# XDF

## XML-based Desktop Application Framework

## ユーザガイド

## (version 1.00)

開発メンバ：
髙木 俊一、児玉 敦、袁 頻

主担当教員：秋口教授
副担当教員：加藤准教授、土屋助教

# はじめに

本書は、チーム SPP(産業技術大学院大学 2008 年度秋口 PBL 袁・児玉・髙木)による Java アプリケーション開発フレームワーク
XML-based Desktop Application Framework (以下「XDF」又は「本フレームワーク」)の、アプリケーション開発者向け解説書である。

1 章には、本フレームワークの概要を記した。

2 章には、簡単な GUI アプリケーションを例として開発手順を解説した。

3 章には、本フレームワークが提供する諸機能の仕様を詳説した。

# 目次

# 1. フレームワーク紹介

## 1.1 フレームワーク概要

GUI アプリケーション構築フレームワーク XML-based Desktop Application Framework（以下、「XDF」又は「本フレームワーク」）は Java による GUI アプリケーション開発を支援するためのフレームワークである。

XDF は従来の Swing アプリケーション開発で問題となる

・開発生産性の低さ

・機能修正や拡張の難しさ

を解決し、さらに、迅速なプロトタイピングを可能とするための機能を提供する。

また、本フレームワークを利用することによって GUI ツールキット（Swing, Qt など）に依存しないアプリケーションを構築することが可能となる。これによって、プラットフォーム間の移植性とパフォーマンスの両立が実現される。

### 1.1.1 Java Swing アプリケーションの開発生産性の向上

現状の Java Swing アプリケーションでは、アプリケーションの内部ロジック以上のコード行が GUI 構築（注 1）に費やされている。そこで XDF では、GUI 構築のコード行を削減することによって、アプリケーション全体の開発生産性の大幅な向上を実現している。

注 1）ここで GUI 構築とは、GUI コンポネントの生成および各種の属性設定、GUI コンポネントの画面レイアウト設定、GUI コンポネントとアプリケーションロジックとの関連づけなどのことを意味している。

### 1.1.2 アプリケーションのメンテナンス性向上

現状の Java Swing アプリケーションの開発では、事前に十分な設計を行わない限り、アプリケーションの内部ロジックと GUI レイアウトのコードが入り交じってしまう。そのため、アプリケーションのメンテナンス性は開発者の設計に依存してしまう。また、その設計を行うためには非常に高度な Java や Swing に関する知識・経験が必要になるため、十分な設計ができないことが多い。

そこで、XDF によってアプリケーションの内部ロジックと GUI レイアウトの構築コードを十分に分離する仕組みを提供することにより、開発者の設計に依存せずにアプリケーションのメンテナンス性を向上させることを目指している。

### 1.1.3 GUI プロトタイプ構築の迅速化

現実の業務において、GUI 部分に対するプロトタイプを迅速に提供できるということが、アプリケーションの要件抽出・仕様策定に対して重要である。

しかし、現状の Java Swing API では迅速に GUI プロトタイプを提供することは難しい。そこで、本フレームワークの XML 画面定義およびスタイルシート機能によって、GUI プロトタイプを迅速に提供することを容易にする。

### 1.1.4 移植性とパフォーマンスの両立

Swing による GUI アプリケーション開発で常に問題として挙げられるのが、パフォーマンスの問題であり、実際、最近の Swing ではパフォーマンスが向上したとはいえ、ネイティブ GUI のパフォーマンスには劣っている。そのため Swing を用いず他のネイティブ GUI ツールキット（Qt など）を利用するケースもあるが、それではプラットフォーム間の移植性に問題が出る。

そこで、本フレームワークでは UI の記述方式を GUI ツールキットに依存しない形式として提供することにより、ひとつのアプリケーション実装が別の任意の GUI ツールキット上でも動作できるようにする。それによって、すべてのプラットフォーム（Windows, Macintosh, Linux など）でネイティブな GUI ツールキットを利用したアプリケーションを提供することが可能となる。

## 1.2 提供される機能 (本フレームワークが提供する使い勝手)

本フレームワークは、エンドユーザに利用される実行アプリケーション(以下「目的アプリケーション」とも)のテンプレートとして機能する。具体的には、以下に述べる各ホットスポットを記述することが、本フレームワークを用いる場合のアプリケーション作成作業となる。

### 1.2.1 画面定義文書(XML)

本フレームワークでは画面定義文書(XML)によって目的アプリケーションの GUI を定義する。この画面定義文書の構文および使用可能なタグ情報などの詳細については 3.1.1 章に記述されるため、そちらを参照のこと。

この画面定義文書は、目的アプリケーション中の画面 1 個の定義をひとつの画面定義文書によって行う。ひとつの画面定義文書中で定義可能な GUI 情報には以下のものがある。

- コンポネント構成の表現（コンポネントの種類、配置順序、包含関係）
- メニュー構成
- スタイルシートの参照
- アクションの設定

以上の各 GUI 情報で記述できる詳細な内容および記述の方法については 3.1 章に記述する。

### 1.2.2 スタイル定義

前述の画面定義文書のうちレイアウト等の属性に関する記述を、スタイル定義と呼ぶ別ファイルに抽出し、画面定義文書とは独立させて取扱うことができる。

例えば、外観の一貫性を確保する目的で、1 つのスタイル定義を幾つもの画面定義文書に適用するといったことが可能である。

### 1.2.3 アクション定義

本フレームワークでは、ウィンドウ、ボタン、コンボボックス、・・・といった画面内コンポネントをユーザが操作（クリック、キー入力など）することによって発生するイベントの処理を「アクション」として記述し、そのアクションを画面定義文書中において特定のコンポネントのイベントに設定することで、ユーザ操作に対応した処理を記述することができる。また、「アクション」と呼ばれる処理の内容は Java コードによって記述する。

詳細なアクション記述方法については 3.2 章に記述する。

### 1.2.4 ライフサイクルの記述

本フレームワークでは、アプリケーションの起動や終了などのライフサイクルイベントを検出し、イベントに応じた処理を記述することが可能である。

本フレームワークで対応しているライフサイクルイベントは以下の 4 項目である。

・アプリケーションの起動
・アプリケーションの終了
・ウィンドウのオープン
・ウィンドウのクローズ

詳細なライフサイクル処理の記述方法は 3.1 章に記述する。

### 1.2.5 その他の機能

#### (1)外部コンポネントの取込み

本フレームワークでは基本的な画面コンポネント（ボタンやテキストフィールドなど）はデフォルトで提供している。さらに、デフォルト提供の画面コンポネントに含まれないコンポネントについても、容易にフレームワーク内に取り込むことが可能となるような仕組みを備えている。

そのため、アプリケーション開発者が独自に開発したコンポネントなどもフ

レームワーク内に取り込むことが容易になっている。

詳細な取り込みの手法については 3.3 章に記述する。

**(2) ツールキットの変更**

本フレームワークではデフォルトの GUI ツールキットとして Swing をサポートしているが、プラグインを用いることでその他の GUI ツールキット（Qt、SWT など）上でアプリケーションを動作させることも可能である。

## 1.3 フレームワーク構成

本フレームワークはフレームワークのコア機能を提供する XDF コアフレームワークおよび XDF で利用されるコンポネントの実装を提供する XDF コンポネントプラグインの二つによって成り立っている。

それぞれのフレームワーク要素の関連およびフレームワークの動作を図 1 に示す。

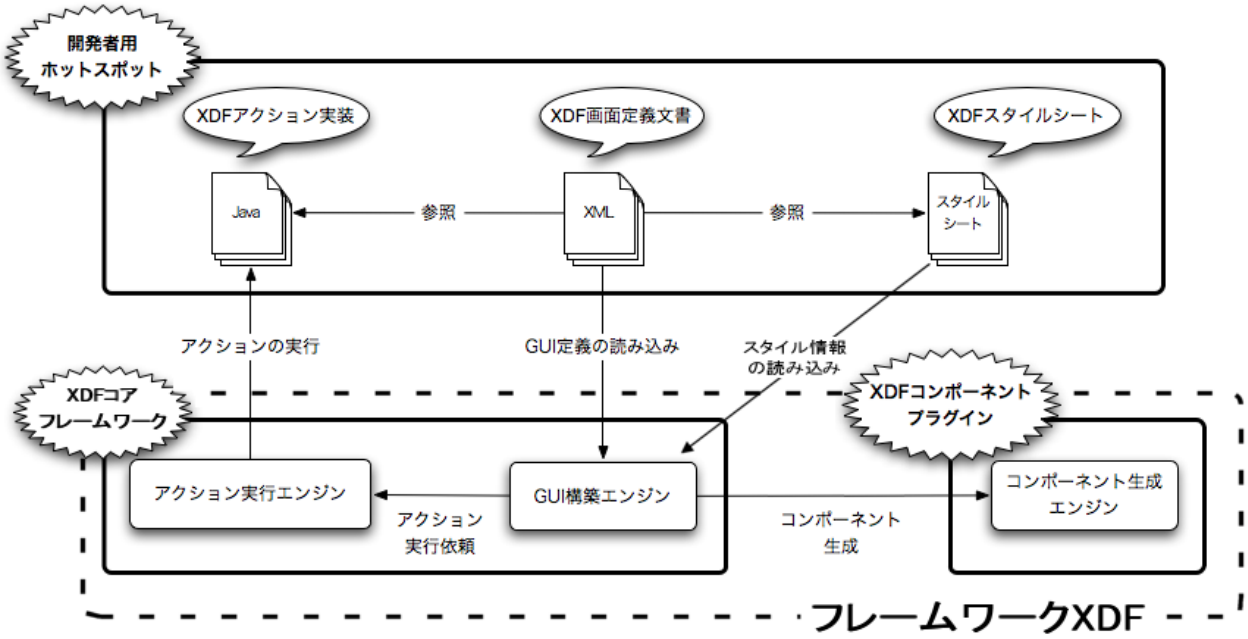



*図 1：XDF 動作図*

### 1.3.1 開発者用ホットスポット

1.2 でも述べたように、本フレームワークを利用する開発者が利用すべき重要なフ

レームワークのホットスポットとして以下の 3 点が挙げられる。

・XDF 画面定義文書
・XDF スタイルシート
・XDF アクション実装

本フレームワークは上記 3 点のホットスポットに設定された情報をもとに GUI の構築やアクションの設定／呼び出しなどを行う。

### 1.3.2 XDF コア

XDF コアは GUI 構築エンジンとアクション実行エンジンから構成される本フレームワークのコア機能を提供するものである。

GUI 構築エンジンは、XDF 画面定義文書に指定された設定に基づき、XDF コンポネントプラグインを介して GUI コンポネントの生成を行う役割を持つ。また、それとともに、指定されたイベントに対するアクションの実行が画面定義文書による指定通りに動作するように、アクション実行エンジンに対してアクションの実行を命じる役割を持つ。

アクション実行エンジンは GUI 構築エンジンによって命じられたアクションを実行するとともにそのアクションに対して発生したイベントに関する情報を紐づける役割を持つ。

### 1.3.3 XDF コンポネントプラグイン

XDF コンポネントプラグインは、XDF コアによって要求されたコンポネントを GUI プラットフォームが提供する実コンポネントによって実現する役割を持つ。

また、この XDF コンポネントプラグインを変更することによって、異なる GUI ツールキットへの対応が可能になる。

# 2. アプリケーション開発手順

本章では XDF を用いた GUI アプリケーションの開発についてその手順を記述する。

## 2.1 XDF フレームワークの内容

配布される XDF フレームワークの内容およびディレクトリ構成を図 2 に示す。

```
xdf
  ├ README.txt
  ├ xdfdemo090210.zip（XDF のデモ環境）
  ├ lib（本フレームワークにおける全ての jar を含む）
  |   └ xdf.jar
  ├ deps（本フレームワークが使用する API 群）
  |   ├ logback-classic-0.9.13.jar
  |   ├ logback-core-0.9.13.jar
  |   └ slf4j-api-1.5.6.jar
  ├ docs（本フレームワークに関するドキュメント類）
  |   ├ XDF UserGuide.pdf
  |   ├ xdf-samples.zip
  |   └ xdf-javadoc
  └ src(.zip)（本フレームワークのソースコード一式）
```

*図 2:XDF フレームワークの構成*

## 2.2 アプリケーション開発例

本章では、XDF を用いたアプリケーション開発について、画面定義文書およびスタイルシートを利用してアプリケーションの GUI 構築を行い、その GUI 上でアクションを動作させるまでの手順を示す。

### 目的アプリケーションについて

下記のような「エコーアプリ」の開発を例にして、本フレームワークによるアプリケーション開発の方法を示す。

#### (1)外観

開発例とする「エコーアプリ」は図 3 のような外観のアプリケーションである。「エコーアプリ」と標題されたウィンドウ内に、「エコー文字列」との標記の右にテキストフィールド(以下「エコー文字列欄」)、その右に「エコー」と記されたボタン(以下「エコーボタン」)、下段「今までのエコー」との標記の下にテキストエリア(以下「今までのエコー欄」)を有する。

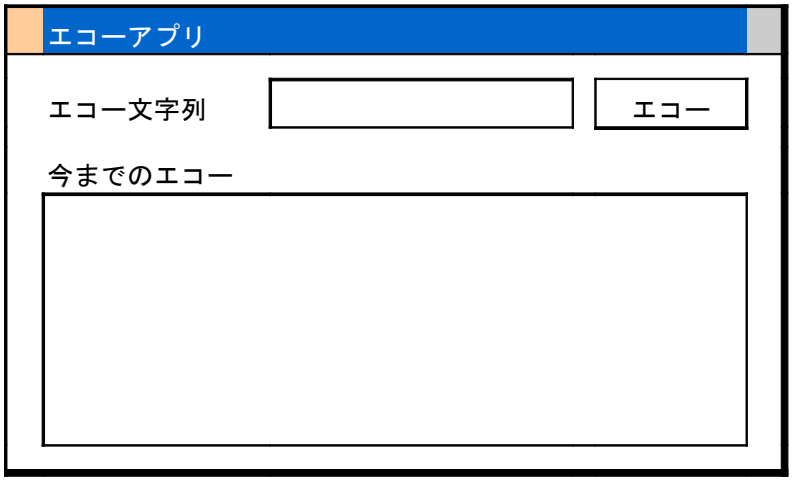



*図 3:エコーアプリ外観*

**(2)動作**

図 3 のアプリケーションの動作は、エコーボタンを押したときに、エコー文字列欄に指定された文字列が今までのエコー欄に表示されるものである。

**(3)ファイル構成**

上記エコーアプリを動作させるために必要となるファイル（以下、2.2.2 章、2.2.3 章、2.2.4 章で作成する）について、その構成を図 4 に示す。

```
echo-app
  ├ sample.echoapp
  │   ├ Main.java
  │   └ EchoAction.java
  ├ echo-app.xml
  └ echo-app.style
```

*図 4:エコーアプリの構成*

**(4)作成手順の概要**

上記エコーアプリの作成手順は概ね下記の通りである。

・画面定義文書作成
・スタイルシート作成
・アクション作成

### 2.2.1 画面定義文書の記述

開発例であるエコーアプリを開発するにあたって、まず、主に外観の論理的構造(配置物、順序等)に関する定義を画面定義文書に記述する。図4中のecho-app.xmlがこれに該当する。

本例における画面定義文書echo-app.xmlの内容を図5に示す。図5中に用いられるXMLタグおよびそのタグ属性が意味する内容を3.1.1章、表1ー4に示す。

```
<xdf-window>
   <frame title="エコーアプリ">
      <group class="echoInput">
         <label text="エコー文字列" />
         <textfield />
         <button text="エコー" />
      </group>
      <label text="今までのエコー" />
      <textarea id="result" />
   </frame>
</xdf-window>
```

*図 5:エコーアプリの構成（echo-app.xml）*

また、アプリケーション例であるエコーアプリを実行するためのソースコードを図6に示す。図4中のMain.javaがこれに該当する。

図6で使用されているメソッドの意味・動作については添付Javadocに示す。

```
package sample.echoapp;
import java.io.FileNotFoundException;
import jp.ac.aiit.xdf.application.Application;
import jp.ac.aiit.xdf.component.swing.SwingTagReferences;
public class Main {
    public static void main(String[] args)
                throws FileNotFoundException {
        //アプリケーションインスタンスの取得
        Application app
            = Application.getInstance(new SwingTagReference());
        //ウィンドウ名をmainとして、echo-app.xmlを登録
        app.registWindow("main", "echo-app.xml");

        //ウィンドウ名がmainのウィンドウを表示
        app.open("main");
 }
}
```

*図 6:エコーアプリ起動クラス（Main.java）*

この時点で Main.java を起動してアプリケーションを実行すると以下の図 7 のような画面が表示される。このとき表示される画面は図 3 に示した外観とは大きく異なっている。この外観を図 3 のものと一致させるための手順を次章 2.2.3 に示す。


*図 7:エコーアプリ画面（画面定義のみ）*

### 2.2.2 スタイルシートの記述

2.2.2 で作成した GUI に対する詳細なレイアウトを指定するためのスタイルシートを図 8 に示す。このファイルが図 4 に示したファイルの echo-app.style となる。図 8 の記述方法に関する詳細は 3.1.2 章に示す。

```
//echoInput クラスの group タグの中で、
//label 要素と textfield 要素の次の要素は 10 ポイント空けて右に配置させる
group.echoInput label, group.echoInput textfield {
    follows: right; margin-right: 10;
}

//echoInput クラスの group タグの中で、
//textfield の幅を 240px、高さは 24px に指定
group.echoInput textfield {
    width:240; height: 24;
}

//result という id の textarea の幅は 240px で、
//10 行分のテキストを記述できるように指定
textarea#result {
    width:240; rows: 10;
    margin-bottom: 5;
}
```

*図 8:エコーアプリ用スタイルシート（echo-app.style）*

また、このスタイルシートを GUI に適用するには、そのスタイルシートを参照するという指定を画面定義文書内に記述する必要がある。そのため、図 8 の echo-app.style を図 5 の echo-app.xml に適用するように echo-app.xml の修正（図 9）を行う。

具体的な修正内容としては、head タグを設けその中にスタイルシートを指定する style タグを設ける。そして、その style タグの src 属性に対して参照するスタイルシート echo-app.style の指定を行う。

```
<xdf-window>
   <head>
      <style src="echo-app.style" />
   </head>
   <frame title="エコーアプリ">
      <group class="echoInput">
         <label text="エコー文字列" />
         <textfield />
         <button text="エコー" />
      </group>
      <label text="今までのエコー" />
      <textarea id="result" />
   </frame>
</xdf-window>
```

*図 9:修正済み画面定義文書（echo-app.xml）*

そして、この時点で Main.java を実行すると、図 10 のような GUI が表示される。以上で XDF を用いた GUI 構築は終了であり、次節 2.2.3 にはこのアプリケーションを実際に動作させるためのアクション定義などの方法を説明する。

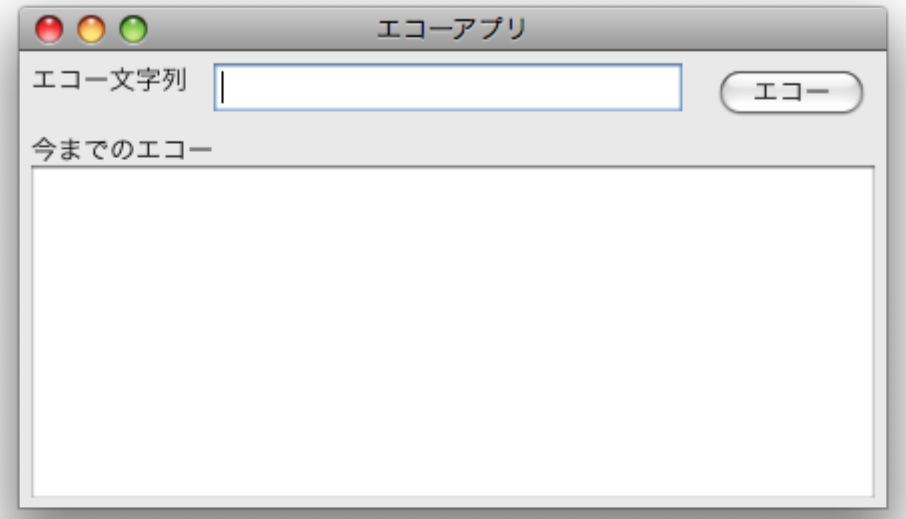



*図 10:エコーアプリ画面（スタイルシート適用済み）*

ところで、図 10 のエコーアプリ画面を、スタイルシートを用いることなく画面定義文書のみを用いて表示させるには、画面定義文書の内容を図 11 のようにする。

```
<xdf-window>
   <frame title="エコーアプリ">
      <group class="echoInput">
         <label text="エコー文字列"
            follows="right" margin-right="10" />
         <textfield
            follows="right" margin-right="10"
            width="240" height="24" />
         <button text="エコー" />
      </group>
      <label text="今までのエコー" />
      <textarea id="result"
         width="240" rows="10"
         margin-bottom="5" />
   </frame>
</xdf-window>
```

*図 11:スタイルシートを用いない画面定義文書*

### 2.2.3 アクション記述

本章において、2.2.3 章までで作成したエコーアプリの GUI に対して、実際にボタ

ンを押したときにエコーの表示を行うアクションの記述を追加する。

### (1) アクションクラスの実装

まず、エコー表示用のアクションを jp.ac.aiit.xdf.core.action.Action を実装するクラス（「アクションクラス」と呼ぶ)として作成し、そのクラスにアクションの具体的内容を記述した execute メソッド を実装する。エコーアプリにおけるエコー表示アクションの実装を図 12 に示す。このファイルが図 4 における EchoAction.java の内容である。

```
package sample.echoapp;
import java.util.List;
import jp.ac.aiit.xdf.application.Application;
import jp.ac.aiit.xdf.core.action.Action;
import jp.ac.aiit.xdf.core.action.Event;
import jp.ac.aiit.xdf.core.model.ObjectModel;
import jp.ac.aiit.xdf.core.selector.Selector;

public class EchoAction implements Action {

  @Override
  public void execute(Event e) {
    Application app = Application.getInstance();
    ObjectModel window = app.getWindow("main");

    System.out.println("ECHO");

    //エコー入力用テキストフィールドを探す
    List<ObjectModel> models =
      Selector.find("group.echoInput textfield", window);
    //結果出力用テキストエリアを探す
    ObjectModel result =
      Selector.find("#result", window).get(0);

    for(ObjectModel m : models) {
      //テキストエリア内テキストの最後にエコーテキストをつける
      result.attr("value"
        , ((String) result.attr("value"))
          + m.attr("value") + "¥n");
      //エコー入力用テキストフィールドをクリア
      m.attr("value", "");
    }
  }
}
```

*図 12:エコー表示用アクション（EchoAction.java）*

### (2) アクション実行のための XML 編集

作成した GUI 上で図 12 のアクションクラスを動作させるには、そのアクションクラスを利用するという指定を画面定義文書に記述する必要がある。そのため、図 12 のアクションが GUI 上で動作するように図 9 の画面定義文書を変更したものが以下の図 13 の画面定義文書である。

ここでの具体的な変更内容は head 領域に利用アクションクラスを指定する action タグを設けることである（action タグについての詳細は後述する）。

```
<xdf-window>
  <head>
    <style src="echo-app.style" />
    <action target="group.echoInput button" event="click"
        classname="sample.echoapp.EchoAction" />
  </head>
  <frame title="エコーアプリ">
    <group class="echoInput">
      <label text="エコー文字列" />
      <textfield />
      <button text="エコー" />
    </group>
    <label text="今までのエコー" />
    <textarea id="result" />
  </frame>
</xdf-window>
```

*図 13:修正済み画面定義文書（echo-app.xml）*

### (3) 動作確認

以上で、サンプルアプリケーション「エコーアプリ」の開発は完了である。実際に完成した「エコーアプリ」を実行すると図 14 の画面が現れる。

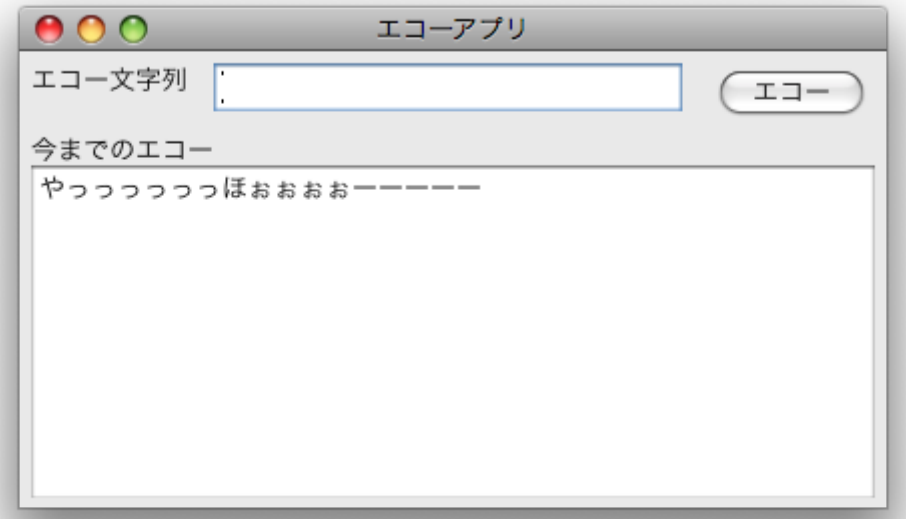



*図 14:エコーアプリ画面（完成）*

# 3. 各機能の詳細

## 3.1 GUI 構築機能

ここでは、本フレームワークを用いたアプリケーションの GUI 構築の為に利用する画面定義文書およびスタイルシートについての機能詳細を記述する。

### 3.1.1 画面定義文書

本フレームワークの画面定義文書は専用のスキーマを持った XML（eXtensible Markup Language）フォーマットのテキストファイルによって記述される。ファイルの基本的な記述方法（タグおよびその属性）については XML 標準の記法に準ずる。

また、本フレームワークにおける画面定義文書はひとつの定義ファイルが実ウィンドウと一対一で対応する関係にある。

画面定義文書の基本構成は図 15 の通りである。

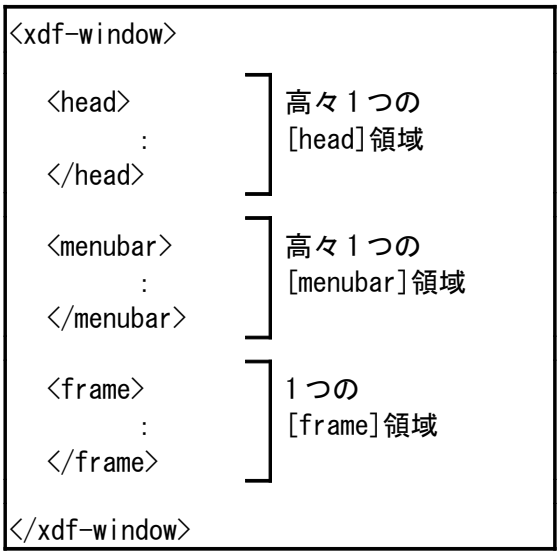



*図 15:画面構成文書の基本構成*

また、図 15 中の[head]領域、[menubar]領域、[frame]領域内で利用可能なタグ(head 領域タグ、menubar 領域タグ、frame 領域タグ)に関する情報を以下の表に示す。

#### (1) head 領域タグ

head タグの開始タグ(<head>)に始まり閉じタグ(</head>)に終わる記述領域を head 領域と呼ぶ。head 領域には、その画面定義が参照するスタイルシート、アクション定義等を記述する。head 領域内に含むことのできるタグ(以下

「head 領域タグ」)および各 head 領域タグに関する属性を表 1 に示す。

*表 1:head 領域タグ一覧*

| タグ文字列 | タグの説明 | |
|---|---|---|
| | 属性 | 属性の説明 |
| style | 参照するスタイルシートに関する設定。任意個指定可能。 | |
| | src | 参照するスタイルシートのファイルを classpath ルートからの相対パスによって指定する。 |
| action | アクションを特定のコンポネントに発生したイベントに結びつける設定を行う。任意個指定可能。 | |
| | classname | 利用するアクションクラスを FQCN によって表す。 |
| | target | アクションを適用する対象コンポネントをセレクタ表現で指定する。 |
| | event | アクションを適用するイベントを指定する。指定可能な値とイベントの関係について表 7 及び表 8 に示す。 |

(head 領域タグ記述例)

```
<xdf-window>
    <head>
        <style src="echo-app.style" />
        <action target="group.echoInput button" event="click"
            classname="sample.echoapp.EchoAction" />
    </head>
    <frame title="エコーアプリ">
        :
    </frame>
</xdf-window>
```

### (2) menubar 領域タグ

ウィンドウ端部などに置かれる [ファイル(F)], [表示(V)], ‥ といった「メニュー」の内容構成を、menubar タグの開始タグ(<menubar>)に始まり閉じタグ(</menubar>)に終わる記述領域(以下「menubar 領域」)に記述する。

menubar 領域内に含むことのできるタグ(以下「menubar 領域タグ」)および各 menubar 領域タグに関する属性を表 2 に示す。

*表 2:menubar 領域タグ一覧*

| タグ文字列 | タグの説明 | |
|---|---|---|
| | 属性 | 属性の説明 |
| menugroup | 複数個のメニュー項目をサブメニューとして保有するメニュー項目を表す。このタグの内部には menugroup タグおよび menuitem タグを指定することが可能である。 | |

| | | |
|---|---|---|
| | text | メニュー項目の表示名 |
| | image | メニュー項目に表示するアイコン |
| | access | ショートカットキーの指定 |
| menuitem | メニュー項目を表現する。このタグの内部にタグを記述することは不可能である。 | |
| | text | メニュー項目の表示名 |
| | image | メニュー項目に表示するアイコン |
| | access | ショートカットキーの指定 |

(menubar 領域タグ記述例)

```
<xdf-window>
        :
    <menubar>
        <menugroup text="ファイル(F)" access="F">
            <menuitem text="閉じる(C)" access="C">
            <menuitem text="印刷(P)" access="P">
            <menuitem text="終了(X)" access="X">
        </menugroup>
        <menugroup text="ヘルプ(H)" access="H">
            <menuitem text="バージョン情報(A)" access="A">
        </menugroup>
    </menubar>
    <frame title="エコーアプリ">
        :
    </frame>
</xdf-window>
```

### (3) frame 領域タグ

frame タグはアプリケーションにおける 1 つのウィンドウに該当し、frame タグの開始タグ(<frame>)に始まり閉じタグ(</frame>)に終わる記述領域(以下「frame 領域」)内に、ウィンドウにおけるコンポネント構成を記述する。

frame 領域内に含むことのできるタグ(以下「frame 領域タグ」)および各 frame 領域タグに関する属性を表 3 に示す。なお同表冒頭には、frame タグに関して記述する。また、レイアウトに関する属性等、各タグに亘って共通な属性について表 4 に示す。

*表 3:frame 領域タグ一覧*

| タグ文字列 | タグの説明 | |
|---|---|---|
| | 属性 | 属性の説明 |

| | | |
|---|---|---|
| frame | frame 領域のルートタグであり、ウィンドウを表現するタグ。<br>このタグの内部には、任意個の<br>・tabgroup タグ　・group タグ　・本表 button タグ以降に記すタグ<br>を任意の順で含みうる。 | |
| | title | ウィンドウのタイトルバーに表示する標題を表す。 |
| group | 描画するコンポネントの集まりを表現するコンテナを表す。<br>このタグの内部には、任意個の<br>・tabgroup タグ　・group タグ　・本表 button タグ以降に記すタグ<br>を任意の順で含みうる。 | |
| | border | コンテナの周囲を囲む境界線の種類を指定する。<br>指定可能な値と意味はそれぞれ以下の通り<br>solid：直線で周囲を囲む<br>groove：周囲を立体的に窪んだ線で囲む<br>ridge：周囲を立体的に隆起した線で囲む<br>inset：コンテナ全体が窪んだ様に表示する<br>outset：コンテナ全体が隆起した様に表示する |
| tabgroup | 任意個の「タブ」(次段「tab」タグ参照)を含んで構成されるタブ区画を表現するタグであり、ウィンドウ内の任意の描画領域をタブ構成にする場合に用いる。<br>このタグの内部には任意個の tab タグを含みうる。 | |
| | tabplace | タブ区画におけるタブの配置を表す。上、下、左、右への配置にそれぞれ対応する top, bottom, left, right の 4 種類の値から指定可能である。 |
| tab | タブ区画におけるタブの 1 つを表す。このタグの内部に記述されたタグがタブ内部に表示される画面構成となる。<br>このタグの内部には、任意個の<br>・tabgroup タグ　・group タグ　・本表 button タグ以降に記すタグ<br>を任意の順で含みうる。 | |
| | title | タブの「つまみ」部に表示されるタイトルを表す。 |
| | img | タブの「つまみ」部に表示されるアイコンイメージを表す。 |
| | access | タブへのショートカットキーを表す。<br>(例)<br>access="altR"　access="Alt-R"<br>access="alt-r"　access="r" |
| button | 画面中のボタンコンポネントを表現する。 | |
| | text | このボタンに表示される文字列を表す。 |
| | img | このボタンに表示されるアイコンイメージを表す。 |
| textfield | 1 行で構成される改行なしのテキスト入力フィールドを表す。 | |
| | value | フィールドに表示されるデフォルト値 |

| | | |
|---|---|---|
| | border | コンテナの周囲を囲む境界線の種類を指定する。<br>指定可能な値と意味はそれぞれ以下の通り<br>solid：直線で周囲を囲む<br>groove：周囲を立体的に窪んだ線で囲む<br>ridge：周囲を立体的に隆起した線で囲む<br>inset：コンテナ全体が窪んだ様に表示する<br>outset：コンテナ全体が隆起した様に表示する |
| | editable | このテキストフィールドが編集可能であるかを指定する。<br>指定可能な値は true 又は false であり、それぞれ編集可能、編集不可能であることを示す。 |
| textarea | 複数行で構成される改行を含んだテキスト入力エリアを表す。 | |
| | value | テキスト入力エリアに表示されるデフォルト値 |
| | rows | このテキスト入力エリアの高さを行数で指定する。 |
| | columns | このテキスト入力エリアの幅を行当り字数で指定する。 |
| | border | コンテナの周囲を囲む境界線の種類を指定する。<br>指定可能な値と意味はそれぞれ以下の通り<br>solid：直線で周囲を囲む<br>groove：周囲を立体的に窪んだ線で囲む<br>ridge：周囲を立体的に隆起した線で囲む<br>inset：コンテナ全体が窪んだ様に表示する<br>outset：コンテナ全体が隆起した様に表示する |
| | editable | このテキストフィールドが編集可能であるかを指定する。<br>指定可能な値は true 又は false であり、それぞれ編集可能、編集不可能であることを示す。 |
| label | 画面上のラベル要素を表す。 | |
| | text | ラベルに表示するタイトル |
| | img | ラベルに表示するアイコン |
| combobox | 複数項目から一つの項目を選択するためのプルダウンリストを表す。<br>選択肢となる値集合を values 属性で指定する。 | |
| | values | 複数の選択肢をカンマ区切りのテキストとして指定する。<br>(例)values="blue, red" |
| | selected | 現在選択されている値を表す。選択項目は選択肢のインデックス値によって指定する。 |
| checkbox | チェックボックスを表す。 | |
| | text | チェックボックスに表示するラベル文字列 |

| | | |
|---|---|---|
| | checked | そのチェックボックスが選択されているか(選択されている「true」／選択されていない「false」)<br>(例)checked="true" |
| radio | 択一用のラジオボタンを表す。name の一致する 1 つ以上のラジオボタンによって 1 つの選択肢グループが構成される。 | |
| | name | ラジオボタンのグループを識別するための名前 |
| | text | ラジオボタンに表示するラベル文字列 |
| | checked | そのラジオボタンが選択されているか(選択されている「true」／選択されていない「false」)<br>(例)checked="true" |
| list | 複数項目を表示するためのリストを表現する。リストに表示される要素を values 属性によって指定する。 | |
| | values | 複数の選択肢をカンマ区切りのテキストとして指定する。<br>(例)values="blue, red" |
| | selected | 現在選択されている値(値集合における位置)を表す。 |
| | border | コンテナの周囲を囲む境界線の種類を指定する。<br>指定可能な値と意味はそれぞれ以下の通り<br>solid：直線で周囲を囲む<br>groove：周囲を立体的に窪んだ線で囲む<br>ridge：周囲を立体的に隆起した線で囲む<br>inset：コンテナ全体が窪んだ様に表示する<br>outset：コンテナ全体が隆起した様に表示する |
| treetable | ツリーテーブルコンポネントを表現する | |
| | model | TreetableModel インタフェースを実装したクラスを FQCN で指定する。 |

レイアウトに関するもの等、下記の各属性については、多くのタグに亘って共通的に指定可能である。

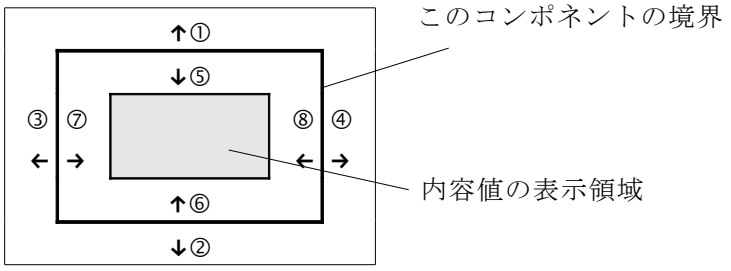
*表 4:frame 領域タグの共通属性一覧*

| 属性 | 説明 | 属性値(使用例) |
|---|---|---|
| id | 要素を一意に識別する名称を表す。 | 自由文字列 |
| class | 同一の特徴を持った要素グループを定義する | 自由文字列 |
| follows | 後続する要素の配置方向を表す。 | bottom ： 次の要素を下に配置する<br>right ： 次の要素を右に配置する<br>(例)follows="right" |

| | | |
|---|---|---|
| width | 幅を固定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)<br>width="100" 又は width="100px" |
| min-width | 最小幅を指定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)<br>min-width="60"<br>max-width="160px"<br>pref-width="100" |
| max-width | 最大幅を指定する | |
| pref-width | 推奨幅を指定する | |
| height | 高さを固定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)height="100px" |
| min-height | 最小高さを指定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)<br>min-height="16px"<br>max-height="48"<br>pref-height="24px" |
| max-height | 最大高さを指定する | |
| pref-height | 推奨高さを指定する | |
| margin | 上下左右のマージン(隣接コンポネントとの間隔)に関する指定を行う | 最大 4 個のピクセル値（数字のみ又は px の単位付き）<br>1 個目のピクセル値：margin-top の値<br>2 個目のピクセル値：margin-right の値<br>3 個目のピクセル値：margin-bottom の値<br>4 個目のピクセル値：margin-left の値<br>(例)<br>margin="10 20 10 20" 又は<br>margin="10px 20px 10px 20px" |
| margin-left | 左マージンを指定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)<br>margin-left="5px"<br>margin-right="15"<br>margin-top="25"<br>margin-bottom="20px" |
| margin-right | 右マージンを指定する | |
| margin-top | 上マージンを指定する | |
| margin-bottom | 下マージンを指定する | |

| | | |
|---|---|---|
| padding | 上下左右のパディング(コンポネント内部の余白)に関する指定を行う | 最大 4 個のピクセル値（数字のみ又は px の単位付き）<br>1 個目のピクセル値：padding-top の値<br>2 個目のピクセル値：padding-right の値<br>3 個目のピクセル値：padding-bottom の値<br>4 個目のピクセル値：padding-left の値<br>(例)<br>padding="3 5 3 5" |
| padding-left | 左パディングを指定する | ピクセル値（数字のみ又は px の単位付き）<br>(例)<br>padding-left="0px"<br>padding-right="2"<br>padding-top="3"<br>padding-bottom="2px" |
| padding-right | 右パディングを指定する | |
| padding-top | 上パディングを指定する | |
| padding-bottom | 下パディングを指定する | |
| tooltip | ツールチップの文字列 | 自由文字列 |
| bgcolor | 背景色を指定する | 16 進カラーコード<br>(例)<br>bgcolor="#F0F8FF"<br>fgcolor="#228B22" |
| fgcolor | 前景色を指定する | |
| font | 使用するフォントを指定する | フォント名、ポイント数、スタイルをそれぞれカンマ区切りで記述する。ポイント数、スタイルは省略可。<br>フォント名：フォント識別文字列（Java API を参照のこと）<br>ポイント数：数字+"pt"<br>スタイル：italic、bold、normal のどれか<br>(例)<br>font="MS 明朝,12pt,bold"<br>font="Lucida Console,16pt"<br>font="Courier New" |

補足説明：マージンとパディング




①margin-top ②margin-bottom ③margin-left ④margin-right
⑤padding-top ⑥padding-bottom ⑦padding-left ⑧padding-right

(frame 領域タグ記述例)

```
<xdf-window>
        :
   <frame title="エコーアプリ">
      <group class="echoInput">
         <label text="エコー文字列"
            follows="right" margin-right="10" />
         <textfield
            follows="right" margin-right="10"
            width="240" height="24" />
         <button text="エコー" />
      </group>
      <label text="今までのエコー" />
      <textarea id="result"
         width="240" rows="10"
         margin-bottom="5" />
   </frame>
</xdf-window>
```

### 3.1.2 スタイルシート

本フレームワークにおけるスタイルシートとは、画面定義文書中で所定の条件に該当する一群の XML 要素に対して XML 属性を一様に適用するために利用するものである。画面定義文書との関係を図 16 に示す。

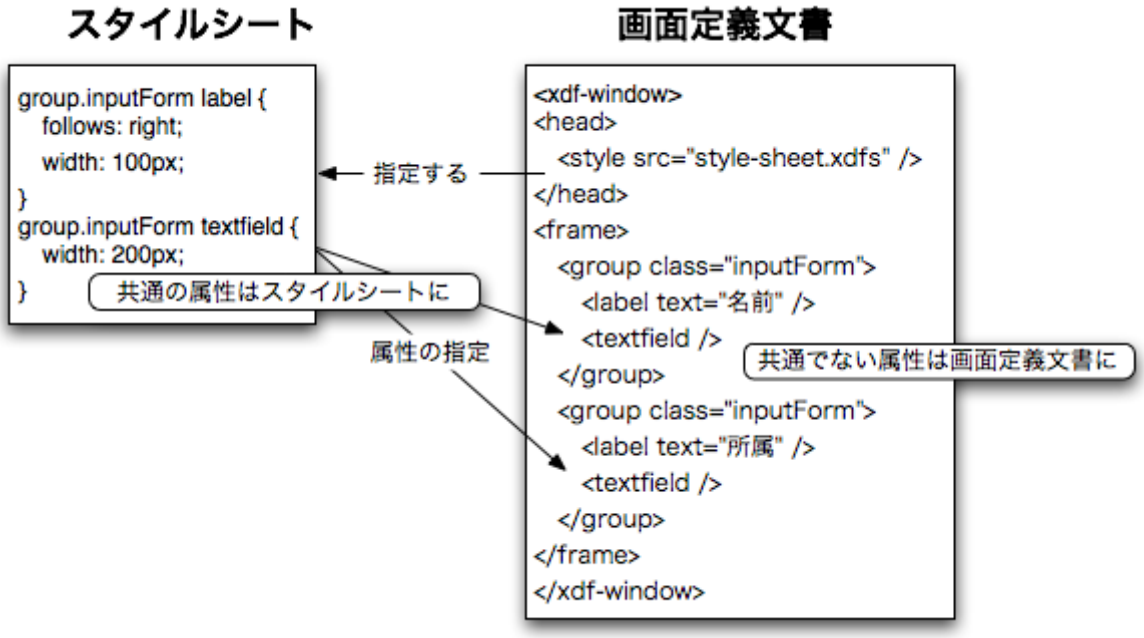



*図 16:画面定義文書とスタイルシートの関連*

このスタイルシートの構文を BNF 状の記法で示すと図 17 のようになる。

```
<空白文字> ::= 半角スペース | 改行 | タブ                                    [1]
<スタイルシート> ::= <スタイルユニット>*                                      [2]
<スタイルユニット>
  ::= <セレクタ> '{' <属性ユニット>[';' <属性ユニット>]* ';'? '}'            [3]
<セレクタ>
  ::= <ルールシーケンス>[(半角スペース)<ルールシーケンス>]*                   [4]
<ルールシーケンス> ::= <TAG ルール>?(<ID ルール> | <CLASS ルール>)*          [5]
<ルールテキスト> ::= ^((空白文字) | [,.#;{}])*                             [6]
<TAG ルール> ::= <ルールテキスト>                                           [7]
<ID ルール> ::= '#'<ルールテキスト>                                         [8]
<CLASS ルール> ::= '.'<ルールテキスト>                                      [9]
<属性ユニット> ::= <属性名> ':' <属性値>                                    [10]
<属性名> ::= ^((空白文字) | [.#;}])*                                       [11]
<属性値> ::= ^[;}]*                                                        [12]
```

*図 17:スタイルシートの構文*

【補足説明】

[1] 半角スペース、改行及びタブを<空白文字>とする。

[2] <スタイルシート>は<スタイルユニット>を全く含まなくてもよい。

[3] <スタイルユニット>の{}内最後の<属性ユニット>は、後に'}'を伴わなくてもよい。

[4] <セレクタ>は、半角スペースを挟んで<ルールシーケンス>を並べたもの。

[5] <ルールシーケンス>は、<空白文字>を挟むことなく下記定義の<ルール>を連結したものとも言い換えられる。

<ルール> ::= <TAG ルール> | <ID ルール> | <CLASS ルール>
<ルールシーケンス> ::= <ルール>+

ただし、<ルール>の後の<ルール>として<TAG ルール>を置くことはできない。

[6] <ルールテキスト>は、(空白文字)「.」「#」「;」「{」「}」以外で構成される文字列とする。

[11]<属性名>は、(空白文字)「.」「#」「;」「}」以外で構成される文字列とする。

[12]<属性値>は、「;」「}」以外で構成される文字列とする。

※ <スタイルユニット>,<セレクタ>,<ルール語>,<属性ユニット>,<属性名>,<属性値>の前後には任意個の<空白文字>を置いてよい。

### (1)セレクタ

本フレームワークにおいてセレクタとは、画面定義文書中の XML 要素をその構造によって特定するための構文および検索機能をさす。スタイルシートによる属性設定やアクションの指定などを行う際は、このセレクタにより特定される XML 要素に対して行う。

セレクタは XML タグ名、ID 属性値、タグの CLASS 属性値の 3 種類の指定を組み合わせて XML 要素を指定する(図 17[4][5]参照)。この 3 種類の指定のことをルールと呼び(図 17[5]の欄外補足参照)、タグ名についての指定を TAG ルー

ル、ID 属性値についての指定を ID ルール、CLASS 属性値についての指定を CLASS ルールと呼ぶ。

また、ルールを組み合わせる方法として、AND と SEQUENCE の 2 種類の方法を提供している。

・AND

空白文字を挟むことなく各ルールを連接させたものを指す(図 17[5]の欄外補足第 2 式参照)。指定した複数ルールのすべてを満たす XML 要素を取得する。

[例] `textfield#result.left`

id が"result"であり、かつ class 属性に"left"が指定されている textfield を探す。

・SEQUENCE

AND 結合されたルール又は単独のルール(<ルールシーケンス>)を、1 つ以上の半角スペースを挟んで並べたものを指す(図 17[4]の欄外補足参照)。指定された複数のルールに対して、前のルールに一致した XML 要素の中で次のルールに一致する XML 要素を取得するように動作する。

[例] `group.left button`

class 属性に"left"が指定された group を探した結果、得られた XML 要素の中（その要素をルートとしたツリー上）で button タグを探す。

### (2)属性ユニット

図 17 中の属性ユニットにおいて指定可能な属性名および属性値については、3.1.1 画面定義文書に詳述する通りである。

### (3) スタイルシートの記述例

図 18 の画面定義文書に対して、図 19 のようなスタイルシートを記述した場合の動作について説明する。

```
<xdf-window>
<head>
   <style src="sample-stylesheet.xdfs" />
</head>
<frame>
   <group id="inputs">
      <group id="searchPathInput">
         <label text="サーチパス:" />
         <textfield />
      </group>
      <group id="keywordInput">
```

```
        <label text="キーワード:" />
        <textfield />
      </group>
   </group>
   <button id="execute" />
</frame>
</xdf-window>
```

*図 18:サンプル画面定義文書*

```
//スタイルシート利用例
group#inputs group {
   follows: right;
}
```

*図 19:サンプルスタイルシート 1*

図 19 の例においてはセレクタを group#inputs group と指定しているため、group タグでかつ inputs の ID を持つ XML 要素の下の group タグすべてに対して follows 属性を right に設定するという意味となる。この例のセレクタによって特定される XML 要素を、図 18 中に太字で示した。

この図 18 に示すスタイルシートを用いる場合、アプリケーションは図 20 のような画面を表示することとなる。

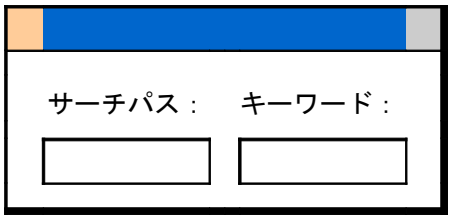



*図 20:サンプル画面 1*

一方、この例におけるスタイルシートを図 21 のように改めた場合は、図 22 のような画面が表示されることとなる。

```
//スタイルシート利用例
group#inputs group label{
   follows: right;
}
```

図 *21:*サンプルスタイルシート *2*

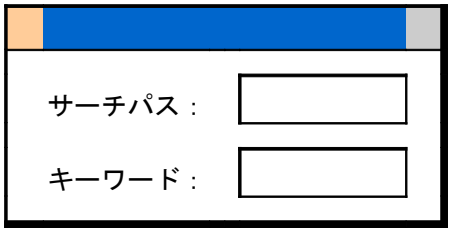



*図 22:サンプル画面 2*

## 3.2 アクション設定

ここでは、本フレームワーク中におけるアクションの実装方法およびそのアクションを特定のイベントに結びつける方法について記述する。

### 3.2.1 アクションの実装

本フレームワーク上で動作するアクションを作成する為には、XDF 用のアクション基底インタフェース `jp.ac.aiit.xdf.core.action.Action` を実装したアクションクラスを作成する必要がある。

実装すべき基底インタフェースの内容を図 23 に示す。

```
package jp.ac.aiit.xdf.core.action;

/**
 * @author Pin Yuan, Shunichi Takagi
 *
 * アプリケーション開発者がアクションを実装するためのインタフェース定義
 * （あるイベントに対応づけられた）アプリケーション内で動作するアクション
を実装する場合はこのインタフェースを implements する必要がある。
 */
public interface Action {
    public void execute(Event e);
}
```

*図 23:アクション基底インタフェース*

Action インタフェースの execute メソッドには、そのアクションの起動イベントに関する情報が渡される。Event から取得可能な情報を表 5 に示す。また、イベントに応じて取得可能なパラメタの詳細を表 6 に示す。

*表 5:Event から取得できる情報*

| 情報 | 説明 | 取得メソッド |
|---|---|---|
| モデル | イベントが発生したコンポーネントを表す。 | ObjectModel Event#getObjectModel(); |
| イベントタイプ | 発生したイベントの種類を表す。返り値の文字列は action タグで指定した event 属性値の文字列である。 | String Event#getEventType(); |
| パラメタ | 発生したイベントに関するパラメタ情報。（詳細は表 6） | Object Event#getParam(); |

*表 6:イベントに関するパラメタ情報*

| イベントタイプ | パラメタの内容 |
|---|---|
| click | そのイベント(click)に関する座標等の情報をあらわす Dimension。なお、現時点では未実装である。 |
| press-enter | そのイベント(press-enter)に関するキーコード等の情報をあらわす。なお、現時点では未実装である。 |
| | |

ここで、Event からそのアクションが実行されるために発生したイベントの種類（イベントタイプ）、イベントが発生したコンポーネントおよびパラメタ情報が取得できるようになっている。

これは action タグの target 属性に対してセレクタを用いることで一つのアクションを複数の XML 要素に対応づけることができ、また、イベント属性に複数のイベントタイプを指定できるため、実際にそのアクションが呼び出されるに至ったコンポーネントとイベントを見分けるために用意されているものである。

ただし、表 5 に示した Event から取得できる情報については、これを利用することによってそのアクションが特定のコンポーネントとイベントに依存してしまうため、利用は推奨しない。

### 3.2.2 イベントへの結びつけ

3.2.1 章で説明した通りにアクションを実装した後、そのアクションを特定のイベントが発生したときに起動させるように設定する方法を記述する。

本フレームワークにおいて、作成されたアクションをイベントに結びつけるには画面定義文書中の head 領域において action タグを用いて設定する。action タグに関

する詳細は表 1 を参照のこと。

また、action タグ中でイベントタイプを指定する event 属性に指定可能な値について、すべてのコンポネントに共通で指定可能なイベントを表 7 に示し、コンポネント毎に指定可能なイベントを表 8 に示す。

*表 7:コンポネント共通イベント*

| イベント名 | イベントの説明 |
|---|---|
| click | マウスでクリックした際に起動するイベント |
| doubleclick | マウスでダブルクリックした際に起動するイベント |
| focus | コンポネントにフォーカスがあたったときに起動するイベント |
| focuslose | コンポネントがフォーカスを失ったときに起動するイベント |

*表 8:コンポネント別イベント*

| 対象コンポネント | イベント名 | イベントの説明 |
|---|---|---|
| textfield | edit | 編集時にテキストが変化するたびに発生するイベント |
| | press-enter | テキストフィールドにフォーカスがあたっている状態でエンターキーを押したときに発生するイベント |
| textarea | edit | 編集時にテキストが変化するたびに発生するイベント |
| combobox | change | コンボボックスの選択状態が変化したときに発生するイベント |
| checkbox | check | 対象のチェックボックスが選択されたときに発生するイベント |
| | uncheck | 対象のチェックボックスの選択が解除されたときに発生するイベント |
| radio | check | 対象のラジオボタンが選択されたときに発生するイベント |
| | uncheck | 対象のラジオボタンの選択が解除されたときに発生するイベント |
| list | change | リスト上の選択項目が変化したときに発生するイベント |

### 3.2.3 アクション定義例

本フレームワークのアクション設定機能を用いたアクションの作成例を図 24 および図 25 に示す。

```
package sample;
import jp.ac.aiit.xdf.core.action.Action;
import jp.ac.aiit.xdf.core.action.Event;
import jp.ac.aiit.xdf.core.model.ObjectModel;
import jp.ac.aiit.xdf.core.selector.Selector;
public class HelloWorldAction implements Action {
  public void execute(Event e) {
    ObjectModel src = e.getObjectModel(); //イベント発生源(ボタン or 入力欄)のハンドル  [1]
    ObjectModel output //出力欄 textarea をハンドルする
      = Selector.find("textarea", src.parent()).get(0);                    [2]
    String type = e.getEventType(); //イベント種類                        [3]
    if( type.equals("click") ) { //ボタン押下の場合
      ObjectModel input //入力欄 textfield をハンドルする
        = Selector.find("textfield", src.parent()).get(0);
      output.attr("value" //出力欄の内容を書き換える
        , "Hello World " + input.attr("value") + "さん！");                [4]
    } else { //その他(press-enter)の場合
      output.attr("value", "(" + src.attr("value") + "さん)");             [5]
        }
    }
}
```

*図 24:アクション定義サンプル*

【補足説明】

[1] Event#getObjectModel() により、イベントが発生したコンポーネントの ObjectModel をハンドルしている(表 5 参照)。

[2] クラスメソッド Selector.find(String, ObjectModel) は、第 1 引数の文字列をセレクタ(3.1.2(1)「セレクタ」参照)として第 2 引数の ObjectModel 以下を検索する。ただしこのメソッドの戻り値は、セレクタに該当した全ての ObjectModel を表す List<ObjectModel>であるので、List#get(0)により 1 要素(この場合、ただ 1 つと判っているうえで先頭要素)をハンドルしている。

また、第 2 引数を src.parent()と与えているが、ObjectModel#parent() は、その ObjectModel の「親」要素(コンテナ要素の ObjectModel)を提供する。

[3] Event#getEventType() により、イベント種類を得ている(表 5 参照)。

[4] ObjectModel#attr(String) は、その ObjectModel(output)における、第 1 引数の名を持つ属性(「value」属性)の値を提供する。

[5] ObjectModel#attr(String, Object) は、その ObjectModel(output)における、第 1 引数の名を持つ属性(「value」属性)の値を、第 2 引数のものとする。

※ObjectModel の詳細な仕様については Javadoc を参照のこと。

```
<xdf-window>
<head>
   <style src="styles/hw.style" />
   <action target="#inputB" event="click"
      classname="sample.HelloWorldAction" />
   <action target="#inputF" event="press-enter"
      classname="sample.HelloWorldAction" />
</head>
<frame>
```

```
  <label /><textfield id="inputF" />
  <button id="inputB" text="Show HelloWorld!" />
  <textarea />
</frame>
</xdf-window>
```

【styles/hw.style の内容】

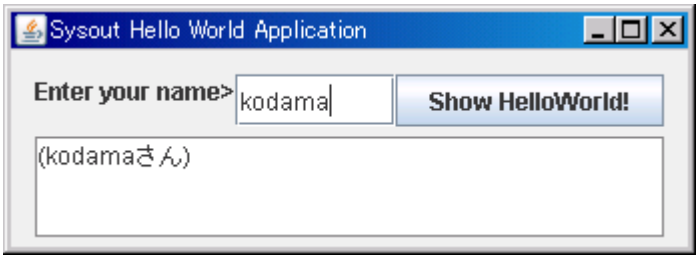
```
frame{title:Sysout Hello World Application;}
label{text:Enter your name>; follows:right;}
textfield{width:80; follows:right;}
textarea{
  columns:20; rows:3; border:solid;
  margin-top:5; margin-bottom:5;
}
```

*図 25:アクション設定サンプル*

この例の場合、動作は次のようになる。

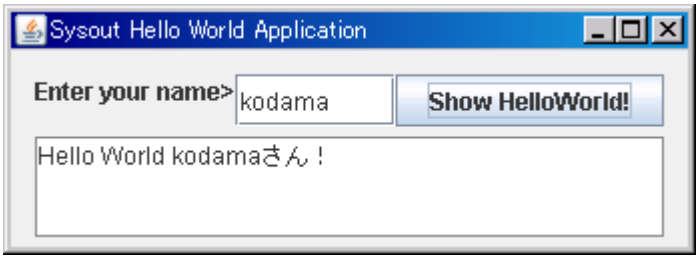
(a) テキストフィールド入力及び Enter キー押下 時

入力内容(「○○」とする)を用いて、出力欄に「(○○さん)」と表示される。




(b)「Show HelloWorld!」ボタン押下時

入力内容を用いて、出力欄に「Hello World ○○さん!」と表示される。



## 3.3 独自コンポネントの取り込み

本フレームワークは、アプリケーション開発者が独自に作成したコンポネント(独自コンポネント)をフレームワーク上で動作させるための仕組みを提供する。ここでは、その独自コンポネントを取り込むために必要な記述方法について示す。

### 3.3.1 独自タグの定義

アプリケーションに独自コンポネントを取り込むためには画面定義文書内でその独自コンポネントを参照／設定するためのタグが必要になる。そのため、独自コンポネントを取り込むにあたって、その独自コンポネントを表現するタグ(以下「独自タグ」)の表記を、図 26 に示す構造の定義ファイルによって定義する。

```
<tagdef>
    <tag name="[TAG_NAME]">
        <modelclass>[MODEL_CLASS]</modelclass>
        <component-mapper env="[TOOLKIT_ENV]"
            class="[COMPONENT_MAPPER_CLASS]" />
    </tag>
</tagdef>
```

*図 26:独自タグの定義ファイル構造*

上記定義ファイル構造中の[TAG_NAME]、[MODEL_CLASS]などに関する説明を以下、表 9 に示す。

*表 9:独自タグの定義要素*

| 要素名 | 説明 |
|---|---|
| [TAG_NAME] | 独自タグの名称、フレームワーク内でユニークな名称になる必要がある |
| [MODEL_CLASS] | 独自タグの属性情報を格納するモデルクラスを指定する。ここに指定されたクラスは jp.ac.aiit.xdf.core.model.ObjectModel インタフェースを実装する必要がある。 |
| [TOOLKIT_ENV] | その独自タグが動作するツールキット環境を示す文字列 |
| [COMPONENT_MAPPER_CLASS] | 独自タグをコンポネントにマップする役割を持つクラスを指定する。ここに指定されたクラスは jp.ac.aiit.xdf.component.ComponentMapper インタフェースを実装する必要がある。 |

また、上記の記述方式に従って作成したタグ定義ファイルをフレームワーク側に取り込むためには図 27 に示す jp.ac.aiit.xdf.application.TagReferences を実装したクラスを準備する。

```
package jp.ac.aiit.xdf.application;
import java.util.Map;
import jp.ac.aiit.xdf.core.tags.Tagdef.Tag;

public interface TagReferences {
    public String getReferenceId();
    public Map<String, Tag> getDefinedTags();
}
```

*図 27:タグ定義読み込み用インタフェース*

そして、実際にフレームワーク内に定義した独自タグを取り込ませるためには jp.ac.aiit.xdf.application.Application クラスのインスタンス取得メソッド Application#getInstance()メソッドの引数に上記図 27 のインタフェースを実装したクラスを指定する必要がある。

### 3.3.2 独自コンポネントのマッピング

本フレームワーク内で独自コンポネントを利用するためには上記 3.3.1 の独自タグ定義を行うとともに、XML 要素として設定された属性情報などを実際のコンポネントにマッピングする処理が必要になる。

そのマッピング処理は jp.ac.aiit.xdf.component.ComponentMapper インタフェースを実装したクラスを作成し、そのクラスに記述する。実装すべきインタフェースの内容を図 28 に示す。

```
package jp.ac.aiit.xdf.component;
import jp.ac.aiit.xdf.core.model.ObjectModel;

/**
 * @author Shunichi Takagi
 */
public interface ComponentMapper {
    public void setModel(ObjectModel model);
    public ObjectModel getModel();
    public void realize();
    public boolean isRealized();
    public Object getComponent();

    public Object typeConvert(String name, String value);
```

```
    public Object getAttr(String name);

    public void setRealize(boolean realize);
}
```

*図 28:ComponentMapper インタフェースの内容*

### 3.3.3 コンポネントの取り込み例

実際に XDF 基本コンポネントには含まれていない JSeparator を例にとって、コンポネントを取り込むために必要なタグ定義ファイルおよび ComponentMapper の実装をそれぞれ図 29、図 30 に示す。

```
package sample.mappers;
import jp.ac.aiit.xdf.component.swing.mappers.SwingComponentMapperTemplate;
public class JSeparatorMapper extends SwingComponentMapperTemplate {          [1]
    @Override
    protected Map<String, AttributeProcessingUnit> initProcessingUnits() {  [2]
        //共通属性　※線の長さなどはこれに含まれる。
        Map<String, AttributeProcessingUnit> result
            = CommonAttributeStore.commonAttributes();
        //セパレータ方向を表す「orientation」属性の定義
        result.put("orientation"
            , new AttributeProcessingUnit(
                new SetterAttributeProcessor("setOrientation", true)
                , new GetterAttributeProcessor("getOrientation")
                , new IntegerConverter()));
        return result;
    }
    @Override protected Object newComponent() {                                [3]
        return new JSeparator();
    }
    @Override protected void processingChildComponents() { }                   [4]
    @Override protected Map<String, EventHandler> intiProcessingAction() {  [5]
        return null;
    }
}
```

*図 29:JSeparator に対する ComponentMapper の実装*

【補足説明】

[1] Swing のコンポネントプラグインを拡張する場合は通常、インタフェース ComponentMapper を予め実装しているテンプレートクラスである SwingComponentMapperTemplate を利用して行う。

[2][3][4][5] これらは何れも SwingComponentMapperTemplate のホットスポットである。

※テンプレートクラス SwingComponentMapperTemplate の詳細については、ソース及び Javadoc を参照のこと。

```
<tagdef>
<!-- 区切り線に対応する新たなタグ「separator」 -->
    <tag name="separator">                                          [1]
        <modelclass>jp.ac.aiit.xdf.core.model.DefaultObjectModel    [2]
        </modelclass>
        <component-mapper env="swing"                               [3]
            class="sample.mappers.JSeparatorMapper" />              [4]
    </tag>
</tagdef>
```

*図 30:JSeparator 取り込み用タグ定義ファイル*

【補足説明】

[1] 画面定義でタグ文字列「separator」を用いる際の定義を開始している。

[2] modelclass タグの内側には、このタグに対応させる ObjectModel のクラスを FQCN で指定する。ただし殆どの場合、jp.ac.aiit.xdf.core.model.DefaultObjectModel としてよい。

[3] このタグと実コンポネントとをつなぐ ComponentMapper についての定義を開始し、コンポネントプラグインの識別名(この場合、Swing を表す「swing」)を指定している。

[4] コンポネントプラグインの ComponentMapper となるクラスを FQCN で指定する。

※DefaultObjectModel, ComponentMapper の詳細については、それらのソース及び Javadoc を参照のこと。