プログラミングが容易な簡易GUIフレームワーク・コンポーネントウエア lab.Lab（仮称）の概要

2013-07-22

産業技術総合研究所

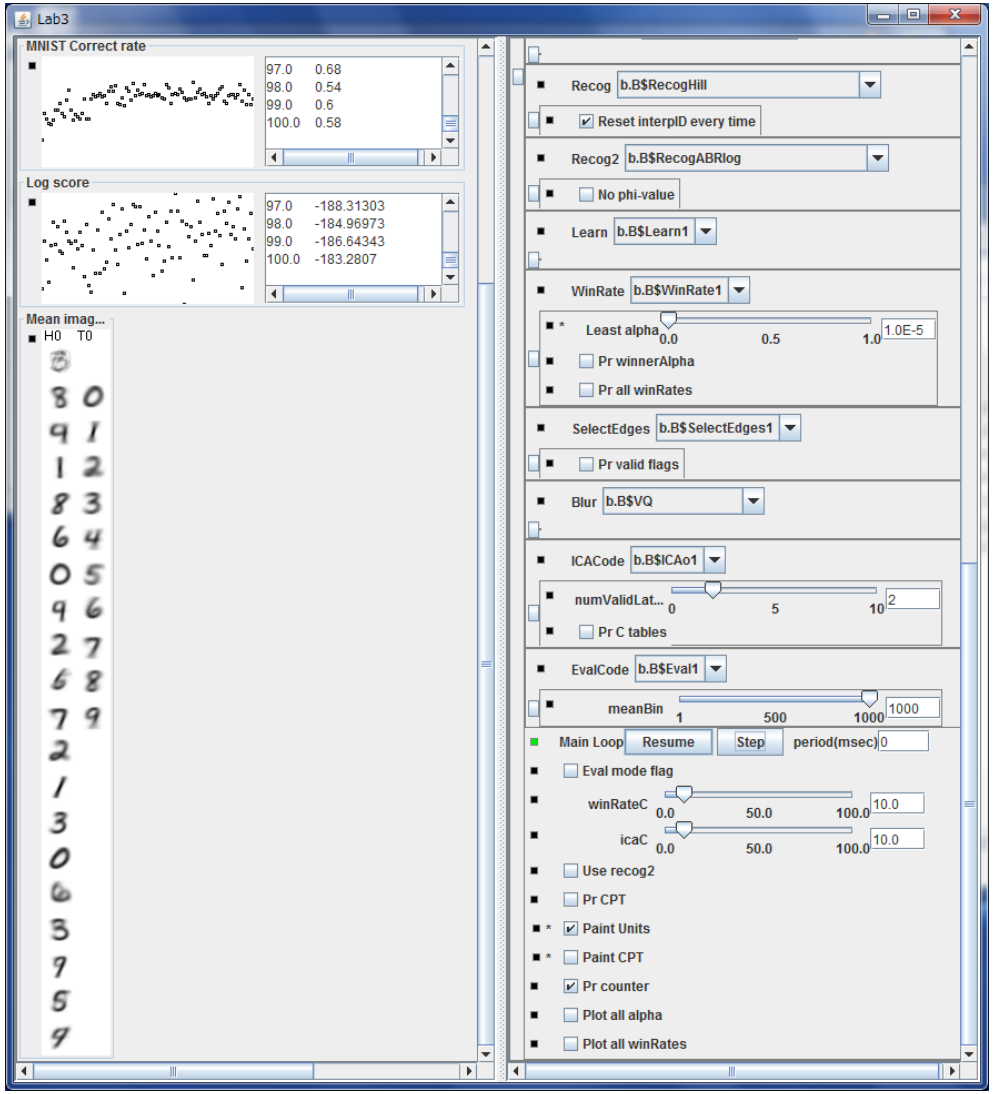
一杉裕志

注：本資料中のスクリーンショットとAPIは開発中のもので、最新版とは異なる場合があります。

lab.Lab とは

- **プログラミングしやすい簡易GUIフレームワーク。**
  - GUI部品が一行で追加できる。
- **プログラミングしやすい簡易コンポーネントウエア。**
  - モジュール（アプリケーションの部品）の追加・選択が容易。
- Java 言語上での複雑なアルゴリズム開発を支援する目的で開発。
- Swing 上で実装。
- とてもシンプル。将来の機能拡張の余地が大きい。

本フレームワークを使ったアプリケーションのスクリーンショット

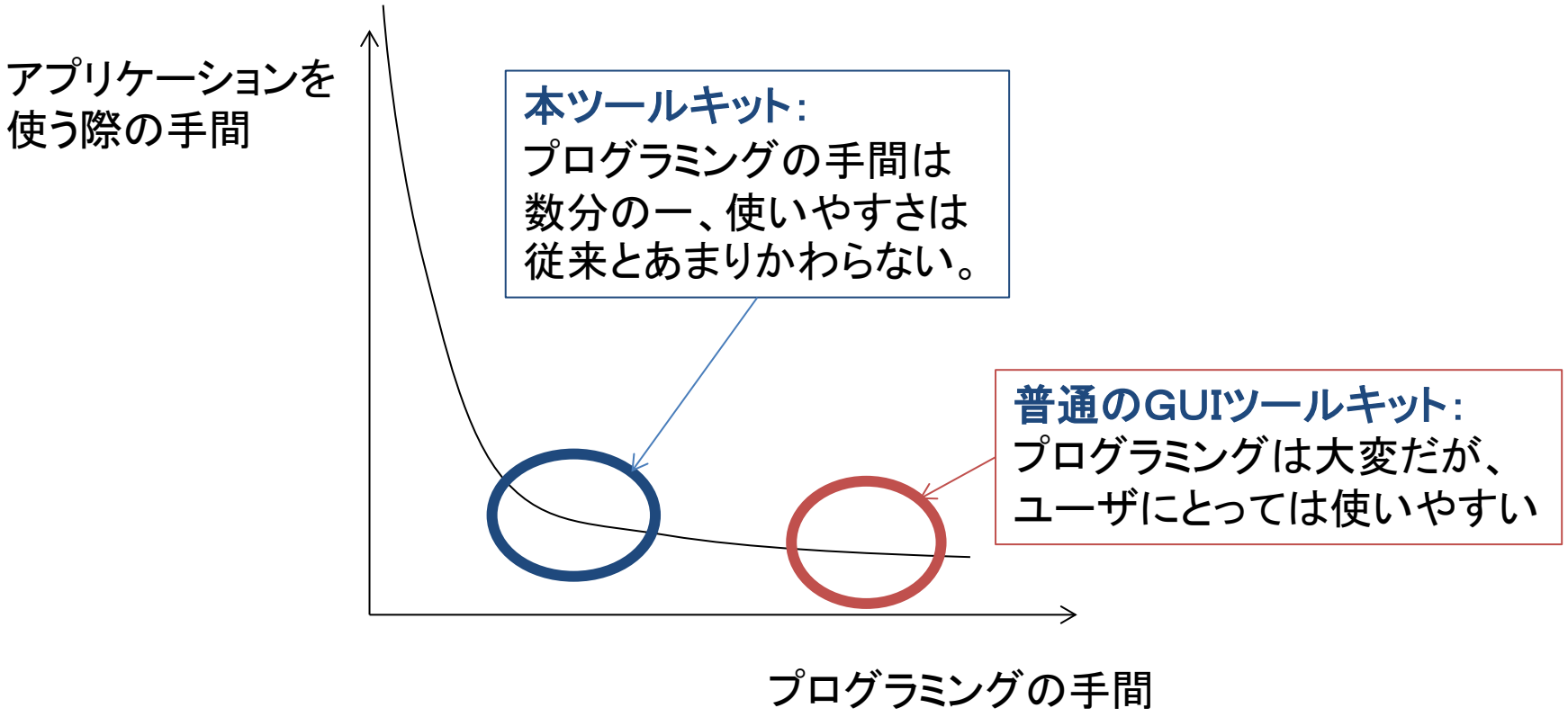


設計方針

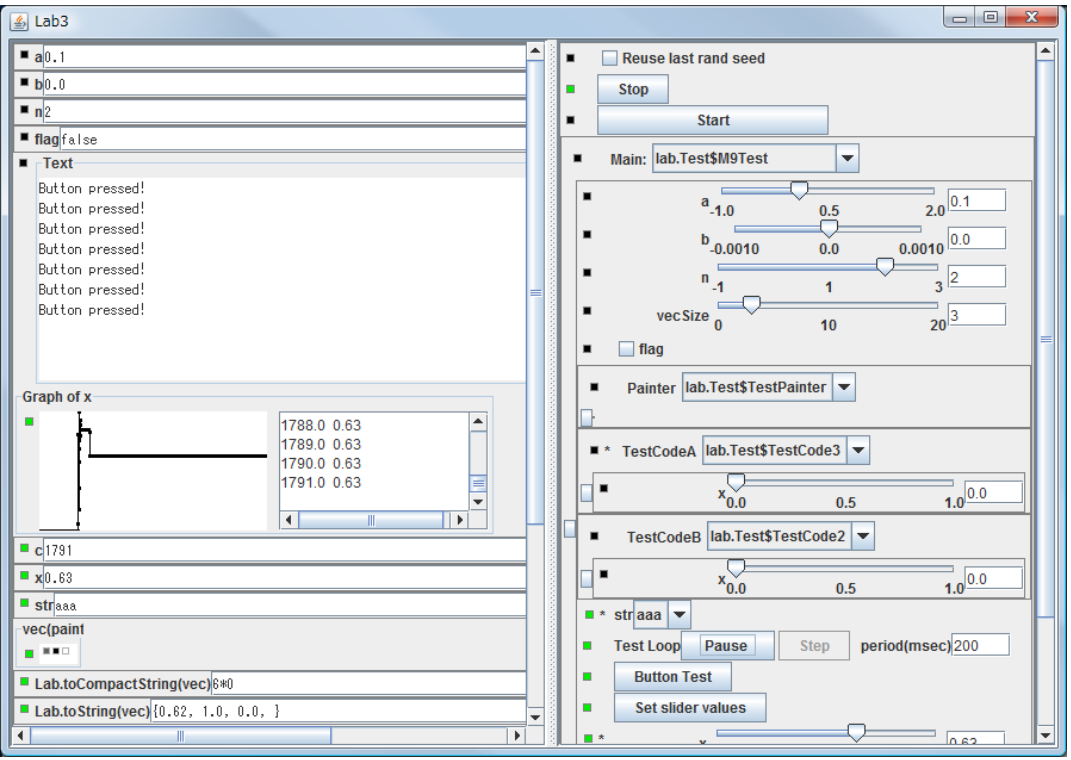
開発の動機

- アルゴリズム開発の際に、複数のアルゴリズム、テストプログラム、評価方法などのモジュールを、さまざまな組み合わせで試したい。
- それぞれのモジュールはパラメタがたくさんある。最適なパラメタ値を時には手動、時には自動で探索したい。
- GUIを使いたいが、従来のGUIビルダは使いにくい。特にプログラム自身の構造が頻繁に変わるときにGUIの保守性が悪いのをなんとかしたい。

プログラマーの手間とユーザの手間のトレードオフ



入出力をできるだけ簡単に



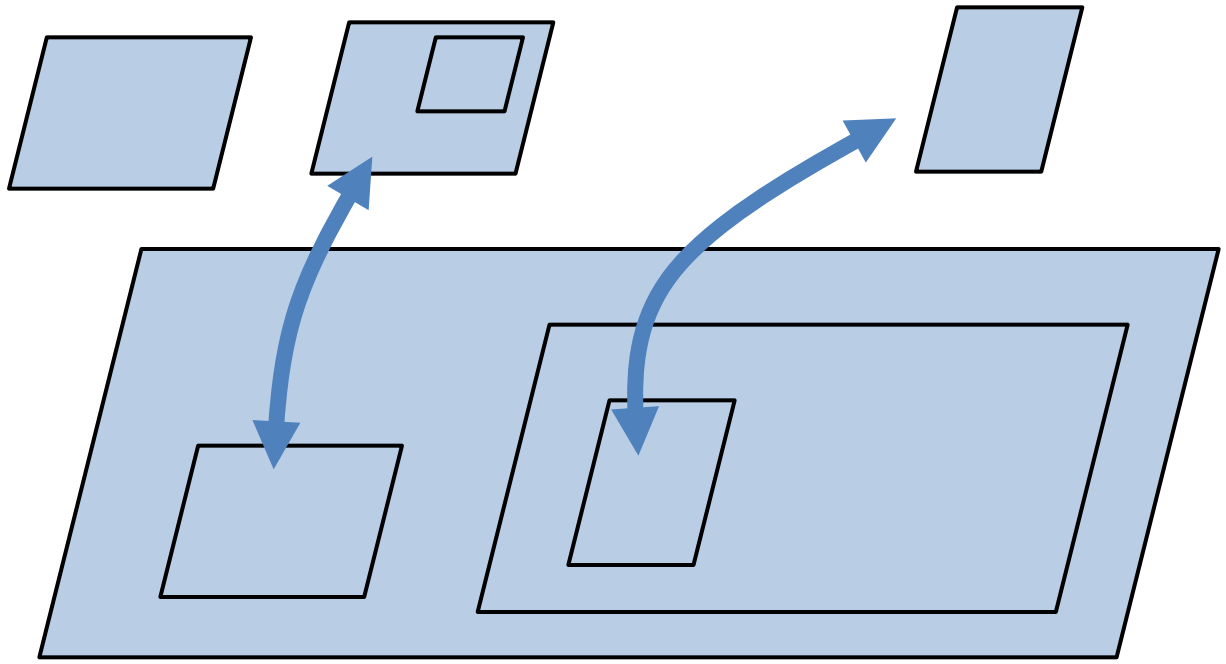
スライダ、チェックボックス、セレクタなどが1行で追加可能。

グラフ、配列の可視化なども1行で追加可能。

```
float a = panel.getFloat("a", 0.1f, -1, 2);
float b = panel.getFloat("b", 0, -0.001f, 0.001f);
int n = panel.getInt("n", 2, -1, 3);
int vecSize = panel.getInt("vecSize", 3, 0, 20);
boolean flag = panel.flag("flag");
TestPainter painter = panel.getCode("Painter", TestPainter.class);
```

メタファーその１

- パソコンの部品のように再利用性を高めたい。
  - 階層構造（モジュールの中にまたモジュール）
  - パラメタ（部品内の半固定抵抗やディップスイッチ）



メタファーその２

- C プリプロセッサの #define, #ifdef でやることをGUIでやりたい。

コードのバリエーションの組み合わせ

```
#define DEBUG
...


#ifdef DEBUG
...
#endif
```

チェックボックス、メニュー

定数定義

```
#define BUFSIZE 1024
#define ALPHA  0.1
...
```

スライダ

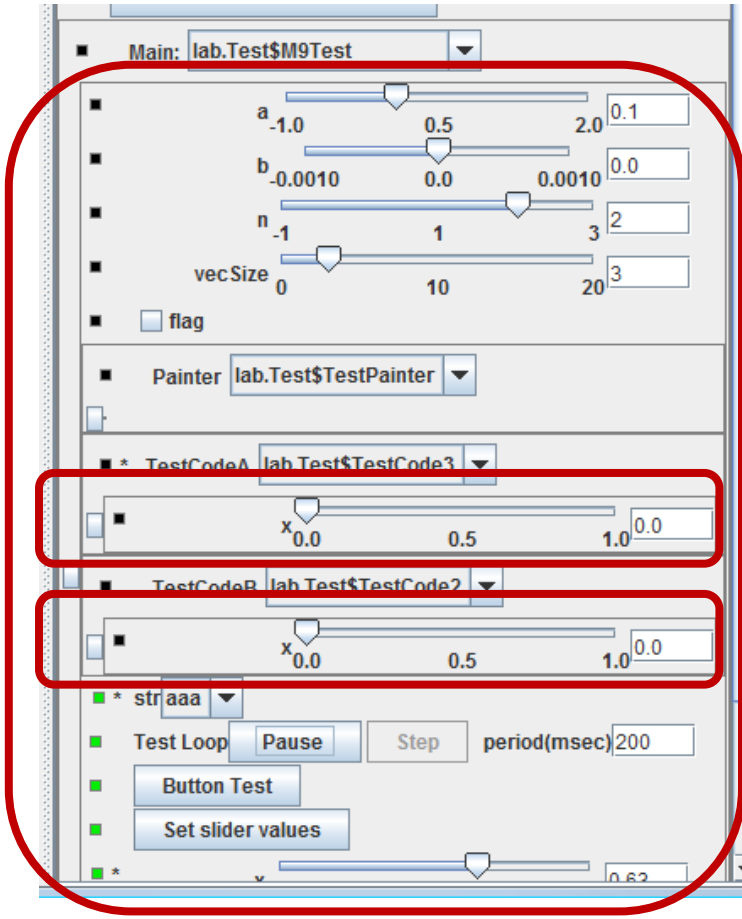
GUIフレームワークとしての機能

少ない労力で手軽にGUI

- Java GUI プログラミングの**落とし穴**になりがちな**レイアウトマネージャ**と **EDT** (event dispatch thread)を**隠蔽**。
  - レイアウトをほとんど気にしなくてよい。
  - マルチスレッドをほとんど気にしなくてよい。
    - かつての BASIC 言語の input 文のように気軽に入力、print 文のように気軽に出力。

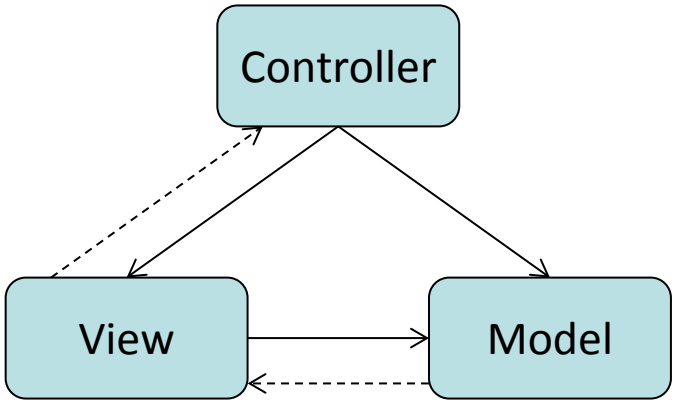
自動レイアウト

- **極限までシンプル**で、**そこそこ使いやすい**レイアウトを提供。
- 部品は必ず panel の最下部に追加。
- 入力部品は tree 状に配置。
  - **panel はネストする**。
  - モジュール（後述）の階層構造を panel の階層構造に対応づけることで、**自動的に自然なレイアウト**になる。



panel のネスト

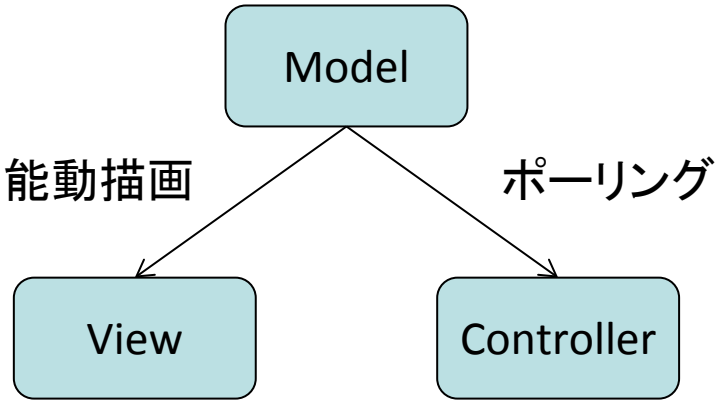
EDT(event dispatch thread)の隠ぺい、MVCの分離、簡潔な呼び出し関係



従来の Model-View-Controller

イベントドリブン。
「Contoroller = ユーザの視点」が中心。
コールバックがからんで複雑！

向いている用途：
一般ユーザによるドキュメント等の作成



本フレームワーク

アプリケーションスレッドドリブン。
「Model =プログラマの視点」が中心。
GUIメソッドの素直な呼び出し。

向いている用途：
プログラマによるアルゴリズム開発

イベントドリブン vs. アプリケーションスレッドドリブン

- **ドキュメント作成アプリケーション**の場合：
  - モデルの一貫性が崩れた状態は一時的にしか発生しない。
  - ユーザの操作はモデルに速やかに反省させたい。
  - ゆえに**イベントドリブン**の実装が向いている。
- **複雑なアルゴリズムを長時間動作させるプログラム**の場合：
  - モデルに一貫性があって可視化可能な状態が時々しか発生しない。
  - モデルの一貫性を崩さずにパラメタ変更を反映可能な瞬間も時々しか発生しない。
  - ゆえに**アプリケーションスレッドドリブン**で能動描画・ポーリングするスタイルが向いている。
    - 描画できる時に描画し、値を使える時に値を取得するというスタイル。APIがシンプルになるだけでなく、関係するコードが分散しなくてすむので**プログラムの保守性が上がる**という利点もある。

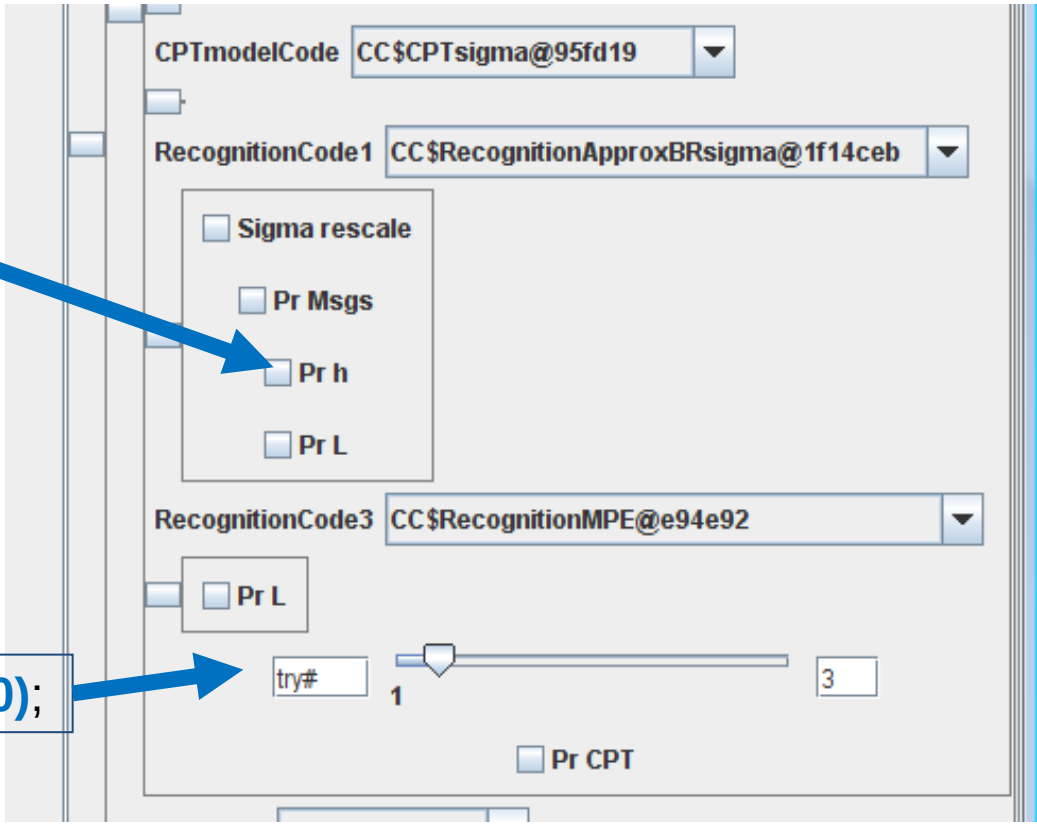
入力部品からの値の取得

- 値が必要になった時に**その場所に1行書くだけ**で、GUI部品の生成、レイアウト、値の取得のすべてが自動で行われる。
  - （従来のGUIツールキットでは3行程度必要。しかも**場所は分散**。）
- 例：

  int x = panel.getInt(“label”, default, min, max);
- GUI部品は panel に登録され、ラベルで識別される。
  - 2度目以降の呼び出しでは部品は生成されず、値の取得のみが行われる。
  - 同じラベルでも異なる panel では別の部品として扱われる。

チェックボックス、スライダ、ボタン

```
if (panel.flag("Pr h")){
        Lab.printArray("   h=", h);
}
```




```
int tryNum = panel.getInt("try#", 3, 1, 20);
```

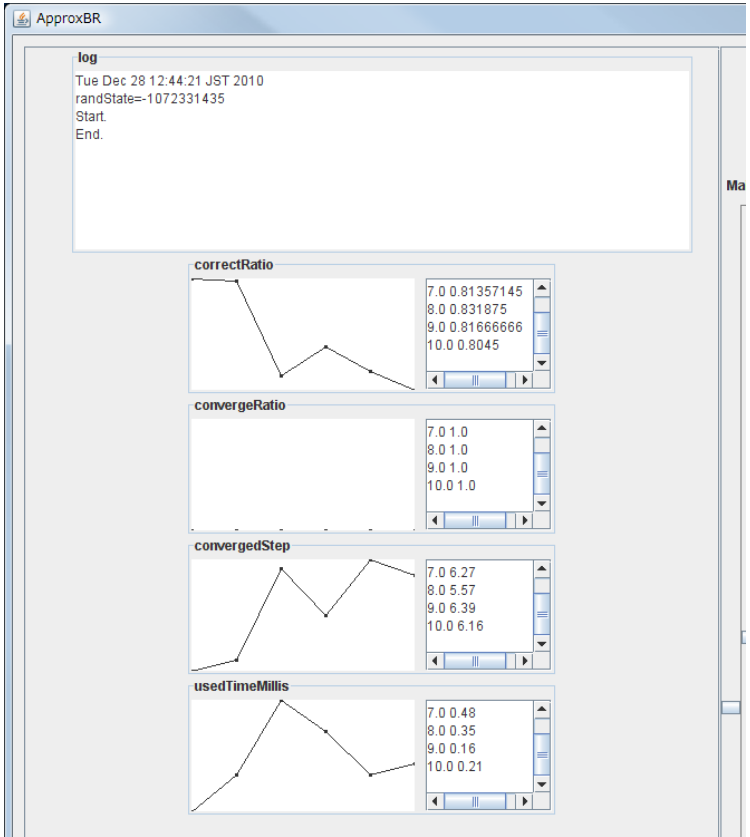
```
if (panel.button("Button Test")){
    System.out.println("Button pressed!");
}
```

押されたボタンは、値が読みだされると押されていない状態に戻る。ボタンを一度押すごとに一度だけ true が返ることになる。

グラフの出力

```
for (int i = 0; ...){
        ...
        env.viewPanel.linePlot("correctRate",  i,  correctRate);
        env.viewPanel.linePlot("convergenceRate",  i,  convergenceRate);
        env.viewPanel.linePlot("convergenceTime",  i,  convergenceTime);
        env.viewPanel.linePlot("usedTimeMillis",  i,  usedTimeMillis);
}
```

・メインループの中に plot 文を埋め込んで使う。
・最初の呼び出しでグラフが出力パネルに追加される。
・２度目以降の呼び出しで、グラフはリアルタイムに更新される。



コンポーネントウエアとしての機能

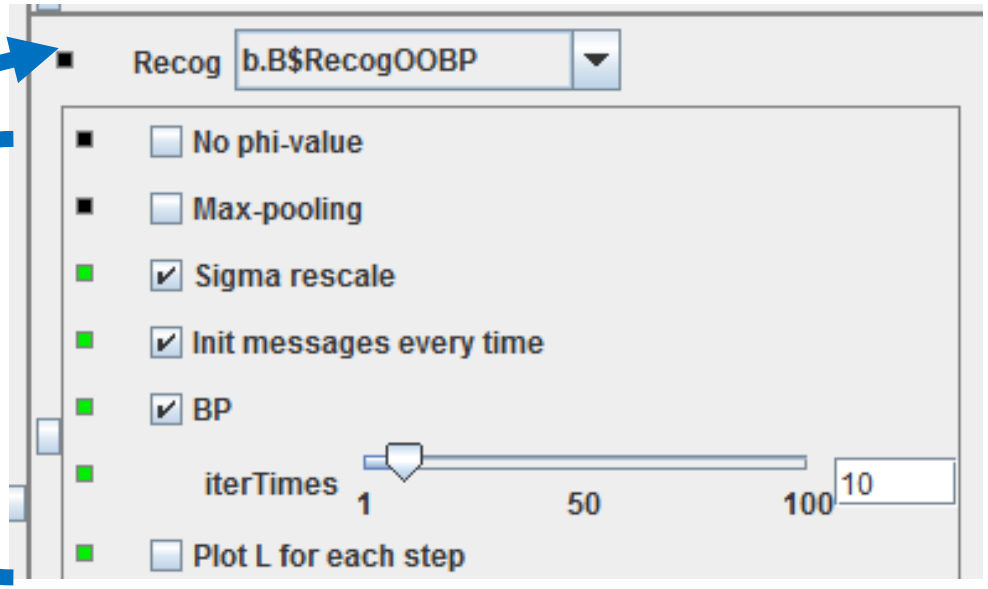
コードセレクタ（モジュールの選択）

- メニューを使って、登録されているクラスの中から1つを選択。
  - 指定したクラスのサブクラスのみが選択可能。
  - **型的に安全。**

```
RecogCode rc = panel.getCode("Recog", RecogCode.class);
rc.calcMPE(net);
```

ユーザがメニューで選択したクラスのインスタンスが返される。あとは普通のオブジェクトとして、メソッド呼び出しで使う。

コードセレクタの下には、選択されたモジュールのパラメタを入力する panel が表示される。



モジュール定義

- 定義方法はシンプル。class lab.Lab.Code を継承したクラスを定義し、モジュール登録するだけ。
  - 登録クラスの public static member class も再帰的に自動登録される。
    - モジュール登録し忘れのトラブルをなくせる。
  - 変数 panel の値は自動的に設定される。

```
package  b;
public class B {
   public static abstract class RecogCode extends lab.Lab.Code {
      public abstract void calcMPE(Net net);
      …
   }
   public static class RecogOOBP extends RecogCode {   // モジュール定義
       public boolean noPhiValueFlag = panel.flag(“No phi-value");
       …
   }
}
…
  lab. Lab.addSelectableClass(b.B.class);  // モジュール登録
```

オブジェクト指向 vs. 手続き指向

- 本フレームワークでは**手続き指向パラダイム**でのアプリケーション記述になりやすい。
  - モジュール＝手続きの集合。
    - アプリケーションを構成するコードの断片（手続きの集合）を、メニューで切り替える、というイメージ。
  - 本フレームワークの主目的であるアルゴリズム開発においては、データ構造よりも手続きの切り替えの頻度の方が高いので、この方が便利ではある。

使い心地について

自分で1年以上使ってみて実感した効果

- モジュール作成時には思いもしなかったモジュール・パラメタの組み合わせを、ふと思いついた時に気軽に試せる。
  - その際、コードを壊す恐れがないという安心感がある。
  - すべてのパラメタ（定数定義）をスライダにしておく癖がつく。その際の手間は従来と比べてとても小さい。
- デバッグ用コードをコメントアウトせずに残しておける。問題が起きた時にチェックボックスをオンにするだけで有効にできて助かる。
  - 記述時にカーソル移動が不要なので #ifdef より使いやすい。
  - 思い付いたデバッグ用コードを実装する心理的障壁が低い。
- アルゴリズムの動作をリアルタイムに可視化すると、パラメタ変更の影響を理解しやすい。
- 実験的に既存のモジュールを改変したモジュールを追加するのも容易。バージョン管理システムよりも気楽に複数のバージョンを切り替えて試せる。
- GUIへのアクセスが素直に書けて便利。

本フレームワークのデメリット

- プログラマにとってのデメリット
  - オブジェクト指向パラダイムを捨てなければならない。
    - データ構造の拡張には既存コードの修正が必要になる。
    - データ構造のメンバは原則 public にすることになる。
- ユーザにとってのデメリット
  - 入力部品・出力部品の数が増えると使いにくくなってくる。
    - ユーザがドラッグアンドドロップで部品のレイアウトを変更できるようにするなど、将来なんらかの工夫が必要。
  - 入力部品・出力部品は実行時に次々追加されていくので戸惑う。
  - イベントドリブンでないので、GUIの使い勝手には限界がある。（ただし、いまのところそれほど困ってはいない。）

関連技術など

関連技術とその問題点

- アルゴリズムアニメーションシステム
  - 昔よく研究されていた。主に教育用。専用言語で記述。アルゴリズム開発には不向き。
- プリプロセッサ(CPP)、構成管理ツール(CVS, subversion, ...)
  - 言語非依存（＝型的に安全でない）。
  - 実行中にプログラム自身から構成を変更できない。
- Matlab や python Matplotlib の plot 文
  - リアルタイムのグラフ表示ができない。
  - 引数が配列だと Java では少し使いにくい。

本フレームワークを設計する際に参考にした技術文書

- 「マルチスレッドツールキット: 見果てぬ夢？」Graham Hamilton http://www.artonx.org/collabo/backyard/?FailedDreamOrMultitaskingGuiTool
- 「Inversion of Control コンテナと Dependency Injection パターン」マーチン・ファウラー http://kakutani.com/trans/fowler/injection.html

似た目的の過去のシステム

- Squeak, Scratch
- JavaBeans などのコンポーネントウエア
- 私が作ったもの：MixJuice 、チャミー

- （最近のシステムについては不勉強です。本フレームワークと似たものをご存知の方は教えてください。）