ものづくり・IT 融合化推進技術の研究開発

MZ Platform
コンポーネント開発ガイド

= Component Developer's Manual =

Revision 3.5 [MZ Platform.3.5]

＝目次＝

1. 概要 1

1.1. プラットフォームの概要 1
1.2. コンポーネントの構築 2
1.3. コンポーネント間の接続 2

2. MZ PLATFORM 3

3. コンポーネント共通機能 4

3.1. 共通データ構造 4
  3.1.1. データ構造クラス 4
  3.1.2. オブジェクトの複製（clone） 4
  3.1.3. オブジェクトの同一性（equals） 5
3.2. イベント 5
  3.2.1. イベント処理 5
  3.2.2. イベント一覧 6
  3.2.3. イベント関連クラス 7
3.3. 例外処理 11
  3.3.1. 例外一覧 11
  3.3.2. 各コンポーネントの例外処理 11
  3.3.3. エラーメッセージ表示 11
3.4. メッセージダイアログ 12
3.5. メモリ負荷状況 13
3.6. ログ出力 14
  3.6.1. ログ出力処理 14
  3.6.2. ログ内容 14
3.7. マルチロケール対応 15
  3.7.1. 固定文字列のマルチロケール対応 15
  3.7.2. データ文字列のマルチロケール対応 16
  3.7.3. コンポーネントのマルチロケール対応 16
3.8. シリアライズの互換性 17
  3.8.1. シリアライズデータの互換性 17
  3.8.2. オブジェクトのシリアライズ互換性確保 18
3.9. システムプロパティの利用 19

4. コンポーネントの開発概要 20

4.1. コンポーネント開発環境 20
4.2. コンポーネント開発の手順 21
4.3. サンプルコンポーネントの提供 23

5. コンポーネントの実装手順 24

5.1. 画面表示を伴うコンポーネントの開発 24
  5.1.1. 設計にあたって 24
  5.1.2. プログラム開発手順 25

5.1.3. 実装の留意点 26
5.2. 画面表示を伴わないコンポーネントの開発 28
5.2.1. 設計にあたって 28
5.2.2. プログラム開発手順 29
5.2.3. 実装の留意点 30
5.3. ネイティブ処理を行うコンポーネントの開発 31
5.3.1. 設計にあたって 31
5.3.2. プログラム開発手順 32
5.3.3. 実装の留意点 34
5.3.4. 共有ライブラリ生成時の留意点 35
5.4. XML 入出力機能の実装 36
5.4.1. XML 入出力インターフェイス 36
5.4.2. XML 入出力の実装方法 38
5.4.3. XML 入出力機能実装のサンプル 42

1. 概要

設計・製造支援アプリケーションのための共通プラットフォーム研究開発では、ソフトウェアのコンポーネント化（部品化）を行い、システム構築や変更を利用者自身による組み立て作業によって実現することを最終目的としています。共通プラットフォームは様々なシステムで共通に使用される基本コンポーネントと、それら部品を使用してシステムを構築／利用するための環境を提供します。

1.1. プラットフォームの概要

本プラットフォーム上ではソフトウェアをコンポーネント化することで、ソフトウェアの保守性を高めるだけでなく、コンポーネントの接続／構成を容易に、かつ動的に行うことによって、アプリケーションシステム全体をより拡張性のあるものにします。

プラットフォーム上のアプリケーションは機能単位に分割されたコンポーネントによって構成され、コンポーネント間は互いに依存性の無い形で関係付けを行います。プラットフォームの基本アーキテクチャを下図に示します。

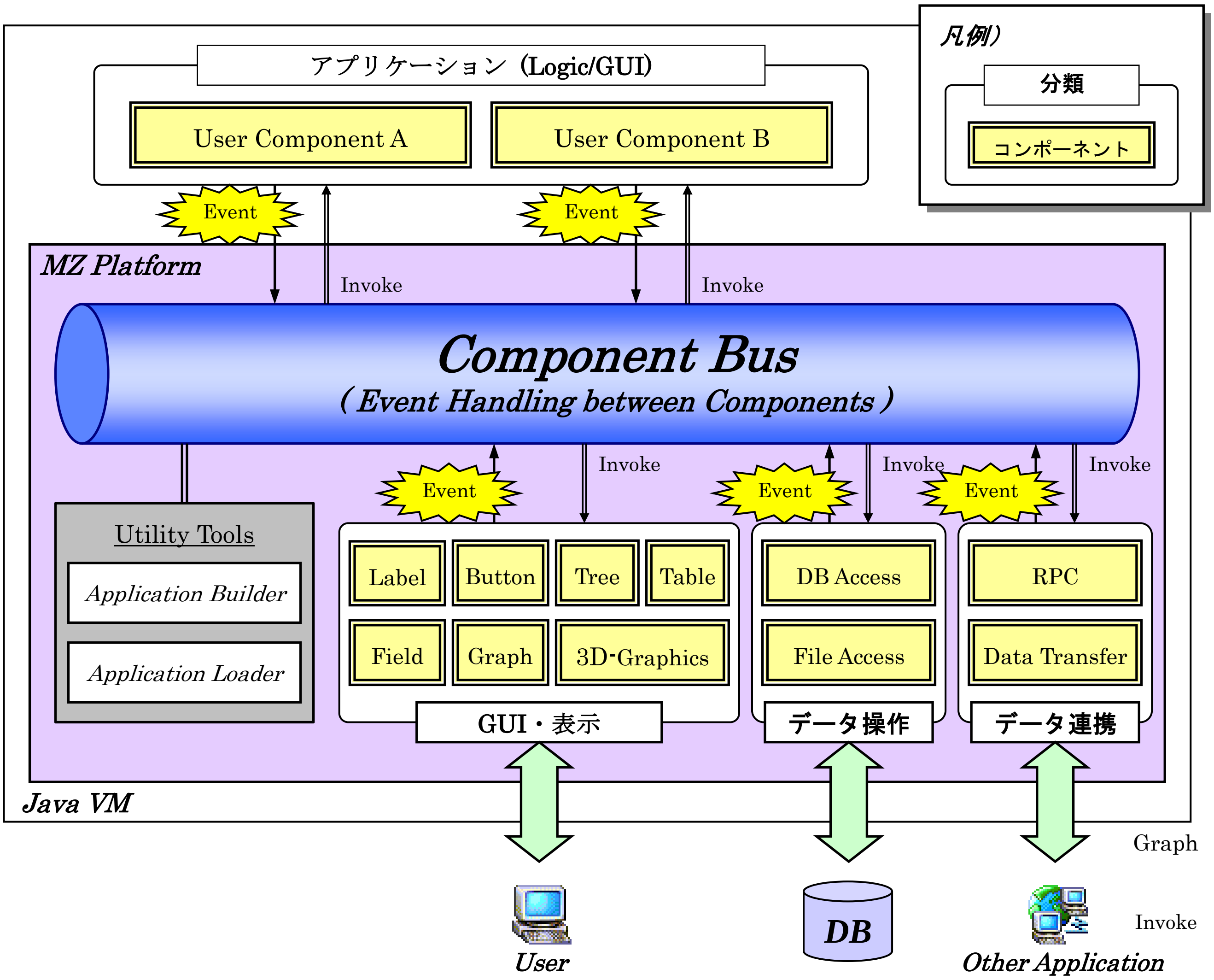


図 1 プラットフォーム基本構造

1.2. コンポーネントの構築

本プラットフォームでは互いに独立したコンポーネント間を接続する仕組みを提供し、各コンポーネント単位での保守性／再利用性を確保します。これを実現させるために、各コンポーネントは以下のルールに従って構築するものとします。

1)Bean として構築

Java のコンポーネント実装である JavaBeans として構築します。[1]

・引数なしのコンストラクタを実装
・直列化が可能（Serializable）
・転送イベントモデル（Delegation Event Model）の使用
・JavaBeans 構成規則に準拠（クラス／メソッドなどの命名規則）

2)共通インターフェイスの実装

プラットフォームが提供する『コンポーネントインターフェイス』を実装します。

3)データ構造

コンポーネント間で交換するデータの構造は、プラットフォームが提供するものに限定します。

4)イベント

コンポーネントから発生させるイベントは、プラットフォームが処理対象とするイベントのみに限定します。

5)その他

・表示／出力のマルチロケール対応（多国語対応）
・他のコンポーネントの機能／データ構造に非依存

1.3. コンポーネント間の接続

JavaBeans の規定に従い、コンポーネント間の接続は転送イベントモデルを使用し、すべての連携はイベント発生をトリガーにした処理起動によって行われます。プラットフォームはあるコンポーネントからイベントを受け、他のコンポーネントの処理を起動します。コンポーネント接続のための機能として、プラットフォームは以下の機能を提供します。

1)接続関係の定義

コンポーネント間の接続関係を定義するためのツールとして、アプリケーションビルダーを提供します。このツールは GUI や簡易定義言語などを利用して、プログラムを書かずにアプリケーションを構築することができるユーティリティです。

2)動的な処理起動

コンポーネントの接続はアプリケーション実行中でも変更可能とし、アプリケーションの実行を止めることなく仕様変更／動作確認が可能です。そのために、コンポーネント間接続関係はプログラムソース内に埋め込まず、実行時のデータとして管理することによって、動的に処理制御を変更することができるようにします。

---

[1] JavaBeans の詳細規定については
http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html を参照

2. MZ Platform

本プラットフォームは、アプリケーション構築のベースとなるコンポーネント開発を支援するための開発環境を提供し、コンポーネント開発におけるフレームワークを定めるものです。（下図太枠が提供範囲）

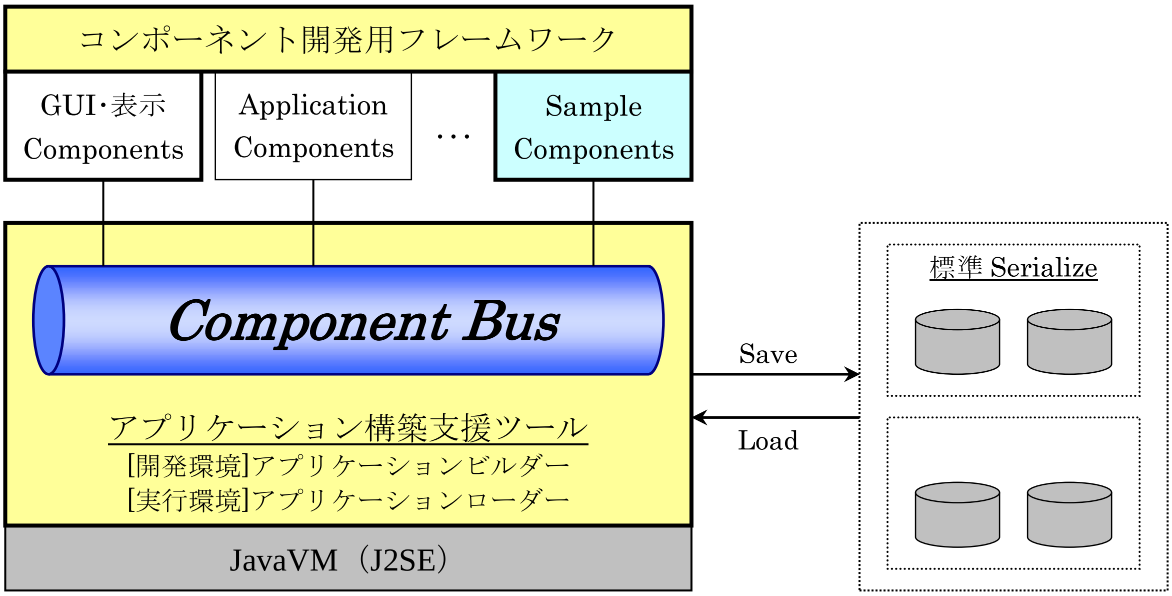


図 2 MZ Platform 提供範囲

本プラットフォームでは以下を提供します。

1)コンポーネント開発用フレームワーク

コンポーネント構築のルールに基づいた、共通クラス／インターフェイス群を提供します。また、コンポーネント開発用に必須メソッドなどを記述した、テンプレートソースを提供します。

2)アプリケーション構築支援ツール

①アプリケーションビルダー

ルールに基づいて構築されたコンポーネントをアプリケーションとして組み立てる機能をもつユーティリティツールを提供します。このツール上ではコンポーネントの貼り付け／属性変更、画面レイアウト設定、コンポーネント間の接続が可能です。また、ここで作成したアプリケーションはローカルファイルに保存し、再利用が可能です。

②アプリケーションローダー

アプリケーションビルダーによって構築／保存された Java-Application を、ファイルからロードし、実行します。

3)Component-Bus

アプリケーションでのコンポーネント管理／コンポーネント接続を行うための、プラットフォーム基幹機能です。

4)サンプル

コンポーネント開発のサンプルとして、サンプルコンポーネントソースを提供します。

3. コンポーネント共通機能

3.1. 共通データ構造

3.1.1. データ構造クラス

【パッケージ】

”jp.go.aist.dmrc.platform.util”

【データクラス】

Data Class	説明
PFDataStructure	共通データ構造（基底クラス） プラットフォーム上で扱われるすべてのデータ構造のベースとなるクラス。
PFObjectList	リスト構造 任意のオブジェクトのリスト構造（1 次元配列）を保持するためのデータ構造クラス。含まれるオブジェクトはすべて同じクラスである必要はなく、NULL 値も可。
PFOblectTable	二次元テーブル構造 オブジェクトを、二次元の表構造で保持するためのデータ構造クラス。列には列名、列型の属性があり、すべての行において同一列のオブジェクトは列型に指定されているクラスである必要がある。NULL 値も可。
PFObjectTree	ツリー構造 階層構造を持った任意のオブジェクトの関係を保持するためのデータ構造クラス。親ノードは複数の子ノードを持ち、最上位のノードはルートノードと呼ばれる。含まれるオブジェクトはすべて同じクラスである必要はなく、NULL 値も可。

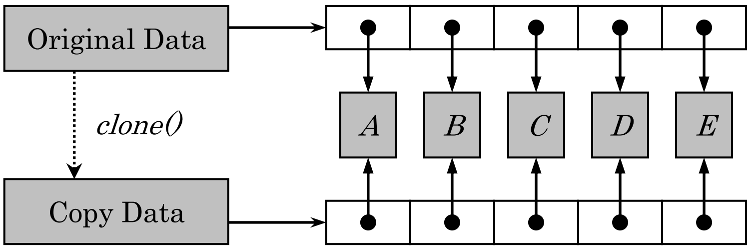
3.1.2. オブジェクトの複製（clone）

共通データ構造は複製可能（Cloneable）とし、メソッド “clone()” では以下のルールで複製します。

①データ構造全体を複製する

②データ構造内のオブジェクト実体は複製しない（参照の複製）

例）リスト構造(PFObjectList)の場合



3.1.3. オブジェクトの同一性（equals）

共通データ構造の同一判断を行うメソッド“equals()”は、以下のルールで判断を行います。

①データの構造が一致すること（構造／サイズなど）

②データ構造内のオブジェクトがすべて同一（メソッド“equals()”）と判断されること

例）リスト構造(PFObjectList)の場合

ListA.equals(ListB)の条件

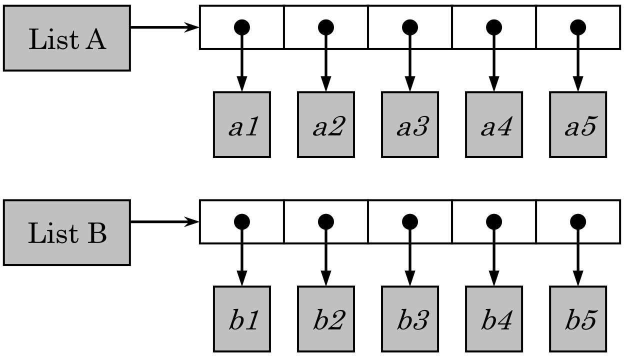
・リストサイズが同じ

・a1.equals(b1)

・a2.equals(b2)

:

・a5.equals(b5)



3.2. イベント

プラットフォーム上のコンポーネント間で伝播するイベントは限定し、それらイベントクラス、およびそれに関連するインターフェイス／クラスはプラットフォームが提供します。

3.2.1. イベント処理

あるコンポーネントのイベント発生を他コンポーネントに伝播させる処理はプラットフォーム（Component-Bus）が行います。このときの連携先をイベント発生時の状態や種類などにあわせて、分岐させるためにイベントにはイベント番号を持たせ、イベント番号ごとに連携先を変える仕組みを提供します。

1)イベント発生側

イベント発生側コンポーネントはイベント発生時にイベント番号を設定します。例えば、『データ更新イベント』にイベント番号を設定することにより、更新対象のデータや更新内容を外部に知らせることが可能です。イベント番号は、イベントオブジェクトを生成する時点で設定可能です。イベント番号を設定しない場合はデフォルト値として“0”が設定されます。なお、イベント番号は実行時のデータやタイミングなどで変わらない方が、アプリケーションの構築作業がわかりやすくなりますので、固定値を使用することを推奨します。

2)イベント連携先分岐（Component-Bus 機能）

処理分岐の設定はアプリケーション構築時に行い、アプリケーションビルダーによって連携対象のイベント番号を設定できます。

3.2.2. イベント一覧

プラットフォームが提供するイベントを以下に示します。プラットフォーム上の処理のきっかけとなるイベントはこれらに限定されており、コンポーネントの状態変更などの情報は、これらのイベントの発生によって実装します。

新規にコンポーネントを作成する場合は、下記一覧表の種別欄に“コンポーネント用”と示されているものから適したものを選択して使用します。また、独自の GUI コンポーネントを作成する場合は、“GUI コンポーネント用”のイベントを使用します。

イベント	説明	種別
アプリケーション開始イベント	アプリケーションの開始を示すイベント	システム用
アプリケーション終了イベント	アプリケーションの終了を示すイベント	システム用
処理要求イベント	他のコンポーネントに対する処理要求を行うイベント	コンポーネント用
処理完了イベント	処理の完了を示すイベント	コンポーネント用
アクションイベント	アクションが発生したことを示すイベント	コンポーネント用
フォーカスイベント	フォーカス取得などの動作が発生したことを示すイベント	GUI コンポーネント用
マウスボタンイベント	マウスボタンの操作が発生したことを示すイベント	GUI コンポーネント用
マウスモーションイベント	マウス移動など動作が発生したことを示すイベント	GUI コンポーネント用
マウスホイールイベント	マウスホイールの操作が発生したことを示すイベント	GUI コンポーネント用
キーイベント	キーボードからの入力が発生したことを示すイベント	GUI コンポーネント用
スクロールイベント	スクロールされたことを示すイベント（１次元）	GUI コンポーネント用
縦横スクロールイベント	スクロールされたことを示すイベント（２次元）	GUI コンポーネント用
ピックイベント	ビューワ表示上で図形がピックされたことを示すイベント	GUI コンポーネント用
ロケートイベント	ビューワ表示上でロケート（背景選択）されたことを示すイベント	GUI コンポーネント用
ビュー変更イベント	ビューワ表示のビューが変更されたことを示すイベント	GUI コンポーネント用
データドロップイベント	画面上にオブジェクトがドロップされたことを示すイベント	GUI コンポーネント用
データ生成イベント	データが生成されたことを示すイベント	コンポーネント用
データ設定イベント	データがセットされたことを示すイベント	コンポーネント用
データ更新イベント	データが更新されたことを示すイベント	コンポーネント用
データ選択イベント	データが選択されたことを示すイベント	コンポーネント用
コンポーネント連携結果通知イベント	コンポーネント連携の処理結果が通知されたことを示すイベント	システム用
PULL 型コンポーネント転送結果通知イベント	PULL 型コンポーネント転送の処理結果が通知されたことを示すイベント	システム用
PUSH 型コンポーネント転送結果通知イベント	PUSH 型コンポーネント転送の処理結果が通知されたことを示すイベント	システム用
PULL 型コンポーネント転送送信イベント	PULL 型コンポーネント転送を行ったことを示すイベント	システム用
PUSH 型コンポーネント転送受信イベント	PUSH 型コンポーネント転送を受けたことを示すイベント	システム用

3.2.3. イベント関連クラス

1)イベントクラス

・基底クラス

イベント		イベント内包データ			
イベント	Event Class	説明	型	取得メソッド	設定メソッド
イベント（すべてのイベントの基底クラス）	PFEvent	イベント発生元コンポーネント	java.lang.Object	getSource()	ー（コンストラクタにて設定）
		イベント番号	int	getEventNo()	setEventNo()
抽象マウスイベント（マウスボタン、マウスモーション、マウスホイールの基底クラス）	PFAbstractMouseEvent	マウスＸ座標	int	getX()	ー（コンストラクタにて設定）
		マウスＹ座標	int	getY()	ー（コンストラクタにて設定）
		イベント種別	int	getType()	ー（コンストラクタにて設定）

・イベントクラス

イベント		イベント内包データ			
イベント	Event Class	説明	型	取得メソッド	設定メソッド
アプリケーション開始イベント	PFApplicationStartEvent	ー	ー	ー	ー
アプリケーション終了イベント	PFApplicationTerminateEvent	ー	ー	ー	ー
処理要求イベント	PFProcessRequestEvent	処理要求データ	java.lang.Object	getRequestData()	setRequestData()
処理完了イベント	PFProcessTerminateEvent	処理結果データ	java.lang.Object	getResultData()	setResultData()
アクションイベント	PFActionEvent	ー	ー	ー	ー
フォーカスイベント	PFFocusEvent	ー	ー	ー	ー
マウスボタンイベント	PFMouseButtonEvent	マウスボタン種別	int	getMouseButton()	ー（コンストラクタにて設定）
		修飾キー	int	getMouseModifiers()	ー（コンストラクタにて設定）
		マウスクリック回数	int	getClickCount()	ー（コンストラクタにて設定）
マウスモーションイベント	PFMouseMotionEvent	ー	ー	ー	ー
マウスホイールイベント	PFMouseWheelEvent	マウスホイールの回転数	int	getWheelRotation()	ー（コンストラクタにて設定）
キーイベント	PFKeyEvent	イベント種別	int	getType()	ー（コンストラクタにて設定）
		キーコード	int	getKeyCode()	ー（コンストラクタにて設定）
		入力文字	char	getKeyChar()	ー（コンストラクタにて設定）
		修飾キー	int	getKeyModifiers()	ー（コンストラクタにて設定）
スクロールイベント	PFScrollEvent	スクロール位置	int	getScrollPosition()	setScrollPosition()
縦横スクロールイベント	PFScroll2Devent	縦スクロール位置	int	getVerticalScrollPosition()	setVerticalScrollPosition()
		横スクロール位置	int	getHorizontalScrollPosition()	setHorizontalScrollPosition()
ピックイベント	PFViewPickEvent	ピック情報	java.lang.Object	getPickData()	setPickData()
ロケートイベント	PFViewLocateEvent	ロケート情報	java.lang.Object	getLocateData()	setLocateData()
ビュー変更イベント	PFViewUpdateEvent	変更後ビューマトリクス	java.lang.Object	getViewMatrix()	setViewMatrix()
データドロップイベント	PFDataDropEvent	ドロップデータ	java.lang.Object	getDroppedData()	setDroppedData()
データ生成イベント	PFDataCreateEvent	生成されたデータ	java.lang.Object	getSourceData()	setSourceData()
データ設定イベント	PFDataSetEvent	セットされたデータ	java.lang.Object	getSourceData()	setSourceData()
データ更新イベント	PFDataUpdateEvent	更新対象のデータ	java.lang.Object	getSourceData()	setSourceData()
		更新情報	java.lang.Object	getUpdatedData()	setUpdatedData()
データ選択イベント	PFDataSelectEvent	選択対象のデータ	java.lang.Object	getSourceData()	setSourceData()
		選択情報	java.lang.Object	getSelectedData()	setSelectedData()

・データ連携のイベントクラス

イベント		イベント内包データ			
イベント	Event Class	説明	型	取得メソッド	設定メソッド
コンポーネント連携結果通知イベント	PFComponentCooperationResultEvent	リクエストID	java.lang.String	getRequestID()	－（コンストラクタにて設定）
		ステータスコード	int	getStatusCode()	－（コンストラクタにて設定）
		コンポーネント連携の結果	java.lang.Object	getResult()	－（コンストラクタにて設定）
		コンポーネント連携の結果の型	java.lang.Class	getResultType()	－（コンストラクタにて設定）
		エラーの発生場所	java.lang.String	getErrorLocation()	－（コンストラクタにて設定）
		エラーメッセージ	java.lang.String	getErrorMessage()	－（コンストラクタにて設定）
		詳細なエラーメッセージ	java.lang.String	getDetailedErrorMessage()	－（コンストラクタにて設定）
PULL 型コンポーネント転送結果通知イベント	PFPullComponentTransferResultEvent	リクエストID	java.lang.String	getRequestID()	－（コンストラクタにて設定）
		取得したコンポーネント	PFComponent	getComponent()	－（コンストラクタにて設定）
		ステータスコード	int	getStatusCode()	－（コンストラクタにて設定）
		エラーの発生場所	java.lang.String	getErrorLocation()	－（コンストラクタにて設定）
		エラーメッセージ	java.lang.String	getErrorMessage()	－（コンストラクタにて設定）
		詳細なエラーメッセージ	java.lang.String	getDetailedErrorMessage()	－（コンストラクタにて設定）
PUSH 型コンポーネント転送結果通知イベント	PFPushComponentTransferResultEvent	リクエストID	java.lang.String	getRequestID()	－（コンストラクタにて設定）
		ステータスコード	int	getStatusCode()	－（コンストラクタにて設定）
		エラーの発生場所	java.lang.String	getErrorLocation()	－（コンストラクタにて設定）
		エラーメッセージ	java.lang.String	getErrorMessage()	－（コンストラクタにて設定）
		詳細なエラーメッセージ	java.lang.String	getDetailedErrorMessage()	－（コンストラクタにて設定）
PULL 型コンポーネント転送送信イベント	PFPullComponentTransferSentEvent	リクエストID	java.lang.String	getRequestID()	－（コンストラクタにて設定）
		転送タイプコード	int	getTransferTypeCode()	－（コンストラクタにて設定）
		送付したコンポーネントのリスト	PFObjectList	getComponentList()	－（コンストラクタにて設定）
		ステータスコード	int	getStatusCode()	－（コンストラクタにて設定）
		エラーの発生場所	java.lang.String	getErrorLocation()	－（コンストラクタにて設定）
		エラーメッセージ	java.lang.String	getErrorMessage()	－（コンストラクタにて設定）
		詳細なエラーメッセージ	java.lang.String	getDetailedErrorMessage()	－（コンストラクタにて設定）
PUSH 型コンポーネント転送受信イベント	PFPushComponentTransferReceivedEvent	リクエストID	java.lang.String	getRequestID()	－（コンストラクタにて設定）
		転送タイプコード	int	getTransferTypeCode()	－（コンストラクタにて設定）
		送付されたコンポーネントのリスト	PFObjectList	getComponentList()	－（コンストラクタにて設定）
		ステータスコード	int	getStatusCode()	－（コンストラクタにて設定）
		エラーの発生場所	java.lang.String	getErrorLocation()	－（コンストラクタにて設定）
		エラーメッセージ	java.lang.String	getErrorMessage()	－（コンストラクタにて設定）
		詳細なエラーメッセージ	java.lang.String	getDetailedErrorMessage()	－（コンストラクタにて設定）

2)イベント関連クラス

イベント	イベントソース		イベントリスナ	
	interface	interface 実装クラス	interface	interface 実装クラス
アプリケーション開始イベント	PFApplicationStartEventSource	PFApplicationStartEventSourceImpl	PFApplicationStartListener	PFApplicationStartListenerImpl
アプリケーション終了イベント	PFApplicationTerminateEventSource	PFApplicationTerminateEventSourceImpl	PFApplicationTerminateListener	PFApplicationTerminateListenerImpl
処理要求イベント	PFProcessRequestEventSource	PFProcessRequestEventSourceImpl	PFProcessRequestListener	PFProcessRequestListenerImpl
処理完了イベント	PFProcessTerminateEventSource	PFProcessTerminateEventSourceImpl	PFProcessTerminateListener	PFProcessTerminateListenerImpl
アクションイベント	PFActionEventSource	PFActionEventSourceImpl	PFActionListener	PFActionListenerImpl
フォーカスイベント	PFFocusEventSource	PFFocusEventSourceImpl	PFFocusListener	PFFocusListenerImpl
マウスボタンイベント	PFMouseButtonEventSource	PFMouseButtonEventSourceImpl	PFMouseButtonListener	PFMouseButtonListenerImpl
マウスモーションイベント	PFMouseMotionEventSource	PFMouseMotionEventSourceImpl	PFMouseMotionListener	PFMouseMotionListenerImpl
マウスホイールイベント	PFMouseWheelEventSource	PFMouseWheelEventSourceImpl	PFMouseWheelListener	PFMouseWheelListenerImpl
キーイベント	PFKeyEventSource	PFKeyEventSourceImpl	PFKeyListener	PFKeyListenerImpl
スクロールイベント	PFScrollEventSource	PFScrollEventSourceImpl	PFScrollListener	PFScrollListenerImpl
縦横スクロールイベント	PFScroll2DEventSource	PFScroll2DEventSourceImpl	PFScroll2DListener	PFScroll2DlistenerImpl
ピックイベント	PFViewPickEventSource	PFViewPickEventSourceImpl	PFViewPickListener	PFViewPickListenerImpl
ロケートイベント	PFViewLocateEventSource	PFViewLocateEventSourceImpl	PFViewLocateListener	PFViewLocateListenerImpl
ビュー変更イベント	PFViewUpdateEventSource	PFViewUpdateEventSourceImpl	PFViewUpdateListener	PFViewUpdateListenerImpl
データドロップイベント	PFDataDropEventSource	PFDataDropEventSourceImpl	PFDataDropListener	PFDataDropListenerImpl
データ生成イベント	PFDataCreateEventSource	PFDataCreateEventSourceImpl	PFDataCreateListener	PFDataCreateListenerImpl
データ設定イベント	PFDataSetEventSource	PFDataSetEventSourceImpl	PFDataSetListener	PFDataSetListenerImpl
データ更新イベント	PFDataUpdateEventSource	PFDataUpdateEventSourceImpl	PFDataUpdateListener	PFDataUpdateListenerImpl
データ選択イベント	PFDataSelectEventSource	PFDataSelectEventSourceImpl	PFDataSelectListener	PFDataSelectListenerImpl
コンポーネント連携結果通知イベント	PFComponentCooperationResultEventSource	PFComponentCooperationResultEventSourceImpl	PFComponentCooperationResultListener	PFComponentCooperationResultListenerImpl
PULL 型コンポーネント転送結果通知イベント	PFPullComponentTransferResultEventSource	PFPullComponentTransferResultEventSourceImpl	PFPullComponentTransferResultListener	PFPullComponentTransferResultListenerImpl
PUSH 型コンポーネント転送結果通知イベント	PFPushComponentTransferResultEventSource	PFPushComponentTransferResultEventSourceImpl	PFPushComponentTransferResultListener	PFPushComponentTransferResultListenerImpl
PULL 型コンポーネント転送送信イベント	PFPullComponentTransferSentEventSource	PFPullComponentTransferSentEventSourceImpl	PFPullComponentTransferSentListener	PFPullComponentTransferSentListenerImpl
PUSH 型コンポーネント転送受信イベント	PFPushComponentTransferReceivedEventSource	PFPushComponentTransferReceivedEventSourceImpl	PFPushComponentTransferReceivedListener	PFPushComponentTransferReceivedListenerImpl

3)イベント発生（イベントソース）

イベント	イベントソース	リスナー追加／削除メソッド	イベント発生メソッド	
			メソッド	説明
アプリケーション開始イベント	PFApplicationStartEventSource	addPFApplicationStartListener() removePFApplicationStartListener()	fireApplicationStarted()	アプリケーションの開始
アプリケーション終了イベント	PFApplicationTerminateEventSource	addPFApplicationTerminateListener() removePFApplicationTerminateListener()	fireApplicationTerminated()	アプリケーションの終了
処理要求イベント※	PFProcessRequestEventSource	addPFProcessRequestListener() removePFProcessRequestListener()	fireProcessReqested()	処理の要求
処理完了イベント	PFProcessTerminateEventSource	addPFProcessTerminateListener() removePFProcessTerminateListener()	fireProcessTerminated()	処理の完了
アクションイベント	PFActionEventSource	addPFActionListener() removePFActionListener()	fireActionPerformed()	アクションの発生
フォーカスイベント	PFFocusEventSource	addPFFocusListener() removePFFocusListener()	fireFocusChanged()	フォーカスの変更
マウスボタンイベント	PFMouseButtonEventSource	addPFMouseButtonListener() removePFMouseButtonListener()	fireButtonActionPerformed()	マウスボタンの操作
マウスモーションイベント	PFMouseMotionEventSource	addPFMouseMotionListener() removePFMouseMotionListener()	fireMotionPerformed()	マウス移動操作
マウスホイールイベント	PFMouseWheelEventSource	addPFMouseWheelListener() removePFMouseWheelListener()	fireWheelActionPerformed()	マウスホイールの操作
キーイベント	PFKeyEventSource	addPFKeyListener() removePFKeyListener()	fireKeyActionPerformed()	キーの操作
スクロールイベント	PFScrollEventSource	addPFScrollListener() removePFScrollListener()	fireScrolled()	スクロール
縦横スクロールイベント	PFScroll2DEventSource	addPFScroll2DListener() removePFScroll2DListener()	fireScrolled()	スクロール
ピックイベント	PFViewPickEventSource	addPFViewPickListener() removePFViewPickListener()	fireViewPicked()	ピック
ロケートイベント	PFViewLocateEventSource	addPFViewLocateListener() removePFViewLocateListener()	fireViewLocated()	ロケート（背景選択）
ビュー変更イベント	PFViewUpdateEventSource	addPFViewUpdateListener() removePFViewUpdateListener()	fireViewUpdated()	ビューの変更
データドロップイベント	PFDataDropEventSource	addPFDataDropListener() removePFDataDropListener()	fireDataDroped()	オブジェクトのドロップ
データ生成イベント	PFDataCreateEventSource	addPFDataCreateListener() removePFDataCreateListener()	fireDataCreated()	データの生成
データ設定イベント	PFDataSetEventSource	addPFDataSetListener() removePFDataSetListener()	fireDataSet()	データの設定
データ更新イベント	PFDataUpdateEventSource	addPFDataUpdateListener() removePFDataUpdateListener()	fireDataUpdated()	データの更新
データ選択イベント	PFDataSelectEventSource	addPFDataSelectListener() removePFDataSelectListener()	fireDataSelected()	データの選択
コンポーネント連携結果通知イベント	PFComponentCooperationResultEventSource	addPFComponentCooperationResultListener() removePFComponentCooperationResultListener()	fireComponentCooperationResultInformed()	コンポーネント連携の実行
PULL 型コンポーネント転送結果通知イベント	PFPullComponentTransferResultEventSource	addPFPullComponentTransferResultListener() removePFPullComponentTransferResultListener()	firePullComponentTransferResultInformed()	プル型コンポーネント転送の実行
PUSH 型コンポーネント転送結果通知イベント	PFPushComponentTransferResultEventSource	addPFPushComponentTransferResultListener() removePFPushComponentTransferResultListener()	firePushComponentTransferResultInformed()	プッシュ型コンポーネント転送の実行
PULL 型コンポーネント転送送信イベント	PFPullComponentTransferSentEventSource	addPFPullComponentTransferSentListener() removePFPullComponentTransferSentListener()	firePullComponentTransferSent()	プル型コンポーネント転送によるコンポーネントの送信
PUSH 型コンポーネント転送受信イベント	PFPushComponentTransferReceivedEventSource	addPFPushComponentTransferReceivedListener() removePFPushComponentTransferReceivedListener()	firePushComponentTransferReceived()	プッシュ型コンポーネント転送によるコンポーネントの受信

※処理要求イベント

処理要求イベントは要求先の処理結果を受け取るために、要求先の処理結果をイベント発生メソッド“fireProcessReqested()”の戻り値として受け取ることができます。他のイベント発生メソッドは戻り値がありません。

3.3. 例外処理

3.3.1. 例外一覧

Exception Class	説明	発生箇所
PFException	MZ Platform 例外	※1
PFComponentException	コンポーネントで発生の例外	コンポーネント
PFSystemException	プラットフォーム内部の例外	※1
PFMethodException	メソッド起動に関する例外	※1
PFMethodFaildException	起動したメソッドで例外発生	Component-Bus
PFMethodInvokeException	メソッド起動に失敗	Component-Bus
PFMethodParameterException	メソッド引数の取得に失敗	Component-Bus
PFRuntimeException	その他の例外	Component-Bus
※1：これらは基底クラスであるため実際には発生しない		

3.3.2. 各コンポーネントの例外処理

あるコンポーネント内のメソッドで発生した例外は、呼び出し元まで伝播させます。これを実現させるために、各コンポーネントでは以下のような方針で例外処理を行ってください。なお、例外処理のサンプルコードは、5.コンポーネントの実装手順にありますので、そちらを参考にしてください。

①ユーザ操作時に発生したエラー／例外（GUI コンポーネントの場合）

【方針】

画面上にエラーメッセージを表示する。

【内容】

プラットフォームが提供するエラーメッセージ表示ユーティリティを使用して、画面上にエラーメッセージダイアログを表示する。

②他コンポーネントからのメソッド起動によって発生したエラー／例外

【方針】

例外を呼び出し元にスローする。

【内容】

プラットフォーム例外（PFException）を受けた場合は、そのままスローする。その他の例外やエラーについては、新たに PFComponentException オブジェクトを作成しスローする。

このときエラーメッセージを必ず設定し、発生原因となった例外があればそれも設定する。

【注意】

プラットフォーム例外を受けとった場合に、新たな例外を作成してスローしてしまうと、エラーの原因となった情報が呼び出し元まで伝達されないため、GUI コンポーネントが表示するエラーメッセージダイアログに正しいメッセージが表示できないことがある。必ずこのルールで例外の伝播処理を行っておくこと。

3.3.3. エラーメッセージ表示

GUI コンポーネントなど、イベント発生の元となるコンポーネントでは、エラーが発生した場合に利用者にエラー内容を表示して通知しなければならないケースがあります。エラーメッセージ表示にはプラットフォームが提供する以下のユーティリティを使用します。エラーメッセージのマルチロケール対応については各コンポーネント内で必ず行ってください。

クラス	jp.go.aist.dmrc.platform.util.PFUtility	
メソッド	エラー表示	displayErrorMessage(Component parent, PFException ex)
		displayErrorMessage(Component parent, String message)
		displayErrorMessage(Component parent, Throwable ex)
	詳細表示	displayDetailErrorMessage(Component parent, PFException ex)
	※第 1 引数はエラーメッセージを表示するベースコンポーネント	

3.4. メッセージダイアログ

先述のエラーメッセージ表示のように、メッセージを表示するためのダイアログ表示を共通機能として提供します。表示されるメッセージのマルチロケール対応については各コンポーネント内で必ず行ってください。また、GUI コンポーネントとしても、メッセージダイアログが提供されていますので、拡張性を持たせたいなどの必要に応じてそちらも利用してください。

1)エラーメッセージ表示

クラス	jp.go.aist.dmrc.platform.util.PFUtility
メソッド	public static void displayErrorMessage(Component parent, PFException ex) public static void displayErrorMessage(Component parent, String message) public static void displayErrorMessage(Component parent, Throwable ex) public static void displayDetailErrorMessage(Component parent, PFException ex)
	※第 1 引数はエラーメッセージを表示するベースコンポーネント
イメージ	 エラーメッセージ

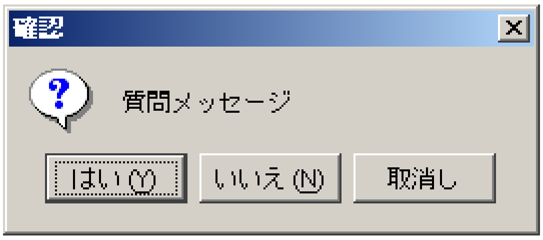
2)情報メッセージ表示

クラス	jp.go.aist.dmrc.platform.util.PFUtility
メソッド	public static void displayInformationMessage(Component parent, String message)
	※第 1 引数はエラーメッセージを表示するベースコンポーネント
イメージ	 情報メッセージ

3)警告メッセージ表示

クラス	jp.go.aist.dmrc.platform.util.PFUtility
メソッド	public static void displayWarningMessage(Component parent, String message)
	※第 1 引数はエラーメッセージを表示するベースコンポーネント
イメージ	 警告メッセージ

4)質問メッセージ表示

クラス	jp.go.aist.dmrc.platform.util.PFUtility
メソッド	public static int displayQuestionMessage(Component parent, String message)
	public static int displayQuestionMessage(Component parent, String message, boolean cancel)
	※第 1 引数はエラーメッセージを表示するベースコンポーネント ※第 3 引数はキャンセルボタンの有無 ※戻り値は押されたボタンの情報（詳細は以下） PFUtility.ANSWER_YES : [はい](Yes)ボタン PFUtility.ANSWER_NO : [いいえ](No)ボタン PFUtility.ANSWER_CANCEL : [取消し](Cancel)ボタン PFUtility.ANSWER_CLOSED : ダイアログのクローズ
イメージ	 確認 質問メッセージ ①キャンセルボタンなし
	 確認 質問メッセージ ②キャンセルボタンあり

3.5. メモリ負荷状況

特にアプリケーション構築中の動作確認、性能評価などの目的のため、アプリケーション実行／アプリケーション構築中の JavaVM が使用しているメモリサイズを取得する機能を提供します。使用されるメモリ量にあわせて、アプリケーションの実行時にメモリの初期確保サイズ、拡張最大サイズなどの設定を行う必要があります。（Java コマンドのオプション）

クラス	jp.go.aist.dmrc.platform.util.PFUtility	
メソッド	public static long getUsedMemory()	※使用中のメモリサイズ（Byte）
	public static long getAllocatedMemory()	※確保されているメモリサイズ（Byte）

3.6. ログ出力

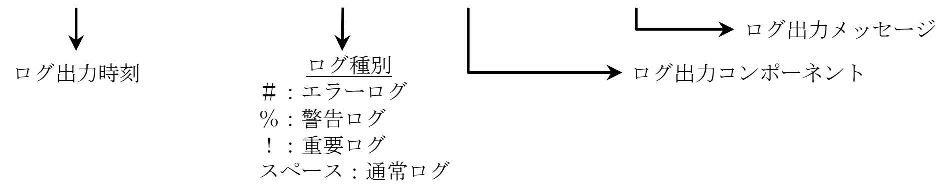
アプリケーションの処理トレース、エラー情報取得のために、ログ出力機能を提供します。ログにはその出力目的／内容によって出力時に区別し、実行時のログ出力レベル設定にあわせて出力されます。ログ出力レベルによる各ログ種別の出力対象を下表に示します。

ログ種別	ログ出力レベル			
	すべて出力	重要ログのみ	エラーログのみ	出力なし
通常ログ	◎	×	×	×
重要ログ	◎	◎	×	×
警告ログ	◎	◎	◎	×
エラーログ	◎	◎	◎	×

3.6.1. ログ出力処理

クラス	jp.go.aist.dmrc.platform.util.PFUtility
メソッド	public static void logWrite(int type, PFComponent component, String message, Throwable ex)
	public static void logWrite(int type, PFComponent component, String message)
	public static void errorLogWrite(PFComponent component, String message, Throwable ex)
	※引数 type はログ種別 PFUtility.LOG_TYPE_NORMAL　通常ログ PFUtility.LOG_TYPE_IMPORTANT　重要ログ PFUtility.LOG_TYPE_WARNING　警告ログ PFUtility.LOG_TYPE_ERROR　エラーログ ※引数 component はログ出力コンポーネント ※引数 ex はエラー発生時の例外オブジェクト（NULL 可）

3.6.2. ログ内容

ファイル名	Platform.log　※実行時のカレントフォルダに保管
出力項目	[YYYY.MM.DD hh:mm:ss:lll] [#] [Component]　Message 
出力例	[2002.08.31 22:41:17:286][ ][コンポーネントバス] RECEIVE: PFApplicationStartEvent(コンポーネントバス ［ ID : 0]) [2002.08.31 22:41:17:286][!][コンポーネントバス] INVOKE : ダイアログ ［ ID : 12]::show() [2002.08.31 22:41:21:402][ ][コンポーネントバス] RECEIVE: PFActionEvent(ボタン(Sample) [ID : 1]) [2002.08.31 22:41:21:402][!][コンポーネントバス] INVOKE : フィールド(Sample) [ID : 9]::clearText() [2002.08.31 22:41:21:402][!][コンポーネントバス] INVOKE : データ管理(Sample) [ID : 11]::sendData()

3.7. マルチロケール対応

本プラットフォーム、およびプラットフォーム上で動作するコンポーネント／アプリケーションはマルチロケールに対応し、英語環境や日本語環境など様々な環境下での動作において、その環境にあわせた振る舞いを行う必要があります。

3.7.1. 固定文字列のマルチロケール対応

【対応対象】

固定ポップアップメニュー／エラーメッセージ／警告メッセージ／派生ダイアログ 他

【対応方針】

①Java が提供する ResourceBundle 機能を使用する

②各国語のリソースファイルについては、各コンポーネント開発時に準備する

③リソース文字列の取得については、プラットフォームが提供する共通ユーティリティを使用する

【対応方法】

プラットフォームやコンポーネントが表示／出力する固定文字列は、リソースファイルとしてプログラムから切り出し、実行時に環境にあわせたリソースを取得する。

①リソースファイル作成

少なくとも以下の３つのリソースファイルを作成する。

<table>
<tr><th>ロケール（言語）</th><th>ファイル名</th><th>備考</th></tr>
<tr><td>日本語</td><td><Resource Name>_ja.properties</td><td></td></tr>
<tr><td>英語</td><td><Resource Name>_en.properties</td><td></td></tr>
<tr><td>デフォルト</td><td><Resource Name>.properties</td><td>デフォルトは英語</td></tr>
<tr><td colspan="3">特記事項）
実装時に <Resource Name>_ja.properties と <Resource Name>_en.properties を作成しておいてください。<Resource Name>.properties は <Resource Name>_en.properties のコピーで十分です。
class ファイルおよび jar の作成時には、J2SE が提供する native2ascii コマンドを使って変換したリソースファイルを利用してください。</td></tr>
<tr><th colspan="3">リソースファイル例</th></tr>
<tr><td colspan="2">ErrorMessage_ja.properties</td><td>ErrorMessage_en.properties</td></tr>
<tr><td colspan="2">ErrorDialogTitle=エラーメッセージ

ERRNO-001=ファイル {0} が見つかりません。
ERRNO-002=ファイル名が不正です。</td><td>ErrorDialogTitle=Error Message

ERRNO-001=File {0} not found.
ERRNO-002=File name is invalid.</td></tr>
</table>

※{ } で囲まれた部分には、プログラム側で任意の値を設定することができます。
詳細については、②リソースの取得 pubilc String get(String resourceName, String key, Object[] arg) および java.text.MessageFormat の Javadoc を参照してください。

②リソースの取得

リソース管理ユーティリティを使用し、文字列の表示／出力時に実行ロケール環境に合わせた文字列を取得／使用する。

<table>
<tr><th>クラス</th><td colspan="2">jp.go.aist.dmrc.platform.util.PFResourceUtility</td></tr>
<tr><th rowspan="2">メソッド</th><td>文字列取得</td><td>public String get(String *resourceName*, String *key*)
※リソースファイル／データが見つからない場合は null を返す</td></tr>
<tr><td>文字列取得
(引数付)</td><td>public String get(String *resourceName*, String *key*, Object[] *arg*)
※引数付きリソース文字列を取得する
引数については、java.text.MessageFormat の Javadoc を参照
※リソースファイル／データが見つからない場合は null を返す</td></tr>
</table>

3.7.2. データ文字列のマルチロケール対応

【対応対象】

コンポーネント名称／GUI 部品のラベル文字列／タイトル文字列 など

【対応方針】

①マルチロケール対応文字列のためのデータ型を提供する

②このデータ型に対してアプリケーション構築時に複数ロケールの文字列データを設定可能とする

③このデータ型から文字列を取得すると、実行時ロケールに対応した文字列が取得できる

【対応方法】

コンポーネントの属性として表示／出力する文字列データは、マルチロケール対応文字列データとして定義し、マルチロケール対応する。

①マルチロケール対応文字列型

マルチロケール対応文字列データを“マルチロケール対応文字列型”として定義する。

クラス	jp.go.aist.dmrc.platform.util.PFMultiLocaleString	
メソッド	**文字列設定**	public void setString(String *str*, Locale *locale*) **※引数のロケールは言語のみ有効で、国名は無視する**
	文字列取得	public String getString() **※実行ロケール（言語）に対応するデータが見つからない場合、** **デフォルトとして英語ロケール用文字列を返し、** **それも存在しなければ null を返す**

②マルチロケール対応文字列データの設定

各ロケールにあわせた文字列を設定するには、上記のメソッド“setString()”を使用する。引数のロケールには以下を設定する。

日本語：java.util.Locale.JAPANESE
英　語：java.util.Locale.ENGLISH

③マルチロケール対応文字列データの表示

表示時に上記メソッド“getString()”を使用し、稼働ロケールに対応した文字列データを取得する。

3.7.3. コンポーネントのマルチロケール対応

【対応対象】

全コンポーネント

【対応方針】

①実行中にロケールが変更された場合、それに合わせて表示などが変更されるようにする

【対応方法】

コンポーネントインターフェイスに外部からのロケール変更通知を受けるためのメソッドを定義し、全コンポーネントはこのメソッドを実装しなければならない。

①メソッド“localeUpdated()”の実装

表示されている文字列の変更、固定ポップアップメニューの再作成など、必要に応じて内部データ変更を行う。対応が不要な場合は何もしない空のメソッドを作成する。

インターフェイス	jp.go.aist.dmrc.platform.base.PFComponent
メソッド	public void localeUpdated()

3.8. シリアライズの互換性

開発中の作業効率性や将来のバージョンアップへの対応を考えると、シリアライズにおける互換性について意識する必要があります。これは、各オブジェクト（各コンポーネントなど）の責任で行うものであり、外部から対応できる問題ではありません。

各コンポーネント、およびコンポーネント内で扱うデータ構造やデータ型など、プラットフォーム上でシリアライズされる可能性のあるオブジェクト（Serializable 実装しているクラス）については、シリアライズに対する考慮が必要となります。

3.8.1. シリアライズデータの互換性

クラスに変更を行った場合、旧バージョンで作成したシリアライズデータをデシリアライズできるかどうか（シリアライズデータに互換性があるか）は、その変更内容によって異なります。

1)互換性のない変更
 ①フィールドの削除
 ②クラス継承階層の上下移動
 ③インスタンス変数（非 static 変数）からクラス変数（static 変数）への変更
 ④非 transient 変数から transient 変数への変更
 ⑤データ型の変更
 ⑥writeObject()／readObject()の実装変更（シリアライズされるデータに変更がある場合）
 ⑦Serializable から Externalizable への変更（またはその逆）
 ⑧Serializable／Externalizable の実装取り消し

2)互換性のある変更
 ①フィールドの追加
 ②クラス継承階層の追加
 ③クラス継承階層の削除
 ④writeObject()／readObject()の追加
  （ただし defaultWriteObject()/defaultReadObject()の呼び出しは必須）
 ⑤writeObject()／readObject()の削除
 ⑥Serializable の実装追加
 ⑦フィールドのアクセス変更（public/package/protected/private）
 ⑧クラス変数（static 変数）からインスタンス変数（非 static 変数）への変更
 ⑨transient 変数から非 transient 変数への変更

3.8.2. オブジェクトのシリアライズ互換性確保

各コンポーネントなどのオブジェクトシリアライズについては、以下のような対応を行うことによって互換性を確保することができます。

1)シリアライズバージョン番号の定義

シリアライズバージョン番号を明示的に定義することで、新旧バージョンを同一に扱えます。

現行はシリアライズバージョン番号を自動発番しているために、互換性がなくなっています。

【対応方法】

①シリアライズバージョン番号を決定する

新規に作成するクラスでは、シリアライズバージョン番号をプログラムのバージョンや作成日、などの固定数値に設定しておきます。既存クラスでシリアライズ互換性を保持するためには、以下のコマンドで従来のシリアライズバージョン番号を取得します。

コマンド： serialver.exe ＜*class_name*＞

コマンド(実行ファイル)が存在するフォルダ： (J2SDK インストール先フォルダ)¥bin

環境　：環境変数“CLASSPATH”を対象クラスの保管場所に設定すること

実行例：コマンド入力

```
>serialver jp.go.aist.dmrc.platform.util.PFObjectList
jp.go.aist.dmrc.platform.util.PFObjectList: static final long
serialVersionUID = -3563129031580938021L;
```

②該当クラスにフィールド”serialVersionUID”を定義し、上記の番号を設定する

実行例：PFObjectList.java

```
public class PFObjectList {
    …
    private static final long serialVersionUID = -3563129031580938021L;
    …
}
```

③フィールド”serialVersionUID”が変わらなければ、同一データとして扱いが可能

※”serialVersionUID”は一度決定したら互換性のある間は変更しない

2)シリアライズ／デシリアライズ処理での意識

先述の『互換性のある変更』であれば、上記シリアライズバージョンの定義によって新旧バージョン間での互換性は確保できます。ただし、旧バージョンのデータを入力とする場合を想定する必要があります。(シリアライズ実装方針の維持が必要)

<u>例：新たにフィールドが追加された場合</u>

前提：旧バージョンでシリアライズされたデータが存在

変更：新たにフィールドを追加

処理：新バージョンで旧データをデシリアライズする

・通常のデシリアライズ処理が可能（互換性のある変更）

・新たに追加されたフィールドの値は設定されない（旧データに存在しない）

対応：（必要であれば）新たに追加されたフィールドの初期値（デフォルト値）を設定する

3.9. システムプロパティの利用

アプリケーション動作を実行時のパラメタで制御したい場合には、プラットフォームが提供するシステムプロパティ機能を使用します。プラットフォーム実行パラメタが格納されている初期設定ファイル（導入フォルダ¥etc¥Platform.ini）には自由にパラメタの設定が可能で、そこで設定されたパラメタはアプリケーション実行中、Java のシステムパラメタとして参照が可能になります。ただし、初期設定ファイルは起動時に入力されますので、起動後に編集しても反映されません。

アプリケーション単位に独自の初期設定ファイルを作成し、それを入力する処理をアプリケーションで実装する方法もありますが、システムプロパティ機能を使用することで簡単に実行環境の動作設定を実現することが可能です。

■Platform.ini

```
LogLevel=3
LookAndFeel=
RuntimeLocale=
ComponentListFile_ja=PlatformComponents_ja.ini
ComponentListFile_en=PlatformComponents_en.ini
ComponentListFile=PlatformComponents_en.ini
ComponentInformationFolder=components
UseDataCooperation=false
BrokerAddress=
PlatformName=
datamanagement.default_componentcooperation_policy=true
datamanagement.default_componentpulltransfer_policy=true
datamanagement.default_componentpushtransfer_policy=true
java.rmi.server.codebase=
java.security.policy=java.policy
brokermonitor.broker_access_log=
brokermonitor.broker_debug_log=
MyComponentParameter=100     ← パラメタを追加
```

■使用方法（パラメタ取得処理）

```
public class MyComponent implements PFComponent {
    …
    public void method() {
        …
        // パラメタ取得
        String parameter = System.getProperty(“MyComponentParameter”);
        …
    }
    …
}
```

4. コンポーネントの開発概要

プラットフォーム上で動作するコンポーネントの開発を行う手順について説明します。コンポーネントの開発を行うために必要な情報や環境を、コンポーネント開発環境として提供しますので、それを使用して開発を行ってください。

4.1. コンポーネント開発環境

コンポーネント開発環境をインストールすると、導入先のフォルダに以下のファイルが提供されます。

◇開発環境フォルダ名

導入先フォルダ¥developer　※導入先はインストール時に指定

◇提供内容

項目	ファイル名	内容
マニュアル	manual¥	コンポーネント開発関連のドキュメント ・コンポーネント開発ガイド（本資料）
サンプル	sample¥	コンポーネントサンプル ・非GUIコンポーネントサンプル（加算） ・GUIコンポーネントサンプル（ボタン） ・Nativeコンポーネント（加算）
テンプレート	templates¥Component.template	非GUIコンポーネント用テンプレートソース
	templates¥GUIComponent.template	GUIコンポーネント用テンプレートソース
ツール	tools¥MakeResourceFile.bat	リソースファイル変換用バッチ（サンプル）
	tools¥CompileSample.bat	コンパイル用バッチ（サンプル）

※補足

サンプルコンポーネントは、sampleフォルダに以下のような構成で提供されます。

項目	ファイル名	内容
ソースファイル	sample¥src	サンプルコンポーネントのソース ・非GUIコンポーネントサンプル（加算） ・GUIコンポーネントサンプル（ボタン） ・Nativeコンポーネント（加算）
Nativeサンプル ソースファイル	sample¥src_native	サンプルコンポーネントのNativeソース ・Native処理（Native加算コンポーネント）
サンプル アプリケーション	sample¥AP_DATA	サンプルコンポーネントを使用した サンプルアプリケーションファイル ・加算アプリケーション ・Native版加算アプリケーション
JARファイル	sample¥jars	サンプルコンポーネントのJARファイル ※Javaソースをコンパイルしたもの
DLLファイル	sample¥lib	サンプルコンポーネントのDLL ・Native処理（Native加算コンポーネント） ※Cソースをコンパイルしたもの
コンパイル用ファイル	sample¥build.bat	コンパイル用バッチファイル（サンプル）
	sample¥build.xml	Ant用ビルドファイル（サンプル）
	sample¥build-gcc.mak	GCC用Makefile（サンプル）

4.2. コンポーネント開発の手順

1)コンポーネントプログラムの記述

コンポーネントのプログラムを記述します。プログラム作成を行うために、開発環境としてコンポーネント開発を行うためのテンプレートソースファイルを提供します。開発時にはこのテンプレートファイルをコピーして修正を行うことで、本プラットフォームが規定するフレームワークに従った開発が容易に可能となります。テンプレートの使用方法、プログラム記述については、次章の記述を参照してください。

【関連する開発環境ファイル】

- コンポーネント開発ガイド（本資料）
- コンポーネント開発に関連するクラスのドキュメント（Javadoc）
- コンポーネント用テンプレートソース
- サンプルコンポーネント（ソースファイル）

2)コンポーネントプログラムのコンパイル

作成したコンポーネントのコンパイルには、MZ Platform が提供するいくつかのクラスが必要になります。jars フォルダ内 mzplatform.jar をコンパイル時のクラスパスに設定してください。

【関連する開発環境ファイル】

- サンプルコンポーネント（コンパイル用バッチファイル）

3)プラットフォームからの利用

作成したコンポーネントを MZ Platform 上で使用するには、以下の手順が必要となります。これらの設定については、『アプリケーションビルダー操作説明書』に記述されていますので、詳細はそちらを参考にしてください。

①class/jar ファイルの配置【必須】

作成したコンポーネントの class ファイルについては、プラットフォームの導入フォルダの直下以外に格納してください。作成したコンポーネントの jar ファイルについては、任意のディレクトリに格納することができますが、プラットフォームのホームディレクトリの jars ファイルに格納することを推奨します。

②クラスパスの設定【必須】

プラットフォーム実行時のクラスパスに、作成したクラスを格納したディレクトリ名もしくは jar ファイル名を追加します。プラットフォーム実行時のクラスパス設定には、“導入フォルダ¥etc” にあるクラスパス設定ファイル “PlatformClassPath.ini” に記述します。

なお、環境変数 CLASSPATH には、プラットフォームに依存する class/jar ファイルを設定しないでください。

③コンポーネントの登録【必須】

新規コンポーネントを基本コンポーネントと同じようにアプリケーションビルダー上で使用するために、コンポーネント一覧に登録しておくと便利です。コンポーネント一覧は、導入フォルダ内にあるコンポーネント一覧ファイル “PlatformComponents_xx.ini”（xx はロケール名）に記述します。コンポーネント名とクラス名をタブ区切りで一行に記述します。

④コンポーネント情報の登録

アプリケーションビルダー上でメソッド情報やイベント情報を表示させるには、コンポーネント情報を登録する必要があります。アプリケーションビルダーが提供するコンポーネント情報の登録機能を使用して、情報を登録してください。

⑤共有ライブラリの配置

『5.3 ネイティブ処理を行うコンポーネントの開発』で作成された共有ライブラリについては、プラットフォームのホームディレクトリの lib ディレクトリに格納してください。

4.3. サンプルコンポーネントの提供

本プラットフォームのフレームワークに従って構築されたコンポーネントのサンプルを提供します。コンポーネントの構築方法、直列化の実装方法などについて、参考にして下さい。

コンポーネント	クラス
ボタンコンポーネント（GUI）	samples.SampleButtonComponent
加算コンポーネント（非 GUI）	samples.SampleAdditionComponent
加算コンポーネント（C 言語実装）	samples.SampleNativeAdditionComponent

1)ソースファイル

sample¥src¥samples¥SampleButtonComponent.java
sample¥src¥samples¥SampleAdditionComponent.java
sample¥src¥samples¥SampleNativeApplicationComponent.java
sample¥src¥sample¥samples_en.properties
sample¥src¥sample¥samples_ja.properties
sample¥src_native¥samples_SampleNativeAdditionComponent.h
sample¥src_native¥samples_SampleNativeAdditionComponent.c

2)JAR ファイル

sample¥jars¥sample.jar	※コンポーネントのクラス／リソースを含む

3)Native ライブラリファイル

sample¥lib¥sample.dll	※Native コードのライブラリ

4)サンプルアプリケーション

sample¥AP_DATA¥SampleApplication.apl	※Java シリアライズ形式データ
sample¥AP_DATA¥SampleApplication.xml	※独自 XML 形式データ
sample¥AP_DATA¥SampleNativeApplication.apl	※Java シリアライズ形式データ
sample¥AP_DATA¥SampleNativeApplication.xml	※独自 XML 形式データ

5)その他のファイル

sample¥build.bat	※コンパイル/jar 作成用バッチファイルのサンプル
sample¥build.xml	※コンパイル/jar 作成用 ANT ビルドファイル
sample¥build-gcc.mak	※MinGW で共有ライブラリ作成するための makefile

5. コンポーネントの実装手順

5.1. 画面表示を伴うコンポーネントの開発

5.1.1. 設計にあたって

画面表示を伴うコンポーネントの設計時には、通常の機能定義／画面定義に加えて、以下の点を必ず明確にして下さい。

1)継承する Java GUI-Component クラスの決定

これらコンポーネントはApplet や Frame、Panel など、Java 標準の Container クラスに貼り付けられます。従って、必ず ”java.awt.Component” の派生クラスである必要があります。

例）ボタンコンポーネント → “java.awt.Button”, “javax.swing.JButton” などを継承

例）自作描画コンポーネント → “java.awt.Canvas”, “javax.swing.JComponent” などを継承

2)発生するイベント

対象コンポーネントから発生させるイベントを確定します。継承した Java GUI-Component からは様々なイベントが発生されますが、外部との連携を行うイベントを絞り、プラットフォームが提供するイベントのどれを発生させるかを決定してください。（『3.2 イベント』参照）

3)内部データ

コンポーネント内部で保有する表示のためのデータを確定します。Java が提供する GUI コンポーネント（Button など）を継承する場合、表示に関するデータは親クラス内で保持されていますが、グラフ、図面など、自作の描画コンポーネントの場合、表示のためのデータをコンポーネント属性として準備する必要があります。また、表示文字列についてはマルチロケール対応について考慮する必要があり、必要に応じてマルチロケール対応文字列型（PFMultiLocaleString）を使用します。また、他コンポーネントとの連携を行うデータ（グラフの表示データなど）については、プラットフォームが提供するデータ構造である必要があります。（『3.1 共通データ構造』参照）

5.1.2. プログラム開発手順

No	作業内容
1	ソースファイル準備 概要：コンポーネント作成用にテンプレートファイルを提供。 テンプレートファイルには実装しなければならない処理が記述されており、 開発時にはそれに対して必要な修正を加えて開発する。 手順：①下記のファイルをコピーして Java ソースファイルを作成 templates¥GUIComponent.template ②このコンポーネントが使用するリソースファイルを作成する（他との共用可能） 日本語：<ResourceName>_ja.properties 英語　：<ResourceName>_en.properties
2	テンプレート修正 概要：コンポーネント内容に合わせて、テンプレートファイルの修正が必要。 手順：①クラス名 作成するクラスにあわせて、関連箇所を修正。 ・ファイル名 ・class 文（クラス名／継承クラス名） ・各種コメントなど ②コンポーネント名称設定 アプリケーションビルダーで表示されるコンポーネント名称を設定。 メソッド”getComponentName()”の実装部分を修正。（マルチロケール対応対象） ③コンポーネントキー情報設定 アプリケーションビルダーで表示されるコンポーネントキー情報を設定。 複数の同一コンポーネントを識別できる情報を指定する。（ボタンのラベルなど） メソッド”getComponentKey()”の実装部分を修正。
3	イベント対応実装（テンプレート修正） 概要：テンプレートには全てのイベント実装が含まれているため、不要なものを記述削除する。 手順：不要なイベント実装（複数）について以下の記述を削除 ①class 文の implements 句 ②対応するインスタンス変数 ③イベントソース interface 実装メソッド（リスナ取得／リスナ追加／リスナ削除） 【具体例】アクションイベントが不要な場合 ①class 文の implements 句から ”PFActionEventSource” を削除する ②インスタンス変数 ”actionEventSource” を削除する ③”PFActionEventSource” interface 実装メソッドを削除する ・リスナ取得メソッド ”getPFActionListenerList()” を削除する ・リスナ追加メソッド ”addPFActionListener()” を削除する ・リスナ削除メソッド ”removePFActionListener()” を削除する
4	実装 概要：機能設計に従って実装。 手順：作業手順は任意。リソースファイルも編集すること。
5	コンパイル／Jar 作成 概要：ソースコードをコンパイルした後、コンポーネント Jar ファイルを作成する 手順：①ソースコードのコンパイル javac コマンドを使って、ソースコードをコンパイルする ※生成される class ファイルを、ソースコードとは別のフォルダに格納すること ②リソースファイルの変換/コピー native2ascii コマンドを使って、リソースファイルを Unicode エスケープ文字列に変換し、①で生成された class ファイルと同じフォルダにコピーする ③Jar ファイルの作成 jar コマンドを使って、①②で生成されたすべての class ファイル/リソースファイルを Jar ファイルに圧縮する ※サンプルのコンパイル用バッチファイルおよび ANT ビルドファイルを参照

5.1.3. 実装の留意点

画面表示を伴うコンポーネントの実装にあたっては以下の項目に注意してください。これらに従っていない場合、アプリケーションビルダーからの利用ができません。

1)直列化（シリアライズ）

プラットフォーム上の全てのコンポーネントは直列化されます。従って、各コンポーネントは属性や状態をすべてシリアライズでき、またそのデータからデシリアライズ可能でなければなりません。デフォルト機能（ObjectOutputStream::defaultWriteObject(), defaultReadObject()）で対応可能であれば特に対応は不要ですが、必要な場合は writeObject(), readObject(), writeExternal(), readExternal()などのメソッドをオーバーライドしてください。直列化については以下が注意点となります。

- 親クラスのデータを直列化対象とするか（親クラスが直列化可能か）
- デシリアライズ時に行う初期処理がないか
  ※Externalizable 実装の場合はコンストラクタが呼ばれた直後に readExternal()が呼び出されるが、Serializable 実装の場合はコンストラクタは呼ばれないので注意。
- 表示用データは直列化対象とするか（他で保存されていれば保存不要な場合もある）

画面表示を伴うコンポーネントの場合、画面配置情報（配置座標／表示サイズ）もアプリケーション情報として扱われるべきデータとなりますので、必ず直列化対象としてください。

2)イベント発生

イベント発生を行うためのメソッドがイベントソース実装に提供されていますので、各コンポーネント内でイベントを発生させたいタイミングで、そのメソッドを呼び出してください（『3.2.3 イベント関連クラス』参照）。また、イベント発生メソッドは例外をスローしますので、必ず”try～catch”で例外発生を検知し、適切なエラー処理を行ってください（『3.3 例外処理』参照）。

■サンプル

アクションイベントを発生させる処理の例

- **インスタンス変数 “actionEventSource”（PFActionEventSourceImpl）のメソッド起動**
- **例外発生時にはエラーメッセージ表示ユーティリティを使用して、エラー画面表示**

```
try {
   actionEventSource.fireActionPerformed(new PFActionEvent(this));
} catch (PFException ex) {
   // プラットフォーム例外の発生
   PFUtility.displayDetailErrorMessage(this, ex);
} catch (Exception ex) {
   // 例外の発生
   PFUtility.displayDetailErrorMessage(this, "Catch Exception.", ex);
} catch (Error ex) {
   // エラーの発生
   PFUtility.displayDetailErrorMessage(this, "Catch Error.", ex);
}
```

3)派生 Window

コンポーネントから新たに表示される Window はモーダル化し、親コンポーネント（親フレーム）を指定してください。Dialog などの Window は以下のように親 Container をたどって親フレームを指定してください。エラーメッセージ表示（PFUtility::displayErrorMessage()）も、必ず第 1 引数にベースとなるコンポーネントの指定が必要です。

```
Component cmp = this;
while (!(cmp instanceof Frame)) {
    cmp = cmp.getParent();
}
JDialog dialog = new JDialog(cmp, "Sample Dialog", true);
```

4)実行モードと編集モード

アプリケーションビルダーには『アプリケーション実行』モードと『アプリケーション実行（設定可能)』モードの２つの実行モードがあり、コンポーネントの属性に対する設定を許すかどうかを切り替えられるようにしています。よって、各画面表示コンポーネントが提供する属性編集機能は、このモードによって使用可否を判断してもらう必要があります。

ソーステンプレートファイルには、真偽値型のインスタンス変数として“editableFlag”が記述されていますので、このフラグに従って編集機能実装を行ってください。なお、このフラグの設定はアプリケーションビルダーから行います。（メソッド setPropertyEditable()）

```
/** 編集可否フラグ */
private boolean editableFlag;
```

```
/** アクションリスナーの登録 **/
addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        // 右クリック時のみポップアップメニューを表示
        if (SwingUtilities.isRightMouseButton(e)) {
            if (editableFlag == true) {
                // 属性編集用ポップアップメニューの表示
            }
        }
    }
});
```

5)サイズの指定

表示されるサイズは、そのコンポーネントが貼り付けられるパネルに合わせて自動的に設定されますが、本プラットフォームでは『手動配置』機能を提供するため、コンポーネントの生成時（コンストラクタ内)、表示サイズに影響のある属性変更時に、サイズを明示的に設定しておく必要があります。

5.2. 画面表示を伴わないコンポーネントの開発

5.2.1. 設計にあたって

画面表示を伴わないコンポーネントの設計時には、通常の機能定義／画面定義に加えて、以下の点を必ず明確にして下さい。

1)発生するイベント

対象コンポーネントから発生させるイベントを確定します。外部との連携を行うイベントを選択し、プラットフォームが提供するイベントのどれを発生させるかを決定してください。(『3.2 イベント』参照)

2)内部データ

コンポーネント内部で保有するデータを確定します。他コンポーネントとの連携を行うデータ（グラフの表示データなど）については、プラットフォームが提供するデータ構造である必要があります。(『3.1 共通データ構造』参照)

5.2.2. プログラム開発手順

No	作業内容
1	ソースファイル準備 概要：コンポーネント作成用にテンプレートファイルを提供。 テンプレートファイルには実装しなければならない処理が記述されており、 開発時にはそれに対して必要な修正を加えて開発する。 手順：①下記のファイルをコピーして Java ソースファイルを作成 templates¥Component.template ②このコンポーネントが使用するリソースファイルを作成する（他との共用可能） 日本語：<ResourceName>_ja.properties 英語　：<ResourceName>_en.properties
2	テンプレート修正 概要：コンポーネント内容に合わせて、テンプレートファイルの修正が必要。 手順：①クラス名 作成するクラスにあわせて、関連箇所を修正。 ・ファイル名 ・class 文（クラス名／継承クラス名） ・各種コメントなど ②コンポーネント名称設定 アプリケーションビルダーで表示されるコンポーネント名称を設定。 メソッド”getComponentName()”の実装部分を修正。（マルチロケール対応対象） ③コンポーネントキー情報設定 アプリケーションビルダーで表示されるコンポーネントキー情報を設定。 複数の同一コンポーネントを識別できる情報を指定する。（ボタンのラベルなど） メソッド”getComponentKey()”の実装部分を修正。
3	イベント対応実装（テンプレート修正） 概要：テンプレートには全てのイベント実装が含まれているため、不要なものを記述削除する。 手順：不要なイベント実装（複数）について以下の記述を削除 ①class 文の implements 句 ②対応するインスタンス変数 ③イベントソース interface 実装メソッド（リスナ取得／リスナ追加／リスナ削除） 【具体例】アクションイベントが不要な場合 ①class 文の implements 句から ”PFActionEventSource” を削除する ②インスタンス変数 ”actionEventSource” を削除する ③”PFActionEventSource” interface 実装メソッドを削除する ・リスナ取得メソッド ”getPFActionListenerList()” を削除する ・リスナ追加メソッド ”addPFActionListener()” を削除する ・リスナ削除メソッド ”removePFActionListener()” を削除する
4	実装 概要：機能設計に従って実装。 手順：作業手順は任意。リソースファイルも編集すること。
5	コンパイル／Jar 作成 概要：ソースコードをコンパイルした後、コンポーネント Jar ファイルを作成する 手順：①ソースコードのコンパイル javac コマンドを使って、ソースコードをコンパイルする ※生成される class ファイルを、ソースコードとは別のフォルダに格納すること ②リソースファイルの変換/コピー native2ascii コマンドを使って、リソースファイルを Unicode エスケープ文字列に変換し、①で生成された class ファイルと同じフォルダにコピーする ③Jar ファイルの作成 jar コマンドを使って、①②で生成されたすべての class ファイル/リソースファイルを Jar ファイルに圧縮する ※サンプルのコンパイル用バッチファイルおよび ANT ビルドファイルを参照

5.2.3. 実装の留意点

画面表示を伴わないコンポーネントの実装にあたっては以下の項目に注意してください。これらに従っていない場合、アプリケーションビルダーからの利用ができません。

1)直列化（シリアライズ）

プラットフォーム上の全てのコンポーネントは直列化されます。従って、各コンポーネントは属性や状態をすべてシリアライズでき、またそのデータからデシリアライズ可能でなければなりません。デフォルト機能（ObjectOutputStream∷defaultWriteObject(), defaultReadObject()）で対応可能であれば特に対応は不要ですが、必要な場合は writeObject(), readObject(), writeExternal(), readExternal()などのメソッドをオーバーライドしてください。直列化については以下が注意点となります。

- 親クラスのデータを直列化対象とするか（親クラスが直列化可能か）
- デシリアライズ時に行う初期処理がないか

※Externalizable 実装の場合はコンストラクタが呼ばれた直後に readExternal()が呼び出されるが、Serializable 実装の場合はコンストラクタは呼ばれないので注意。

2)イベント発生

イベント発生を行うためのメソッドがイベントソース実装に提供されていますので、各コンポーネント内でイベントを発生させたいタイミングで、そのメソッドを呼び出してください（『3.2.3 イベント関連クラス』参照）。また、イベント発生メソッドは例外をスローしますので、必ず”try～catch”で例外発生を検知し、適切なエラー処理を行ってください（『3.3 例外処理』参照）。

■サンプル
アクションイベントを発生させる処理の例
- **インスタンス変数 “actionEventSource”（PFActionEventSourceImpl）のメソッド起動**
- **例外発生時には呼び出し下に例外を通知**

```
try {
    actionEventSource.fireActionPerformed(new PFActionEvent(this));
} catch (PFException ex) {
    // プラットフォーム例外の発生
    throw ex;
} catch (Exception ex) {
    // 例外の発生
    throw new PFComponentException(this, ex, "Catch Exception.");
} catch (Error ex) {
    // エラーの発生
    throw new PFComponentException (this, ex, "Catch Error.");
}
```

3)派生 Window

コンポーネントから新たに表示される Window はモーダル化し、親コンポーネント（親フレーム）を指定してください。Dialog などの Window は以下のように親 Container をたどって親フレームを指定してください。エラーメッセージ表示（PFUtility∷displayErrorMessage()）も、必ず第 1 引数にベースとなるコンポーネントの指定が必要です。

```
Component cmp = this;
while (!(cmp instanceof Frame)) {
    cmp = cmp.getParent();
}
JDialog dialog = new JDialog(cmp, "Sample Dialog", true);
```

5.3. ネイティブ処理を行うコンポーネントの開発

本プラットフォームは Java によって実装されており、コンポーネントも Java で作成する必要があります。したがって Java 以外の言語で記述されたコンポーネントをそのまま利用することはできません。

Java 以外の言語で記述されたコンポーネントを利用するには、Java の JNI 機能（Java Native Interface）を使用してください。JNI を使用すれば、

1)Java 以外の言語で作成したプログラムを呼び出す
2)OS 依存の処理を行う
3)既存のライブラリにアクセスする

などといったことが可能になります。

ここでは、C 言語を使ってネイティブ処理を行うコンポーネントの開発について説明します。

5.3.1. 設計にあたって

ネイティブ処理を行うコンポーネント開発を行う際には、通常の Java コンポーネント開発(5.1.1 設計にあたってぉよび 5.2.1 設計にあたってを参照)を行う場合に加えて、以下の点に注意する必要があります。

1)Java とのインターフェイス

Java で記述されたクラス内で、ネイティブで処理を行うメソッド(ネイティブメソッド)を宣言してください。

JNI に関する具体的なことについては以下を参照してください。

Java2SDK Documentation: Java Native Interface
http://docs.oracle.com/javase/1.4.2/docs/guide/jni/index.html (英語)
http://docs.oracle.com/javase/jp/1.4/guide/jni/index.html (日本語)

5.3.2. プログラム開発手順

No	作業内容
1	Java でソースコードを作成 概要：コンポーネントを Java で作成する。 手順：①5.1.2 プログラム開発手順および 5.2.2 プログラム開発手順の手順でコンポーネントを作成する。 ②ネイティブメソッドを宣言する。 【例】

```
package sample;
public class SampleClass {
    …
    // ネイティブメソッドの宣言
    public native void method1();
    public native void method2();
}
```

2	Java で書かれたソースコードをコンパイル 概要：No.1 で記述されたコンポーネントのソースコードをコンパイルする。 手順：①javac コマンドを利用して、Java で記述されたソースコードをコンパイルする。 ②class ファイルが生成されているかどうかを確認する。
3	ヘッダファイルを生成 概要：JNI スタイルのヘッダファイルを生成する。 手順：①javah コマンドを利用して、ヘッダファイル(.h ファイル)を生成する。 ②ヘッダファイルが生成されているかどうかを確認する。 【例】パッケージ名 sample、クラス名 SampleClass にネイティブメソッド method1, method2 が宣言されている場合、 ・ヘッダファイル sample_SampleClass.h が生成される。 ・sample_SampleClass.h に以下の関数プロトタイプが宣言されている。

```
JNIEXPORT void JNICALL
Java_sample_SampleClass_method1(JNIEnv *, jobject);
JNIEXPORT void JNICALL
Java_sample_SampleClass_method2(JNIEnv *, jobject);
```

4	C 言語でソースコードを作成 概要：C 言語でネイティブメソッドを実装する。 手順：①C 言語のソースファイルを準備する。 ソースファイル名は任意。拡張子を.c とすること。 ②ソースファイルの先頭で、No.3 で生成されたヘッダファイルをインクルードする。 ③ヘッダファイルに宣言されている関数プロトタイプに基づいて、関数を実装する。 【例】No.3 でヘッダファイル sample_SampleClass.h が生成されたものとする。 sample_SampleClass.c を作成して、以下のように記述する。

```
#include “sample_SampleClass.h”

JNIEXPORT void JNICALL
Java_sample_SampleClass_method1(JNIEnv *env, jobject obj) {
    // 実装すること
}
JNIEXPORT void JNICALL
Java_sample_SampleClass_method2(JNIEnv *env, jobject obj) {
    // 実装すること
}
```

5	C 言語で書かれたソースコードをコンパイル/共有ライブラリを生成 概要：No.3 で生成されたヘッダファイルと No.4 で記述されたソースコードをコンパイルして、共有ライブラリを生成する。 手順：①コンパイルして、共有ライブラリを生成する。 Windows ならば Visual C++、Borland C、Cygwin、MinGW 等を使ってコンパイルする。Linux ならば gcc でコンパイルする。 ②共有ライブラリが生成されていることを確認する。

5.3.3. 実装の留意点

ネイティブ処理を行うコンポーネントを実装するにあたり、まず通常の Java コンポーネント開発時の留意点(5.1.3 実装の留意点と 5.2.3 実装の留意点を参照)に注意してください。また、以下にあげる項目に注意してください。これらに従っていない場合、ネイティブメソッドを呼び出すことはできません。

1)ネイティブメソッド宣言(Java 側)

ネイティブメソッドを宣言するときは、修飾子 native を記述する必要があります。

ネイティブメソッドの実装を記述するとコンパイルエラーになります。

【例】メソッド public void method1()をネイティブメソッドとして宣言する。

○

```
public native void method1();
```

×

```
public native void method1() {
    // ネイティブメソッドの中に処理を記述してはならない
}
```

2)ライブラリのロード(Java 側)

ネイティブメソッドを呼び出す前にライブラリをロードする必要があります。

ライブラリをロードするには、System.loadLibrary()メソッドを呼び出してください。

System.loadLibrary() の引数には、ライブラリ名を指定してください。

【例】ライブラリ名を sample としたとき、

```
try {
    System.loadLibrary(“sample”);
} catch (Throwable t) {
    // 共有ライブラリファイルが存在しなかった場合の処理
}
```

System.loadLibrary()の呼び出しについては、static イニシャライザーで行うことが多いです。

3)ヘッダファイル(C/C++側)

ネイティブメソッドの名称・引数・戻値・修飾子、またはネイティブメソッドを含んでいるクラス・パッケージの名称が変更された場合は、必ず javah でヘッダファイルを生成し直し、さらに C/C++側の関数定義を、生成し直したヘッダファイルに記述されている関数プロトタイプに合うように修正してください。

javah によって生成されたヘッダファイル内の関数プロトタイプ宣言を絶対に変更しないでください。

5.3.4. 共有ライブラリ生成時の留意点

共有ライブラリを生成するときは、以下の点に注意してください。

1)コンパイル

コンパイルするには、J2SDK に含まれている jni.h や jni_md.h などが必要です。コンパイラのオプションとして、以下のディレクトリをインクルードディレクトリに指定してください。

Windows の場合
(J2SDK のインストール先ディレクトリ)¥include
(J2SDK のインストール先ディレクトリ)¥include¥win32
Linux の場合
(J2SDK のインストール先ディレクトリ)/include
(J2SDK のインストール先ディレクトリ)/include/linux

2)ライブラリ名と共有ライブラリファイル名の関係(共有ライブラリ生成時)

共有ライブラリファイルを生成する際、共有ライブラリファイル名を以下のように命名する必要があります。

OS	共有ライブラリファイル名
Windows2000/XP/Vista/7/8	(ライブラリ名).dll
Linux	lib(ライブラリ名).so

例. ライブラリ名が sample であるとき、共有ライブラリファイル名は以下のとおり。
Windows2000/XP/Vista/7/8 のとき、sample.dll
Linux のとき、libsample.so

5.4. XML 入出力機能の実装

本プラットフォームのアプリケーション保存形式には、Java 標準シリアライズ形式と XML テキスト形式の 2 種類が準備されています。Java バージョンアップやコンポーネントプログラム変更などが発生した場合でもアプリケーションの互換性を確保するため、Version 2.0 からは XML テキスト形式が標準となりました。したがって、コンポーネントプログラムを作成する場合には、XML テキスト形式による入出力機能を用意することが必要です。

5.4.1. XML 入出力インターフェイス

XML テキストによる入出力機能を用意するため、各コンポーネントにて以下のインターフェイスを実装します。

◇インターフェイス名

`jp.go.aist.dmrc.platform.base.xml.PFXMLSerializable`

◇定義メソッド

```
public void writeXML(PFXMLGenerator out)
public void readXML(PFXMLLoader in)
```

このインターフェイスを実装したコンポーネントに対し、MZ Platform は XML 入出力時に以下のような動作を行います。

出力時：標準の属性出力終了後、上記インターフェイスの writeXML()メソッドを呼び出す
入力時：標準の属性出力終了後、上記インターフェイスの readXML()メソッドを呼び出す

付属のテンプレートファイル（Component.template、GUIComponent.template）には、標準として以下の実装がなされています。

```
public void writeXML(PFXMLGenerator out) throws IOException {
  out.defaultWriteXML();
}

public void readXML(PFXMLLoader in){
  in.defaultReadXML();
}
```

この標準の XML 入出力機能では、コンポーネントの属性情報は setter/getter メソッドの組を検出し、それを文字列表現によって入出力を行っています。標準では対応していないデータ型の属性や setter/getter メソッドの存在しない属性の保存処理、属性設定時の処理を行うためには、5.4.2 以降の説明に従い、必要な処理を記述する必要があります。

<u>setter と getter について</u>

コンポーネントなどのオブジェクトに対して、ある属性値を設定するメソッドを setter、取得するメソッドを getter といいます。標準 XML 入出力機能では、以下の条件をすべて満足する２つの public メソッドが存在するとき、これらを組になる setter/getter と見なします。

(1) “set”で始まる１引数メソッドと”get”で始まる０引数メソッド
(2) メソッド名の４文字目以降が大文字／小文字の区別も含めて同一

(3) “set”で始まるメソッドの引数の型と、”get”で始まる０引数メソッドの戻り値の型が同一

ただし、属性の型が boolean の場合に限り、以下の３つの条件を満足するメソッドも setter/getter の組と見なされます。

(1) “set”で始まる１引数メソッドと”is”で始まる０引数メソッド
(2) “set”で始まるメソッド名の４文字目以降と”is”で始まるメソッド名の３文字目以降が大文字／小文字の区別も含めて同一
(3) “set”で始まるメソッドの引数の型と、”get”で始まる０引数メソッドの戻り値の型がともに boolean

例えば、以下の①、②、③は組になる setter/getter ですが、④と⑤はそうではありません。

① void setCount(int)と int getCount()（組になる setter/getter）
② void setEnabled(boolean)と boolean isEnabled()（組になる setter/getter）
③ void setTime(Date)と Date getTime()（組になる setter/getter）
④ void setLength(long)と double getLength()（引数と戻り値の型が一致しません）
⑤ void setValueAt(Integer, int)と Integer getValueAt(int)（引数の数が、それぞれ１と０ではありません）

setter 名および getter 名は、Java コーディング規約に基づき、その４文字目（メソッド名が”is”で始まる場合は３文字目）を大文字としてください。

5.4.2. XML 入出力の実装方法

各コンポーネントでXML入出力インターフェイスを実装するには、XML入出力用に提供されるユーティリティのメソッドを使用します。

1)XML 出力メソッド

PFXMLSerializable インターフェイスにて定義されるメソッド writeXML(PFXMLGenerator)の実装として、出力したい属性を以下のメソッドを使用して記述します。それぞれのメソッドの第一引数は属性名、第二引数はデータ値です。

出力ユーティリティ（PFXMLGenerator）メソッド
void writeBigDecimalPropertyValue(String, BigDecimal)
void writeBigIntegerPropertyValue(String, BigInteger)
void writeBooleanPropertyValue(String, Boolean)
void writeBorderPropertyValue(String, Border)
void writeBytePropertyValue(String, Byte)
void writeCharacterPropertyValue(String, Character)
void writeClassPropertyValue(String, Class)
void writeColorPropertyValue(String, Color)
void writeDatePropertyValue(String, Date)
void writeDimensionPropertyValue(String, Dimension)
void writeDoublePropertyValue(String, Double)
void writeFloatPropertyValue(String, Float)
void writeFontPropertyValue(String, Font)
void writeIconPropertyValue(String, Icon)
void writeImageIconPropertyValue(String, ImageIcon)
void writeImagePropertyValue(String, Image)
void writeIntegerPropertyValue(String, Integer)
void writeLongPropertyValue(String, Long)
void writePFMultiLocaleStringPropertyValue(String, PFMultiLocaleString)
void writePFObjectListPropertyValue(String, PFObjectList)
void writePFObjectNetworkPropertyValue(String, PFObjectNetwork)
void writePFObjectTablePropertyValue(String, PFObjectTable)
void writePFObjectTreePropertyValue(String, PFObjectTree)
void writePFXMLSerializablePropertyValue(String, PFXMLSerializable)
void writePointPropertyValue(String, Point)
void writePrimitiveBooleanPropertyValue(String, boolean)
void writePrimitiveBytePropertyValue(String, byte)
void writePrimitiveCharPropertyValue(String, char)
void writePrimitiveDoublePropertyValue(String, double)
void writePrimitiveFloatPropertyValue(String, float)
void writePrimitiveIntPropertyValue(String, int)
void writePrimitiveLongPropertyValue(String, long)
void writePrimitiveShortPropertyValue(name, short)

void writeSerializableDataPropertyValue(String, Object)
void writeShortPropertyValue(String, Short)
void writeStringPropertyValue(String, String)

2)XML 入力メソッド

PFXMLSerializable インターフェイスにて定義されるメソッド readXML(PFXMLLoader)の実装として、writeXML()の実装に対応した属性を以下のメソッドを使用して入力します。引数は属性名です。

これらのメソッドを使用して属性を入力する場合には、指定した属性名に対応する getter（36 ページ）を実装する必要があります。getter は、以下の条件を満足しなければなりません。

(1) “get”で始まる０引数のメソッドで、戻り値の型が入力する値の型と一致。ただし、型が boolean の場合に限り、”is”で始まる０引数のメソッドでもよい。
(2) メソッド名の４文字目以降（メソッド名が”is”で始まる場合は３文字目以降）が、属性名の頭文字のみ大文字に代えたものと、大文字／小文字の区別を含めて完全に一致。

入力ユーティリティ（`PFXMLLoader`）メソッド
`BigDecimal readBigDecimalPropertyValue(String)`
`BigInteger readBigIntegerPropertyValue(String)`
`Boolean readBooleanPropertyValue(String)`
`Border readBorderPropertyValue(String)`
`Byte readBytePropertyValue(String)`
`Character readCharacterPropertyValue(String)`
`Class readClassPropertyValue(String)`
`Color readColorPropertyValue(String)`
`Date readDatePropertyValue(String)`
`Dimension readDimensionPropertyValue(String)`
`Double readDoublePropertyValue(String)`
`Float readFloatPropertyValue(String)`
`Font readFontPropertyValue(String)`
`Icon readIconPropertyValue(String)`
`ImageIcon readImageIconPropertyValue(String)`
`Image readImagePropertyValue(String)`
`Integer readIntegerPropertyValue(String)`
`Long readLongPropertyValue(String)`
`PFMultiLocaleString readPFMultiLocaleStringPropertyValue(String)`
`PFObjecrList readPFObjecrListPropertyValue(String)`
`PFObjectNetwork readPFObjectNetworkPropertyValue(String)`
`PFObjectTable readPFObjectTablePropertyValue(String)`
`PFObjectTree readPFObjectTreePropertyValue(String)`
`PFXMLSerializable readPFXMLSerializablePropertyValue(String)`
`Point readPointPropertyValue(String)`
`boolean readPrimitiveBooleanPropertyValue(String)`
`byte readPrimitiveBytePropertyValue(String)`
`char readPrimitiveCharPropertyValue(String)`
`double readPrimitiveDoublePropertyValue(String)`
`float readPrimitiveFloatPropertyValue(String)`
`int readPrimitiveIntPropertyValue(String)`

long readPrimitiveLongPropertyValue(String)
short readPrimitiveShortPropertyValue(String)
Object readSerializableDataPropertyValue(String)
Short readShortPropertyValue(String)
String readStringPropertyValue(String)

5.4.3. XML 入出力機能実装のサンプル

以下のコンポーネントについて、XML 入出力機能の実装を行ってみます。

```
 1 public class MyComponent implements PFComponent, PFXMLSerializable {
 2    private int numberField;
 3    private String stringField;
 4    private MyClass objectField;
 5 
 6    public int getNumberField() {
 7       return this.numberField;
 8    }
 9    public void setNumberField(int number) {
10       this.numberField = number;
11    }
12 
13    public String getStringField() {
14       return this.stringField;
15    }
16 
17    public MyClass getObjectField() {
18       return this.objectField;
19    }
20    public void setObjectField(MyClass object) {
21       this.objectField = object;
22    }
23 
24    public void writeXML(PFXMLGenerator out) throws IOException {
25       out.defaultWriteXML();
26    }
27 
28    public void readXML(PFXMLLoader in) {
29       in.defaultReadXML();
30    }
31 
32 }
```

このままの状態で XML 出力すると、標準対応されているデータ型で setter/getter の組が揃っている、“numberField”（Line.2）のみが出力され、setter のない“stringField”（Line.3）や、標準対応されていない MyClass 型の“objectField”（Line.4）については出力されません。

そこで先述のメソッドを使用して writeXML（Line.24）、readXML（Line.28）に追加実装を行うことにより、“stringField”、“objectField”を出力するようにします。ただし、MyClass クラスは Serializable であることを前提としてください。実装例を以下に示します。

```
public class MyComponent implements PFComponent, PFXMLSerializable {
   private int numberField;
   private String stringField;
   private MyClass objectField;

   public int getNumberField() {
      return this.numberField;
   }
   public void setNumberField(int number) {
      this.numberField = number;
   }

   public String getStringField() {
      return this.stringField;
   }

   public MyClass getObjectField() {
      return this.objectField;
   }
   public void setObjectField(MyClass object) {
      this.objectField = object;
   }

   public void writeXML(PFXMLGenerator out) {
      out.defaultWriteXML();
      // 実装追加
       out.writeStringPropertyValue(”stringField”,this.stringField)
;
       out.writeSerializableDataPropertyValue(”objectField”,
                                                 this.objectField);
   }

   public void readXML(PFXMLLoader in) {
      in.defaultReadXML();
      // 実装追加
       this.stringField = in.readStringPropertyValue(”stringField”);
       this.objectField
        =(MyClass)in.readSerializableDataPropertyValue(”objectField”);
   }

}
```