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

CrestMuse

2010年2月15日

本日のメニュー

- 10:00～10:45　イントロダクションとセットアップ
- 11:00～11:45　(1) MusicXMLの入力と処理

～昼休み～

- 13:00～13:45　(2) MIDIデータのリアルタイム処理
- 14:00～14:45　(3) オーディオデータの信号処理
- 14:45～15:00　質問

イントロダクション：
CrestMuseXML Toolkitとは

CrestMuseXML Toolkit (CMX)とは

- 音楽情報処理研究に便利に使えるライブラリ
- MusicXMLなどのデータ入出力、MIDIや音響信号のリアルタイム処理、確率推論などをサポート
- GUI上で便利に使えるアプリケーションではない（あくまでプログラミングのためのライブラリ）
- Javaで開発
- オープンソース

開発の経緯 (1/3)

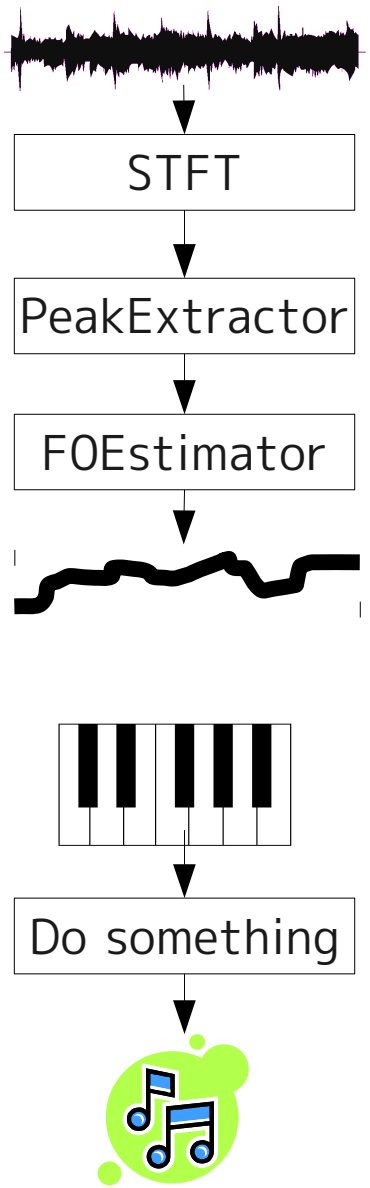
- CrestMuseで名ピアニストの演奏表現DBを作るプロジェクト開始
  - 各音符の打鍵・離鍵時刻、打鍵(・離鍵)の強さを記述
    →これらを記述するXMLフォーマットを開発着手
  - 楽譜データとそこからのずれ(deviation)を分離して記述する方針採用
    →MusicXML + DeviationInstanceXML = 演奏
  - 複数のXMLフォーマットを使い分ける必要性
    →複数のXMLフォーマットをシームレスに扱えるライブラリの開発開始

開発の経緯 (2/3)

- CrestMuseXML Toolkitの開発開始
  - 複数のXMLフォーマットを同じ枠組みで使用可能
    - 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
    - リアルタイムに処理可能
    - 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

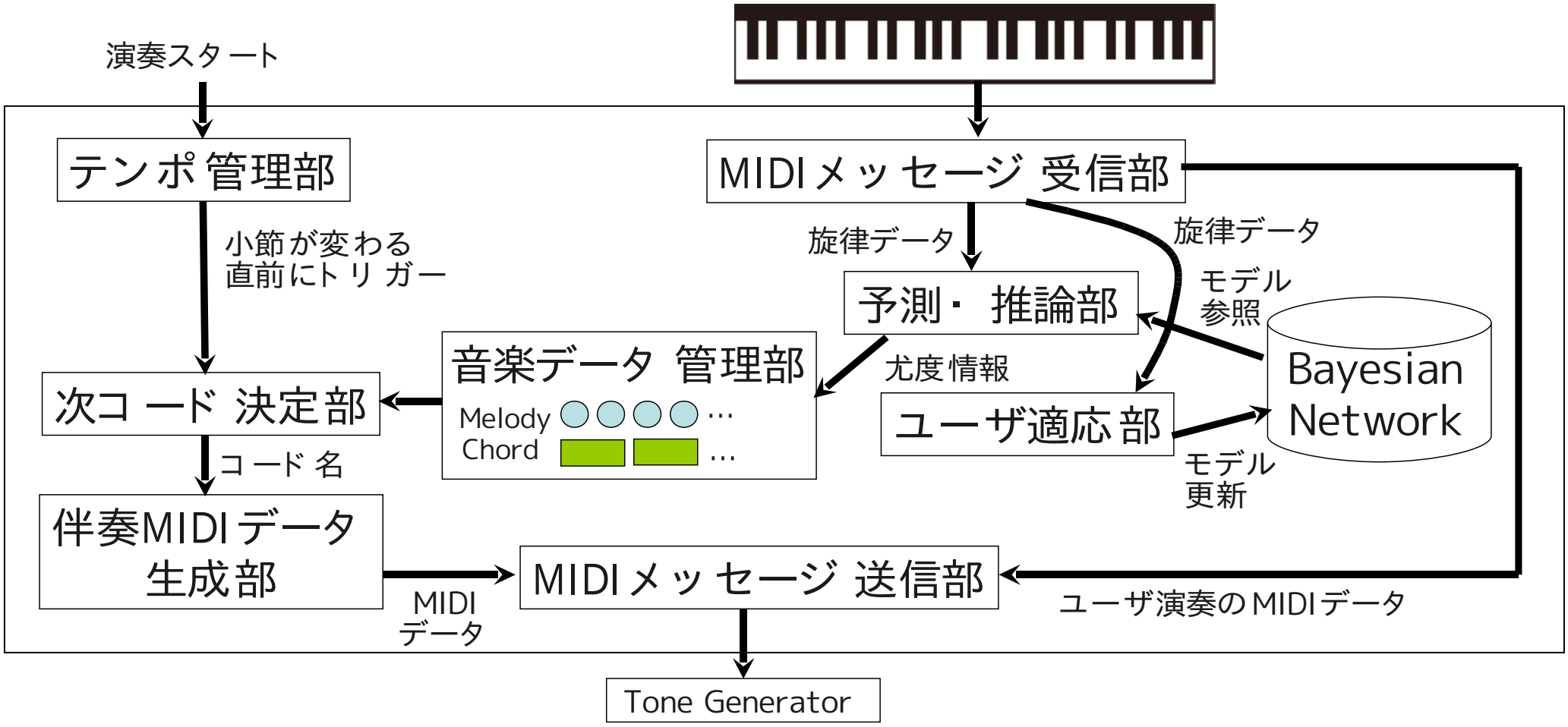


CrestMuseXML Toolkitで何ができるのか

- 各種音楽データの読み書き
  - MusicXML, DeviationInstanceXML, MusicApexXML
    SCCXML, AmusaXML, SMF etc.（一部未対応）
  - 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
  - データフロー型プログラミング
    （モジュールの作成とその結合によるプログラミング）
  - 音楽要素を推論するためのデータ構造を提供
- 音楽音響信号処理
  - 上述のデータフロー型と同じ枠組みで実現

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



イントロダクション：
今回のチュートリアルの方針

今回のチュートリアルの方針

- 実際にプログラムを打ち込んで実行しながらCrestMuseXML Toolkitの理解を深めていただきます。
- プログラミング言語には、Javaの代わりにスクリプト言語「Groovy」（後述）を用います。
- 小難しい話は最小限に抑えたいと思いますが、オブジェクト指向に関する専門用語が出てきます。適宜質問の時間を設けますので、わからなかったら遠慮なく聞いてください。
- それ以外のことも、質問は気軽にどうぞ。

Groovyとは

- Javaをベースにしたスクリプト言語
- Javaでは必須の次の5つをしなくてもいい
  - コンパイルしなくてもいい
  - 型宣言をしなくてもいい
  - 例外処理をしなくてもいい
  - キャストしなくてもいい
  - すべてのメソッドや変数をクラスの中に収めなくてもいい
- Java用のライブラリはすべて使える

つまり…

Javaの機能をフルに使える超お手軽言語

GroovyとCMX

- CMXの最新バージョン(ver.0.52)では、
  Groovyから使うと便利な機能を追加
  - 演算子のオーバーロードによるMATLABライクな行列計算
  - クロージャによる簡潔なループ
- CMXをGroovyに同封したパッケージを用意し、
  インストール作業なしにCMXをGroovyから利用可能

…ということで、

本チュートリアルでは、Groovyを用いて
お手軽にCMXプログラミングを体験

セットアップ

手順1 Javaの実行環境をインストール

JREでググる

↓

「無料Javaのダウンロード」をクリック

↓

指示に従ってインストール

手順2「CrestMuseXML Toolkit 0.52 bundled with Groovy 1.7.1」をダウンロード

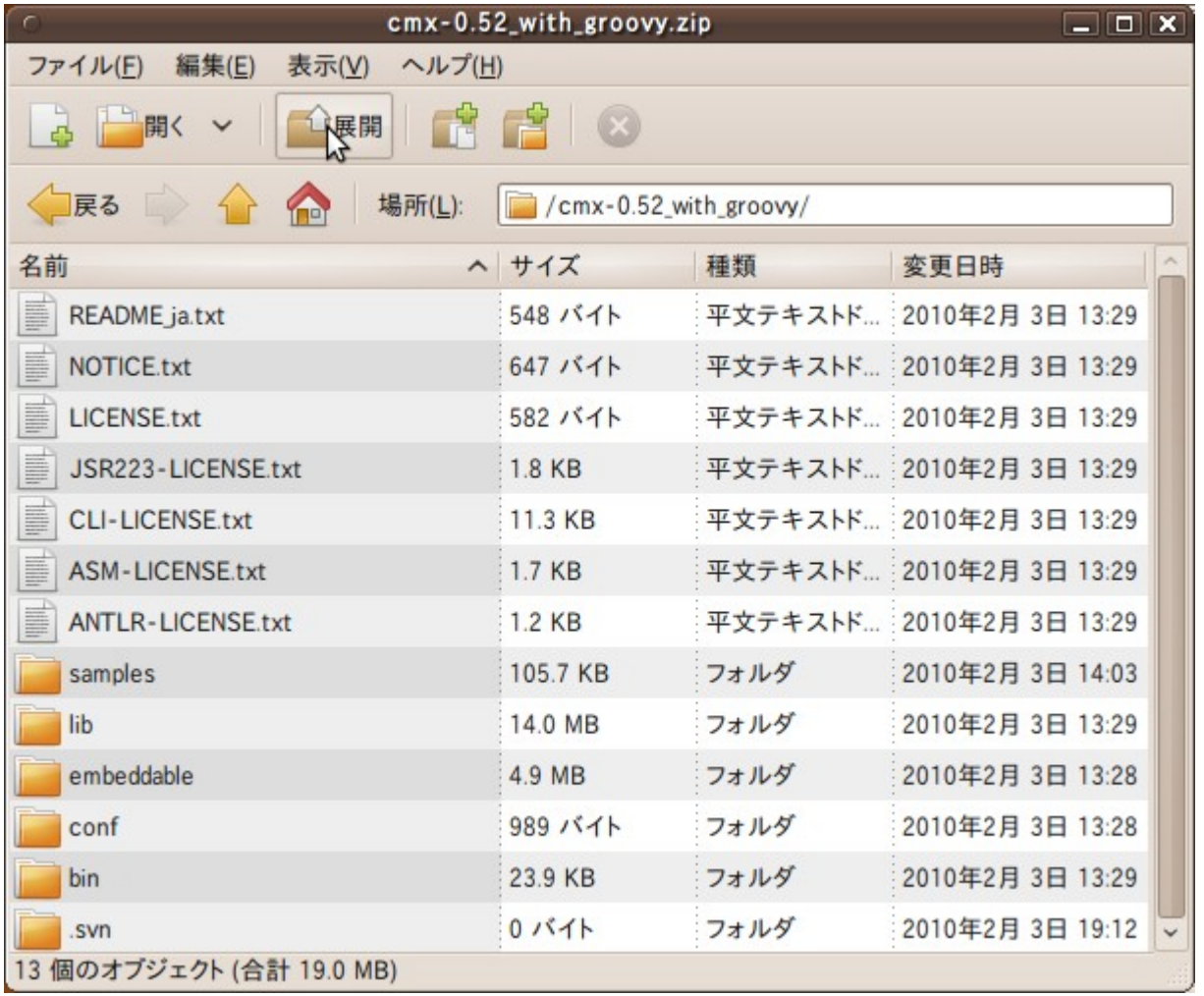
1) http://sourceforge.jp/projects/cmx/ を開く

2) スクロールしたら現れる「ダウンロード」をクリック

3) cmx-0.52_with_groovy.zip をダウンロード

手順3 ダウンロードしたzipファイルを解凍

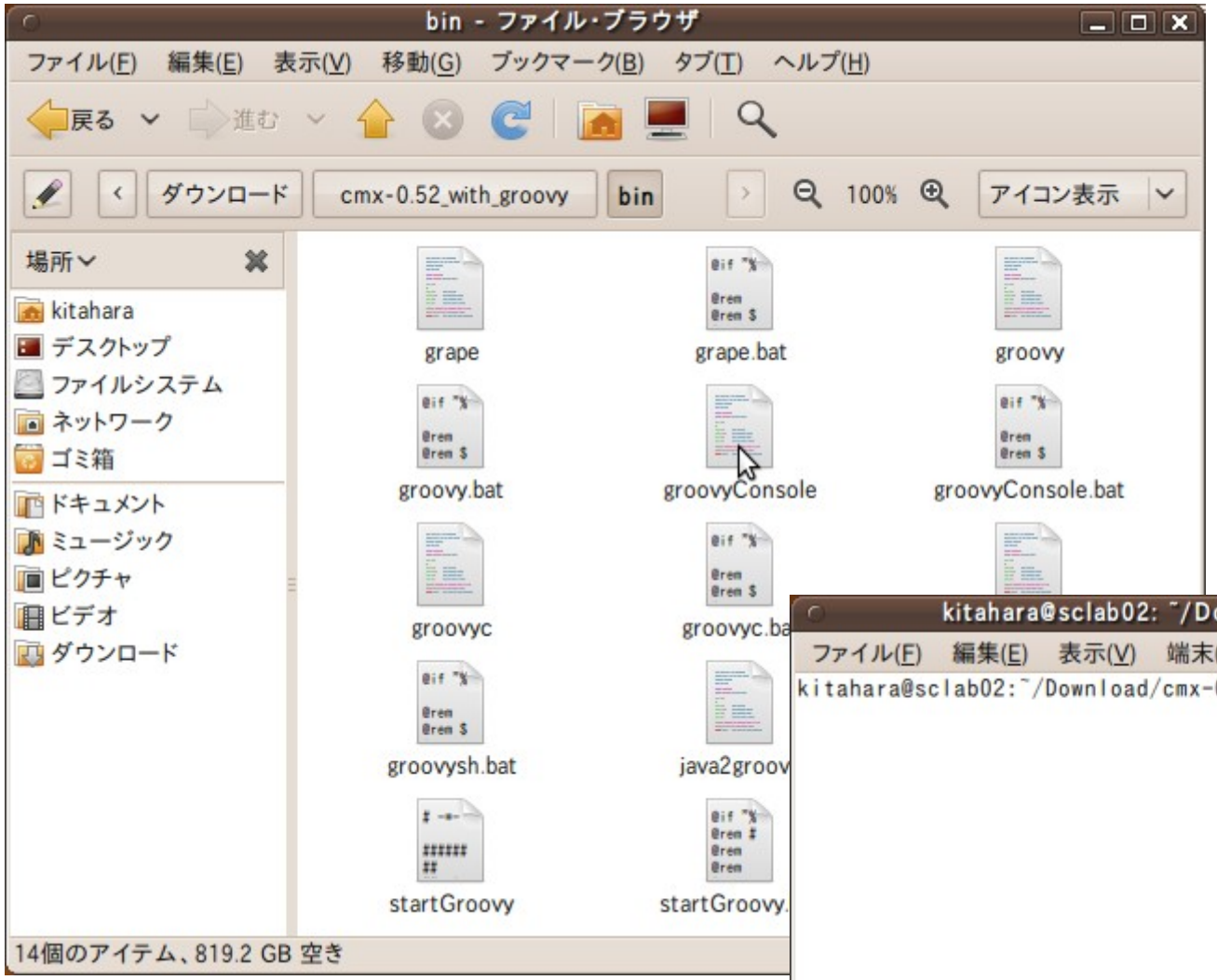
ダウンロードしたcmx-0.52_with_groovy.zipを解凍



実行してみよう

生成されたディレクトリの中のbinの中の次のファイルを実行

- groovyConsole.bat（Windowsの場合）
- groovyConsole（その他の場合）

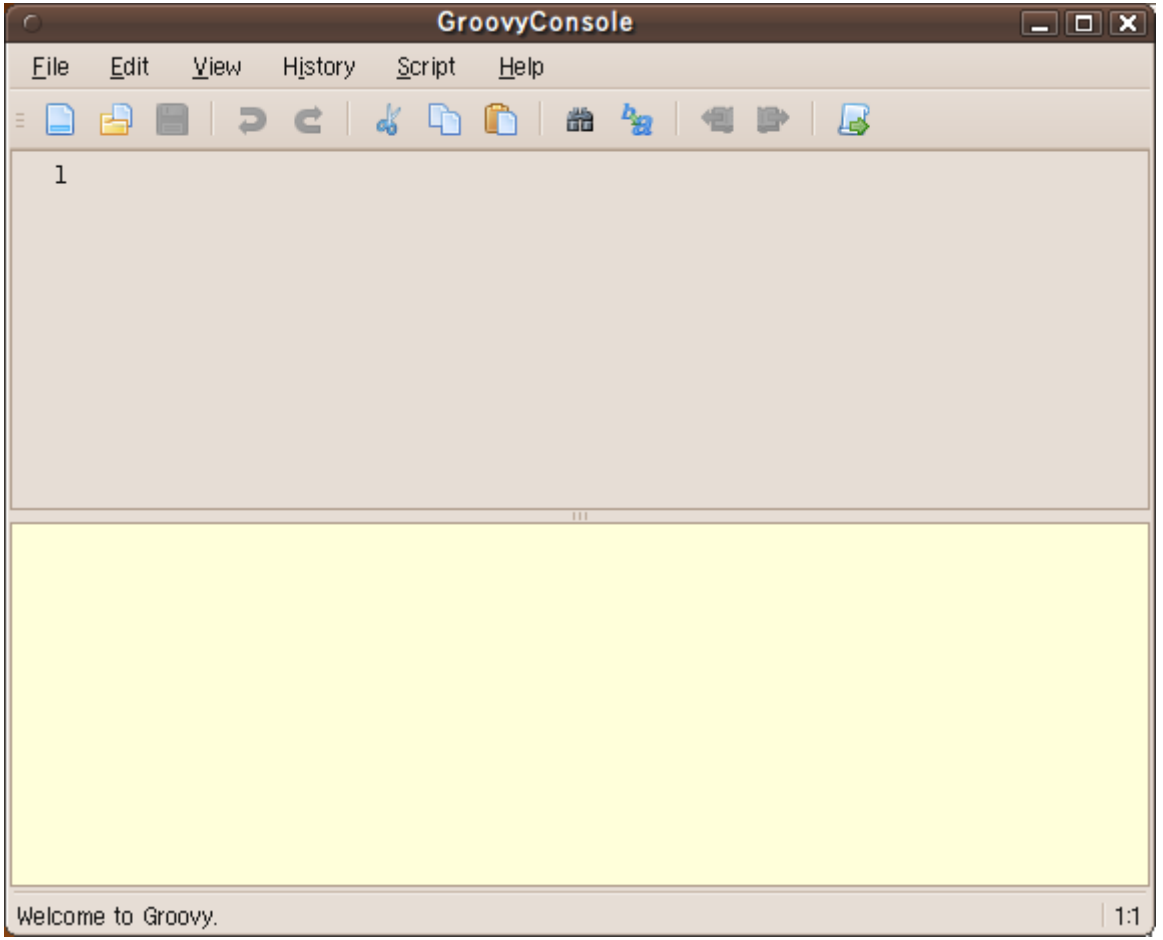


←アイコンをダブルクリック

↓コンソールから

実行できましたか?

こんな画面が出てきたら成功です。



※黒いウィンドウが一瞬出てきて終わってしまう場合は、
Javaの実行環境が正常にインストールできていない可能性大

実習1
MusicXMLの入力と処理

【本実習の目標】
簡単なMusicXML（楽譜情報を表す）を読み込み、付与されている演奏記号に従って演奏を生成するプログラムを作成する。

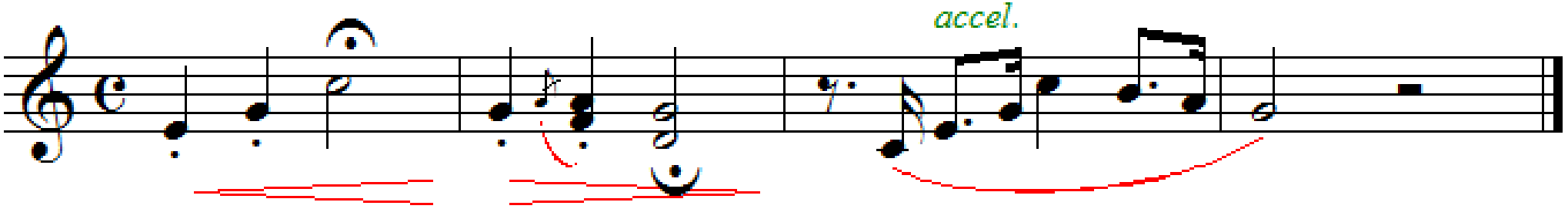
MusicXMLとは

楽譜を記述するためのXMLフォーマット

なんでMIDIファイルじゃダメなの?

- MIDIファイルは、楽譜ではなく「演奏」を記録するフォーマット
  - 小節線や休符という概念がない。
  - スタッカートやフェルマータなどを記述できない。

MusicXMLの例

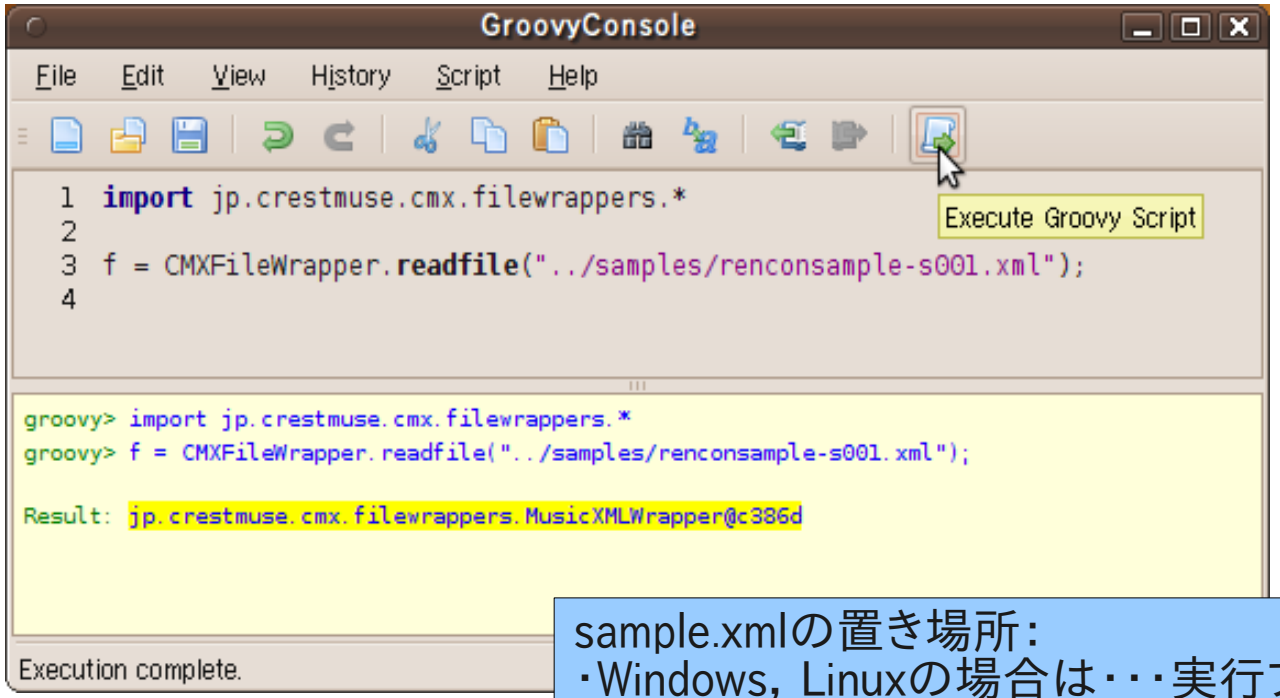



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE score-partwise PUBLIC
"-//Recordare//DTD MusicXML 1.0 Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
    （中略）
  <part-list>
    <score-part id="P1">
      （中略）
    </score-part>
  </part-list>
  <!--=============================-->
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>8</divisions>
          （中略）
      </attributes>
      <direction placement="below">
        <direction-type>
          <wedge default-y="-72" spread="0"
                  type="crescendo"/>
        </direction-type>
        <offset>3</offset>
      </direction>
      <note>
        <pitch>
          <step>E</step>
          <octave>4</octave>
        </pitch>
        <duration>8</duration>
        <voice>1</voice>
        <type>quarter</type>
        <stem>up</stem>
        <notations>
          <articulations>
            <staccato placement="below"/>
          </articulations>
        </notations>
      </note>
      <note>
        <pitch>
          <step>G</step>
          <octave>4</octave>
        </pitch>
        <duration>8</duration>
        <voice>1</voice>
        <type>quarter</type>
        <stem>up</stem>
        <notations>
          <articulations>
            <staccato placement="below"/>
          </articulations>
        </notations>
      </note>
      <note>
        <pitch>
          <step>C</step>
          <octave>5</octave>
        </pitch>
        <duration>16</duration>
        <voice>1</voice>
        <type>half</type>
        <stem>down</stem>
        <notations>
          <fermata type="upright"/>
        </notations>
      </note>
      <direction>
        <direction-type>
          <wedge default-y="-71“
            spread="12" type="stop"/>
        </direction-type>
        <offset>-3</offset>
      </direction>
    </measure>
    <!--=========================-->
    <measure number="2">
      <direction placement="below">
        <direction-type>
以下省略
```

Step 1 MusicXMLファイルを読み込む

sample.xmlをダウンロードして，
下記を入力して実行してみよう

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
```



sample.xmlの置き場所：
・Windows，Linuxの場合は・・・実行ファイルと同じところ
・Macの場合は・・・ホーム（例：/Users/kitahara）
にしてください．

Step 2 読み込んだMusicXMLファイルを画面に表示してみる

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
f.write(System.out)
```

次のステップに行く前に作戦会議

例題の確認

- MusicXMLファイルが入力され、それに対する演奏を生成
- 各音符にスタッカートが付与されていれば、音長を半分に、フェルマータが付与されていれば、テンポを半分にする。

そもそもMusicXMLはどういう構造になっているのか

- → 次のスライド

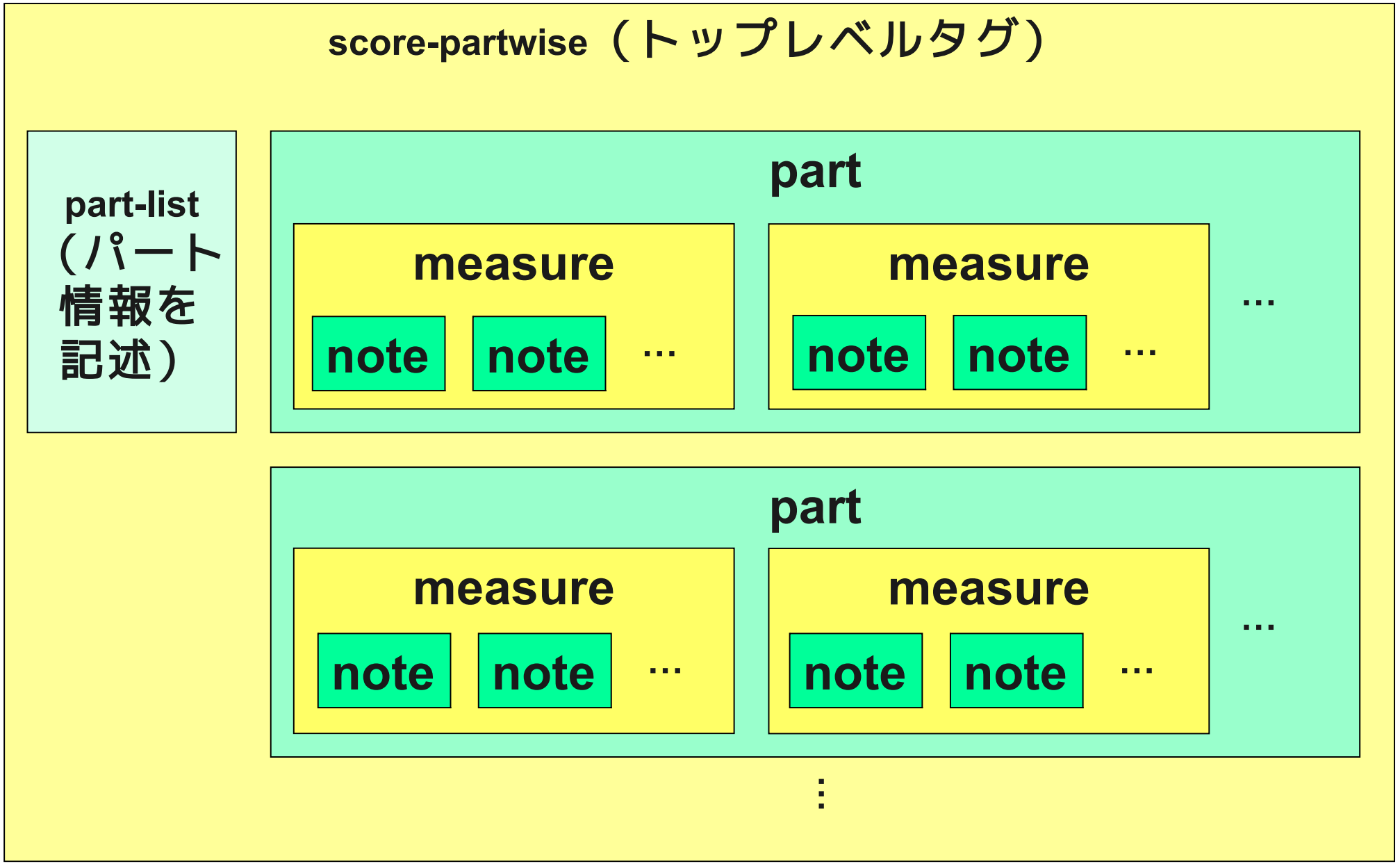
演奏生成はどうやってするのか

- DeviationInstanceXMLデータを生成します。

各音符の情報をどうやって取得するのか

- そのためのメソッドがちゃんと用意されています。

MusicXML(partwise)の基本構造



Step 3 音符ごとに情報を取得してみる

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
f.eachnote { note ->
  println note.pitchStep()
}
```

←XMLファイルを上から読んでいき、note要素が見つかるたびに、{...}を実行

note要素の中身は、noteオブジェクトに用意された各種メソッドを通して取得

利用可能なメソッドの一部

- note.pitchStep()
- note.pitchOctave()
- note.pitchStep()
- note.actualDuration()
- note.chordNotes()
- note.grace()
- note.hasArticulation(文字列)
- note.getFirstNotations()

Step 4 各音符にスタッカートが付与されているかを検査する

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
f.eachnote { note ->
  if (note.hasArticulation("staccato"))
    println("This note has staccato.")
  else
    println("This note does not have staccato.")
}
```

参考：スタッカートが付与されている音符の記述例

```
<note>
  <pitch>
    <step>G</step>
    <octave>4</octave>
  </pitch>
  <duration>2</duration>
  <voice>1</voice>
  <type>eighth</type>
  <stem>up</stem>
  <beam number="1">begin</beam>
  <notations>
      <articulations>
        <staccato placement="below"/>
      </articulations>
  </notations>
</note>
```

Step 5 各音符にフェルマータが付与されているかを検査する

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
f.eachnote { note ->
  notations = note.getFirstNotations()
  if (notations != null && notations.fermata() != null)
    println("This note has fermata.")
  else
    println("This note does not have fermata.")
}
```

参考: フェルマータが付与されている音符の記述例

```
<note>
  <pitch>
    <step>F</step>
    <octave>4</octave>
  </pitch>
  <duration>4</duration>
  <voice>1</voice>
  <type>quarter</type>
  <stem>up</stem>
    <notations>
      <fermata type="inverted"/>
    </notations>
</note>
```

Step 6 各音符に対して、スタッカートとフェルマータの両方の有無を検査する

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
f.eachnote { note ->
  notations = note.getFirstNotations()
  if (note.hasArticulation("staccato"))
    println("Make the duration of this note shorter.")
  if (notations != null && notations.fermata() != null)
    println("Make the tempo slower here.")
}
```

Step 7 演奏データを生成する

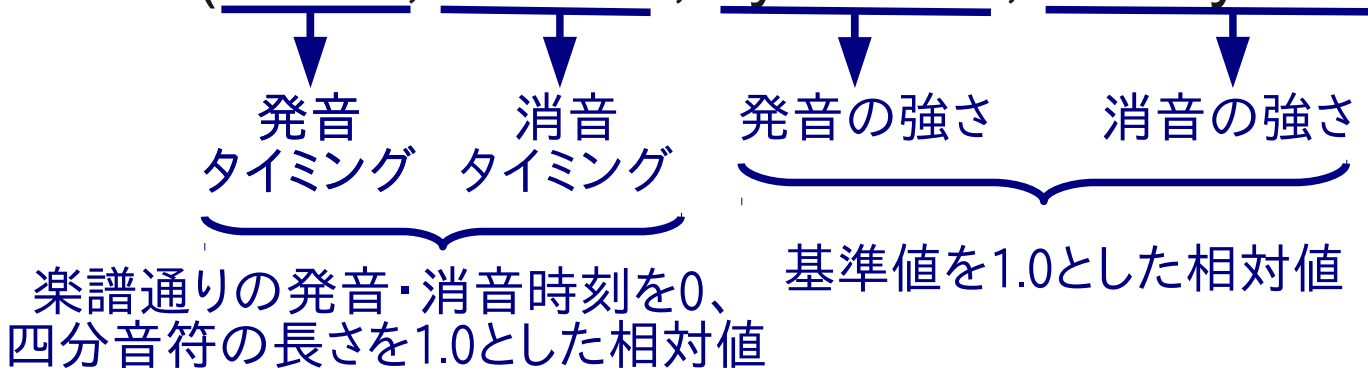
ここでは、実際の演奏データの代わりに、
演奏の楽譜からの差分（deviationデータ）を生成します。

どうやって?

deviationの生成には、DeviationDataSetクラスを使います。

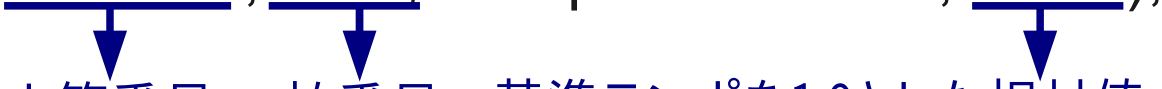
- ある音符の長さを短くするには、addNoteDeviationを使います。

  addNoteDeviation(attack, release, dynamics, endDynamics);



- テンポを変えるには、addNonPartwiseControlを使います。

  addNonPartwiseControl(measure, beat, “tempo-deviation”, value);



Step 7 演奏データを生成する

```
import jp.crestmuse.cmx.filewrappers.*

f = CMXFileWrapper.readfile("sample.xml")
dds = new DeviationDataSet(f)
f.eachnote { note ->
  notations = note.getFirstNotations()
  if (note.hasArticulation("staccato"))
    dds.addNoteDeviation(note, 0.0, -note.actualDuration() / 2,
                                1.0, 1.0)
  if (notations != null && notations.fermata() != null)
    dds.addNonPartwiseControl(note.measure().number(),
                                  note.beat(), "tempo-deviation", 0.5)
}
```

Step 8 演奏データを保存する

```
（前スライドの続き）
dev = dds.toWrapper()
dev.finalizeDocument()
dev.writefile("deviation.xml")
```

Step 9 標準MIDIファイルとして保存する

```
（上記の続き）
dev.toSCCXML(480).toMIDIXML().writefileAsSMF("result.mid")
```

実習2
MIDIデータのリアルタイム処理

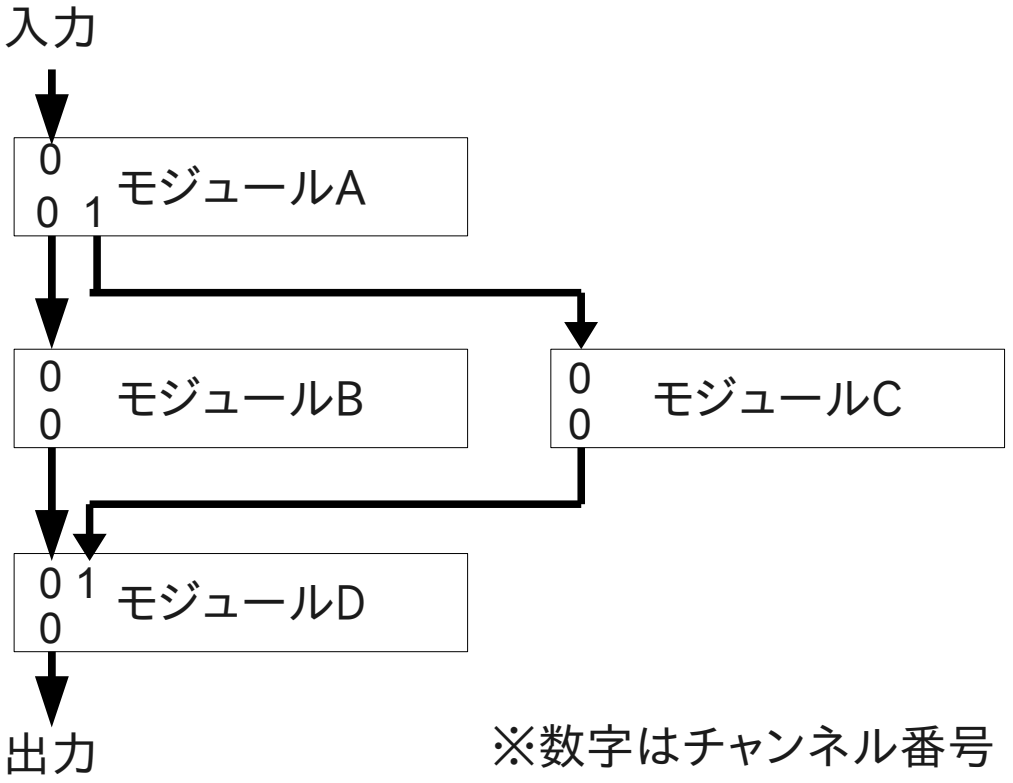
【本実習の目標】
画面に表示されたバーチャルキーボードを使って演奏すると、そのデータがリアルタイムに加工されて出力されるシステムをつくる。

基本的な考え方

処理全体がいくつかの「モジュール」に分割され、
モジュールの中をデータが流れていくものと考える

||

データフロー型プログラミング



※数字はチャンネル番号

「モジュール」の特徴

- 何らかのデータが入力され、処理された後、出力される。
- 複数種のデータが入力or出力されてもいい。
- データが到着するたびに、処理が自動的に行われる。

Step 1 MIDI Inputから来たデータをそのままMIDI Outputに出力

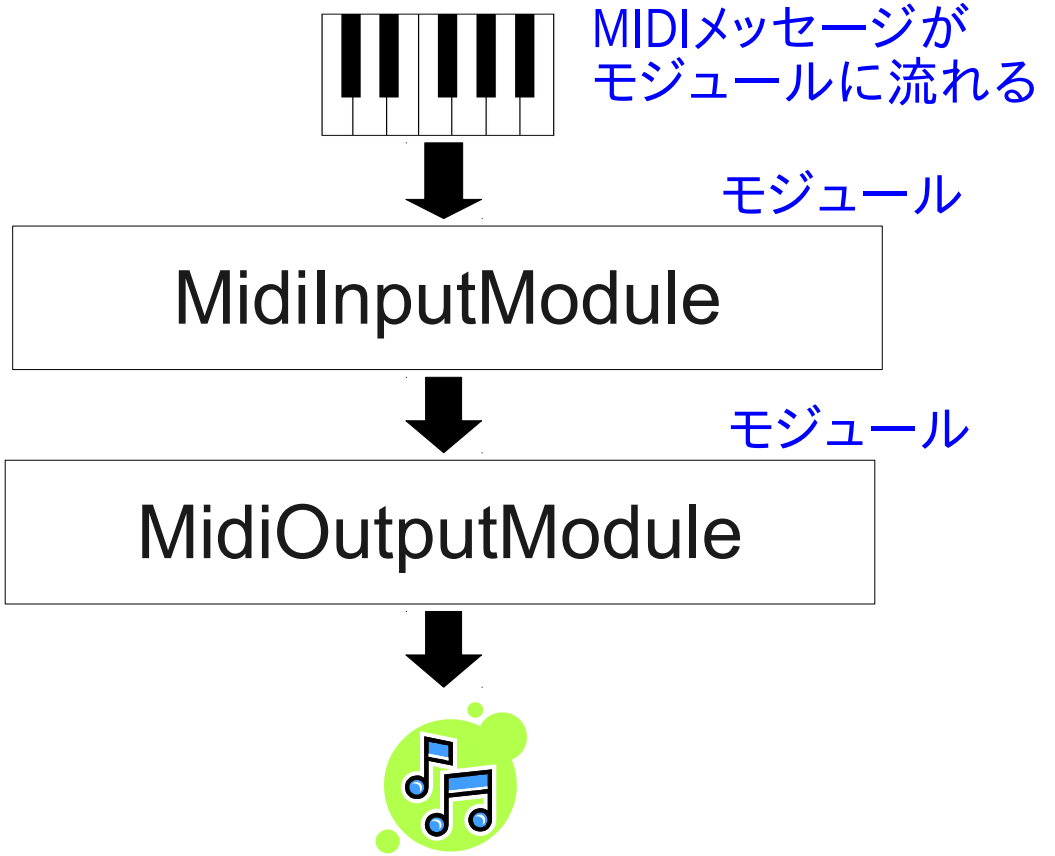
戦略

次のモジュールを使用します。

- MidiInputModule: MIDI Inputからデータ取得
- MidiOutputModule: データをMIDI Outputに出力

方法

- 上のモジュールを使うには、同名のクラスを使います。
- モジュールを登録・実行するには、SPExecutorクラスを使います。



必要な手順をまとめると…

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

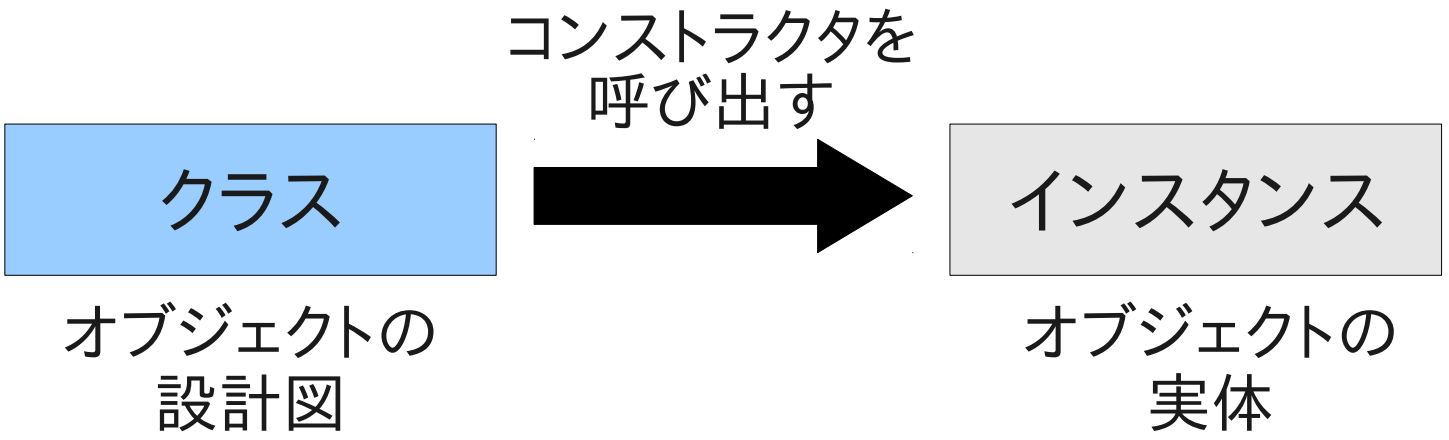
3. 登録したモジュールをつなぐ
（MidiInputModuleの出力をMidiOutputModuleの入力につなぐ）

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？



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

1

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

SPExecutor:

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

MidiInputModule:

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

←PCのキーボードを仮想的にMIDIインプットデバイスにするもの

MidiOutputModule:

```
rcv = MidiSystem.getReceiver()
mo = new MidiOutputModule(rcv)
```

←Java付属のソフトシンセを利用

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)
```

miオブジェクトの出力（チャンネル0）を
moオブジェクトの入力（チャンネル0）につなぐ

4

SPExecutorの実行を開始する

```
exec.start()
```

Step 1の完成版

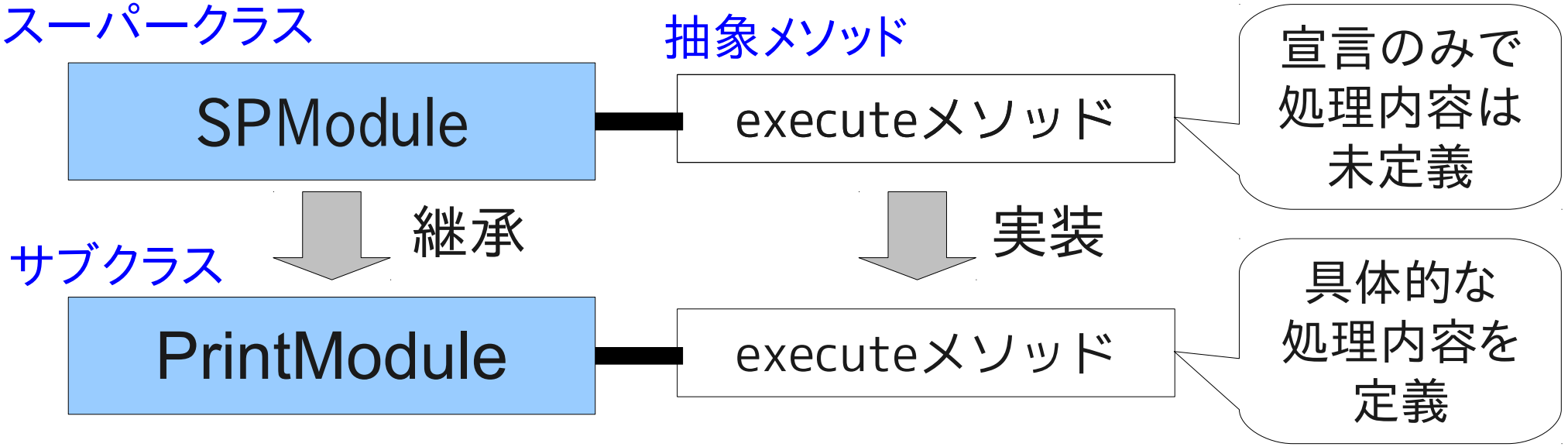
```
import jp.crestmuse.cmx.amusaj.sp.*
import jp.crestmuse.cmx.sound.*
import javax.sound.midi.*

exec = new SPExecutor()
vkb = new VirtualKeyboard()
mi = new MidiInputModule(vkb)
rcv = MidiSystem.getReceiver()
mo = new MidiOutputModule(rcv)

exec.addSPModule(mi)
exec.addSPModule(mo)
exec.connect(mi, 0, mo, 0)
exec.start()
```

Step 2 独自モジュールの作成 (1/2)

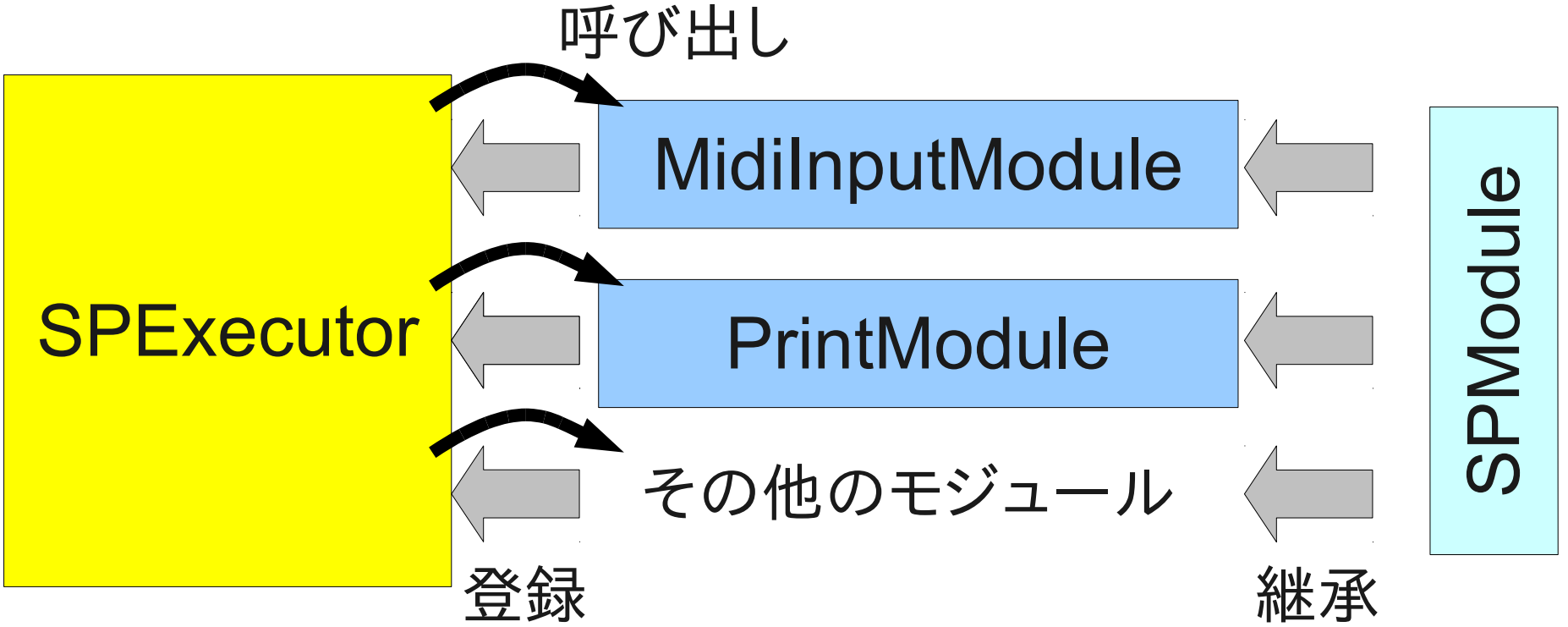
MIDIメッセージを受け取って画面表示する
モジュール「PrintModule」を作成してみよう

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

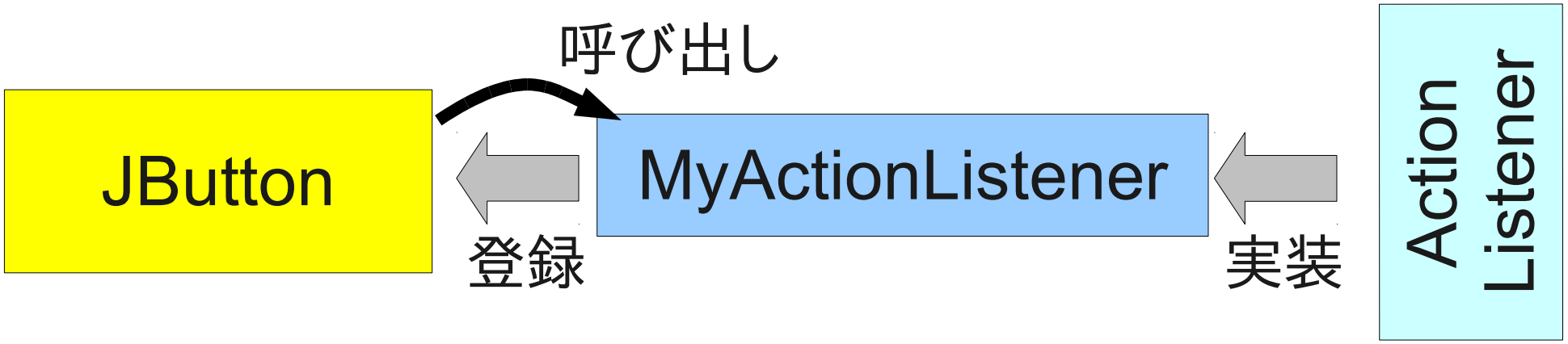


Step 2 独自モジュールの作成 (2/2)



・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。

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
  }
}
```

```
class PrintModule extends SPModule {
  void execute(Object[] src, TimeSeriesCompatible[] dest) {
      ここにモジュールがすべき処理内容を記述する
  }

  Class[] getInputClasses() {
      ここに、このモジュールが受け付けるオブジェクトのクラス名を書く
  }

  Class[] getOutputClasses() {
      ここに、このモジュールが出力するオブジェクトのクラス名を書く
  }
}
```

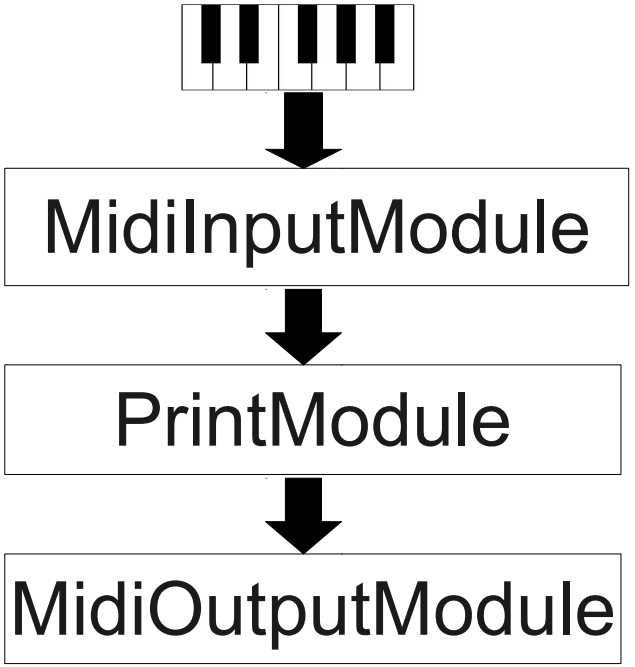
ファイルの最初に次のimport文を追加

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

ここで作戦会議

PrintModuleの仕様は?

- MidiInputModuleからデータを取得
- 取得したデータの中身を画面表示
- 取得したデータをそのまま MidiOutputModuleへ出力



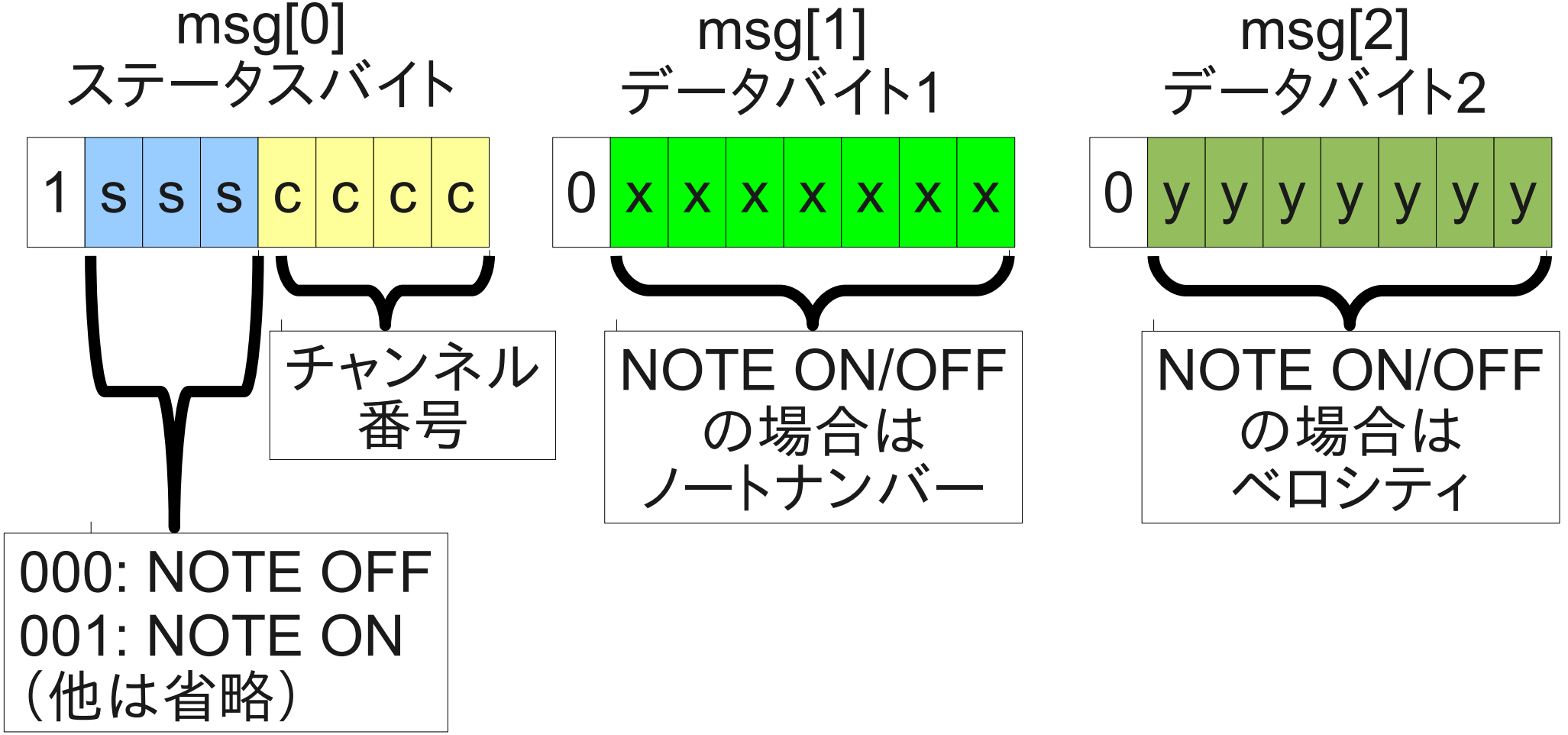
MIDIの情報はどのように扱われるのか

- MIDIEventWithTicktimeというクラスのオブジェクト（このクラスのメソッドを使って情報を取得できる）

そもそもMIDIの情報はどういう風になっているのか

- → 次のスライド

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。

再び作戦会議

おさらい

- モジュールの処理内容は、executeメソッドに書く
- MIDI情報は、MIDIEventWithTicktimeクラスで扱う
- PrintModuleは、入力・出力ともにチャンネル数は1

で、結局executeメソッドはどう書けばいいの?

```
class PrintModule extends SPModule {
  void execute(Object[] src, TimeSeriesCompatible[] dest) {

  }
```

各入力チャンネルから
得られたデータがここにある

今回は、src[0]が
入力されたMIDIデータ

dest[チャンネル番号].add(出力データ)
とすればデータが出力される

今回は、dest[0].add(MIDIデータ)
とすればよい

executeメソッドの答え

```
void execute(Object[] src, TimeSeriesCompatible[] dest) {

  // src[0]からステータスバイトとデータバイトを取得
  def (status, data1, data2) =
                  src[0].getMessageInByteArray()

  // 取得したステータスバイトとデータバイトを画面表示
  println(status + "    " + data1 + "    " + data2)

  // 入力されたデータをそのまま出力
  dest[0].add(src[0])
}
```

executeメソッド以外の答え

```
Class[] getInputClasses() {
  [MidiEventWithTicktime.class]
}

Class[] getOuputClasses() {
  [MidiEventWithTicktime.class]
}
```

完成したPrintModuleを使ってみる

```
exec = new SPExecutor()
vkb = new VirtualKeyboard()
mi = new MidiInputModule(vkb)
rcv = MidiSystem.getReceiver()
mo = new MidiOutputModule(rcv)
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()
```

発展版：入力されたMIDIデータを1オクターブ高くして出力する

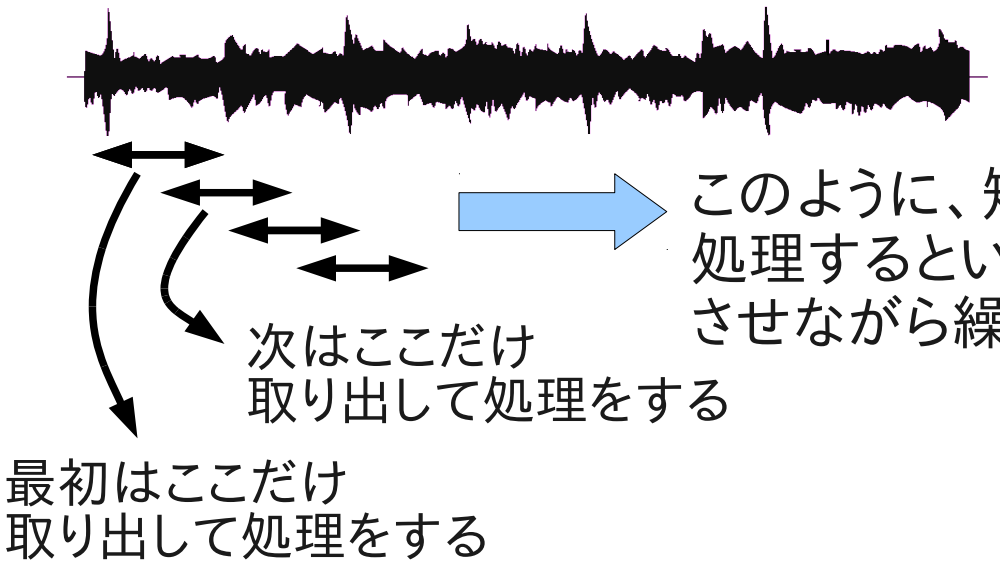
executeメソッドの中身を次のように変更すればOK。説明は省略。

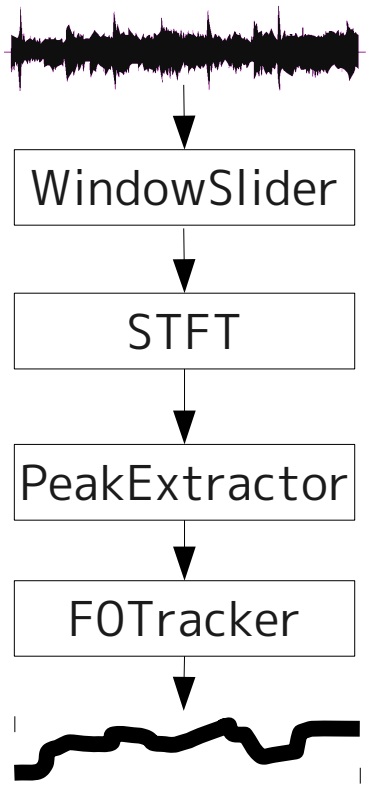
```
void execute(Object[] src, TimeSeriesCompatible[] dest) {
  // src[0]からステータスバイトとデータバイトを取得
  def (status, data1, data2) =
                  src[0].getMessageInByteArray()

  // 1つめのデータバイトに12をたしたMIDIイベントを生成
  def newevent =
    MidiEventWithTicktime.createShortMessageEvent(
    [status, data1 + 12, data2], 0, src[0].music_position)

 // 生成したMIDIイベントをそのまま出力
  dest[0].add(newevent)
}
```

実習3
オーディオデータの信号処理

【本実習の目標】
実習2で用いたAPIはオーディオデータの信号処理にも使えるので、簡単な信号処理を試してみる。ここでは、wavファイルの簡単な基本周波数推定を取り上げる。

基本的な考え方





- 短区間をシフトさせながら取り出すモジュール「WindowSlider」を利用。
- フーリエ変換には「STFT」モジュール、ピーク抽出には「PeakExtractor」モジュールを利用。
- ピークから基本周波数(F0)を抽出するモジュール「F0Tracker」は、自作する。

AbstractWAVAnalyzer

- さきほどと同様にSPExecutorを直接呼び出してもよいが、WAVファイルを読み込んだりと、少々面倒。
- 面倒な部分を自動化してくれるAbstractWAVAnalyzerを使う。

AbstractWAVAnalyzerの使い方

- AbstractWAVAnalyzerは抽象クラスなのでサブクラスを作る。
- 次のメソッドを実装する。
  - getUsedModules() – 使うモジュールを記述
  - getModuleConnections() – モジュールのつなぎ方を記述
  - getAmusaXMLFormat() – 出力データの形式を記述
  - getOutputData() – 出力すべきデータを記述

Step 1 とにかくAbstractWAVAnalyzerのサブクラスを作ってみる

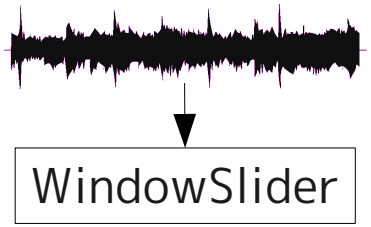
```
import jp.crestmuse.cmx.amusaj.commands.*
import jp.crestmuse.cmx.amusaj.sp.*

class MyAnalyzer extends AbstractWAVAnalyzer {
  ProducerConsumerCompatible[] getUsedModules() {
    []
  }
  ModuleConnection[] getModuleConnections() {
    []
  }
  String getAmusaXMLFormat() {
    ""
  }
  OutputData[] getOutputData() {
    []
  }
}
```

Step 2 作ったサブクラスを実行してみる

```
（前スライドの続き）
(new MyAnalyzer()).start("-conf config.xml sample.wav")
```

（config.xmlとsample.wavをダウンロードしておくこと）

- AbstractWAVAnalyzerは、デフォルトで
  WindowSliderをSPExecutorに登録する。

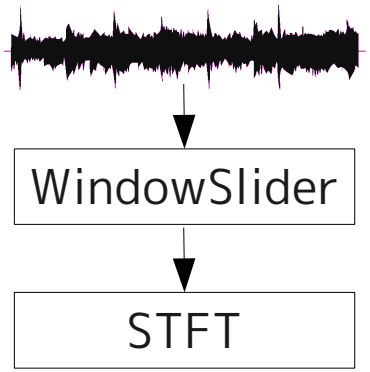


- 他にモジュールを何も登録していないので、
  指定したwavファイルに対してWindowSliderのみ実行。
- getOutputData()に何も指定していないので、
  処理結果はXMLファイルに何も出力されない。
- config.xmlは、様々なパラメータを書いたXMLファイル。

Step 3 WindowSliderの後ろに STFTモジュールをつないでみる

```
class MyAnalyzer extends AbstractWAVAnalyzer {
  def stft
  ProducerConsumerCompatible[] getUsedModules() {
    stft = new STFT(false) ←引数はステレオかどうかの指定
    [stft]
  }
  ModuleConnection[] getModuleConnections() {
    [new ModuleConnection(getWindowSlider(), 0, stft, 0)]
  }
    残り省略
}
```

↑デフォルトで作られる WindowSliderを取得



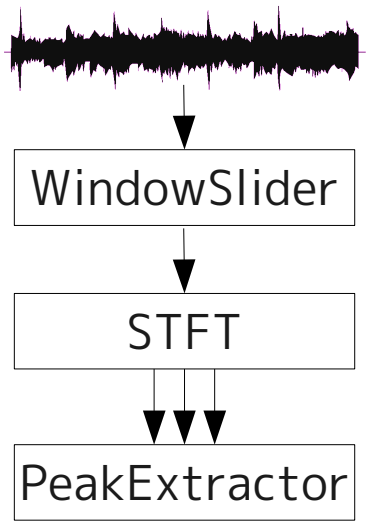
Step 4 STFTモジュールの後ろに PeakExtractorモジュールをつないでみる

```
class MyAnalyzer extends AbstractWAVAnalyzer {
  def stft, peakext
  ProducerConsumerCompatible[] getUsedModules() {
    stft = new STFT(false)
    peakext = new PeakExtractor()
    [stft, peakext]
  }
  ModuleConnection[] getModuleConnections() {
    [new ModuleConnection(getWindowSlider(), 0, stft, 0),
     new ModuleConnection(stft, 0, peakext, 0),
     new ModuleConnection(stft, 1, peakext, 1),
     new ModuleConnection(stft, 2, peakext, 2)]
  }
   残り省略
}
```

←左右ミックス信号

←左チャンネル信号

←右チャンネル信号



- STFTとPeakExtractorの間は、3チャンネル分の接続が必要

Step 5 PeakExtractorの出力を XMLファイルに書き出してみる

```
class MyAnalyzer extends AbstractWAVAnalyzer {
   省略
  String getAmusaXMLFormat() {
    "peaks"
  }
  OutputData[] getOutputData() {
    [new OutputData(peakext, 0)]
  }
}
(new MyCommand()).start(
    "-conf config.xml sample1.wav -o result.xml")
```

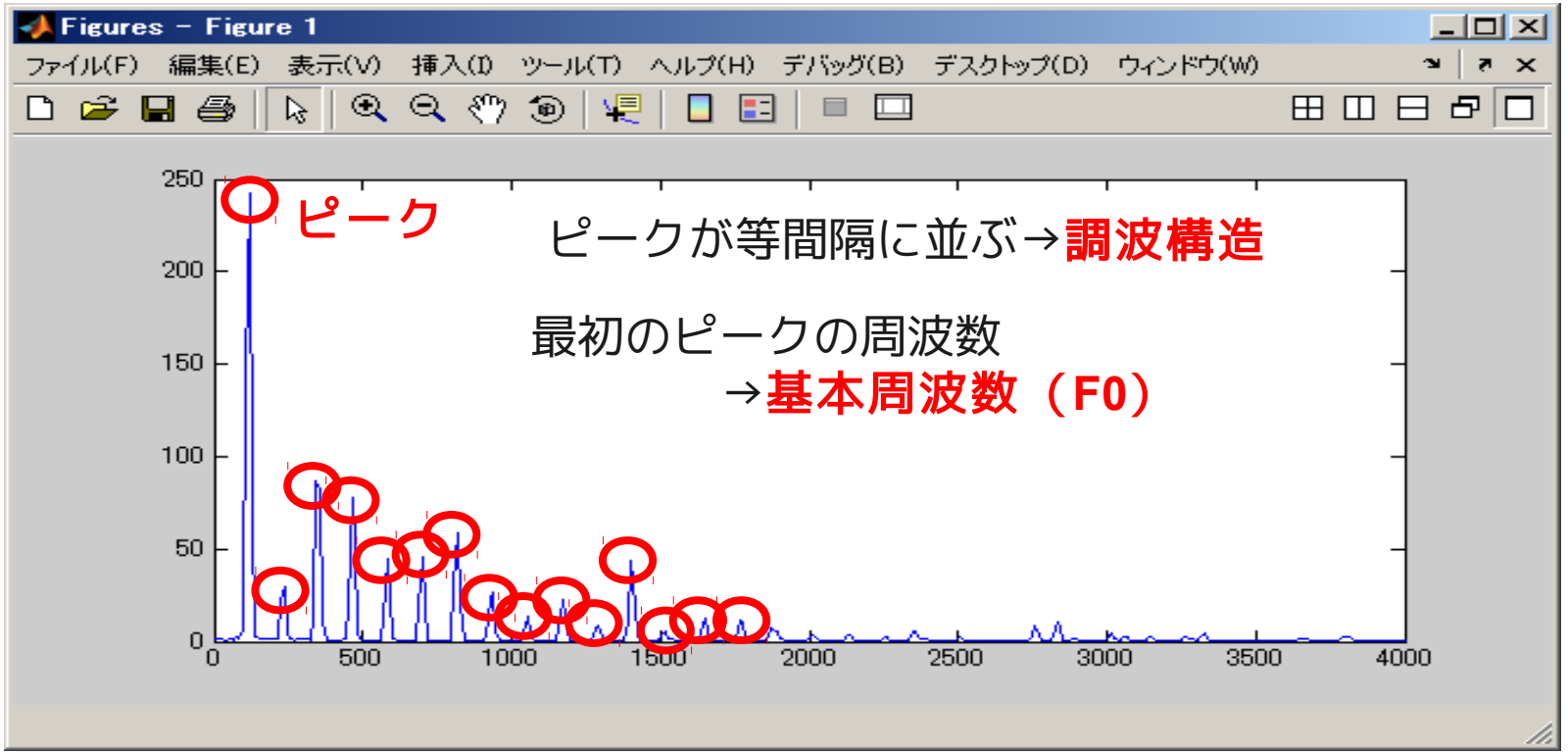
Step 6 抽出したピークからF0を推定するモジュールを作ってみよう。

仮定

- ノイズは十分に小さい。
- 複数音源が同時に発音しない。
  →調波構造が同時に1つしかない。

方針

- 簡単なノイズ除去後、抽出されたピークのうち、最も低い周波数をF0とする。



Step 6 抽出したピークからF0を推定するモジュールを作ってみよう。

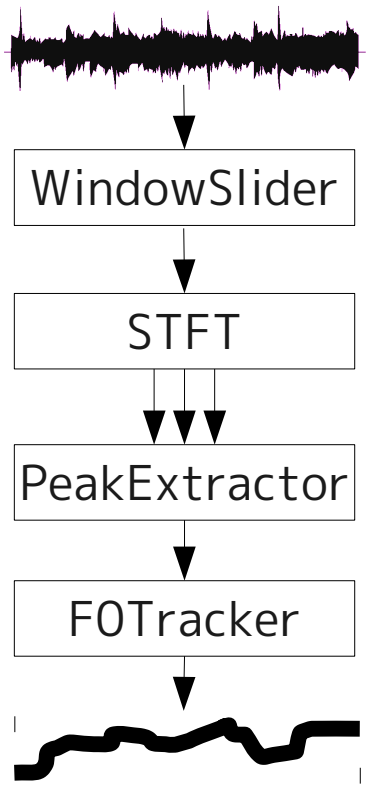
ちょっと複雑なので,

今回はすでに作ってあるものを使いましょう.

MyF0Tracker.groovyをダウンロードしてください.

Step 7 PeakExtractorモジュールの後ろにMyF0Trackerモジュールをつないでみる

```
class MyAnalyzer extends AbstractWAVAnalyzer {
  def stft, peakext, f0tracker
  ProducerConsumerCompatible[] getUsedModules() {
    stft = new STFT(false)
    peakext = new PeakExtractor()
    f0tracker = new MyF0Tracker()
    [stft, peakext, f0tracker]
  }
  ModuleConnection[] getModuleConnections() {
    [new ModuleConnection(getWindowSlider(), 0, stft,
     new ModuleConnection(stft, 0, peakext, 0),
     new ModuleConnection(stft, 1, peakext, 1),
     new ModuleConnection(stft, 2, peakext, 2),
     new ModuleConnection(peakext, 0, f0tracker, 0)]
  }
   残り省略
}
```



Step 8 XMLファイルへの出力を MyF0Trackerからの出力に変更する

```
class MyAnalyzer extends AbstractWAVAnalyzer {
   省略
  String getAmusaXMLFormat() {
    "array"
  }
  OutputData[] getOutputData() {
    [new OutputData(f0tracker, 0)]
  }
}
(new MyCommand()).start(
  "-conf config.xml sample.wav -o result.xml")
```

- できたXMLファイルをMATLABで読み込むプログラムを提供
- ここでは、XMLのタグを手動で消して、Excelで読んでみる。

最後に∝

今回扱わなかった機能

- MusicXML, DeviationInstanceXML以外のXML形式
- 独自XML形式への対応
- SMFを再生しながらMIDI入力処理
- ベイジアンネットワークを用いた音楽データの推論
- マイクから入力された音響信号の処理
- 「jfftw for CMX」を用いたFFTの高速化
- 「jAudio for CMX」を用いた各種音響特徴抽出
- …などなど多数

最後にお願い

- 現時点では十分なドキュメント作成ができていません。ご不明な点は、ぜひ遠慮せずに聞いてください。
- 共同開発に興味のある方は、ぜひ連絡ください。コーディングでなくても、テスト、ドキュメント作成などでも大歓迎です。

お問い合わせ先

- Web: http://www.crestmuse.jp/cmx/
  http://sourceforge.jp/projects/cmx/
- メール：t.kitahara [at] kwansei.ac.jp