻とノートナンバーを取得する部分（PrintModule）は作成済みなので、後はGUI部を追加すればよい。

GUIの部分を作ってみよう (1/2)

```
import java.awt.*;
import java.util.*;
class PianorolI extends Frame {
  ArrayList<TickAndNoteNum> l = new ArrayList<TickAndNoteNum>();
  PianorolI() {
    setSize(600, 300);
    setVisible(true);
  }
  void addNoteOn(long tick, int notenum) {
    l.add(new TickAndNoteNum(tick, notenum));
  }
  public void paint(final Graphics g) {
    super.paint(g);
    for (TickAndNoteNum x : l)
      g.fillRect((int)(x.tick / 25), 900 - 10*x.notenum, 10, 10);
  }
}
```

GUIの部分を作ってみよう (2/2)

```
class TickAndNoteNum {
  long tick;
  int notenum;
  TickAndNoteNum(long tick, int notenum) {
    this.tick = tick;
    this.notenum = notenum;
    }
}
```

PrintModuleを改造しよう (1/2)

```
（import省略）
class PrintModule extends SPModule {
  PianoroII pianoroII;
  class PrintModule(PianoroII pr) {
    pianoroII = pr;
  }
  （中略）
  public void execute(SPElement[] src,
      TimeSeriesCompatible<SPElement>[] dest) throws ... {
      （ここの変更点は次スライドで）
  }
}
```

PrintModuleのexecuteメソッドからピアノロールに描画命令を下すので、PianoroIIオブジェクトへの参照を取得しておく

PrintModuleを改造しよう (2/2)

```
public void execute(SPElement[] src,
    TimeSeriesCompatible<SPElement>[] dest) {
  MidiEventWithTicktime evt
      = (MidiEventWithTicktime)src[0];
  byte[] msg = evt.getMessage().getMessage();
  pianoroII.addNoteOn(evt.music_position, msg[1]);
  dest[0].add(evt);
  pianoroII.repaint();
}
```

executeメソッドの中で、printlnの代わりにピアノロールに描画命令を下している。

※上の例ではNOTE OFFでも同様に描画している。
これを回避する方法は各自考えてみよう。

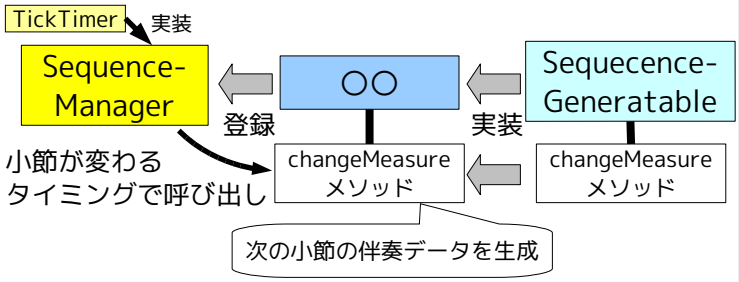
もっと高度なことをするには

今回のチュートリアルで触れなかったより高度な機能について、さわりだけ触れておきます。

1. ユーザの入力に合わせて伴奏を変えたい

基本的な考え方

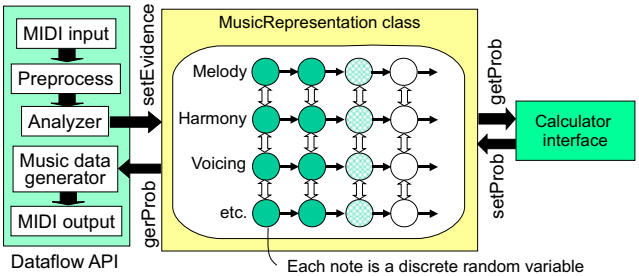
SequenceManagerを使います。



2. 伴奏生成時にユーザの演奏履歴をどう管理するか

基本的な考え方

MusicRepresentationを使います。

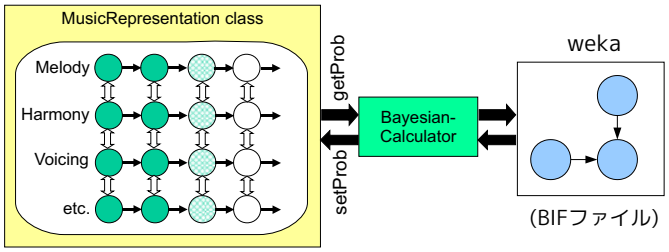


Once an evidence is set to any node, *calculators* are automatically called to update the probabilities of other nodes

3. コーパスから学習したモデルを使いたい

基本的な考え方

BayesianCalculatorを使います。



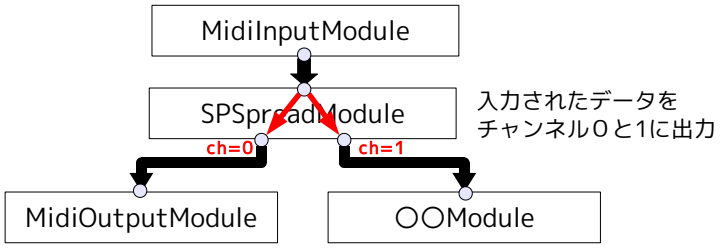
MusicRepresentationのノードの値をwekaにコピー
→ wekaで推論 → 結果をMusicRepresentationにコピー

両者のノードの対応関係はBayesianMappingで記述

4. 同じモジュールの出力を複数のモジュールに入力させたい

基本的な考え方

SPSpreadModuleを使います。



5. 信号処理にも使いたい

基本的な考え方

今回解説したAPI（SPExecutorなど）をそのまま使えます。

WindowSlider　WAVファイルを読み込んで短区間ごとに切って出力

STFT　短時間フーリエ変換

PeakExtractor　パワースペクトルピークを抽出

その他のモジュール

ChromaExtractor, FOPDFCalculatorModule,
FeatureExtractor, HarmonicsExtractor（未テスト?）

6. Javaはタイプ量が多いからイヤ。もっと気軽に使いたい。

基本的な考え方

RubyならJRuby、PythonならJythonを使います。

- Javaで実装したスクリプト言語の実行環境ならJavaのライブラリが使えます。
  - Ruby → JRuby
  - Python → Jython
  - Groovy（Java完全互換のスクリプト言語）
- Max/MSP使いならMax/MSPから使うのもあり

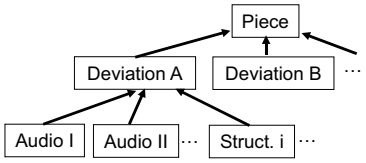
JRubyのみ簡単なテスト済み

CrestMuseXMLについて

CrestMuseXMLとは (1/3)

Extensible framework for describing various types of musical content and knowledge including:

- *Score*: what piece is performed
- *Deviation*: how the piece is performed
- *Audio features*: how the performance sounds
- *Music structure*: what phrasing is intended by the performer
- Others



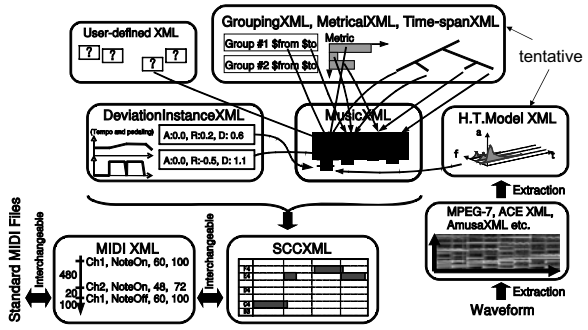
Issues

- Extensibility
- Many-to-one linking
- Compatibility to existing formats

CrestMuseXMLとは (2/3)

Solution

- Data at different layers are described in different formats and linked to each other



CrestMuseXMLとは (3/3)

Main merits of this approach

- Anyone can extend the descriptors by adding new original formats
- Different data (e.g. deviation) can be linked to the same content (e.g. score)
- An existing de facto standard format, if any, can be adopted for each layer

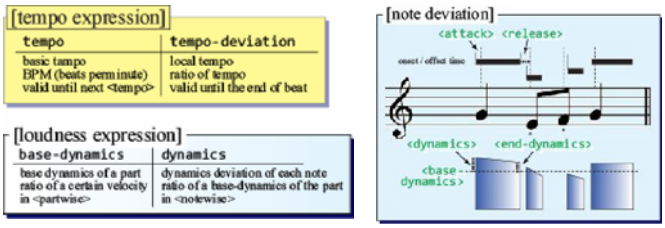
各音楽要素に対する記述フォーマット (1/2)

Description of melody

- *MusicXML* is adopted

Description of deviation

- We designed a new format *DeviationInstanceXML*
  - Basic concept: to separate macro- and micro- deviations



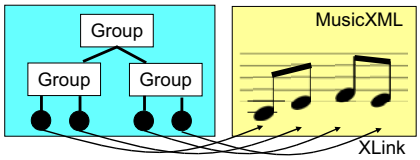
各音楽要素に対する記述フォーマット (2/2)

Description of audio features

- *AmusaXML* (original) and *ACE XML* (by Fujinaga Lab.)

Description of music structure

- *MusicApexXML* (original)
  - *Tree structure* of phrasing in which the leaves link to *note* elements in a MusicXML document



まとめ

CrestMuseXML Toolkitとは（おさらい）

- 音楽情報処理用のオープンソースライブラリ
  - 各種音楽データの読み書き
  - リアルタイム用APIを提供
  - 音楽要素を推論するためのデータ構造を提供
- Javaベース、cross-platform
- 信号ベース、MIDIベースどちらにも有用
- XMLを扱わないアプリにも有用
- CrestMuseXMLは、単一のXMLフォーマットを指し示す言葉ではない

開発体制について

- 現在は、北原＋アルバイト（戸谷、徳網）が実装
- SourceForge.jp上のsubversionで管理
- 常に開発協力者募集中
  - コーディング、テスト、バグ報告、ドキュメンテーション、ドキュメント英訳など

さらなる情報

- http://www.crestmuse.jp/cmx/（ちょっと古い）
- 音情研ペーパー（2007年8月、2008年5月）（さらに古い）
- 来年2月（関西学院大）に実習付チュートリアルをやるかも（詳細は一切未定）
- メーリングリスト cmx-dev
- 気軽に北原までメールをください