IBM ILOG Views

Gadgets V5.3

ユーザ・マニュアル

2009 年 6 月

著作権の告知

商標

IBM、IBM ロゴ、ibm.com、Websphere、ILOG、ILOG のデザイン、および CPLEX は、世界中の多くの国の管轄権で登録されている International Business Machines Corp. の商標または登録商標です。その他の製品およびサービス名は、IBM またはその他の企業の商標です。IBM 社の現在の商標一覧は、http://www.ibm.com/legal/copytrade.shtml にある Copyright and trademark information (著作権と商標についての情報) にあります。

Adobe、Adobe のロゴ、PostScript、および PostScript のロゴは、米国およびその他の国における Adobe Systems Incorporated の商標または登録商標です。

Linux は、米国およびその他の国における Linus Torvalds の登録商標です。

Microsoft、Windows、Windows NT、および Windows のロゴは、米国およびその他の国における Microsoft Corporation の商標です。

Java およびすべての Java に基づいた商標とロゴは、米国およびその他の国の Sun Microsystems, Inc. の商標です。

その他の企業、製品およびサービス名は、その他の企業の商標またはサービス商標です。

告知

詳細は、インストールした製品の <*installdir*>/license/notices.txt を参照してください。

目次

本書について

*このユーザ・マニュアル*では、IBM® ILOG® Views Controls の使用方法について説明します。IBM ILOG Views Controls を構成する 3 パッケージ、IBM ILOG Views Studio、IBM ILOG Views Gadgets、および IBM ILOG Views Application Framework について説明します。

前提事項

本書では、特定のウィンドウシステムを含め、ユーザが IBM® ILOG® Views を使用する PC や UNIX 環境について精通していることが前提となっています。IBM ILOG Views は C++ 開発者用に作成されているため、このマニュアルでは、ユーザが C++ のコードを作成できること、および C++ の開発環境について精通しており、ファイルやディレクトリの操作、テキスト・エディタの使用、C++ プログラムのコンパイルおよび実行ができることも前提となっています。

マニュアル構成

*このユーザ・マニュアル*では、IBM® ILOG® Views Controls に含まれる Gadgets パッケージの使用方法について説明します。このマニュアルは 2 部構成となっており、付録も含まれています。

第 I 部、*IBM ILOG Views Studio を使った GUI アプリケーションの作成*では、Gadgets 拡張機能がインストールされている IBM ILOG Views Studio の使用方法について説明します。次の章から構成されています。

- 1 章 *IBM ILOG Views Studio の Gadgets 拡張機能の概要*
- 2 章*ガジェット・パネルの編集*
- 3 章*アプリケーションの編集*
- 4 章*生成されたコードの使用*
- 5 章 *IBM ILOG Views Studio の Gadgets 拡張機能をカスタマイズする*
- 6 章 *IBM ILOG Views Studio の拡張*
- 7 章*詳細設定クラスの使用*

第 II 部、*IBM ILOG Views Gadgets* では、IBM ILOG Views Gadgets を組み込んだアプリケーション開発に関する情報を提供します。次の章から構成されています。

- 8 章 *IBM ILOG Views Gadgets の概要*
- 9 章*ガジェットの理解*
- 10 章*ダイアログ*
- 11 章*共通ガジェットの使用*
- 12 章*ガジェット・アイテム*
- 13 章*メニュー、メニュー・バーとツールバー*
- 14 章*マトリックス*
- 15 章*ペイン*
- 16 章*ペインおよびコンテナのドッキング*
- 17 章*ビュー・フレーム*
- 18 章*ルック・アンド・フィールのカスタマイズ*

付録 A、*状態の編集* では、IBM ILOG Views Studio のステータス機構の使用例を提供します。

表記法

書体の規則

以下の書体に関する規則は、このマニュアル全体に適用されます。

- コードの引用やファイル名は `courier` 書体で記載されます。

- ユーザが入力する項目は、courier 書体で記載されます。
- 初めて登場する用語の中には、*斜体*で記載されているものがあります。

命名規則

以下の命名規則は、マニュアル全体を通して API に適用されます。

- ライブラリで定義されている型、クラス、関数、マクロの名前は `Ilv` で始まります。
- クラス名、およびグローバル関数は、最初の文字が大文字で表された連結語として記載されます。

  class `IlvDrawingView`;

- 仮想および通常メソッドの名前は小文字で始まります。スタティック・メソッドの名前は大文字で始まります。例 :

  ```
  virtual IlvClassInfo* getClassInfo() const;
  ```

  ```
  static IlvClassInfo* ClassInfo*() const;
  ```

I 部

IBM ILOG Views Studio を使った GUI アプリケーションの作成

第 1 部では、Gadgets 拡張機能がインストールされた IBM® ILOG® Views Studio の使用法について説明します。

章

1

IBM ILOG Views Studio の Gadgets 拡張機能の概要

この章では、IBM® ILOG® Views Studio の Gadgets 拡張機能を紹介します。以下のトピックに関する情報が記載されています。

- *GUI アプリケーションの読み込みおよびGUI generation プラグイン*
- *メイン・ウィンドウ*
- *パレット・パネル*
- *ガジェット拡張機能コマンド*
- *ガジェット拡張機能パネル*

> ***メモ:*** *IBM ILOG Views Studio の Gadgets 拡張機能の使用に関する章では、ユーザが IBM ILOG Views Studio ユーザ・マニュアルに記載されている内容に精通していることを前提としています。*

GUI アプリケーションの読み込みおよび GUI generation プラグイン

IBM® ILOG® Views の Gadgets パッケージをインストールしたら、GUI アプリケーション・プラグインおよび GUI Generaion プラグインを IBM® ILOG® Views Studio で使用することができます。

`ivfstudio` を `-selectPlugIns` コマンド・ライン・パラメータで起動します。IBM ILOG Views Studio プラグイン・ダイアログ・ボックスが表示されたら、GUI アプリケーション (`smguiapp`)、GUI Generation (`smguigen`) チェック・ボックスを選択して [OK] をクリックします。

Studio メイン・ウィンドウから SelectPlugins コマンドを実行して、IBM ILOG Views Views Studio プラグイン・ダイアログ・ボックスを表示することもできます。次に GUI アプリケーション (`smguiapp`) および GUI Generation ((`smguigen`) チェック・ボックスを選択して、[OK] をクリックします。

メイン・ウィンドウ

アプリケーションを起動させると、次のように、IBM® ILOG® Views Studio のメイン・ウィンドウが開きます。

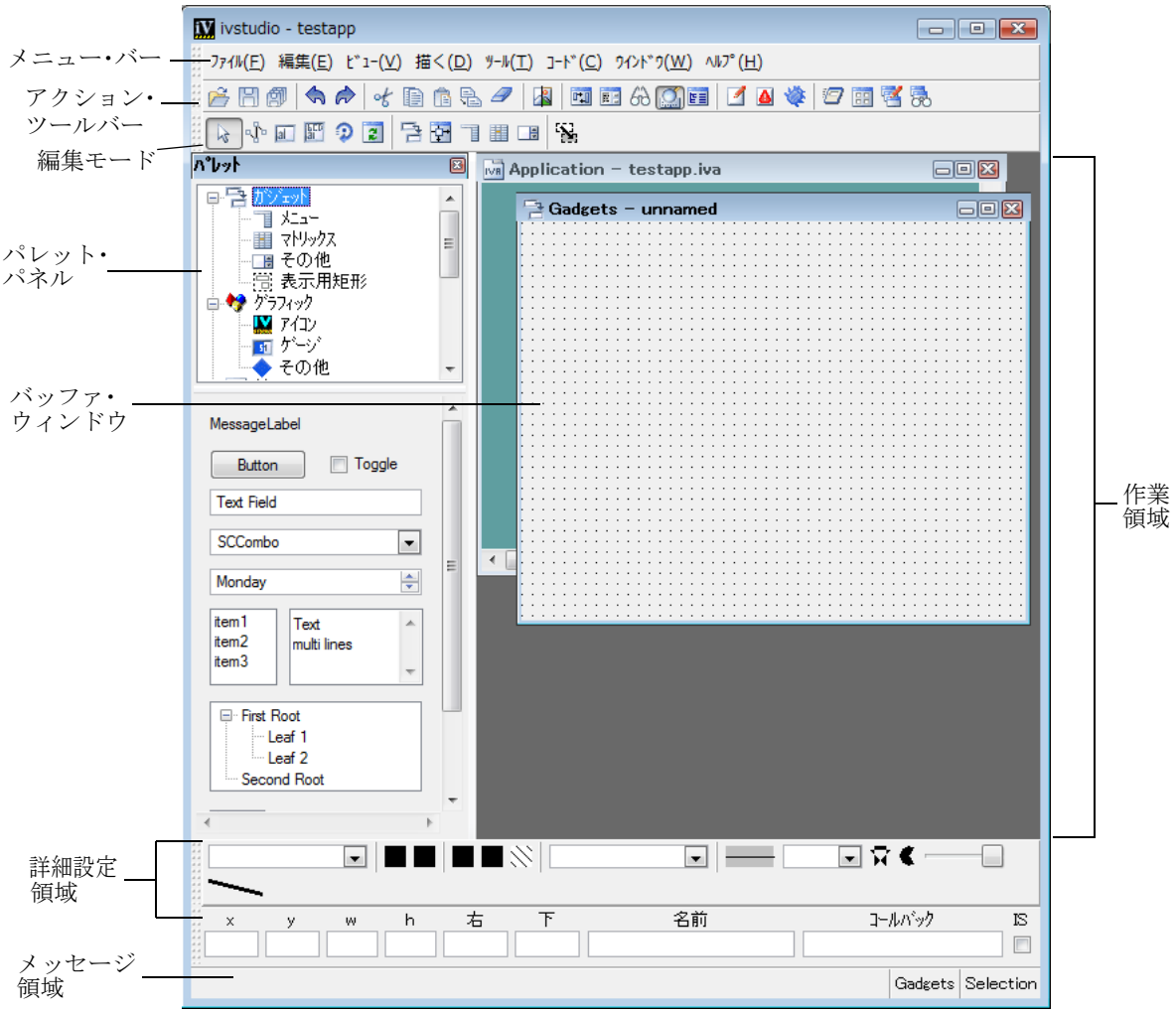


図 1.1 *起動時の IBM ILOG Views Studio メイン・ウィンドウおよびGadgets 拡張機能*

メイン・ウィンドウは、Foundations パッケージだけがインストールされている場合とあまり変わりません。ただし、Gadgets パッケージでは、追加バッファ・ウィンドウ、パレット・パネルでの追加パレット、インターフェースのメニュー・バーおよびツールバーの追加項目へアクセスできるようになっています。

バッファ・ウィンドウ

アプリケーションおよびパネルは、メイン・ウィンドウに表示されているバッファ・ウィンドウに作成されます。カレント・バッファ・タイプはメイン・ウィンドウの一番下に表示されます。

IBM® ILOG® Views Studio の Gadgets 拡張機能では、バッファの次のタイプを編集することができます。

- ガジェット
- 2D グラフィック
- アプリケーション

空の Gadgets バッファおよび空のアプリケーション・バッファが、IBM ILOG Views Studio を起動したときにデフォルトで表示されます。

新規バッファ・ウィンドウを作成するとき、そのタイプ (ガジェット、2D グラフィック、あるいはアプリケーション) を [ファイル] > [新規] メニューを選択して指定します。

メイン・ウィンドウに読み込まれているカレント・バッファ間を切り替えると、次の違いがわかります。

- それぞれのバッファ・タイプには独自の編集モード・セットがあります。カレント・バッファを変更するとき、ツールバーのアイコンとして利用可能な編集モードも変更します。
- 特定のコマンドの振る舞いは、カレント・バッファによって異なります。たとえば、Test コマンドは、ガジェット・バッファを編集している場合、パネルのみをテストしますが、アプリケーション・バッファを編集している場合は、アプリケーションのすべてのパネル・インスタンスをテストします。

Gadgets バッファ・ウィンドウ

Gadgets バッファ・ウィンドウは、パネル・クラスの編集に使用されます。これにより、`IlvGadgetContainer` オブジェクトの内容を編集できます。ガジェットをパレット・パネルからドラッグして、Gadgets バッファ・ウィンドウをアクティブにすることができます。

新規 Gadgets バッファ・ウィンドウを開くには、次の処理を行います。

1. [ファイル] メニューから [新規] を選択します。
2. 表示されたサブメニューで [ガジェット] を選択します。

`IlvGadgetContainer` によって生成された `.ilv` ファイルを開くと、Gadgets バッファ・ウィンドウが自動的に開きます。

Gadgets バッファ・ウィンドウの編集に関する詳細は、2 章 *ガジェット・パネルの編集* を参照してください。

2D Graphics バッファ・ウィンドウ

2D Graphics バッファは、Foundation パッケージのデフォルトです。
IBM ILOG Views Studio の Gadgets 拡張機能で引き続き利用できます。これにより IlvManager あるいは IlvContainer の内容を編集できます。オブジェクトの読み込み、編集、保存に IlvManager を使用します。

新規 2D Graphics バッファ・ウィンドウを作成するには、次の処理を行います。

1. [ファイル] メニューから [新規] を選択します。
2. 表示されたサブメニューで [2D グラフィック] を選択します。

   このウィンドウを開くには、[ツール] メニューから [コマンド] を選択して表示する [コマンド] パネルから NewGraphicBuffer コマンドを実行することもできます。

IlvManager によって生成された .ilv ファイルを開くと、2D Graphics バッファ・ウィンドウが自動的に開きます。

アプリケーション・バッファ・ウィンドウ

アプリケーション・バッファにより、IlvApplication の内容およびプロパティを編集できます。

新規アプリケーション・バッファ・ウィンドウを作成するには、次の処理を行います。

1. [ファイル] メニューから [新規] を選択します。
2. 表示されたサブメニューで [アプリケーション] を選択します。

   [ツール] メニューから [コマンド] を選択して表示する [コマンド] パネルから NewApplication コマンドを実行することもできます。

.iva ファイルを開くと、アプリケーション・バッファはその内容を破棄して新しく開かれたアプリケーションを編集します。

アプリケーションの編集

IBM ILOG ViewsStudio で、アプリケーションは他のタイプのバッファと同じ方法で編集することができます。1 度に開くことができるのは 1 つのアプリケーションだけです。新しいアプリケーションを開くと、自動的に他の開かれているアプリケーションが閉じます。

IBM ILOG Views Studio を起動すると、testapp という名前のデフォルト・アプリケーション・バッファ・ウィンドウが開かれます。

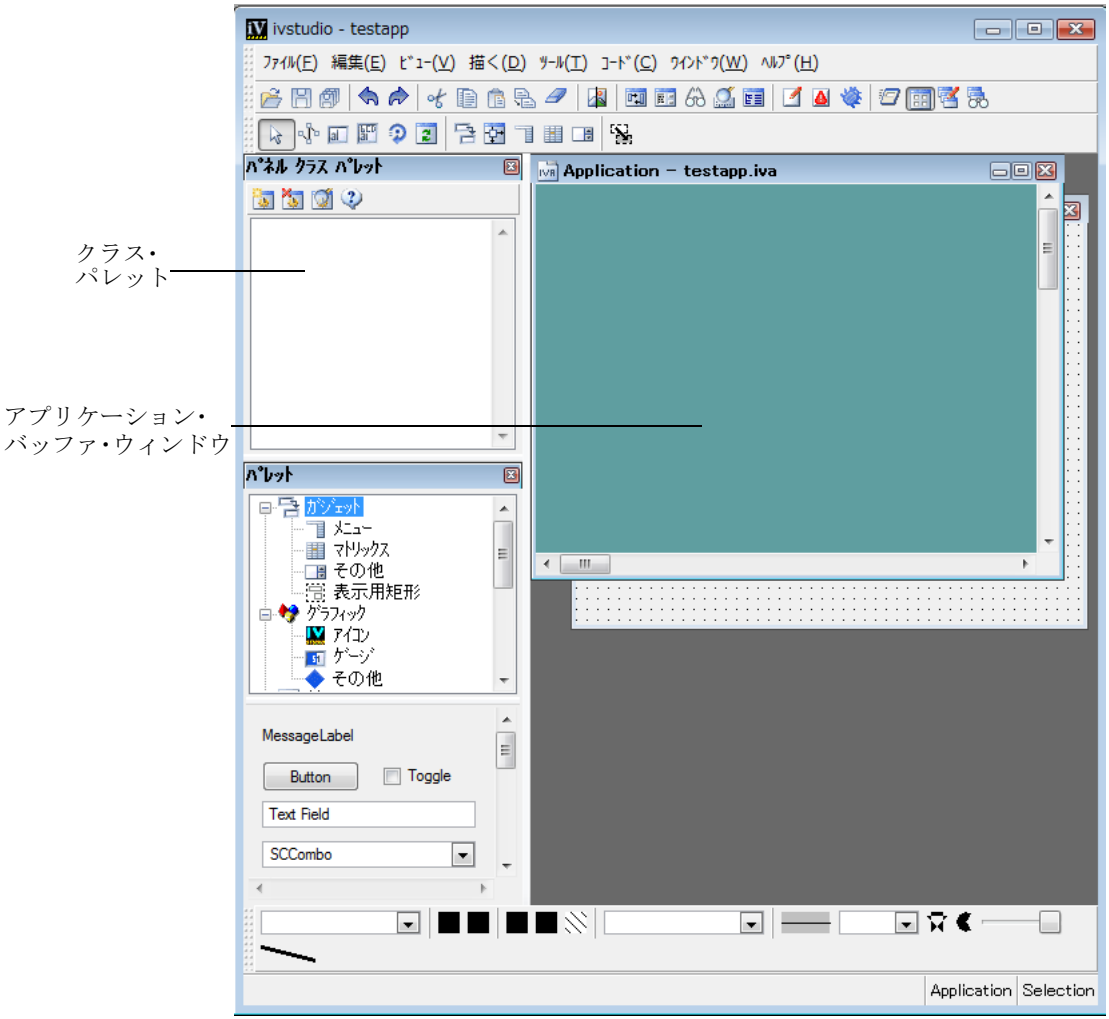


図1.2　アプリケーション・バッファ・ウィンドウ

クラス・パレット経由でパネル・クラス・インスタンスを追加してアプリケーションを編集することができます。クラス・パレットを使って、パネル・クラスの作成、詳細設定または削除を行います。パネル・クラス・インスタンスを作成するために、クラス・パレットで利用できるパネル・クラスを直接アプリケーション・バッファ・ウィンドウにドラッグすることができます。

アプリケーションの編集方法の詳細については、3 章 *アプリケーションの編集* を参照してください。

メニュー・バー

Gadgets パッケージをインストールすると、メイン・ウィンドウのメニュー・バーから追加コマンドが利用できます。特に、アプリケーションの C++ コード生成用コマンドへのアクセスを提供するアプリケーション・メニューが追加されていることがわかります。

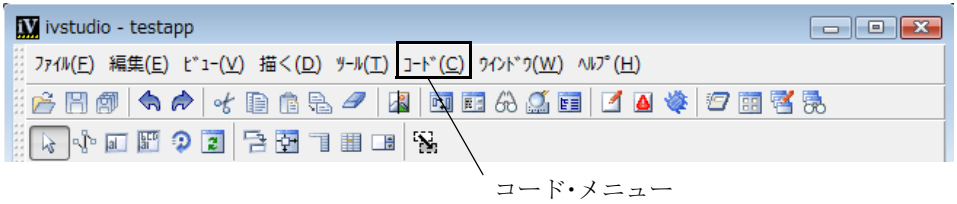


図 1.3 *IBM ILOG Views Studio ガジェット拡張機能メニュー・バー*

以下の表は、メニュー・バーを介して実行できる追加コマンドをまとめたものです。これらのコマンドの詳細については、コマンドをアルファベット順に並べた 40 ページの *ガジェット拡張機能コマンド* を参照してください。

[ファイル] メニュー・コマンド

メニュー項目	コマンド
新規 > ガジェット	*NewGadgetBuffer*
新規 > アプリケーション	*NewApplication*
新規 > デフォルト・アプリケーションの作成	*MakeDefaultApplication*

アプリケーション・メニュー・コマンド

メニュー項目	コマンド
テスト・パネル	*TestPanel*
パネル・クラスへ登録	*NewPanelClass*
クラス・パレット	*ShowClassPalette*
パネル・クラスの詳細設定	*ShowPanelClassInspector*
パネルのサブクラスを生成	*GeneratePanelSubClass*

メニュー項目	コマンド
カレント・バッファを追加	*AddPanel*
詳細設定パネル	*InspectPanel*
アプリケーションの詳細設定	*ShowApplicationInspector*
テスト	*TestApplication*
コード生成	*Generate*
すべて生成	*GenerateAll*
アプリケーションのコード生成	*GenerateApplication*
パネルのクラスを生成	*GeneratePanelClass*
Makefile の生成	*GenerateMakeFile*

ウィンドウ・メニュー・コマンド

メニュー項目	コマンド
アプリケーションの編集	*EditApplication*

アクション・ツールバー

アクション・ツールバーは追加アイコンを含んでおり、これによりガジェット拡張機能のコマンドに迅速にアクセスできます。

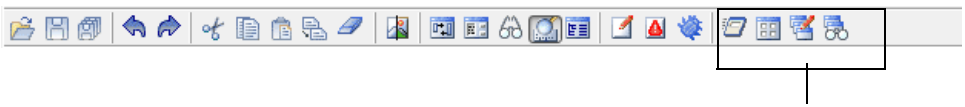


図 1.4 *IBM ILOG Views Studio ガジェット拡張機能アクション・ツールバー*

テスト　カレント・バッファがアプリケーションであればアプリケーションをテストし、カレント・バッファがパネル・バッファであればパネル・データをテストします。50 ページの *TestDocument* を参照してください。

パネル・クラス・エディタ　メイン・ウィンドウのクラス・パレットを表示するか非表示にします。49 ページの *ShowClassPalette* を参照してください。

アプリケーションの編集　アプリケーション・バッファをアクティブ・バッファにして、クラス・パレットを表示します。41 ページの *EditApplication* を参照してください。

アプリケーションの詳細設定　アプリケーションの詳細設定パネルを開きます。49 ページの *ShowApplicationInspector* を参照してください。

編集モード・ツールバー

使用できる編集モードは、編集中のバッファのタイプによります。ガジェット拡張機能で利用できる各バッファのツールバーに表示されているアイコンが異なることがわかります。

Gadgets バッファ編集モード

編集モード・ツールバーは、下図のように、Gadgets バッファが作業領域でアクティブ・ウィンドウであるときに表示されます。

図 1.5　*Gadgets バッファ編集モード・ツールバー*

Gadgets バッファの編集モード・ツールバーは、次のアイコンを含んでいます。

選択モード　選択モードを使用すると、他の一般の編集操作を選択、作成、削除、移動、リサイズ、実行することができます。このモードは、IBM ILOG Views Studio が起動している時に選択できます。

PolySelection モード　このモードを使用すると、`IlvPolyline`、`IlvPolygon`、`IlvSpline`、`IlvFilledSpline`、および `IlvClosedSpline` オブジェクトの異なる点を移動または回転することができます。操作を終了するには、作業領域をダブルクリックするか、別のモードを選択します。

ラベル・モード　このモードを使用すると 、`IlvLabel` オブジェクトを作成、編集することができます。このインタラクタの選択後、作業領域をクリックしてラベル位置を示し、必要な文字列を入力します。Enter キーを押して操作を終了します。

既存の `IlvLabel` オブジェクトを編集するには、このモードを選択して編集する `IlvLabel` オブジェクトをクリックします。

ラベル・リスト・モード　このモードを使用すると、複数行のラベル (`IlvListLabel`) オブジェクトを作成、編集することができます。このインタラクタの選択後、作業領域をクリックしてラベル位置を示し、必要な文字列を入力します。Enter キーを押すと、新しい行に移動することができます。作業領域 (オブジェクトの外の部分) をダブルクリックして操作を終了します。

既存の `IlvListLabel` を編集するには、このモードを選択して編集する `IlvListLabel` をクリックします。

回転モード　このモードを使用すると、バッファ・ウィンドウでオブジェクトを回転することができます。まず、バッファ・ウィンドウで回転させたいオブジェクトを選択します。[編集モード] ツールバーの [回転モード] アイコンをクリックします。次に、バッファ・ウィンドウの左マウス・ボタンをクリックします。バッファ・ウィンドウに矢印が表示されます。マウスをドラッグして、回転角を示します。マウス・ボタンを放すと、オブジェクトは指定された量だけ回転します。

アクティブ・モード　アクティブ・モードを使用すると、オブジェクトの振る舞いをテストして、そのプロパティのいくつかを編集することができます。アクティブ・モードでは、作業領域のオブジェクトはマウス・イベントおよびキーボード・イベントに対応しています。そのため、テキスト・フィールド・ラベルを変更したり、トグル・ボタンの状態を切り替えたりすることができます。

フォーカス・モード　このモードを使用すると、パネルのキーボード・フォーカスのパスを指定することができます。63 ページの *パネルにキーボード・フォーカスを設定する*を参照してください。

アタッチメント・モード　このモードを使用すると、パネルがリサイズされたときに、パネルのオブジェクトの位置および寸法をどのように変更するかを設定することができます。65 ページの *アタッチメント・モードの使用*を参照してください。

メニュー・モード　このモードを使用すると、ポップアップ・メニューをメニュー・バーあるいは他のポップアップ・メニューに付加できます。75 ページの *メニュー・バーにポップアップ・メニューを付加する*を参照してください。

マトリックス・モード　このモードを使用すると、マトリックス・ガジェットに表示されるマトリックス項目を変更することができます。79 ページの *マトリックスの使用*を参照してください。

スピン・ボックス・モード　このモードを使用すると、スピン・ボックス・オブジェクトに表示される項目を指定することができます。83 ページの *スピン・ボックスの編集*を参照してください。

アプリケーション・バッファ編集モード

編集モード・ツールバーは、下図のように、アプリケーション・バッファが作業領域でアクティブ・ウィンドウであるときに表示されます。

図 1.6 *アプリケーション・バッファ編集モード・ツールバー*

Gadgets バッファの編集モード・ツールバーは、次のアイコンを含んでいます。

生成 このモードを使用してアプリケーションおよび変更済みパネルを生成します。

グラフィック・バッファ編集モード

グラフィック・バッファを使用する場合は、Foundation Studio で使用する同じ編集モードにアクセスできます。

図 1.7 *グラフィック・バッファ編集モード・ツールバー*

これらの編集モードについては、*IBM ILOG Views Studio ユーザ・マニュアル*の第 3 章、「*IBM ILOG Views Studio インターフェース*」で説明されています。

パレット・パネル

IBM® ILOG® Views Studio のガジェット拡張機能を使用する場合は、パレット・パネルを通じて追加の定義済みガジェット・オブジェクトにアクセスできます。パレット・パネルのガジェットを、IBM ILOG Views Studio の Foundation パッケージと同じ方法で使用します。ドラッグ・アンド・ドロップ操作または作成モード操作のいずれかの方法を使って、作業領域でオブジェクトを作成します。

ガジェット拡張機能に提供されているパレット・パネルの上側ペインに 5 つの追加パレットがあります。下側ペインの各種ガジェット・オブジェクトにアクセスするには、上側ペインで適切なパレットをクリックします。

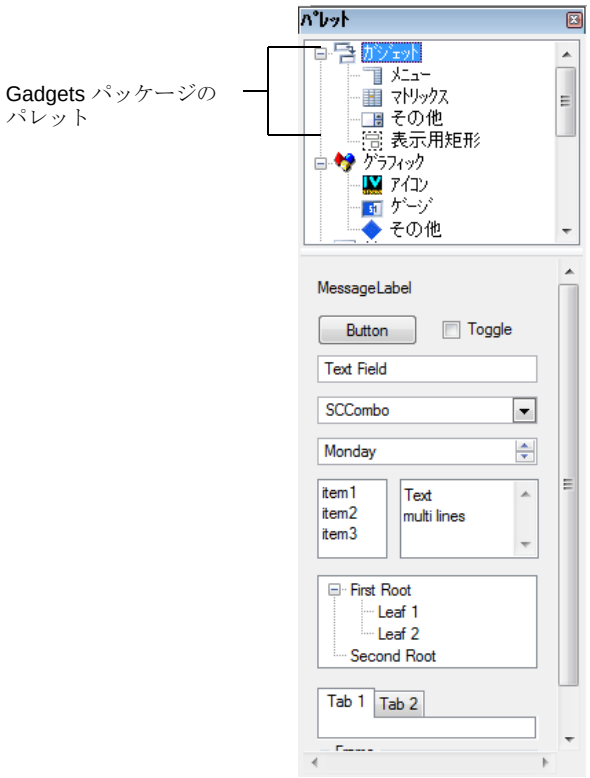
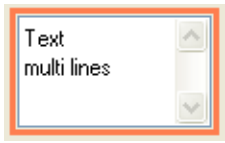
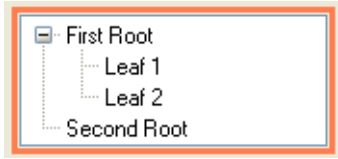


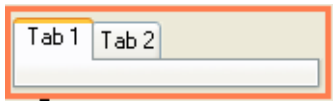
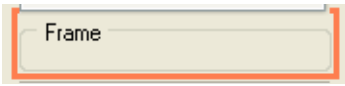
図 1.8 *IBM ILOG Views Studio ガジェット拡張機能パレット・パネル*

次のセクションでは、ガジェット拡張機能で提供されているオブジェクトについて説明します。Foundation パッケージで提供されているオブジェクトの詳細は、*IBM ILOG Views Studio ユーザ・マニュアル*を参照してください。

[ガジェット] パレット

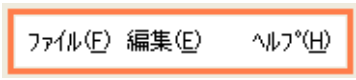
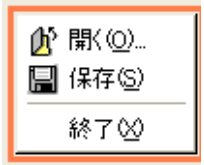
[ガジェット] パレットには、IBM® ILOG® Views Studio の標準作成モード、またはドラッグ・アンド・ドロップ操作で作成できる以下のオブジェクトがあります。

タイプ	アイコン	説明
メッセージ・ラベル		IlvMessageLabel オブジェクトを作成します。
ボタン		IlvButton オブジェクトを作成します。
トグル		IlvToggle オブジェクトを作成します。
テキスト・フィールド		IlvTextField オブジェクトを作成します。
SC コンボ・ボックス		IlvScrolledComboBox オブジェクトを作成します。
スピン・ボックス		IlvSpinBox オブジェクトを作成します。
文字列リスト		IlvStringList オブジェクトを作成します。
複数行テキスト		IlvText オブジェクトを作成します。
ツリー		IlvTreeGadget オブジェクトを作成します。

タイプ	アイコン	説明
ノートブック		IlvNotebook オブジェクトを作成します。
フレーム		IlvFrame オブジェクトを作成します。
立体線		IlvReliefLine オブジェクトを作成します。

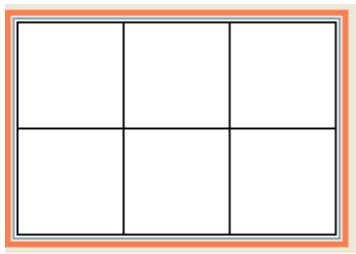
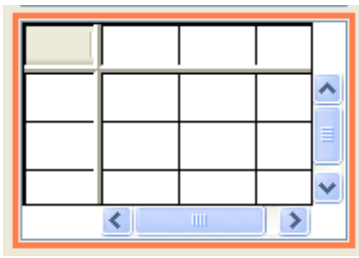
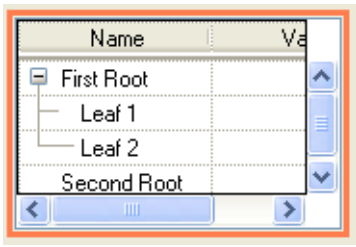
[メニュー] パレット

[メニュー] パレットには、IBM® ILOG® Views Studio の標準作成モード、またはドラッグ・アンド・ドロップ操作で作成できる以下のオブジェクトがあります。

タイプ	アイコン	説明
メニュー・バー		IlvMenuBar オブジェクトを作成します。
ポップアップ・メニュー		IlvPopupMenu オブジェクトを作成します。メニュー・バーあるいは他のポップアップ・メニューにポップアップ・メニューを付加する場合、メニュー編集モードを使用する必要があります。[編集モード] ツールバーの [メニュー] アイコンをクリックします。
ツールバー		IlvToolBar オブジェクトを作成します。

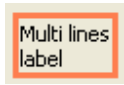
[マトリックス] パレット

[マトリックス] パレットには、IBM® ILOG® Views Studio の標準作成モード、またはドラッグ・アンド・ドロップ操作で作成できる以下のオブジェクトがあります。

タイプ	アイコン	説明
マトリックス		`IlvMatrix` オブジェクトを作成します。
シート		`IlvSheet` オブジェクトを作成します。
階層シート		`IlvHierarchicalSheet` オブジェクトを作成します。

[その他] パレット

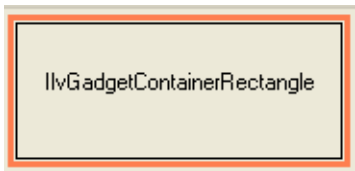
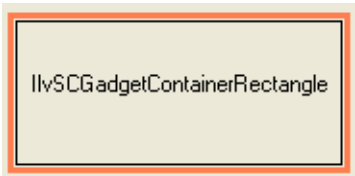
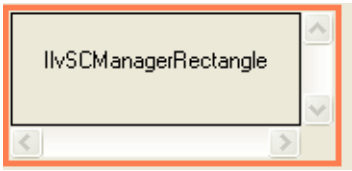
[その他] パレットには、IBM® ILOG® Views Studio の標準作成モード、またはドラッグ・アンド・ドロップ操作で作成できる以下のオブジェクトがあります。

タイプ	アイコン	説明
スライダ		IlvSlider オブジェクトを作成します。スライダに水平方向または垂直方向のいずれかを選択できます。
ラベル		IlvLabel オブジェクトを作成します。
複数行ラベル		IlvListLabel オブジェクトを作成します。
スクロールバー		IlvScrollBar オブジェクトを作成します。スクロールバーに水平方向または垂直方向のいずれかを選択できます。
オプション・メニュー		IlvOptionMenu オブジェクトを作成します。
コンボ・ボックス		IlvComboBox オブジェクトを作成します。
数値フィールド		IlvNumberField オブジェクトを作成します。
日付フィールド		IlvDateField オブジェクトを作成します。
パスワード・フィールド		IlvPasswordTextField オブジェクトを作成します。
色付きトグル		IlvColoredToggle オブジェクトを作成します。

タイプ	アイコン	説明
矢印ボタン		`IlvArrowButton` オブジェクトを作成します。
ガジェット		`IlvGadget` オブジェクトを作成します。

表示用矩形パレット

ビュー矩形パレットには、IBM® ILOG® Views Studio の標準作成モード、またはドラッグ・アンド・ドロップ操作で作成できる以下のオブジェクトがあります。

タイプ	アイコン	説明
ガジェット・コンテナ		`IlvGadgetContainerRectangle` オブジェクトを作成します。
SC ガジェット・コンテナ		`IlvSCGadgetContainerRectangle` オブジェクトを作成します。
SC マネージャ		`IlvSCManagerRectangle` オブジェクトを作成します。

ガジェット拡張機能コマンド

このセクションでは、IBM® ILOG® Views Studio のガジェット拡張機能で使用可能な追加の定義済みコマンドのリストを、アルファベット順に表示します (IBM ILOG Views Studio Foundation の全コマンドも使用できます)。一覧表には、各

コマンドのラベル、コマンドが属するカテゴリ、コマンド・パネル以外からアクセスできる場合は、そのアクセス方法と用途を列挙しています。

コマンド・パネルを表示するには、メイン・ウィンドウの [ツール] メニューから [コマンド] を選択するか、[アクション] ツールバーの [コマンド] アイコン  をクリックします。

AddPanel

ラベル	カレント・パネルを追加
カテゴリ	アプリケーション
アクション	選択したパネル・クラスのパネル・インスタンスを作成して、これを アプリケーションに追加します。

EditApplication

ラベル	アプリケーションの編集
パス	メイン・ウィンドウ : ツールバーの [ツール] メニューおよび [アプリケーションの編集] アイコン
カテゴリ	アプリケーション
アクション	アプリケーション・バッファを選択してクラス・パレットを表示します

Generate

ラベル	コード生成
パス	メイン・ウィンドウ : コード・メニュー

カテゴリ	アプリケーション
アクション	アプリケーション記述ファイルを保存し、アプリケーションおよびその変更したパネル・クラスに C++ コードを生成します。アプリケーション・ファイル名がデフォルト名である場合は、ファイルのセレクタ・パネルが開き、ここに新規ファイル名を入力します。アプリケーションの詳細設定の [メイクファイル] トグル・ボタンがオンである場合、Makefile も生成されます。

GenerateAll

ラベル	すべて生成
パス	メイン・ウィンドウ : コード・メニュー
カテゴリ	アプリケーション
アクション	アプリケーション記述ファイルを保存し、アプリケーションおよびそのパネル・クラスに C++ コードを生成します。アプリケーション・ファイル名がデフォルト名である場合は、ファイルのセレクタ・パネルが開き、ここに新規ファイル名を入力します。アプリケーション・ファイルの [メイクファイル] トグル・ボタンがオンである場合、Makefile も生成されます。

GenerateApplication

ラベル	アプリケーションのコード生成
パス	メイン・ウィンドウ : コード・メニュー
カテゴリ	アプリケーション
アクション	アプリケーション記述ファイルを保存し、アプリケーションに C++ コードを生成します。アプリケーション・ファイル名がデフォルト名である場合は、ファイルのセレクタ・パネルが開き、ここに新規ファイル名を入力します。アプリケーションの詳細設定の [メイクファイル] トグル・ボタンがオンである場合、Makefile も生成されます。

GenerateMakeFile

ラベル	Makefile の生成
パス	メイン・ウィンドウ : コード・メニュー
カテゴリ	アプリケーション
アクション	アプリケーションの詳細設定の [メイクファイル] トグル・ボタンがオンでない場合でも、アプリケーションの Makefile を生成します。

GeneratePanelClass

ラベル	パネルのクラスを生成
パス	メイン・ウィンドウ : コード・メニュー
カテゴリ	アプリケーション
アクション	現在選択されているパネル・クラスに C++ コードを生成します。パネル・クラスを選択するには、クラス・パレットを使用します。

GeneratePanelSubClass

ラベル	パネルのサブクラスを生成
パス	メイン・ウィンドウ : コード・メニュー
カテゴリ	アプリケーション
アクション	現在のパネル・クラスにサブクラス・スケルトンを生成します。このコマンドは、クラス名、ファイル・ベース名ならびにヘッダーとソース・ファイルが生成されるディレクトリを入力するダイアログ・ボックスをアクティブにします。

InspectPanel

ラベル	パネル・クラスの詳細設定
パス	クラス・パレット ツールバー
カテゴリ	アプリケーション、パネル
アクション	選択したパネル・クラスのプロパティの詳細設定を行うパネル・クラスの詳細設定を開きます。

KillTestPanels

ラベル	テスト・パネルを終了
カテゴリ	アプリケーション
アクション	アプリケーションあるいはカレント・バッファのテスト用に作成されたパネルを終了します。

MakeDefaultApplication

ラベル	デフォルト・アプリケーションの作成
パス	メイン・ウィンドウ : [ファイル] メニュー > 新規
カテゴリ	アプリケーション
アクション	カレント・バッファがパネル・バッファである場合、このコマンドはアプリケーションを作成し、カレント・バッファ用パネル・クラスを作成し、そのパネル・クラスにパネル・インスタンスを作成します。

NewApplication

ラベル	アプリケーション
パス	メイン・ウィンドウ : [ファイル] メニュー > 新規

カテゴリ	アプリケーション
アクション	現在のアプリケーションを閉じて新しいアプリケーションを編集します。1 度に編集できるのは 1 つのアプリケーションだけです。

NewGadgetBuffer

ラベル	Gadget
パス	メイン・ウィンドウ : [ファイル] メニュー > 新規
カテゴリ	バッファ
アクション	新規ガジェット・バッファの作成。このバッファが、カレント・バッファになります。

NewPanelClass

ラベル	パネル・クラスへ登録
パス	クラス・パレット
カテゴリ	アプリケーション
アクション	カレント・バッファがコンテナに設計されており、まだアプリケーションの一部となっていない場合、このバッファに新規パネル・クラスを作成します。

OpenApplication

ラベル	開く ...
パス	メイン・ウィンドウ : ツールバーの [ファイル] メニュー、[開く] アイコン
カテゴリ	アプリケーション
アクション	現在のアプリケーションを閉じて、IBM ILOG Views Studio によって以前保存されたアプリケーションを読み込みます。このコマンドは、アプリケーション記述ファイルを選択できるファイルのセレクタ・パネルを開きます。

RemoveAllAttachments

ラベル	すべてのアタッチメントを削除
カテゴリ	アタッチメント
アクション	選択したオブジェクトの全アタッチメントを削除します。このコマンドは、現在のモードがアタッチメント・モードの場合のみ機能します。

RemoveAttachments

ラベル	アタッチメントを削除
カテゴリ	アタッチメント
アクション	カレント・バッファの全アタッチメントを削除します。このコマンドは、現在のモードがアタッチメント・モードの場合のみ機能します。

RemovePanel

ラベル	パネルの削除
パス	アプリケーション・バッファ・ウィンドウのパネル・インスタンス・ウィンドウ。閉じる (X) ボタンあるいはメニュー。
カテゴリ	アプリケーション
アクション	アプリケーションから選択されたパネル・インスタンスを削除します。

RemovePanelClass

ラベル	パネル・クラスの削除
パス	クラス・パレット
カテゴリ	アプリケーション
アクション	選択されたパネル・クラスおよびその全パネル・インスタンスを削除します。

SaveApplication

ラベル	保存
パス	メイン・ウィンドウ : ツールバーの [ファイル] メニュー、[保存] アイコン
カテゴリ	アプリケーション
アクション	現在のアプリケーションの記述をそのデータ・ファイルに保存します。アプリケーションの名前がデフォルト名である場合は、このコマンドは、アプリケーション・ファイル名をユーザが入力できるように、`SaveApplicationAs` コマンドを実行します。新規アプリケーションを初めて保存する、あるいはそのファイル名を変更する場合は、`SaveApplicationAs` コマンドを実行します。

SaveApplicationAs

ラベル	名前を付けて保存 ...
パス	メイン・ウィンドウ : ツールバーの [ファイル] メニュー、[保存] アイコン
カテゴリ	アプリケーション
アクション	アプリケーションに対して新しいファイル名を入力し、その記述をそのファイルに保存するファイルのセレクタ・パネルを開きます。このコマンドにより、アプリケーションのファイル・ベース名 (つまり、その生成された C++ ファイル) およびその場所を変更することができます。

SelectAttachmentsMode

ラベル	アタッチメント
パス	メイン・ウィンドウ : 編集モード・ツールバー (Gadgets バッファの編集時)
カテゴリ	モード、ガジェット
状態	このモードが選択された場合は、True。
アクション	アタッチメント・モードを選択します。65 ページの *アタッチメント・モードの使用*を参照してください。

SelectFocusMode

ラベル	フォーカス
パス	メイン・ウィンドウ : 編集モード・ツールバー (Gadgets バッファの編集時)
カテゴリ	モード、ガジェット
状態	このモードが選択された場合は、True。
アクション	フォーカス・モードを選択します。63 ページの*パネルにキーボード・フォーカスを設定する*を参照してください。

SelectMatrixMode

ラベル	マトリックス
パス	メイン・ウィンドウ : 編集モード・ツールバー (Gadgets バッファの編集時)
カテゴリ	モード
状態	このモードが選択された場合は、True。
アクション	マトリックス・モードを選択します。79 ページの*マトリックスの使用*を参照してください。

SelectMenuMode

ラベル	メニュー
パス	メイン・ウィンドウ : 編集モード・ツールバー (Gadgets バッファの編集時)
カテゴリ	モード
状態	このモードが選択された場合は、True。
アクション	メニュー・モードを選択します。70 ページの *[編集]* メニューを参照してください。

ShowAllTestPanels

ラベル	テスト・パネルをすべて表示
カテゴリ	アプリケーション、パネル
アクション	アプリケーションの現在のパネルをすべて表示します。表示パネルのみを表示する `TestApplication` コマンドと異なり、このコマンドは全パネルを表示します。

ShowApplicationInspector

ラベル	アプリケーションの詳細設定
パス	メイン・ウィンドウ : ツールバーの [ツール] メニューおよび [アプリケーションの詳細設定]
カテゴリ	アプリケーション、パネル
アクション	アプリケーションの詳細設定パネルを開きます。

ShowClassPalette

ラベル	パネル・クラス・エディタ
パス	メイン・ウィンドウ : ツールバーのクラス・アイコン
カテゴリ	アプリケーション、パネル
アクション	パネル・クラスを作成、詳細設定、削除し、パネル・インスタンスを作成できるクラス・パレットを開きます。

ShowPanelClassInspector

ラベル	パネル・クラスの詳細設定
パス	クラス・パレット ツールバー

カテゴリ	アプリケーション、パネル
アクション	選択したオブジェクトのパネル・クラスの詳細設定を表示あるいは非表示にします。

TestApplication

ラベル	テスト・パネル
パス	メイン・ウィンドウ : ツールバー
カテゴリ	アプリケーション
状態	アプリケーションがテストされていれば True。
アクション	アプリケーションがテストされていなければ表示パネルを開き、このコマンドを再び実行するまでそのテストを行うことができます。アプリケーションがテストされている場合は、このコマンドはテスト・パネルを終了してテストを終了させます。

TestDocument

ラベル	テスト
パス	メイン・ウィンドウ : ツールバー
カテゴリ	アプリケーション
アクション	カレント・バッファがアプリケーションである場合はアプリケーションを、パネル・バッファの場合はパネル・データをテストします。

TestPanel

ラベル	テスト・パネル
パス	メイン・ウィンドウ : ツールバー
カテゴリ	バッファ
アクション	カレント・バッファをテストします。

ガジェット拡張機能パネル

ガジェット拡張機能は、独自のパネルおよびアプリケーションを作成するときに使用できる追加パネルおよびダイアログ・ボックスを提供します。

- *アプリケーションの詳細設定*
- *パネル・クラスの詳細設定*
- *パネル・インスタンスの詳細設定*

アプリケーションの詳細設定

アプリケーションの詳細設定は、生成したアプリケーションの設定を編集するために使用します。これにより C++ ファイルの場所、クラス宣言、および生成されたコード用にいくつかのオプションを指定します。このパネルで、生成アプリケーション・クラス・ファイルにコードを挿入することもできます (97 ページのヘッダーおよびソース・ページを参照してください)。

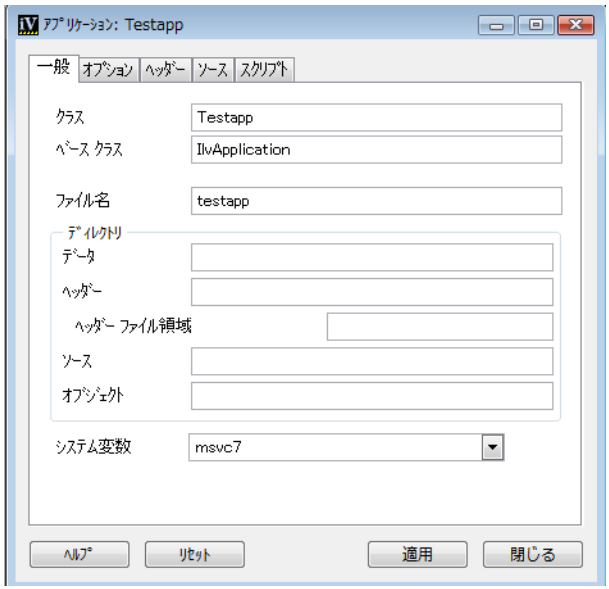
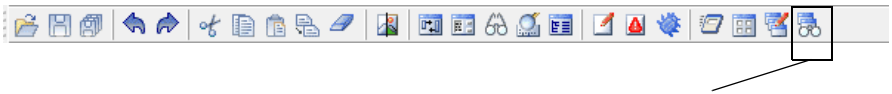


図 1.9 *アプリケーションの詳細設定*

パネルへのアクセス

パネルにアクセスするには

- [アクション] ツールバーの [アプリケーションの詳細設定] アイコンをクリックします。



アプリケーションの詳細設定アイコン

または

- [コード] メニューから [アプリケーションの詳細設定] を選択します。

または

- ツール・メニューからコマンドを選択し、リストから `ShowApplicationInspector` コマンドを選択し、[適用] をクリックします。

アプリケーションの詳細設定を構成する要素

アプリケーションの詳細設定には 5 つのノートブック・ページがあります。一般、オプション、ヘッダー、ソースおよびスクリプト、そして 4 つのボタン、[適用]、[リセット]、[閉じる]、および [ヘルプ] です。各ノートブック・ページおよびページに含まれているフィールドに関する完全な説明は、93 ページの *アプリケーションの詳細設定* を参照してください。

パネル・クラスの詳細設定

パネル・クラスの詳細設定は、ディレクトリおよびパネル・クラス生成用オプションを詳細設定するために使用します。このパネルはまた、独自のコードを生成パネル・クラス・ファイルに挿入するためにも使用します (106 ページの *ヘッダーおよびソース・ページ* を参照してください)。

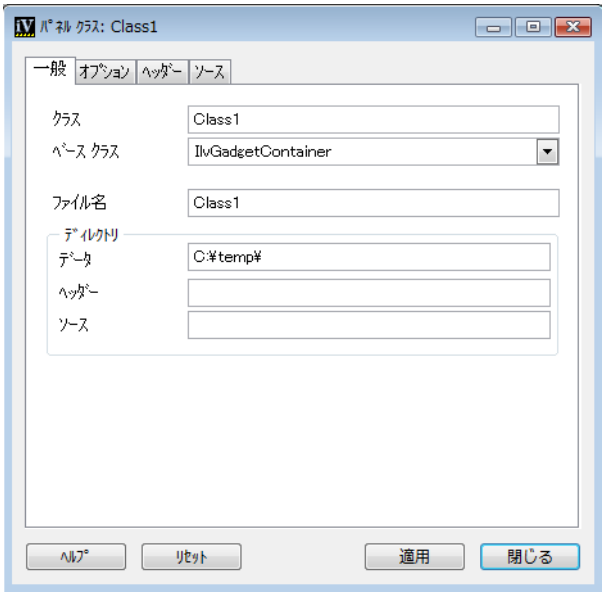


図1.10　パネル・クラスの詳細設定

パネルへのアクセス

パネルにアクセスするには

- [パネル・クラス] パレットの [パネル・クラスの詳細設定] アイコン をクリックします。

または

- [コード] メニューから [パネル・クラスの詳細設定] を選択します。

または

- ツール・メニューからコマンドを選択し、リストから `ShowPanelClassInspector` コマンドを選択し、[適用] をクリックします。

パネル・クラスの詳細設定を構成する要素

パネル・クラスの詳細設定には、4 つのノートブック・ページがあります。一般、オプション、ヘッダー、ソース、そして 4 つのボタン、[適用]、[リセット]、[閉じる]、および [ヘルプ] です。各ノートブック・ページおよびページに含まれているフィールドに関する完全な説明は、103 ページの*パネル・クラスの詳細設定*を参照してください。

パネル・インスタンスの詳細設定

パネル・インスタンスの詳細設定は、選択したパネル・インスタンスのプロパティを編集するために使用します。

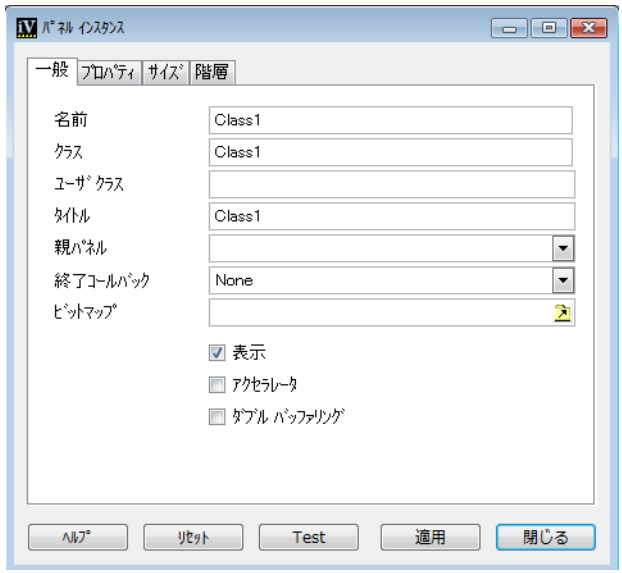


図 1.11 *パネル・インスタンスの詳細設定*

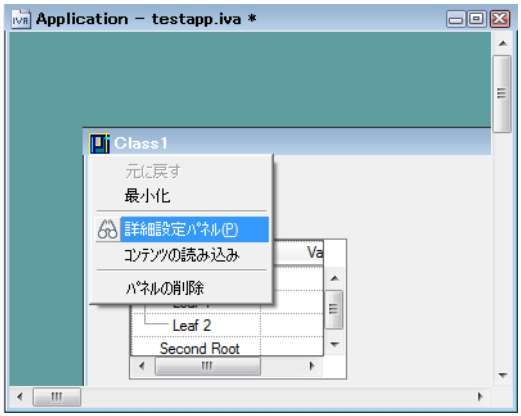
パネルへのアクセス

パネルにアクセスするには

- アプリケーション・バッファ・ウィンドウでパネル・インスタンスのタイトル・バーをダブルクリックします。

  または

- パネル・インスタンスの左上にあるボックスをクリックし、ポップアップ・メニューから [詳細設定パネル] を選択します。



または

- [アプリケーション] メニューから [詳細設定パネル] を選択します。

  または

- ツール・メニューからコマンドを選択し、リストから `InspectPanel` コマンドを選択し、[適用] をクリックします。

パネル・クラス・インスタンスの詳細設定を構成する要素

パネル・クラスの詳細設定には、4 つのノートブック・ページがあります。一般、プロパティ、サイズ、階層および 5 つのボタン、[適用]、[リセット]、[テスト]、[閉じる]、および [ヘルプ] です。各ノートブック・ページおよびページに含まれているフィールドに関する完全な説明は、109 ページの*パネル・インスタンスの詳細設定*を参照してください。

章

2

ガジェット・パネルの編集

この章では、ガジェット・パネル作成に使用する基本的なコマンド、パネル、モードについて説明します。

この章は、以下のトピックに分かれています。

- *新規パネルの作成*
- *ガジェット・オブジェクトの作成*
- *オブジェクトの詳細設定*
- *パネルのテスト*
- *アクティブ・モードの使用*
- *パネルにキーボード・フォーカスを設定する*
- *アタッチメント・モードの使用*
- *[編集] メニュー*
- *マトリックスの使用*
- *スピン・ボックスの編集*

新規パネルの作成

IBM® ILOG® Views Studio が起動すると、編集準備のできた空の Gadgets バッファ・ウィンドウが開きます。この Gadgets バッファ・ウィンドウにパネルを作成することになります。新規 Gadgets バッファ・ウィンドウを作成するために、次の処理を適宜行います。

1. [ファイル] メニューから [新規] を選択します。
2. 表示されたサブメニューで [ガジェット] を選択します。

   新規 Gadgets バッファ・ウィンドウが選択され、編集可能になります。

ガジェット・オブジェクトの作成

パレット・パネルの Gadgets パレットは、多くの定義済みガジェットを提供し、ここからパネルのオブジェクトを作成します。ドラッグ・アンド・ドロップ操作または作成モード機能のいずれかを使用できます。

ドラッグ・アンド・ドロップ操作の使用

ドラッグ・アンド・ドロップ操作を使用してオブジェクトを作成する場合、バッファ・ウィンドウに追加されるオブジェクトは、パレット・パネルにあるようなオブジェクトの正確なコピーです。オブジェクトは、パレット・パネルのオブジェクトと同じ形状およびサイズです。

ドラッグ・アンド・ドロップ操作でオブジェクトを作成するには

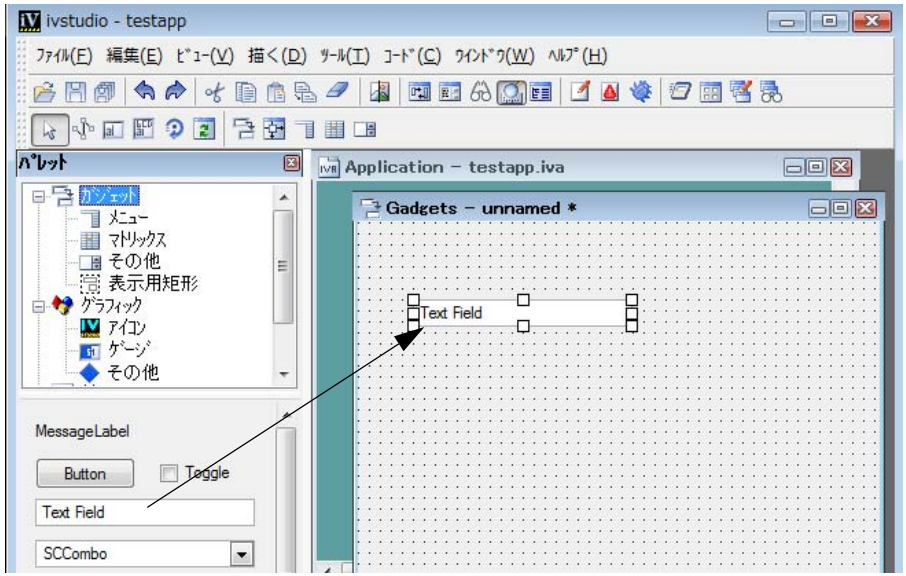
1. パレット・パネルの上側ペインで、作成するガジェットのタイプに対応するツリー・アイテムをクリックします。

   関連するパレットが、下側ペインに表示されます。
2. 目的のオブジェクトをクリックして、Gadgets バッファ・ウィンドウにドラッグします。

   マウス・ボタンを放すと、選択モードになります。オブジェクトはバッファ・ウィンドウで選択されたままであり、必要に応じてオブジェクトを変更することができます。

たとえば、テキスト・フィールドを作成するには次の処理を行います。

1. パレット・パネルの上側ペインで、ツリーの [Gadgets] をクリックします。
2. パレット・パネルの下側ペインで、テキスト・フィールド・ガジェットをクリックします。
3. それを Gadgets バッファ・ウィンドウにドラッグします。



作成モードの使用

作成モードを使用する場合、原則的に、バッファ・ウィンドウでオブジェクトを描画します。オブジェクトのサイズと形状を決定します。作成モードでパレット・パネルで作成するオブジェクトの種類を選択すると、複数のオブジェクトを作成することもできます。

作成モードでオブジェクトを作成するには次の手順に従います。

1. パレット・パネルの上側ペインで、作成するオブジェクトの種類に対応するツリーの項目をクリックします。

   関連するパレットが、下側ペインに表示されます。

2. パレット・パネルの下側ペインで、目的のオブジェクトをクリックします。バウンディング・ボックスがオブジェクトの周りに表示され、作成モードがアクティブであることを示します。

   バッファ・ウィンドウにオブジェクトを 1 つだけ追加する場合は、パレット・ウィンドウでオブジェクトを 1 回クリックします (これで、一時的な作成モードになります。バッファ・ウィンドウでオブジェクトを描いた後、作成モードは自動的に終了します)。

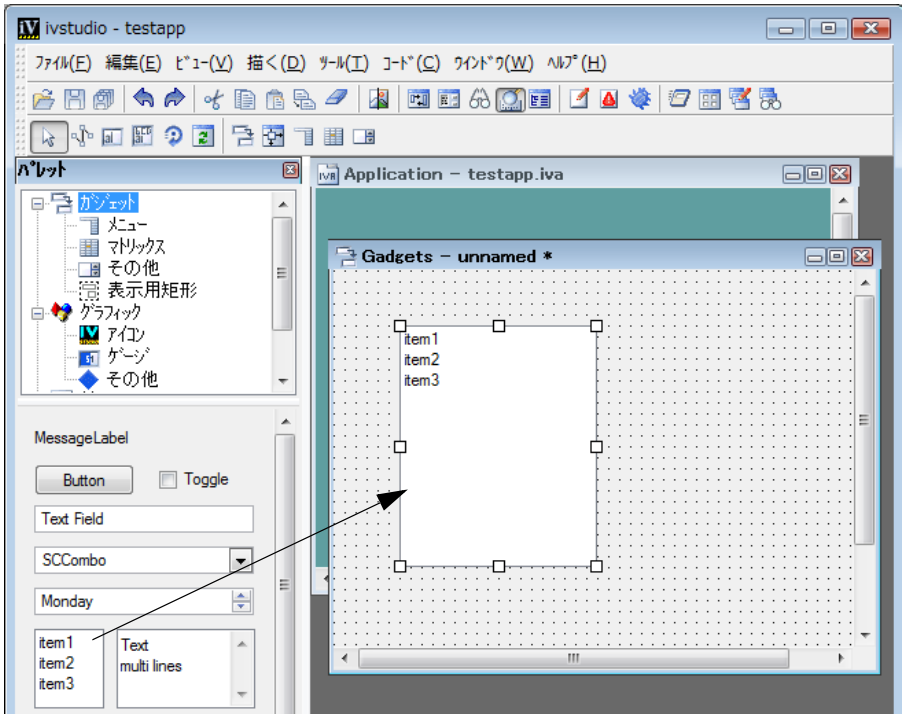
   同じ種類のオブジェクトを複数追加したい場合は、Shift キーを押したまま、パレット・パネルでオブジェクトをクリックします (これで、永続的な作成モードになります。作成モードのままなので、必要なだけオブジェクトを描くことができます。作成モードを終了するには、[編集モード] ツールバーの [選択モード] アイコンをクリックします)。

3. ポインタをバッファ・ウィンドウに移動させます。
4. オブジェクトを配置するバッファ・ウィンドウでクリックして、オブジェクトが目的のサイズと形状になるまでマウスをドラッグします。

たとえば、1 つの文字列リスト・ボックス作成するには次のように行います。

1. パレット・パネルの上側ペインで、ツリーの [Gadgets] をクリックします。
2. パレット・パネルの下側ペインで、`IlvStringList` アイコンを 1 回クリックします。`IlvStringList` アイコンの周りに表示されたバウンディング・ボックスによって、作成モードであることがわかります。
3. Gadgets ウィンドウで、文字列リスト・ボックスを描き始める位置をクリックします。
4. 文字列リスト・ボックスが目的のサイズと形状になるまで、マウスをドラッグします。

   マウスをドラッグすると、オブジェクトの形状とサイズを示すバウンディング・ボックスが表示されます。
5. マウス・ボタンを放します。文字列リスト・ボックスは、描いたバウンディング・ボックスの大きさで表示されます。



マウス・ボタンを放すと自動的に作成モードが解除され、選択モードになります。パレット・パネルの `IlvStringList` アイコンは選択されず、[編集モード] ツールバーの [選択モード] アイコン  が選択されることに注意してください。

必要に応じて、ボックスの形状の変更、リサイズ、移動、または変更を行うことができます。

オブジェクトの詳細設定

オブジェクトの特定のプロパティを詳細設定するには、そのオブジェクトをダブルクリックします。メイン・ウィンドウの [アクション] ツールバーの [詳細設定] アイコンをクリックすることもできます。

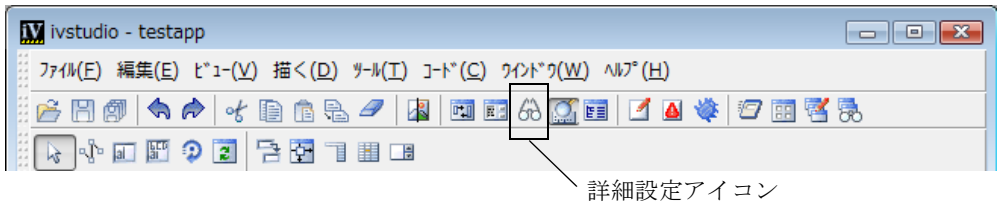


図 2.1 *メイン・ウィンドウ・ツールバーの詳細設定アイコン*

オブジェクト・クラスに関連する詳細設定パネルがある場合、それを使用してオブジェクト・クラスの特定のプロパティを編集することができます。詳細設定の内容は、関連するオブジェクト・クラスによって決まります。

文字列リスト・ガジェットをクリックすると、次の詳細設定パネルが表示されます。

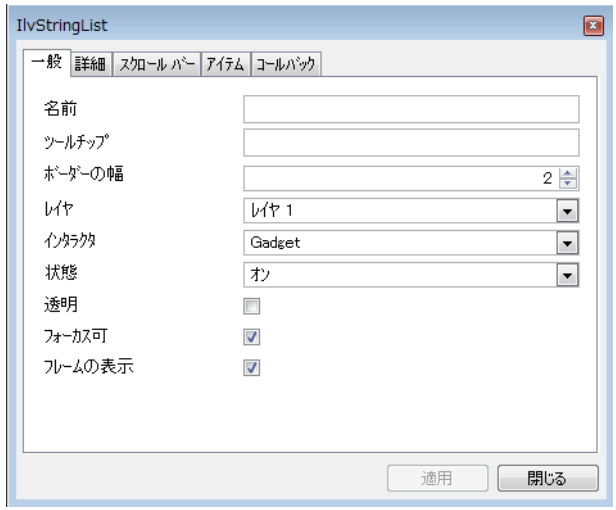


図 2.2 *文字列リストの詳細設定パネル ([一般] ページ)*

オブジェクト・プロパティに行われた変更を確認する場合は [適用] をクリックします。パネルを閉じる場合は、[閉じる] をクリックします。

一度に処理できるのは 1 つのオブジェクトだけです。最初のオブジェクトが詳細設定されている間に同じタイプの別のオブジェクトを選択した場合、新しく選択されたオブジェクトのプロパティが、詳細設定パネルに表示されます。別のタイプのオブジェクトが選択されると、表示されているオブジェクトに代わってその関連の詳細設定が表示されます。

パネルのテスト

パネルの振る舞いをテストするには、メイン・ウィンドウ・ツールバーの [テスト] アイコンをクリックします。

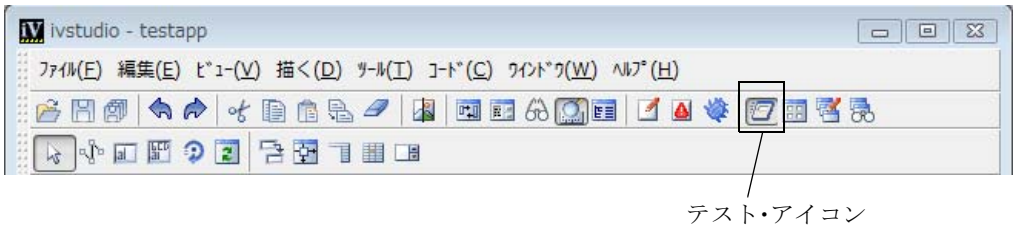


図 2.3　メイン・ウィンドウ・ツールバーのテスト・アイコン

カレント・バッファを表すパネルが表示され、テストを行う準備が整います。テスト・モードを終了するには、同じアイコンをもう一度クリックします。

アクティブ・モードの使用

アクティブ・モードでは、作業領域のオブジェクトはマウス・イベントおよびキーボード・イベントに対応しています。これによって、オブジェクトの振る舞いのテストやオブジェクト・プロパティの編集ができます。たとえば、テキスト・フィールド・ラベルを変更し、トグル・ボタンの状態を切り替えたりすることができます。

アクティブ・モードを選択するには、[編集モード] ツールバーで [アクティブ] アイコンをクリックします。

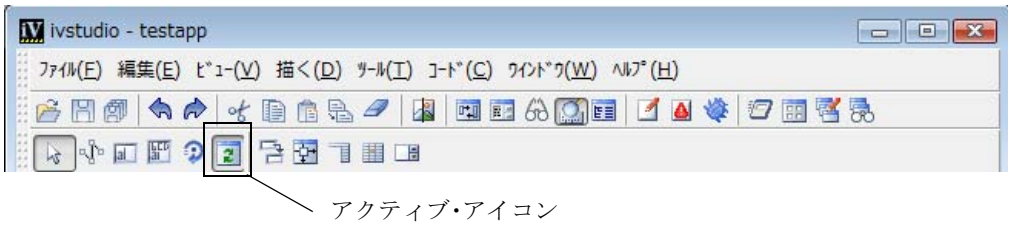


図 2.4　アクティブ・モード・アイコン

パネルにキーボード・フォーカスを設定する

デフォルトでは、キーボード・フォーカスは、オブジェクト・バウンディング・ボックスの位置によって決定されます。このデフォルト・フォーカスは論理的にオブジェクト間を、左から右へ、上から下へと移動します。このデフォルト・パスが常に適切だとはいえないため、IBM® ILOG® Views Studio には、パネルにキーボード・フォーカスのパスを描くことができるフォーカス編集モードが用意されています。

フォーカス・モードを選択するには、[編集モード] ツールバーの [フォーカス] アイコンをクリックします。

図 2.5 *フォーカス・モード・アイコン*

キーボード・フォーカス・モード・パスは一連の矢印で示されます。

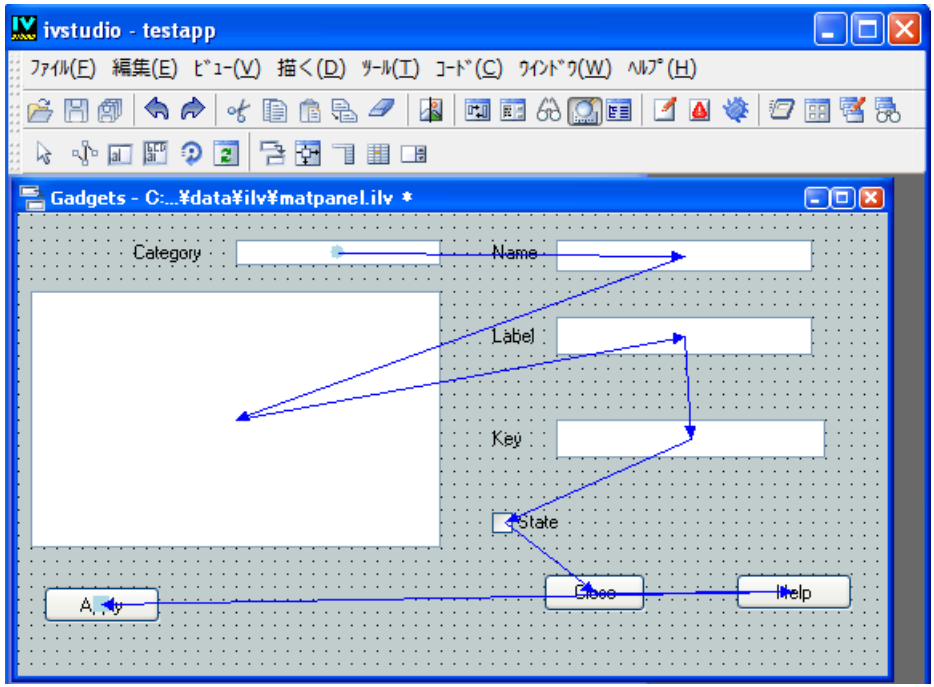


図2.6　フォーカス・パス

次のフォーカス・オブジェクト

フォーカス可能オブジェクト (キーボード・イベントを受け取ることのできるグラフィック・オブジェクト) それぞれに、その次のフォーカス・オブジェクトを指定することができます。これを行うには、オブジェクトからラインをドラッグして、次のフォーカス・オブジェクトにするオブジェクト上にラインがあるときにマウス・ボタンを放します。この操作は、フォーカス可能オブジェクトでのみ機能します。

最初のフォーカス・オブジェクト

パネルを初めてフォーカスする場合、最初のフォーカス・オブジェクトはキーボード・フォーカスを取得するオブジェクトです。最初のフォーカス・オブジェクトを指定するには、(オブジェクトからではなく) 作業領域でラインをドラッグします。最初のフォーカス・オブジェクトとして選択したオブジェクト内にラインがある時に、マウス・ボタンを放します。塗りつぶされた円が、そのオブジェクトの中心に描かれます。ガジェット・コンテナは、最初のフォーカス・グラフィックを 1 つのみ持つことができます。

最後のフォーカス・オブジェクト

最後のフォーカス・グラフィックを指定するには、そのオブジェクトからラインをドラッグし、ラインが (オブジェクトからではなく) 作業領域にあるときにマウス・ボタンを放します。キーボード・フォーカス・チェーンがガジェット・コンテナの最後のフォーカス・グラフィックを放すと、最初のフォーカス・グラフィックに戻ります。しかし、ガジェット・コンテナが他のコンテナにリンクされている場合は、キーボード・フォーカスをそのコンテナに与えます。ガジェット・コンテナは、最後のフォーカス・グラフィックを 1 つ以上持つことができます。

アタッチメント・モードの使用

IBM® ILOG® Views Studio は、パネルがリサイズされたときに、パネル内のオブジェクトの位置および寸法がどのように変更されるかを設定するために使用するアタッチメント・モードを用意しています。このモードをアクティブにするには、[編集モード] ツールバーの [アタッチメント] アイコンをクリックします。

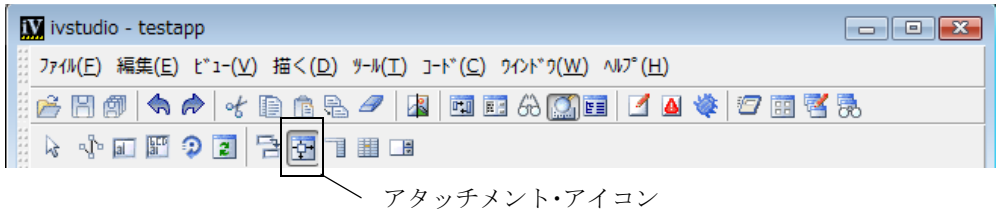


図 2.7　アタッチメント・アイコン

アタッチメントの設定には 2 つの手順があります。

- *ガイドの設定*
- *ガイドにオブジェクトを付加する*

ガイドの設定

初めて [アタッチメント] アイコンをクリックすると、その横に数字が付いているガイドが上部および左の境界に表示されます。これらの数字は、ガイドに対応した重みを示します (下図を参照してください)。

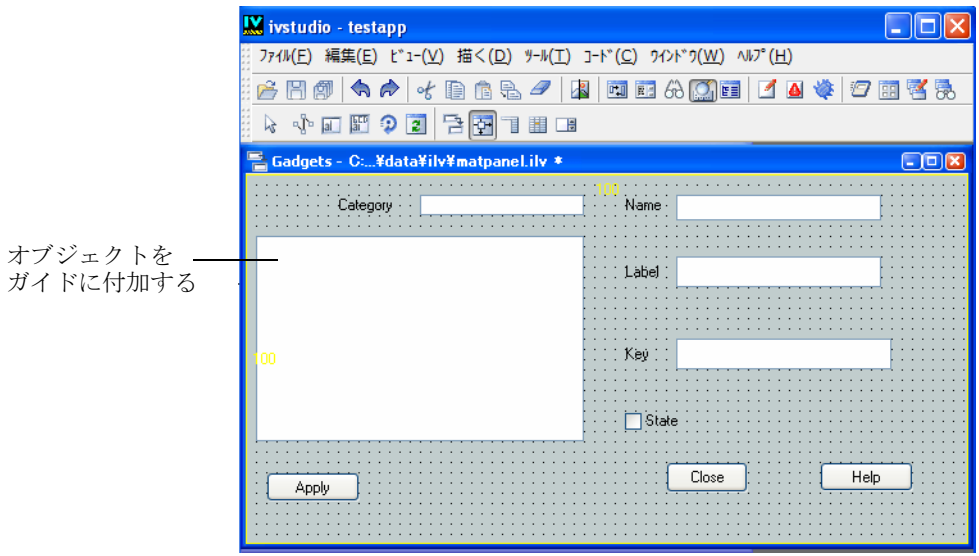


図2.8 ガイドの付加

1. ガイドあるいは重み数値をクリックしてガイドを選択します。
2. 上部あるいは左の境界にある初期ガイドの 1 つを選択してガイドを作成し、新しく作成されたガイドを (マウス・ボタンを押して) 希望位置までドラッグして、マウスボタンを放します。

ガイドは、ガイドの詳細設定パネルで編集可能な 4 つの要素で定義されます。このパネルを開くには、ガイドあるいはその重み数値をダブルクリックします。

図2.9 ガイドの詳細設定パネル

ガイドの詳細設定パネルには、以下のフィールドがあります。

- **位置** 水平ガイドの上部境界からのピクセル数値 垂直ガイドの場合は、左境界からのピクセル数値。

- **サイズ**　水平ガイドの次のガイドまでのピクセル数値。垂直ガイドの場合は、右にある次のガイドまでのピクセル数値。
- **リミット**　ウィンドウがリサイズされたときに、ガイドによって設定されるセクションの最小サイズ (「サイズ」を参照してください)。
- **重み**　ウィンドウがリサイズされたときに、他のセクションに関連するセクションに (ガイドによって区切られる) 割り当てられるウィンドウの数。ウィンドウがリサイズされると各セクションに次の式が適用され、デルタは初期サイズより小さいウィンドウの新しいサイズと等しくなります。

$$\text{新しいセクションのサイズ} = \text{セクションの初期サイズ} + \text{デルタ} \times \frac{\text{セクションを限定するガイドの重み}}{\text{すべてのガイドの重み合計}}$$

ガイドにオブジェクトを付加する

各オブジェクトには、固定 (ダブル・ライン) および伸縮 (シングル・ライン) 位置に設定するための追加コントロールを提供するアタッチメントの詳細設定があります。オブジェクトをダブルクリックすると、アタッチメントの詳細設定が開きます。ここでは既存のアタッチメントの編集しかできません。テキスト・フィールドの数字は、オブジェクトの端から使用されているガイドまでのピクセル数を示します。

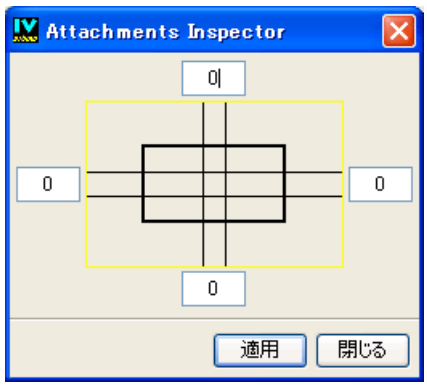


図 2.10　アタッチメントの詳細設定

アタッチメント作成用に 6 つの位置があり、それぞれにタイプが 2 つあります。

- **位置**　アタッチメントの位置は、オブジェクト・バウンディング・ボックスの 4 つの辺のいずれかから、その辺に平行のガイドまでになります。アタッチメントは、オブジェクト内部 (水平、および垂直) で、パネルがリサイズされたときにオブジェクトのサイズを変更するかどうかを指定するために定義することもできます。
- **タイプ**　6 つの位置それぞれで 2 種類のアタッチメント、固定あるいは伸縮が可能です。
  - **固定** (ダブル・ライン) : パネルがリサイズされても、オブジェクトおよびガイドの距離は同じままです。オブジェクト内での固定は、パネルがリサイズされてもオブジェクトのサイズは変更されないことを意味します。
  - **伸縮** (シングル・ライン) : パネルがリサイズされると、オブジェクトおよびガイドの距離もそれに比例して変更されます。オブジェクト内での伸縮は、パネルがリサイズされると、オブジェクトのサイズがそれに比例して変更されることを意味します。

アタッチメント操作

オブジェクトを付加する場合、次のような操作を行うことができます。

- **アタッチメント作成**　付加するオブジェクトを選択し、ラインを (マウス・ボタンを押して) そのオブジェクトの中央のハンドルからその辺に平行なガイドまでドラッグします。ガイドが強調表示されたら、マウス・ボタンを放します。デフォルト・アタッチメントは、反対側およびオブジェクトの内側で行われます。指定したアタッチメントは選択した全オブジェクトに対して適用されます。

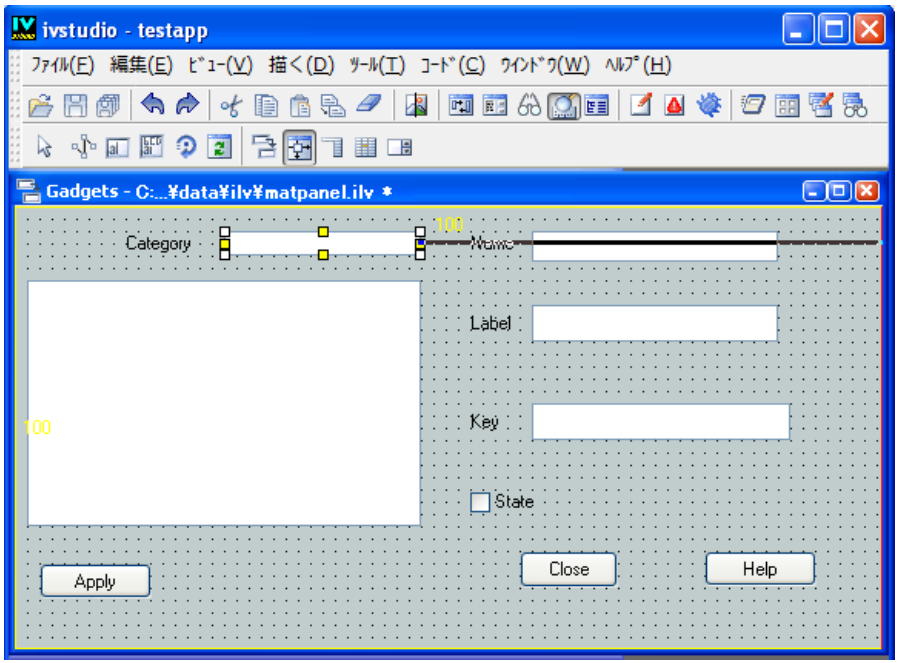


図2.11　オブジェクトをガイドに付加する

> ***メモ:*** *アタッチメントをガイドとオブジェクトの間に作成するとき、オブジェクトがガイドに接近しすぎる問題が生じることがあります。この場合、マウスを反対側にドラッグしてから、ガイドへ戻します。*

- **アタッチメントの削除**　アタッチメントのハンドルの 1 つをドラッグし、新しいラインがガイドに触れていない状態でマウス・ボタンを離します。
- **アタッチメントを別のガイドへに変更**　ラインをアタッチメント・ハンドルから新しいガイドへドラッグします。
- **アタッチメントのタイプを変更**　アタッチメント・ライン上でクリックすると、固定および伸縮の間で切り替わります。これは、アタッチメントの詳細設定で行うこともできます。
- **オブジェクトのアタッチメントの表示**　オブジェクトをアタッチメント編集モードで選択し、オブジェクトをダブルクリックして、アタッチメントの詳細設定を表示させます。

デフォルト

デフォルト選択は、アタッチメントを開始するために使用されたハンドルによって異なります。

- **左および上部ハンドル**　オブジェクトはパネルとともに拡大します (固定、伸縮、固定)。
- **右および下部ハンドル**　オブジェクトの大きさを変えずに移動します (伸縮、固定、固定)。

アタッチメントのテスト

オブジェクトに適用されたアタッチメントは、メイン・ウィンドウのツールバーの [テスト] アイコン  をクリックしてテストすることができます。ウィンドウ・システムを使用してテスト・ウィンドウ・サイズを変更します。パネルのサイズを変更するときに、オブジェクトの振る舞いを確認できます。変更の必要がある場合、[テスト] アイコン  をもう一度クリックしてテスト・パネルを閉じ、アタッチメント・モードで変更を行います。

[編集] メニュー

メニュー・パレットは、パネルで使用できる 3 種類のメニュー・ガジェットを提供します。このセクションでは、これらのオブジェクトの作成方法について説明します。

- *メニュー・バー*
- *ポップアップ・メニュー*
- *ツールバー*

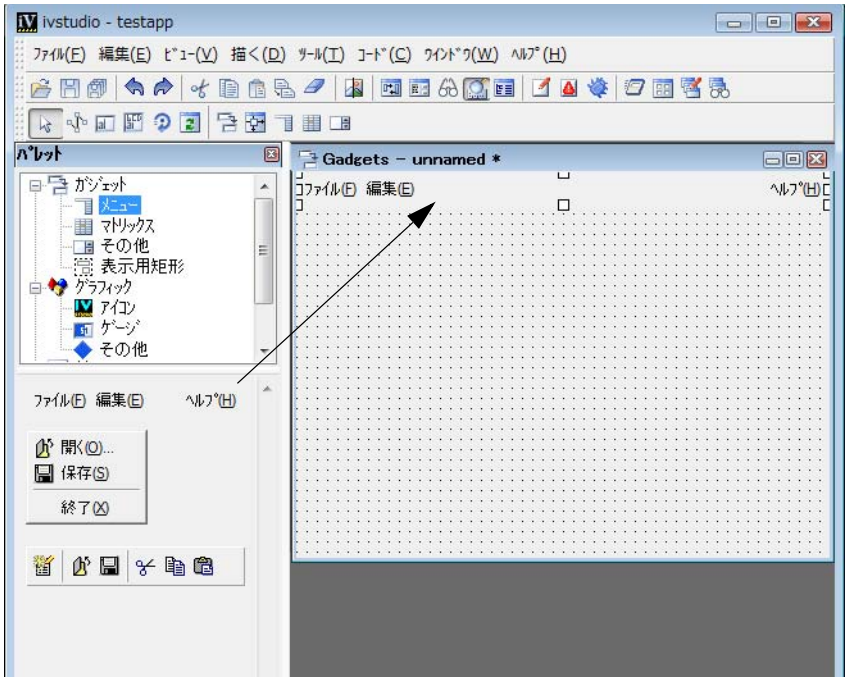
メニュー・バー

パネルにメニュー・バー (`IlvMenuBar`) を挿入するには、次の処理を行います。

1. パレット・パネルのトップ・ペインで、メニューをクリックします。

   メニュー・パレットが、パレット・パネルの下側ペインに表示されます。

2. メニュー・バー・ガジェットをクリックして、これをアクティブの Gadgets バッファ・ウィンドウまでドラッグします。



メニュー・バーは、パネルの幅になるように自動的にリサイズされ、パネル変更を反映させるためにデフォルトの水平アタッチメントが設定されます (固定、伸縮、固定)。

このメニュー・バーの詳細設定を行うには、メイン・ウィンドウ・ツールバーから [詳細設定] アイコンをダブルクリックまたはクリックします。詳細設定は、次のようになります。

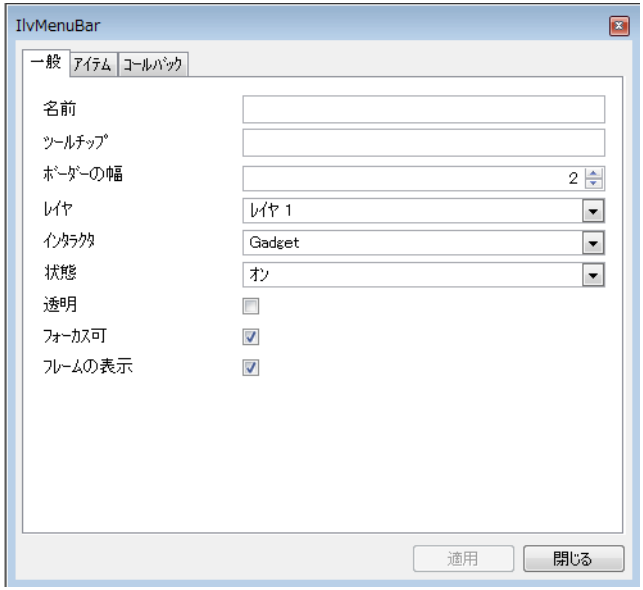


図 2.12 *メニュー・バーの詳細設定パネル ([一般] ページ)*

[アイテム] ページのオプションで、選択したメニュー・バーからアイテムの挿入、追加、削除ができます。メニュー・アイテムのセット間、あるいはポップアップ・メニューにセパレータを追加することもできます。

ページの左側は、メニュー・バーの構造をツリー表示します。メニュー・バー全体あるいはメニュー・バーを構成するアイテムのいずれかに変更を適用するには、ツリーで該当するアイテムを選択して、ページの右側で適宜変更を行います。[適用] をクリックして、変更を確認します。

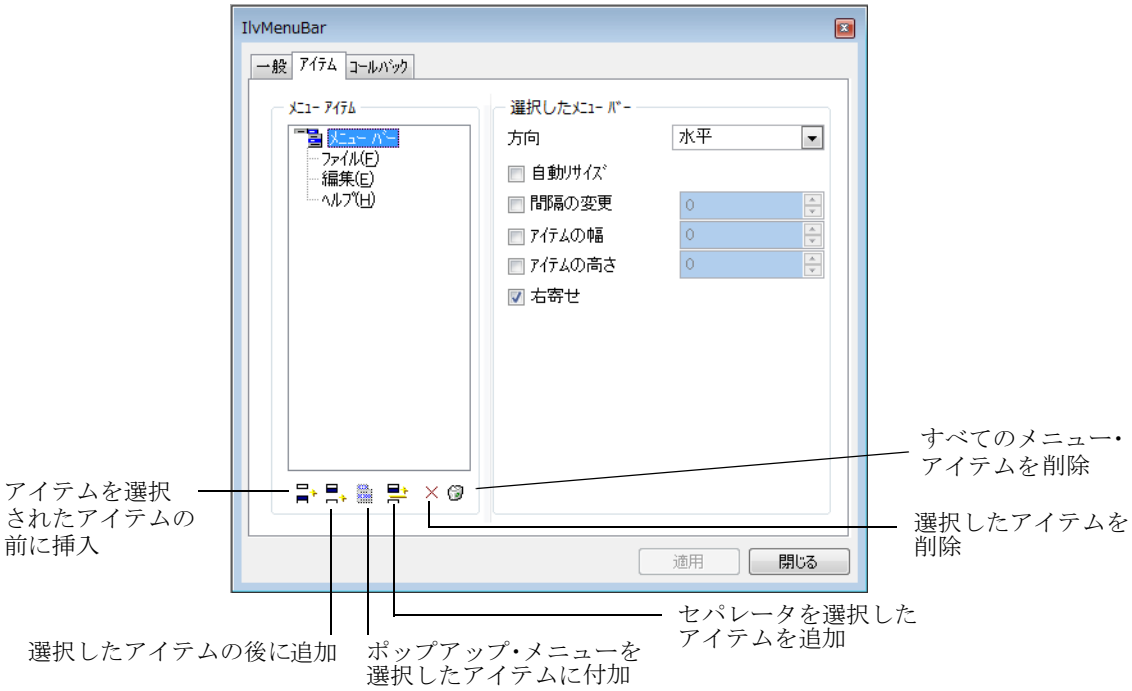


図 2.13 *メニュー・バーの詳細設定パネル ([アイテム] ページ)*

ポップアップ・メニュー

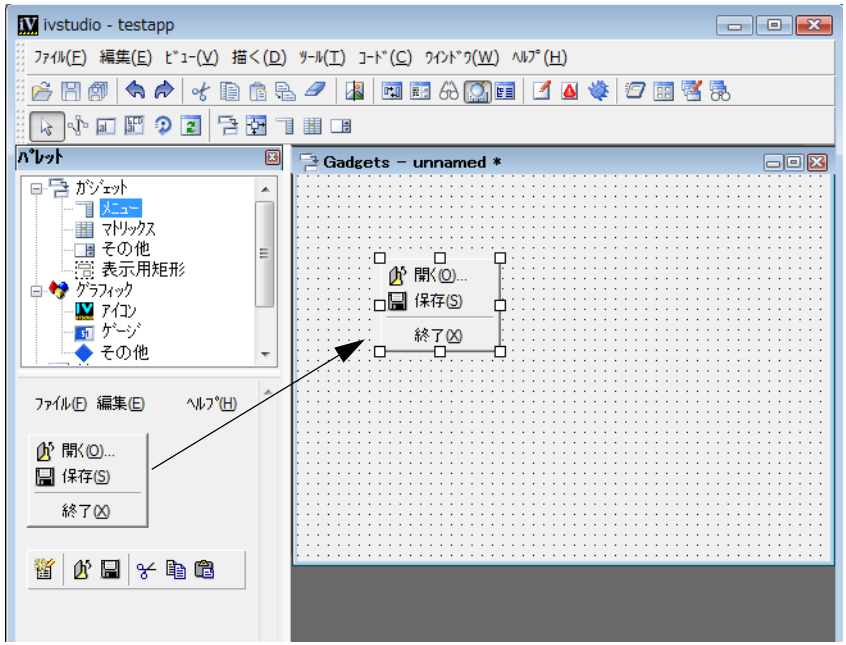
メニュー・バーに付加する前に、ポップアップ・メニューを挿入し、作業領域で編集する必要があります。

ポップアップ・メニュー (`IlvPopupMenu`) を作業領域に挿入するには、次の処理を行います。

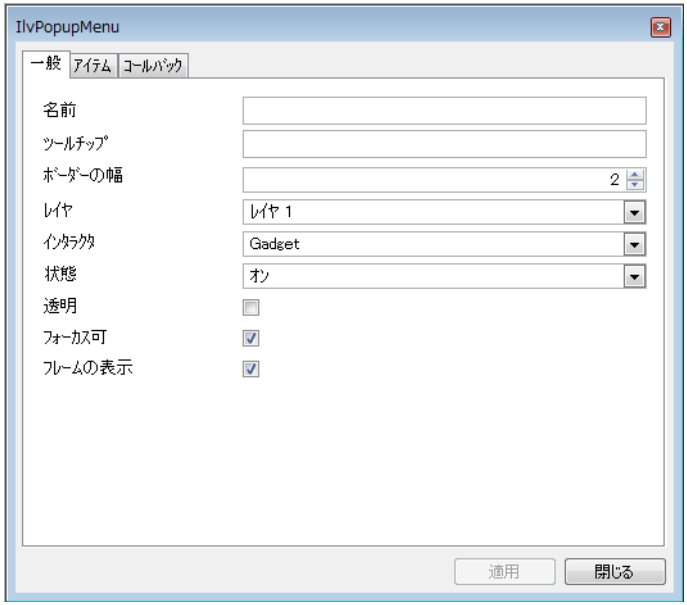
1. パレット・パネルのトップ・ペインで、メニューをクリックします。

   メニュー・パレットが、パレット・パネルの下側ペインに表示されます。

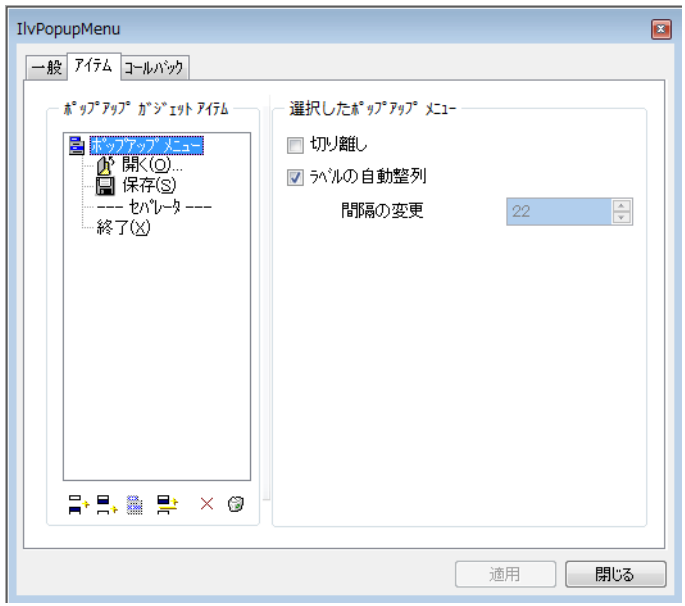
2. ポップアップ・メニュー・ガジェットをクリックして、これを Gadgets バッファ・ウィンドウまでドラッグします。



3. ポップアップ・メニューをダブルクリックしてその詳細設定パネルを表示します。



4. バッファ・ウィンドウのポップアップ・メニューでアイテムを挿入、追加、削除するには、ポップアップ・メニューの詳細設定にある [アイテム] ページを使用します。メニュー・アイテムのセットの間にセパレータを追加することもできます。



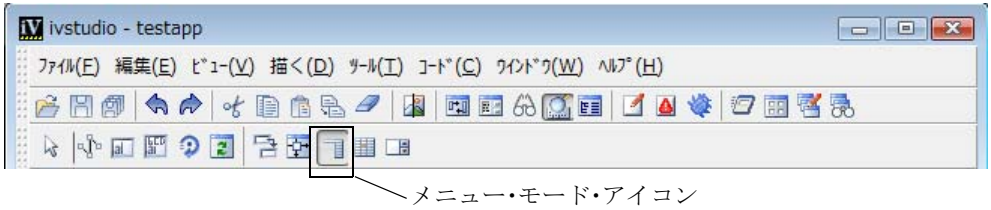
ページの左側は、ポップアップ・メニューの構造をツリー表示します。ポップアップ・メニュー全体あるいはポップップ・メニューを構成するアイテムのいずれかに変更を適用するには、ツリーで該当するアイテムを選択して、ページの右側で適宜変更を行います。

5. [適用] をクリックして、変更を確認します。

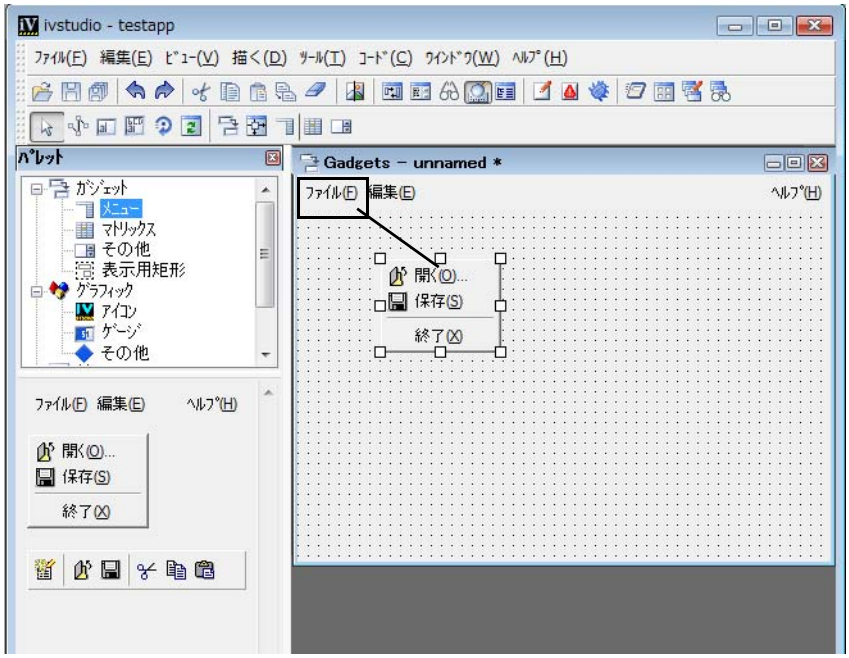
メニュー・バーにポップアップ・メニューを付加する

メニュー・アイテムにポップアップ・メニューを付加するには、次の処理を行います。

1. 編集モード・ツールバーの [メニュー] モードのアイコンをクリックします。



2. ポップアップ・メニューをクリックして、付加するメニュー・バー・アイテムまでマウスをドラッグします。



マウスをドラッグすると、2 つのアイテムをリンクする黒い線が表示されます。ポップアップ・メニューは、マウスボタンを放すと消えます。

メニュー・バーあるいはその他のポップアップ・メニュー・アイテムに付加されたポップアップ・メニューを編集するには、次の処理を行います。

1. [メニュー] モードに戻ります。
2. ポップアップ・メニューを付加するメニュー・アイテムをダブルクリックします。

   そのサブメニューの切り離し、選択、編集ができます。

   - ポップアップ・メニュー・アイテムを変更するには、作業領域からオブジェクトをドラッグして、該当アイテムの上に置きます。そのオブジェクトは、作業領域から削除されます。
   - メニュー・アイテムに使用されるオブジェクトのコピーを取得するには、該当アイテムをドラッグしてメニューの外側に置きます。これでそのコピーを編集し、メニュー・アイテムに戻すことができます。

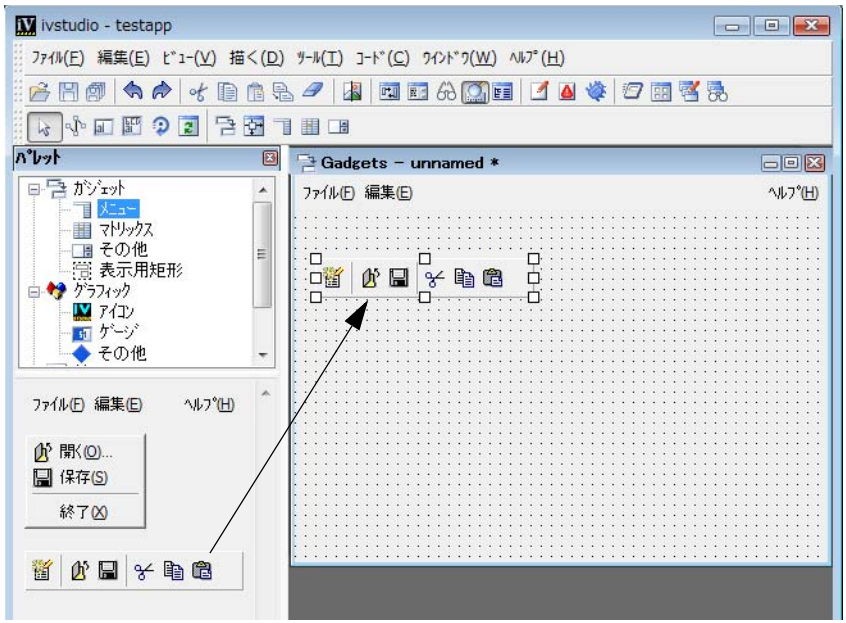
ツールバー

パネルにツールバー (`IlvToolBar`) を挿入するには、次の処理を行います。

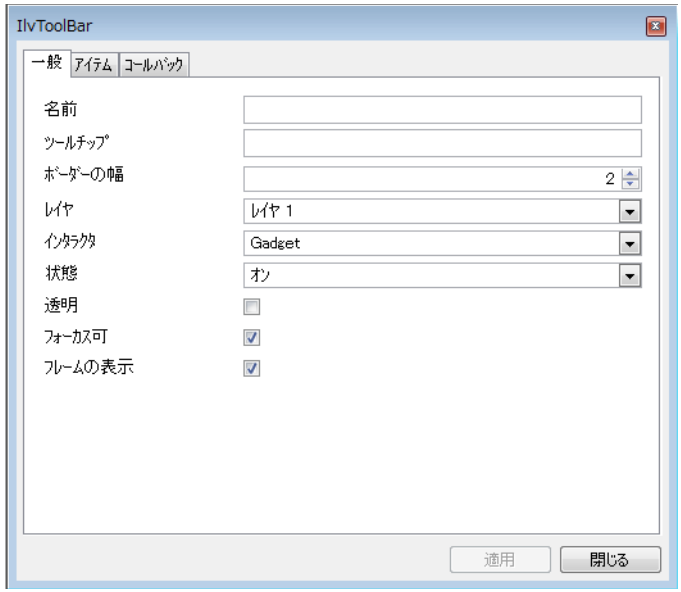
1. パレット・パネルのトップ・ペインで、メニューをクリックします。

   メニュー・パレットが、パレット・パネルの下側ペインに表示されます。

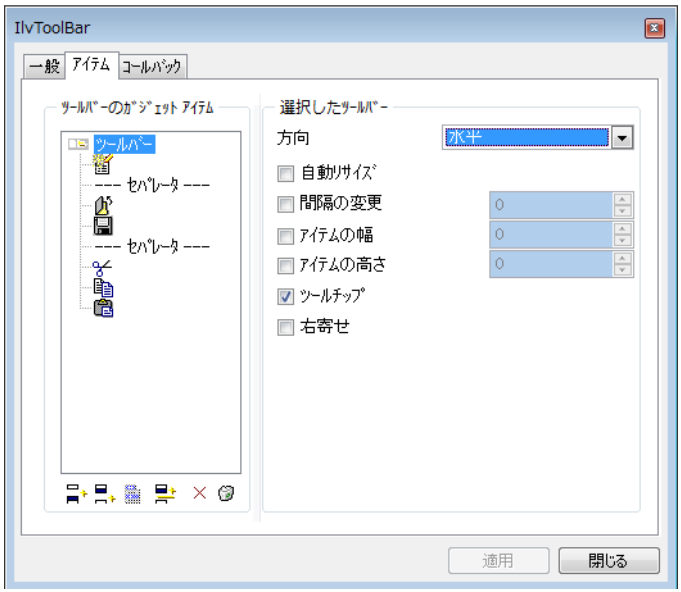
2. ツールバー・ガジェットをクリックして、これを Gadgets バッファ・ウィンドウまでドラッグします。



3. ツールバーをダブルクリックして、その詳細設定パネルを開きます。



4. 選択したツールバーでアイテムを挿入、追加、削除するには、ツールバーの詳細設定にある [アイテム] ページを使用します。ツールバー・アイテムのセットの間にセパレータを追加することもできます。



ページの左側は、ツールバーの構造をツリー表示します。ツールバー全体あるいはツールバーを構成するアイテムのいずれかに変更を適用するには、ツリーで該当するアイテムを選択して、ページの右側で適宜変更を行います。

ツールバーは、水平または垂直に配置することができます。さらに、ツールバー・アイテムでツールチップを表示させたり、ポップアップ・メニューに付加することもできます。

マトリックスの使用

マトリックス・モードを使用すると、`IlvMatrix` あるいは `IlvSheet` オブジェクトや、それらの一般プロパティを編集できる各詳細設定パネルで、アイテムの変更ができます。

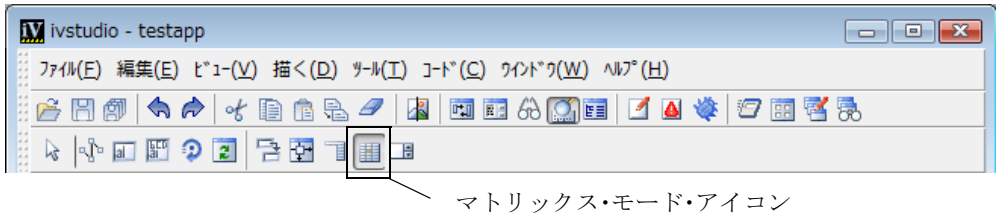
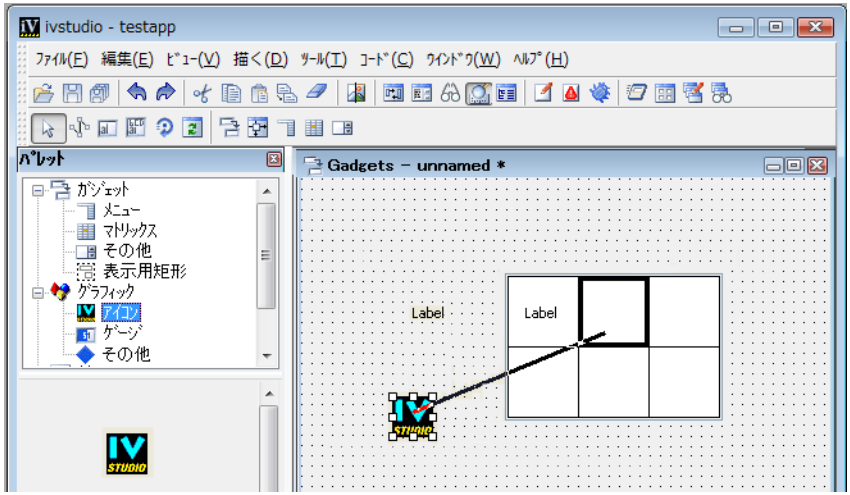


図2.14 *マトリックス・モード・アイコン*

マトリックス・アイテムの設定

作業領域からオブジェクトをドラッグしてこれを希望するアイテム内にドロップすると、マトリックス・アイテムを設定することができます。ドラッグしたオブジェクトのコピーが作成され、マトリックス・アイテム内に置かれます。ソース・オブジェクトは作業領域で引き続き利用可能です。



ドラッグしたオブジェクトが `IlvLabel` オブジェクトの場合、新しいマトリックス・アイテムが `IlvLabelMatrixItem` になります。ドラッグしたオブジェクトがアイコン (クラス `IlvIcon` のもの、あるいは派生クラスのもの) である場合、新しいアイテムが `IlvBitmapMatrixItem` になります。マトリックスおよびアイテムのクラスは、*IBM ILOG Views リファレンス・マニュアル*に文書化されています。

マトリックス・アイテムの抽出

アイテムをドラッグして作業領域にドロップすると、オブジェクトをマトリックスアイテムから抽出することができます。抽出されたオブジェクトは編集して元の場所に戻す、あるいはその他のアイテムにコピーすることができます。

マトリックス・アイテムの詳細設定

マトリックス・モードはアイテムの詳細設定を提供します。セルの詳細設定を行うには、セル上でダブルクリックします。次の詳細設定パネルが表示されます。

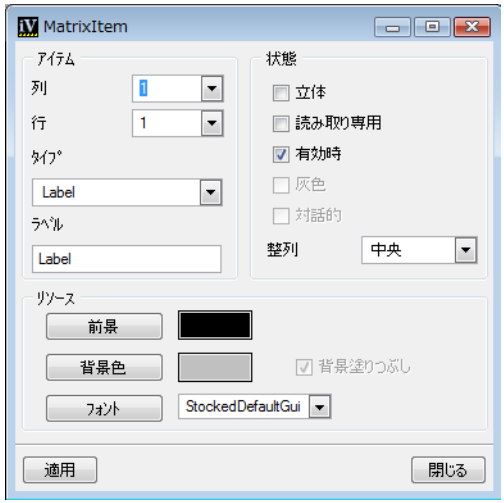


図2.15　マトリックス・アイテムの詳細設定パネル

アイテムの参照

マトリックス・アイテムの詳細設定パネルで次の詳細設定ができます。

- 1 つのセル
- 列のすべての行
- 行のすべての列
- マトリックス全体

列および行フィールドは、詳細設定を行ったセルの座標系を示しています。全列を詳細設定する場合は、列フィールドに「*」を入力します。全行を詳細設定する場合は、行フィールドに「*」を入力します。

アイテムのタイプ

タイプ・オプション・メニューで、詳細設定を行ったセルに対してマトリックス・アイテムのクラスを選択することができます。

オプション	マトリックス・アイテムのクラス	リソースのあるマトリックス・アイテムクラス
空	なし (空のセル)	
ラベル	`IlvLabelMatrixItem`	`IlvFilledLabelMatrixItem`
Int	`IlvIntMatrixItem`	`IlvFilledIntMatrixItem`

オプション	マトリックス・アイテムのクラス	リソースのあるマトリックス・アイテムクラス
浮動小数点	IlvFloatMatrixItem	IlvFilledFloatMatrixItem
倍精度	IlvDoubleMatrixItem	IlvFilledDoubleMatrixItem
ビットマップ	IlvBitmapMatrixItem	
グラフィック	IlvGraphicMatrixItem	
ガジェット	IlvGadgetMatrixItem	

上の表の最後の列は、前景、背景、あるいはラベル用フォント、あるいは数値アイテムを選択したときに使用されるマトリックス・アイテムのクラスを表しています。詳細は、クラスに該当する *IBM ILOG Views リファレンス・マニュアル*のセクションを参照してください。

アイテム・フラグ

有効時、読み取り専用、立体表示、対話的などのトグル・ボタンを使用して、アイテムに該当するフラグを設定します。

トグル・ボタン	参照するクラス	get 関数	set 関数
有効時	IlvMatrix	isItemSensitive	setItemSensitive
読み取り専用	IlvMatrix	isItemReadOnly	setItemReadOnly
灰色	IlvMatrix	isItemGrayed	setItemGrayed
立体	IlvMatrix	isItemRelief	setItemRelief
対話的	IlvGadgetMatrixItem	isInteractive	setInteractive

アイテム・リソース

前景、背景、フォントのフィールドを使用して、選択したアイテムについて色やフォントを設定します。

検査

[適用] をクリックして、マトリックス・アイテムの詳細設定パネルで編集した特性を検査します。

スピン・ボックスの編集

スピン・ボックスをパネルに含める場合は、スピン・ボックスを挿入してスピン・ボックスに表示するアイテムのタイプを指定します。

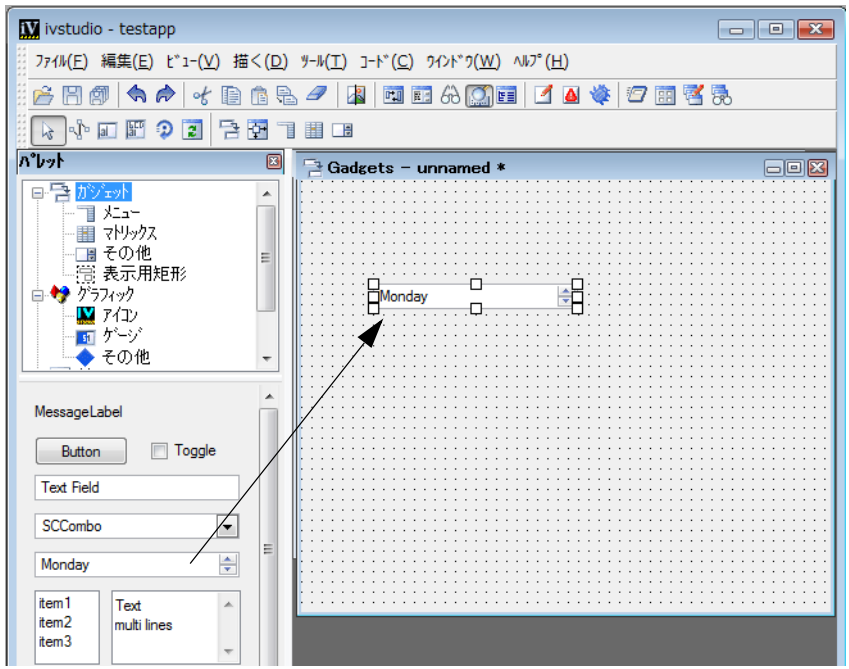
スピン・ボックスの挿入

パネルにスピン・ボックス (`IlvSpinBox`) に挿入するには、次の処理を行います。

1. パレット・パネルの上側ペインで [ガジェット] をクリックします。

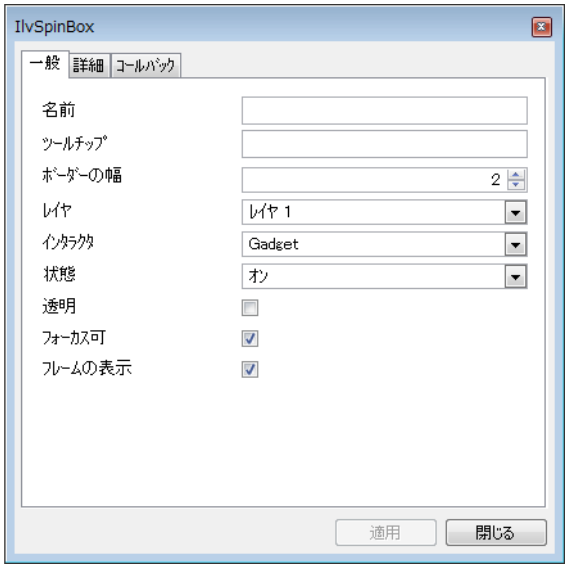
   ガジェット・パレットが、パレット・パネルの下側ペインに表示されます。

2. スピン・ボックス・ガジェットをクリックして、これを Gadgets バッファ・ウィンドウまでドラッグします。



3. スピン・ボックス・ガジェットをダブルクリックして、その詳細設定を表示します。

4. スピン・ボックスの詳細設定を使用して、スピン・ボックスに表示されるアイテムを編集します。



詳細設定の [詳細] ページで、スピン・ボックスにフィールドの追加、フィールドに表示する値の指定、さらにスピン・ボックス内にスピン矢印がどのように表示するかを指定することができます。

スピン・ボックス・アイテムのタイプを設定する

スピン・ボックス・ガジェット内のデフォルト・アイテムは、`IlvTextField` オブジェクトです。スピン・ボックス編集モードを使用して、スピン・ボックス・アイテムとして表示されるガジェット・オブジェクトのタイプを指定することができます。たとえば、`IlvTextField` ではなくて `IlvNumberField` をスピン・ボックス・アイテムにする場合を考えてみましょう。

`IlvNumberField` をスピン・ボックスのアイテムとして設定するには、次の処理を行います。

1. スピン・ボックス・ガジェットをパレット・パネルから Gadgets バッファ・ウィンドウへドラッグします。
2. スピン・ボックス・ガジェットをダブルクリックして、スピン・ボックスの詳細設定を表示させます。
3. [詳細] タブをクリックして [詳細] ページを表示します。
4. フィールド・ボックスの `IlvTextField` アイテムを選択します。
5. フィールド・ボックス下の [削除] アイコン  をクリックします。

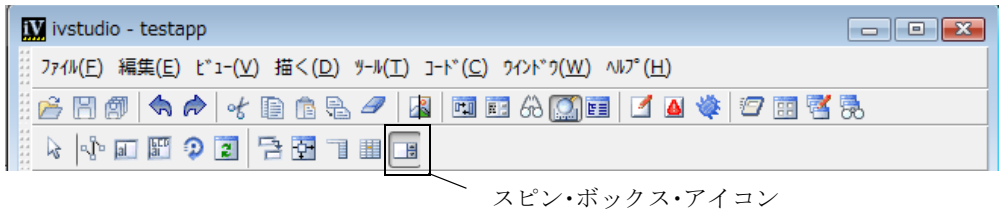
アイテムが、詳細設定の他のフィールドで定義された設定とともに削除されます。

6. パレット・パネルの上側ペインで、[その他] をクリックします。

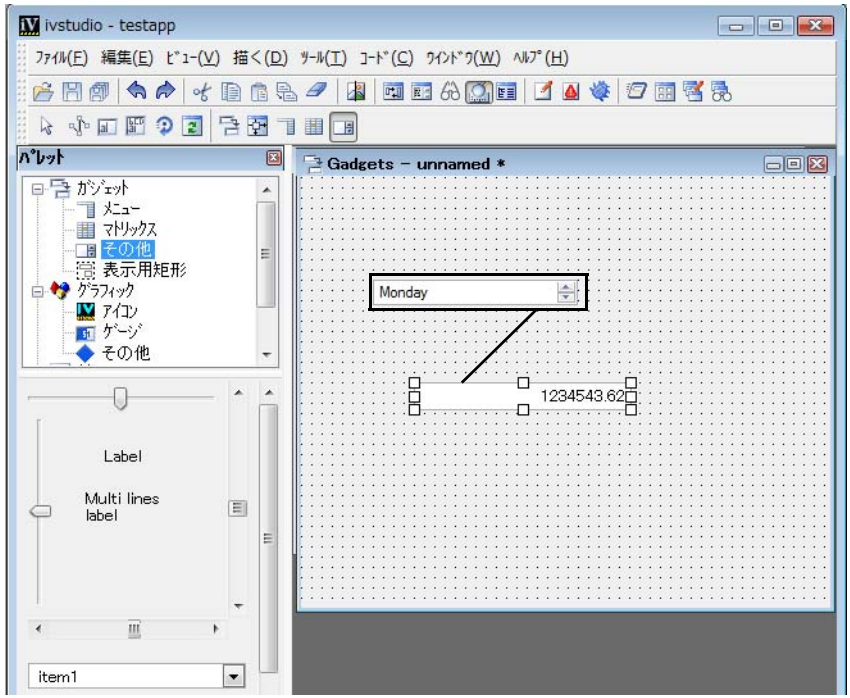
   [その他] パレットが、パレット・パネルの下側ペインに表示されます。

7. `IlvNumberField` ガジェットをクリックして、これを Gadgets バッファ・ウィンドウまでドラッグします。

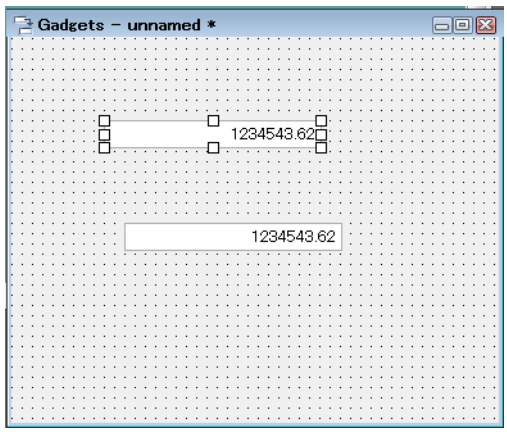
8. [編集モード] ツールバーの [スピン・ボックス] アイコンをクリックします。



9. 数値フィールド・ガジェットからラインをスピン・ボックス・ガジェットにドラッグします。



10. スピン・ボックスは、数値フィールドを含み、スピン・ボックスの詳細設定でアイテムの設定を指定できます。



章

3

アプリケーションの編集

この章では、アプリケーション・バッファの処理方法について説明します。アプリケーション・バッファは、パネル・クラスから派生するパネル・インスタンスを含んでいます。パネル・クラスおよびパネル・インスタンスは、クラス・パレットと呼ばれる特殊パレットを使用して作成、処理されます。

この章は、以下のトピックに分かれています。

- *アプリケーション・バッファ*
- *アプリケーション記述ファイル*
- *他の生成ファイル*
- *アプリケーションの詳細設定*
- *アプリケーションの編集*

アプリケーション・バッファ

IBM® ILOG® Views Studio では、アプリケーション・バッファ・ウィンドウを通じてアプリケーションを編集します。IBM ILOG Views Studio を起動すると、「testapp」という名前のデフォルト・アプリケーションが作成されます。アプリケーション・バッファ・ウィンドウをアクティブにするには、<アプリケーション> をウィンドウ・メニューから選択する、あるいはアプリケーション・バッファ・

ウィンドウをクリックします (デフォルトでは、空のアプリケーション・バッファ・ウィンドウが起動時に表示されます)。1 度に編集できるのは 1 つのアプリケーションだけです。

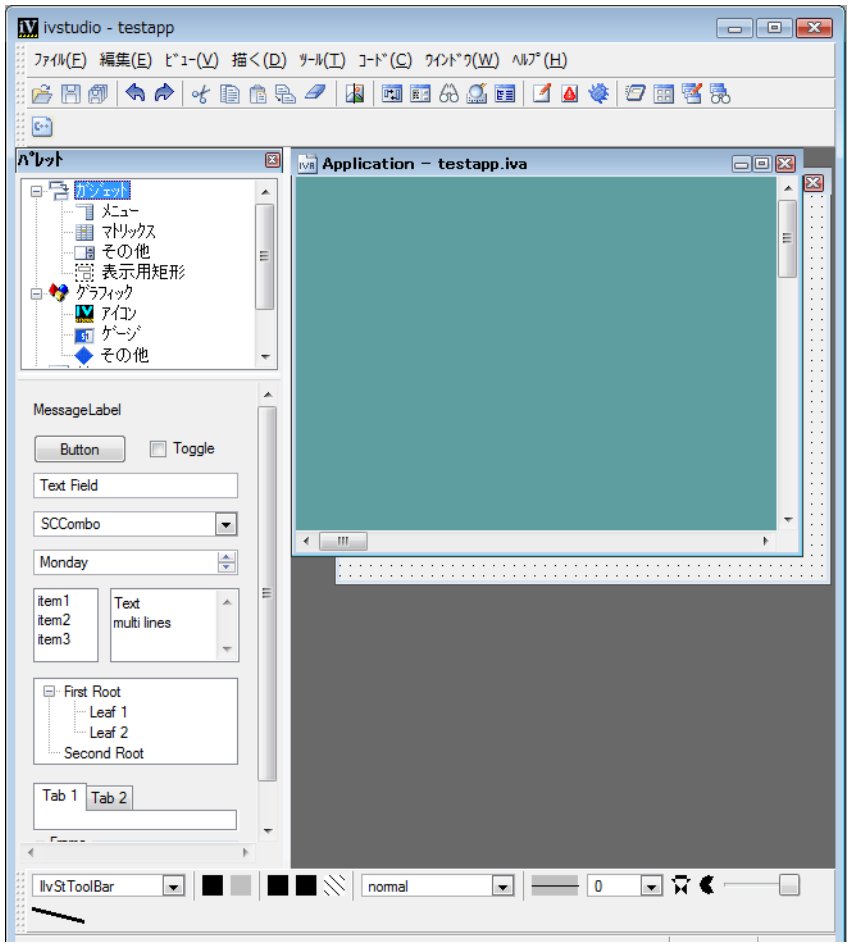


図 3.1 *メイン・ウィンドウ内のアプリケーション・バッファ・ウィンドウ*

アプリケーション・バッファ・ウィンドウがアクティブになると、次のように表示されます。

- メイン・ウィンドウのタイトル・バーがアプリケーション・ファイル名を反映させるため変更され、`<Application>` という語が続きます。
- カレント・バッファ・タイプ、アプリケーションがメイン・ウィンドウの右下に表示されます。

- メイン・ウィンドウの上分に表示される編集モード・ツールバーは、一般コマンドに対応するシングル・アイコンを含んでいます。
- オブジェクト詳細情報が消えます。

アプリケーションを編集する場合は、アプリケーション・バッファ・ウィンドウをクラス・パレットと併用します。アプリケーション・バッファ・ウィンドウおよびクラス・パレットをアクティブにするには、メイン・ウィンドウ上部のツールバーで [アプリケーションの編集] アイコンをクリックします。

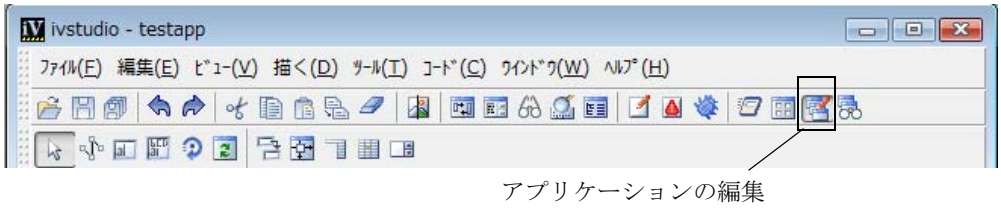


図 3.2 *アプリケーションの編集アイコン*

アプリケーション・バッファを編集するとき、メイン・ウィンドウはほぼ次のように表示されます。

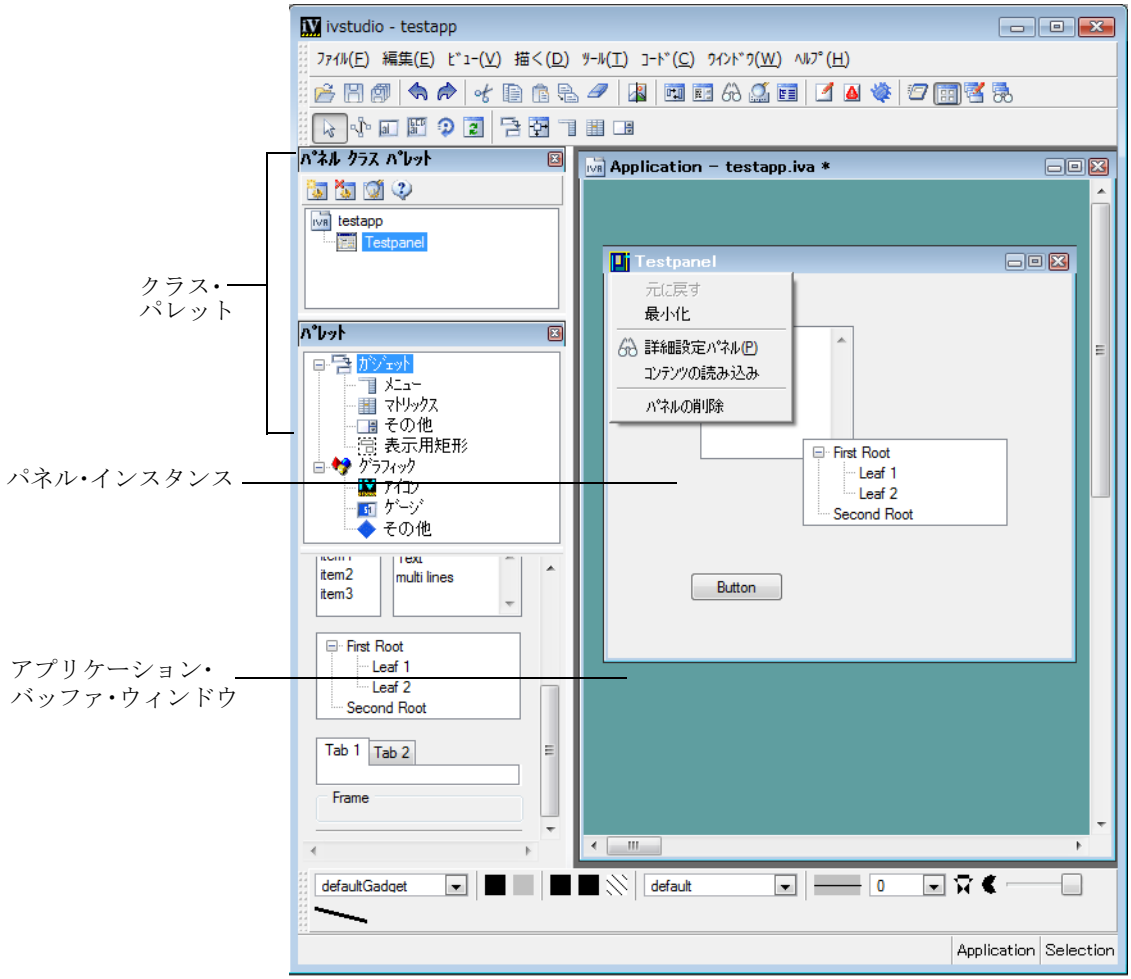


図3.3 *アプリケーション・バッファの編集*

アプリケーション・バッファ・ウィンドウは、これに追加されたあらゆるパネル・インスタンスを含んでいます。

クラス・パレットを使うと、新規パネル・クラスの作成、削除、詳細設定ができます。パレットのアイコンは、現在のアプリケーションに作成されたパネル・クラスを表しています。

アプリケーション記述ファイル

アプリケーションのプロパティ、およびアプリケーションを構成するパネル・クラスおよびパネル・インスタンスは、.iva 拡張子を持つデータ・ファイルに保存されます。このファイル形式はわかりやすくなっていますが、アプリケーション・バッファを編集するには、IBM® ILOG® Views Studio を使用したほうが良いでしょう。

1 度に編集できるのは 1 つのアプリケーションだけです。アプリケーション編集中に新しいアプリケーションを開く場合は、現在のアプリケーションを保存してから新しいアプリケーションを作成したり、以前に保存したアプリケーションを読み込みます。

[ファイル] メニューの以下のコマンドを使用して、アプリケーション記述ファイルの処理を行えます。

新規 > アプリケーション

デフォルトでは、IBM ILOG Views Studio は、起動時に空のアプリケーションを作成します (「testapp」)。< アプリケーション > をウィンドウ・メニューから選択して「testapp」の編集を始めます。あるいはこのウィンドウをクリックしてアクティブにします。

既にアプリケーションを編集していて、新規アプリケーションを作成する場合は、現在のアプリケーションを保存して、[ファイル] メニューから [新規] を選択し、表示されるサブ・メニューの [アプリケーション] を選択します。

名前を付けて保存 ...

アプリケーションを保存する前に、カレント・バッファがアプリケーション・バッファであることを確認します (必要に応じて、ウィンドウ・メニューを使用します)。新しいアプリケーションを初めて保存する場合は、[ファイル] メニューから [名前を付けて保存 ...] を選択します。このコマンドは、ディレクトリにアプリケーション記述ファイルを保存できるファイル・セレクタを開きます。アプリケーション・ファイルは .iva ファイル拡張子で保存されます。

保存

アプリケーションを保存するには、[ファイル] メニューから [保存] を選択します、あるいはツールバーから [保存] アイコン  をクリックします。このコマンドは、アプリケーション記述をデータ・ファイルに保存します。新規アプリケーション・ファイルの保存については、上述の [名前を付けて保存 ...] コマンドを参照してください。

開く ...

IBM ILOG Views Studio によって以前保存されたアプリケーションを読み込むには、[ファイル] メニューから [開く ...] を選択するか、ツールバーから [開く] アイコ

ン  をクリックします。このコマンドは、アプリケーション・ファイルを選択できるファイル・セレクタを開きます。ファイル・セレクタに表示されるファイル・リストにフィルタをかけるには、[ファイル・タイプ] オプション・メニューから [アプリケーション・ファイル] を選択します。

このコマンドで、現在のアプリケーションが破棄されるため、必要に応じて現在のアプリケーションをまず保存してください。

他の生成ファイル

データ・ファイルに加えて、IBM® ILOG® Views Studio では、各アプリケーションに次のファイルを生成します。

- 生成 C++ アプリケーション・クラスのヘッダー・ファイルおよびソース・ファイル
- 各バッファに対応するパネル・クラスのヘッダー・ファイルおよびソース・ファイル
- シンプルな `Makefile`

これらのファイルの位置は個別に指定することができ、特別な詳細設定パネルを使用して独自のコードを生成された各 C++ ファイルに挿入することができます。

アプリケーションの詳細設定

アプリケーション・プロパティを、アプリケーションの詳細設定を使用して表示したり、編集したりできます。現在のアプリケーションの詳細設定を表示するには、メイン・ウィンドウの [アプリケーション] メニューから [アプリケーションの詳細設定] を選択します、あるいはメイン・ウィンドウのツールバーの [アプリケーションの詳細設定] アイコン  をクリックします。

現在のアプリケーションのアプリケーションの詳細設定が開きます。アプリケーションをまだ開いていない場合、これをデフォルト・アプリケーションにすることができます。

アプリケーションの詳細設定には 5 つのノートブック・ページがあります。一般、オプション、ヘッダー、ソースおよびスクリプト、および 4 つのボタン、[適用]、[リセット]、[閉じる]、および [ヘルプ] です。

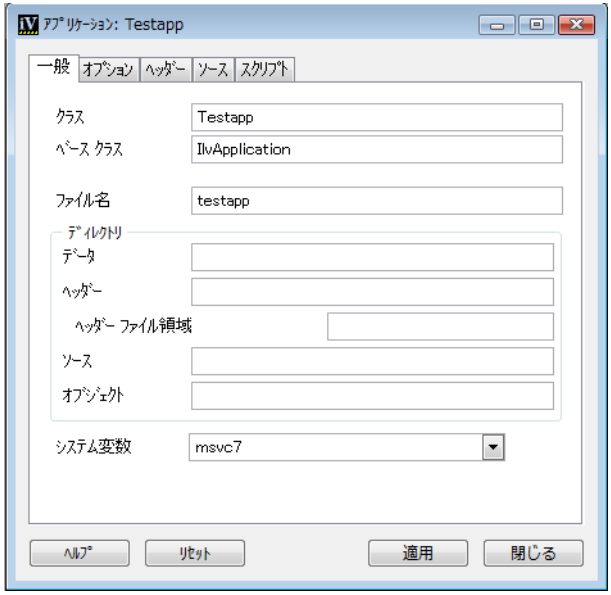


図 3.4 *アプリケーションの詳細設定にある [一般] ページ*

[一般] ページ

アプリケーションの詳細設定にある [一般] ページには次のフィールドが含まれています。

クラス　生成されたアプリケーションのクラス名は、クラス・フィールドで指定できます。デフォルトでは `Testapp` です。このフィールドで指定する名前は、有効な C++ クラス名でなければなりません。

ベース・クラス　アプリケーションのベース・クラスは、ベース・クラス・フィールドで指定できます。デフォルトでは `IlvApplication` です。このフィールドで指定する名前は、有効な C++ クラス名でなければなりません。

`IlvApplication` から生成クラスを派生させる代わりに、独自のベース・クラスを指定することができます。この場合、所定のクラスを `IlvApplication` から派生させる必要があり、互換性のあるコンストラクタを含ませなくてはなりません。もちろん、生成ファイルをコンパイルするときに、ベース・クラスの宣言をコンパイラに通知する必要もあります。つまり、生成ファイルに挿入されるか含まれる必要があります。97 ページのヘッダーおよびソース・ページを参照してください。

ファイル名　アプリケーションを含む `.iva` ファイルの名前を示します。

ディレクトリ

データ　データ・フィールドは、アプリケーション・データ・ファイルが保存されている場所を表示します。このフィールドは編集できません。アプリケーション・データ・ファイルの場所を変更するには、アプリケーション・バッファ・ウィンドウをアクティブにし、メイン・ウィンドウの [ファイル] メニューから [名前を付けて保存 ...] を選択し、これを希望のディレクトリに保存します。

ヘッダー　ヘッダー・フィールドを使用してアプリケーション・ファイルが生成される場所を指定します。デフォルトで、ヘッダー・ファイルは、アプリケーション・データ・ファイルの保存先のディレクトリに生成されます。このディレクトリは、アプリケーション・データ・ファイル・ディレクトリに関連付けられます。

ヘッダー・ファイル領域　ヘッダー・ファイルが生成されるディレクトリは、指定のヘッダー・ディレクトリにヘッダー・ファイル領域を追加して取得します。オプション [ヘッダー・ファイル領域] は、`#include` ステートメントに生成されるサブディレクトリを指定するために使用されます。

アプリケーション・ファイルがディレクトリ `/myappdir` にあると想定して、ヘッダー・ディレクトリは `include`、ヘッダー・ファイル領域は `myapp`、ヘッダー・ファイルはディレクトリ `/myappdir/include/myapp` に生成されます。アプリケーション・ヘッダー・ファイルに対応する生成された `#include` ステートメントは、次のようになります。

```
#include <myapp/file1.h>
#include <myapp/file2.h>
```

次のようにはなりませんので、ご注意ください。

```
#include <file1.h>
#include <file2.h>
```

ソース　ソース・フィールドを使用してアプリケーション・ソース・ファイルが生成される場所を指定します。デフォルトで、ソース・ファイルは、アプリケーション・データ・ファイルの保存先のディレクトリに生成されます。このディレクトリは、アプリケーション・データ・ファイル・ディレクトリに関連付けられます。

オブジェクト　オブジェクト・フィールドを使って、アプリケーションの `makefile` を生成する場所を設定します。デフォルトで、`makefile` は、アプリケーション・データ・ファイルの保存先のディレクトリに生成されます。このディレクトリは、アプリケーション・データ・ファイル・ディレクトリに関連付けられます。

システム　このメニューを使用して、`makefile` を生成するプラットフォームの名前を指定します。デフォルト・プラットフォームは IBM ILOG Views Studio が実行されているプラットフォームです。

Motif　このトグル・ボタンは、システム・オプション・メニューで選択したプラットフォームが X11 プラットフォームの場合のみ表示されます。このトグル・ボタンをオンにすると、生成された Makefile が IBM ILOG Views ライブラリの

Motif バージョンを選択し、libXt および libXm ライブラリを使用アプリケーションにリンクされます。

[オプション] ページ

アプリケーションの詳細設定にある [オプション] ページを次に示します。

図 3.5 *アプリケーションの詳細設定にある [オプション] ページ*

[オプション] ページには次のフィールドがあります。

生成

終了 独立したパネルの [終了] ボタンは、生成したアプリケーションを実行している場合にアクティブにすることができます。このボタンを使うと、アプリケーションを簡単に終了することができます。[終了] ボタンを設定する場合は、このトグル・ボタンをオンにします。

メイン () IBM ILOG Views Studio で簡単な main 関数をアプリケーション・ソース・ファイル内に生成する場合は、このトグル・ボタンをオンにします。生成したアプリケーション・クラスが Testapp で、アプリケーション・ファイル・ベース名が myappli である場合、main 関数は次のようになります。

```
main(int argc, char* argv[])
{
  Testapp* appli = new Testapp("myappli", 0, argc, argv);
  if (!appli->getDisplay())
    return -1;
  appli->run();
```

```
  return 0;
}
```

作成　シンプルな makefile を生成する場合は、このトグル・ボタンをオンにします。

パネル・アクセサ　生成したアプリケーションのメンバ関数で、アプリケーションの特定のパネルにアクセスできます。[パネル・アクセサ] トグル・ボタンをオンにすると、IBM ILOG Views Studio は、各パネル・インスタンスのメンバ関数を生成します。メンバ関数には次の署名があります。

```
MyPanelClass* getMyPanelInstance() const;
```

MyPanelClass が MyPanelInstance という名前のパネル・インスタンスのタイプである場合。パネルの名前は有効な C++ 名でなくてはなりません。

ヘッダーのインクルード　生成したアプリケーション・コードには、アプリケーションのパネル・クラスに生成したヘッダー・ファイルが含まれる必要があります。必要な #include ステートメントは、アプリケーション・ヘッダー・ファイルまたはアプリケーション・ソース・ファイルに生成することができます。パネル・アクセサを生成する場合 (パネル・アクセサ・トグル・ボタンがオン)、パネル・クラスのヘッダーがアプリケーション・ヘッダー・ファイルに含まれる必要があります。この場合、他の選択がないため、[ヘッダーのインクルード] トグル・ボタンは使用できません。それ以外の場合は、#include ステートメントをアプリケーション・ヘッダー・ファイルではなく、アプリケーション・ソース・ファイルに生成できます。アプリケーション全体のコンパイル依存を最小限に抑えるためには、このトグル・ボタンをオフにします。

ビットマップ・リーダ

[オプション] ページの下の部分には、ビットマップ・リーダに関する情報が含まれています。詳細設定のこの部分のトグル・ボタンにより、生成コードに定義済みのビットマップ・リーダを登録できます。

ヘッダーおよびソース・ページ

[ヘッダー] および [ソース] ページをヘッダーおよびソース・コードにコードを追加するために使用することができます。

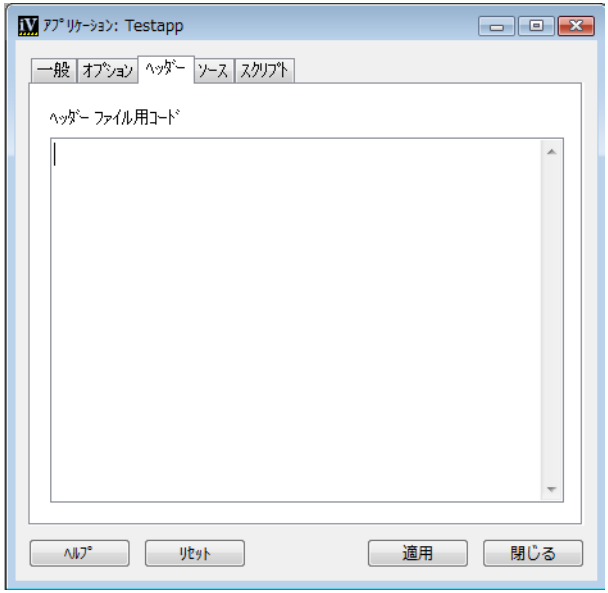


図 3.6 *アプリケーションの詳細設定にある [ヘッダー] ページ*

ヘッダー・ファイル用コード　このページに入力するテキストは、アプリケーション・ヘッダー・バッファで、生成した `#include` ステートメントの後、生成したクラスの宣言の前にそのまま挿入されます。`IlvApplication` 以外のクラスから生成されたクラスをサブクラス化する場合、`#include` ステートメントを挿入してベース・クラスを宣言するファイルに含める必要があります。もちろん、コードを挿入する代わりに、この機能を使ってアプリケーションにコメントを付けることもできます。

ソース・ファイル用コード　このパネルに入力するテキストは、生成したメンバ関数を定義する直前に、アプリケーション・ソース・ファイルにそのまま挿入されます。このテキストを使って、生成ファイルにコメントを付けることも、または任意の C++ コードを挿入することもできます。

[スクリプト] ページ

[スクリプト] ページでは、IBM ILOG Script の使用を指定することができます。

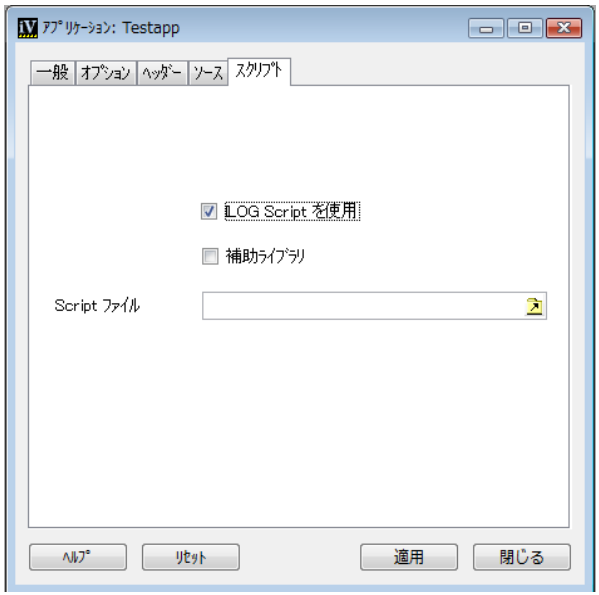


図 3.7　アプリケーションの詳細設定にある [スクリプト] ページ

[スクリプト] ページには次のフィールドがあります。

IBM ILOG Script を使用　IBM ILOG Script を使用する場合は、このトグル・ボタンを選択します。

補助ライブラリ　スクリプト言語を使用してアプリケーション内で IBM ILOG Script for IBM ILOG Views の補助ライブラリを使用したい場合、このトグル・ボタンをオンにします。このライブラリで、ダイアログ・ボックスなどの追加機能を使用することができます。詳細は、*IBM ILOG Views Foundation ユーザ・マニュアル*の「*IBM ILOG Script のプログラミング*」の章を参照してください。

Script ファイル　スクリプト・コードを含むファイル名を入力するか、テキスト・フィールドの横にあるボタンをクリックして、ファイル・セレクタからファイルを選択します。

アプリケーションの詳細設定ボタン

これらのボタンはアプリケーションの詳細設定の下部に表示されます。

適用　アプリケーション・プロパティに対して行われた変更を適用します。

リセット　アプリケーション・プロパティを初期値に戻します。

閉じる　アプリケーションの詳細設定を閉じます。

ヘルプ　アプリケーションの詳細設定のフィールドについてオンライン・ヘルプを表示します。

アプリケーションの編集

IBM® ILOG® Views Studio では、クラス・パレットを通じてアプリケーションを編集します。パネル・クラスをクラス・パレットに追加して、アプリケーション・バッファ・ウィンドウにドラッグしてパネル・インスタンスを作成することができます。最終アプリケーションで表示されるとおりに、パネル・インスタンスが表示されます。パネルの寸法および位置は、アプリケーション・バッファ・ウィンドウで直接編集することができます。

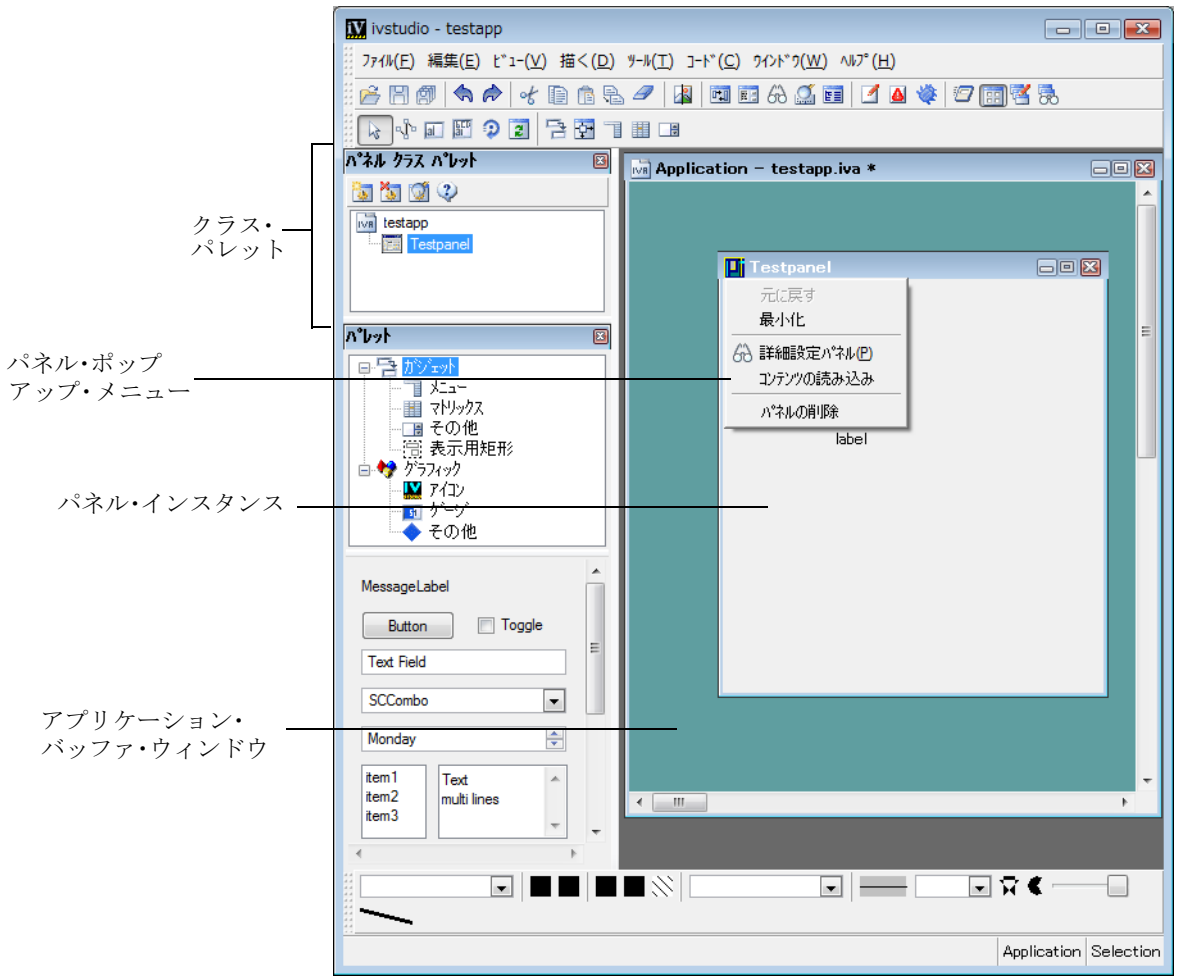


図3.8　クラス・パレット

以下のセクションでは、パネル・クラスの作成方法およびこれらのパネル・クラスのインスタンスをアプリケーションに追加する方法について説明します。

クラス・パレット

クラス・パレットを使って、パネル・クラスの作成、詳細設定または削除を行います。このパレットは、カレント・バッファが何であれ、メイン・ウィンドウのツールバーの [パネル・クラス・エディタ] アイコン  をクリックするか、[コード] メニューから [クラス・パレット] を選択してアクセスすることができます。メイン・ウィンドウのツールバーから [アプリケーションの編集] アイコン  を選択してアプリケーション・バッファとともに開くこともできます。

クラス・パレットは、パネル・クラスを操作するのに使用されるコマンドを含むツールバー、および既存のパネル・クラスを意味するアイコンを示すパネル・クラス・バッファから構成されています。

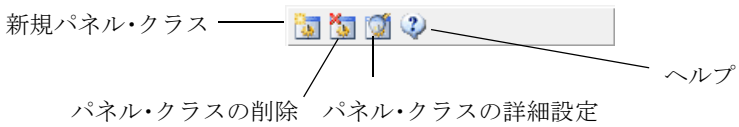


図3.9 *クラス・パレットのツールバー*

パネル・クラス・パレット・ツールバーのコマンドは、次の機能を提供します。

- **新規パネル・クラス**　カレント・バッファから新規パネル・クラスを作成します。カレント・バッファは、既に保存されている必要があります。
- **パネル・クラスの削除**　パレットから選択したパネル・クラスを削除します。
- **パネル・クラスの詳細設定**　選択したパネル・クラスの詳細設定を開きます。
- **ヘルプ**　クラス・パレットのオンライン・ヘルプにアクセスします。

クラス・パレットの [パネル・クラス] アイコンをダブルクリックすると、パネル記述を含むファイルが開き、カレント・バッファとして設定されます。

クラス・パレットの背景をダブルクリックすると ([パネル・クラス] アイコンはクリックしない)、アプリケーション・バッファがカレント・バッファとして設定されます。

パネル・クラス

パネル・クラスは、クラス・パレットを使用して作成できます。これらのクラスは、パネル・インスタンスをアプリケーション・バッファに追加する場合にも使用できるようになります。

編集済みのアプリケーションの一部である各 Gadgets バッファに対して、IBM ILOG Views Studio は `IlvGadgetContainer` から派生する C++ クラスを生成します。生成されたクラスは、次を実行します。

- パネル・オブジェクトを作成するのに使用されるデータを読み込む。
- メソッドとしてコールバックを生成する。
- 名前が付いたオブジェクトにアクセサを生成する。

パネル・クラスの追加

アプリケーションに新規パネル・クラスを追加するには、次の処理を行います。

1. 必要なパネル・バッファが開いており、カレント・バッファとなっているかどうか確認してください。
2. メイン・ウィンドウのツールバーの [パネル・クラス・エディタ] アイコン  をクリックして、クラス・パレットを開きます。
3. [パネル･クラス] パレット･ツールバーの [新規パネル･クラス] アイコン  をクリックします。

   新規パネル・クラスがパレットに追加されます。

パネル・クラスの削除

アプリケーションからパネル・クラスを削除するには、次の処理を行います。

1. クラス・パレットで、削除したいパネル・クラスを選択します。
2. [パネル・クラス] パレット・ツールバーの [パネル・クラスの削除] アイコン  をクリックします。

パネル・クラスの詳細設定

パネル・クラスの詳細設定を行うには、クラス・パレット・ツールバーの [パネル・クラスの詳細設定] アイコン  をクリックします。

パネル・クラスの詳細設定が表示されます。

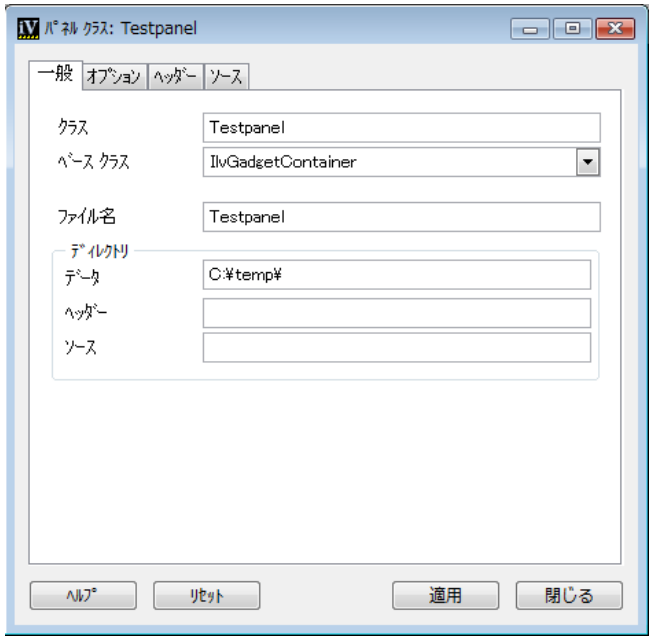


図3.10 *パネル・クラスの詳細設定*

パネル・クラスの詳細設定には、4 つのノートブック・ページがあり、それぞれ詳細設定されたパネル・クラスのプロパティのセットを含んでいます。

[一般] ページ

クラス　このフィールドを使って、C++ パネル・クラスに名前を付けます。このクラス名は、有効な C++ クラス名でなければなりません。デフォルトで、IBM ILOG Views Studio は、対応するバッファ名の頭文字を大文字にしてクラスに名前を付けます。

ベース・クラス　このフィールドを使って、生成したクラスにベース・クラスを指定します。ガジェット・バッファのデフォルトでは、IBM ILOG Views Studio は、`IlvGadgetContainer` から生成クラスを派生します。

`IlvGadgetContainer` から生成パネル・クラスを派生させる代わりに、独自のベース・クラスを指定することができます。この場合、任意のクラスは `IlvGadgetContainer` から派生されたもので、互換性のあるコンストラクタを含んでいなくてはなりません。もちろん、生成ファイルをコンパイルするときに、ベース・クラスの宣言がコンパイラに知らされなくてはなりません。つまり、生成ファイルに挿入されるか含まれる必要があります。106 ページのヘッダーおよびソース・ページを参照してください。

ファイル名　このフィールドには、選択したパネル・クラスを含むファイルの名前が表示されます。これは編集できません。

データ　このフィールドには、パネル・データ・ファイル (.ilv ファイル) が保存されている場所が表示されます。

ヘッダー　このフィールドを使って、パネル・クラス・ヘッダー・ファイルを生成する場所を設定します。このフィールドが空欄の場合、ヘッダー・ファイルは、アプリケーション・ヘッダー・ファイルと同じディレクトリに生成されます。

ソース　このフィールドを使って、パネル・クラス・ソース・ファイルを生成する場所を設定します。このフィールドが空欄の場合、ソース・ファイルは、アプリケーション・ソース・ファイルと同じディレクトリに生成されます。

[オプション] ページ

パネル・クラスの詳細設定にある [オプション] ページを下に表します。

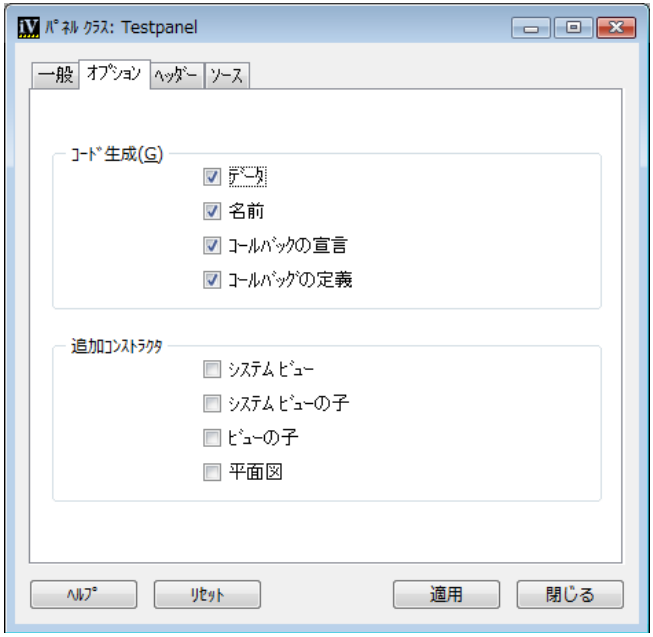


図 3.11　*パネル・クラスの詳細設定にある [オプション] ページ*

[コード生成] セクションは次を含みます。

データ　このトグル・ボタンをオンにして、IBM ILOG Views Studio に C++ パネル・クラス・コードにデータ文字列を生成させます。これにより、実行時にそのコンストラクタでデータ・ファイルを読み込む必要がなくなります。データを使って生成されたコードは、UNIX® プラットフォームでのみ使われます。Windows® システムでは、生成した文字列データは使用されません。

名前　このトグル・ボタンをオンにして、IBM ILOG Views Studio に、パネル内の名前の付いたオブジェクトを返すメンバ関数を生成します。たとえば、パネルに

MyTextField という名前のテキスト・フィールドがある場合、次のメンバ関数が生成されます。

```
IlvTextField* getMyTextField() const
  { return (IlvTextField*)getObject("MyTextField"); }
```

生成されたメンバ関数は常に、次の規則に従って命名されます。

コールバックの宣言　IBM ILOG Views Studio は、コールバックを扱う簡単な方法を用意しています。[コールバックの宣言] トグル・ボタンをオンにすると、IlvGraphicCallback 関数が生成され、デフォルトの仮想メンバ関数が宣言されます。生成した IlvGraphicCallback は、IBM ILOG Views Studio に指定したコールバックと同じ名前の仮想メンバ関数を呼び出します。したがって、コールバックに使用する名前は、有効な C++ 関数の名前でなければなりません。

[コールバックの定義] トグル・ボタンを有効にするには、[コールバックの宣言] トグル・ボタンをオンにする必要があります。

コールバックの定義　これらのコールバック仮想メンバ関数用のデフォルトの定義コード (主関数) を生成して、実際のコールバックを定義する前にアプリケーションのテストを行うことができます。[コールバック定義] トグル・ボタンを選択したら、派生クラス内のコールバックの独自のバージョンを再定義します。

これらの関数の定義を生成しない場合は、[コールバックの定義] トグル・ボタンをオフにします。これは、生成したクラスからクラスを派生させない場合に便利です。この場合、今後のコード生成によって消去されることのない別のファイルに、これらのメンバ関数の独自の定義を書くことができます。

コールバックの登録タスクは、C++ コードで生成されるため、コールバック・メソッドを定義するだけで済みます。

オプションの追加コンストラクタ・セクションには 2 つのトグル・ボタンがあり、ネイティブのシステム・ビュー内で使用するパネル・クラスを生成することができます。

システム・ビュー　既存のシステム・ビューを使ってパネルを作成する場合は、このトグル・ボタンをオンにします。

システム・ビューの子ウィンドウ　既存のシステム・ビューの子ウィンドウとしてパネルを作成する場合は、このトグル・ボタンをオンにします。

ヘッダーおよびソース・ページ

独自のコードを生成パネル・クラスあるいはソース・ファイルに挿入するには、パネル・クラスの詳細設定で [ヘッダー] あるいは [ソース] ページをクリックします。

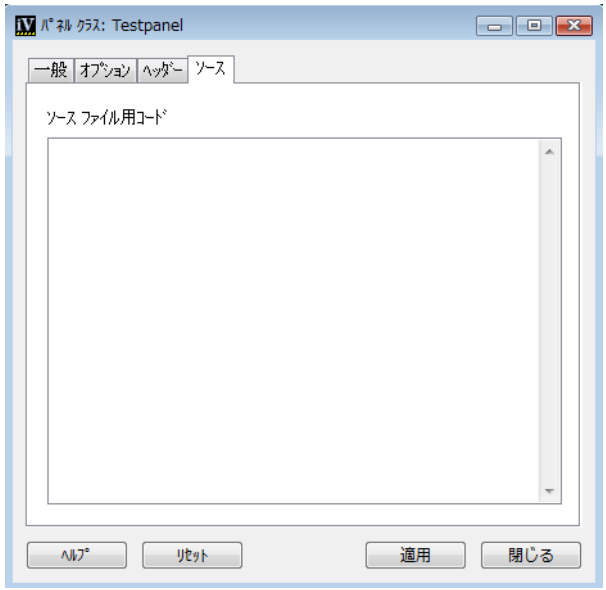


図3.12　パネル・クラスの詳細設定にある [ソース] ページ

ヘッダー・ファイル用コード　このフィールドに入力するテキストは、パネル・クラス・ヘッダー・ファイルに生成した `#include` ステートメントの後、生成したクラスの宣言の前にそのまま挿入されます。`IlvGadgetContainer` 以外のクラスから生成されたクラスをサブクラス化する場合、`#include` ステートメントを挿入してベース・クラスを宣言するファイルに含める必要があります。もちろん、コードを挿入する代わりに、この機能を使ってパネル・クラスにコメントを付けることもできます。

ソース・ファイル用コード　このパネルに入力するテキストは、生成したメンバ関数を定義する直前に、パネル・クラス・ソース・ファイルにそのまま挿入されます。このテキストを使って、生成ファイルにコメントを付けることも、または任意の C++ コードを挿入することもできます。

パネル・インスタンス

パネル・クラスがクラス・パレットで定義されると、これらのクラスのインスタンスを作成し詳細設定することができます。

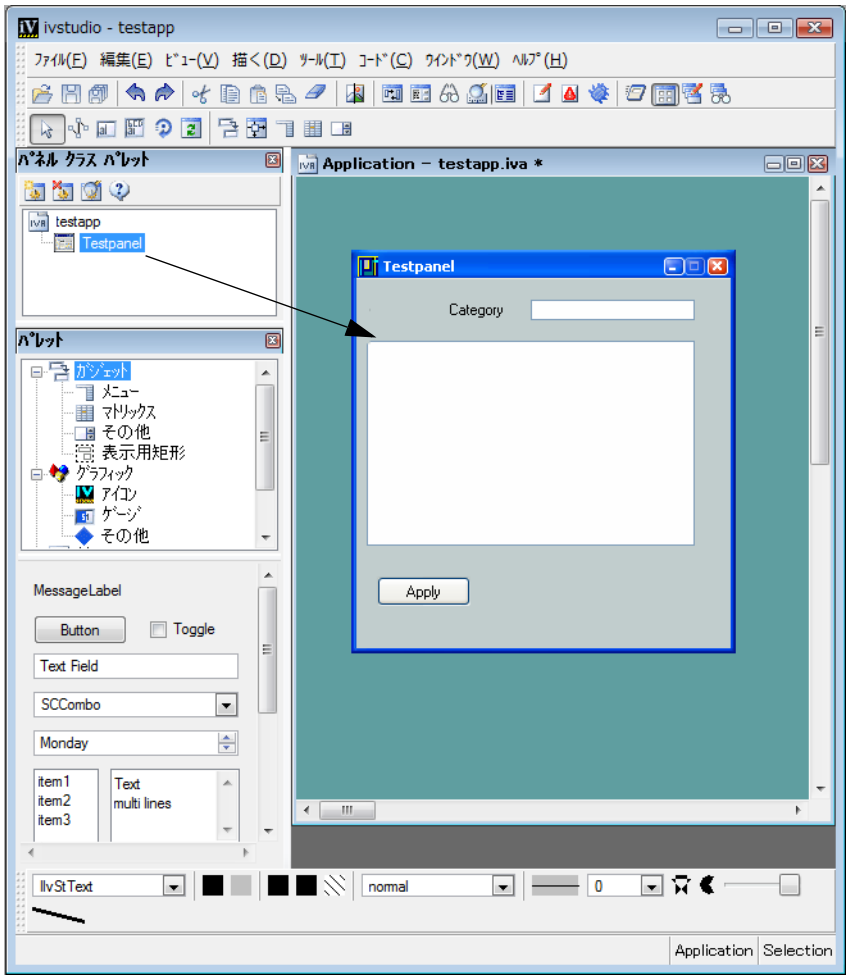
パネルをアプリケーションに追加する

パネルをアプリケーションに追加するには、次の処理を行います。

1. アプリケーション・バッファ・ウィンドウがカレント・ウィンドウであり、クラス・パレットが表示されていることを確認してください。

クラス・パレットを表示させるには、メイン・ウィンドウのツールバーの [パネル・クラス・エディタ] アイコン  をクリックします。

2. クラス・パレットで、パネル・クラスのアイコンを選択し、これを直接アプリケーション・バッファ・ウィンドウまでドラッグします。



アプリケーション・バッファ・ウィンドウにパネルのインスタンスが作成されます。このインスタンスは、ウィンドウとして表示されます。パネル・クラス名は、新規パネルのデフォルト名です。

アプリケーション・バッファでパネル・インスタンスを管理する

パネル・インスタンスがアプリケーション・バッファ・ウィンドウに追加されると、ウィンドウ環境でウィンドウを管理するのと同じ方法でこれらを管理するこ

とができます。各パネル・インスタンス・ウィンドウには、ウィンドウの左上隅をクリックすると表示されるポップアップ・メニューがあります。メニューには、[元に戻す] や [最小化] などの標準ウィンドウ・オプションがあります。

図 3.13 *パネル・ポップアップ・メニュー*

[元に戻す] メニュー・アイテムで、最小化したパネル・インスタンスを戻して、ウィンドウをオリジナルのサイズに戻すことができます。

[最小化] メニュー・アイテムは、パネル・インスタンスをそのタイトル・バーに縮小します。

パネル・インスタンスは、このメニューから [詳細設定パネル] を選択して詳細設定することができます。

[コンテンツの読み込み] メニュー・アイテムで、パネル・インスタンスの内容を読み込むことができます。`noPanelContents` オプションを使用するとき、これは便利です。(5 章　*IBM ILOG Views Studio の Gadgets 拡張機能をカスタマイズする* の「Gadgets 拡張機能設定オプション」の `noPanelContents` オプションの説明を参照してください。

パネルの削除オプションを使うと、パネル・インスタンスがアプリケーション・バッファ・ウィンドウが削除されます。

パネル・インスタンスの詳細設定

パネル・インスタンスの詳細設定を行うには、次の処理を行います。

1. パネル・インスタンスの左上隅のボックスをクリックします。

   ポップアップ・メニューが表示されます。

2. そのメニューから、詳細設定パネルを選択します。

パネル・インスタンス・ウィンドウのタイトル・バーをダブルクリックすることもできます。

パネル・インスタンスの詳細設定が表示されます。

図3.14 *パネル・インスタンスの詳細設定にある [一般] ページ*

パネル・インスタンスの詳細設定にある 4 つのノートブック・ページでパネル・インスタンスのプロパティを編集することができます。

[一般] ページ

名前　このフィールドを使って、パネルに名前を付けます。IBM ILOG Views Studio にアプリケーション用パネル・アクセサを生成する場合、この名前は、有効な C++ 名でなくてはなりません。

クラス　このテキスト・フィールドには、生成したパネルのクラス名が表示されます。このフィールドは編集できません。

ユーザ・クラス　[クラス] フィールドに表示されている、生成したクラスからの派生クラスを使用する場合、このフィールドに、その派生クラスの名前を入力します。この場合、そのクラスを宣言するファイルは、生成したアプリケーション・クラス・ファイル (97 ページの *ヘッダーおよびソース・ページ*を参照) に含まれ、その定義モジュールが最終アプリケーションにリンクされなくてはなりません。

タイトル　このフィールドを使って、パネルにタイトルを設定します。

親パネル　このオプション・メニューを使用して 2 つのパネル間の関係を設定します。このフィールドで既存パネルを選択することは、カレント・パネルが、トラン

ジェント・フィールドで選択されたパネルの前に常に表示されることを指定することになります。

終了コールバック　これは、ウィンドウ・マネージャによってパネルを閉じたときに呼び出されるコールバックです。オプション・メニューを使って、パネルのデフォルト終了コールバックを選択します。

ビットマップ　ビットマップをパネル背景として指定することができます。

表示　アプリケーションの起動時に、パネルが表示されないようにするには、この [表示] トグル・ボタンをオフにします。デフォルトでは、パネルが表示されるようになっています。

アクセラレータ　オンにすると、デフォルトのコンテナ・アクセラレータを使ってパネル・インスタンスが作成されます。つまり、パネル・コンストラクタが useacc パラメータで呼び出され、IlvTrue を設定します。

ダブル・バッファリング　オンにすると、詳細設定したパネルでダブル・バッファリング機構が使用されます。これは次のコードを生成します。

```
cont ->setDoubleBuffering(IlvTrue)
```

[プロパティ] ページ

[プロパティ] ページのトグル・ボタンを使って、ウィンドウ・フレームのプロパティを指定することができます。生成したコードで、オンにしたオプションを組み合わせて、パネル・コンストラクタへの呼び出しの properties パラメータが設定されます。各オプションは、定義済みのプロパティに対応します。

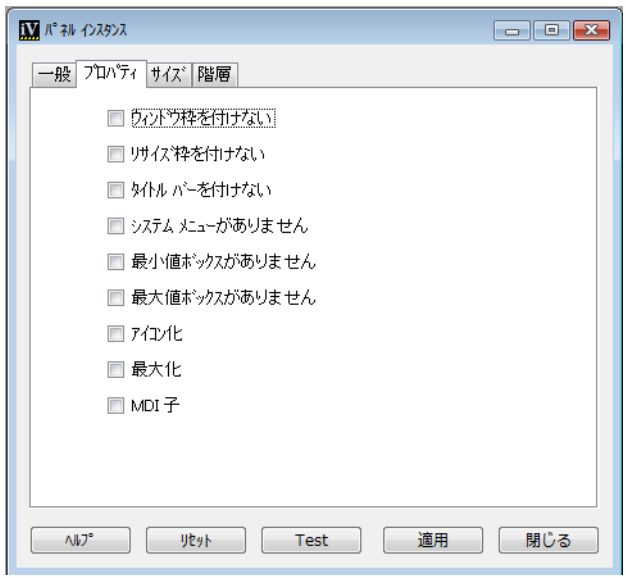


図 3.15　*パネル・インスタンスの詳細設定にある [プロパティ] ページ*

次の表は、パネル・インスタンスの詳細設定にある [プロパティ] ページのトグル・ボタンにリンクされる定義済みプロパティを表しています。

トグル・ボタン	定義済みプロパティ
ウィンドウ枠を付けない	IlvBorder
リサイズ枠を付けない	IIvNoResizeBorder
タイトル・バーを付けない	IlvNoTitleBar
システム・メニューがありません	IlvNoSysMenu
最小値ボックスがありません	IlvNoMinBox
最大値ボックスがありません	IlvNoMaxBox
アイコン化	IlvIconified
最大化	IlvMaximized
MDI 子	IlvMDIChild

[サイズ] ページ

パネル・インスタンスの詳細設定にある [サイズ] ページには、3 つのセクションがあります。バウンディング・ボックス、パネルの最小サイズおよびパネルの最大サイズの 3 つのセクションがあります。

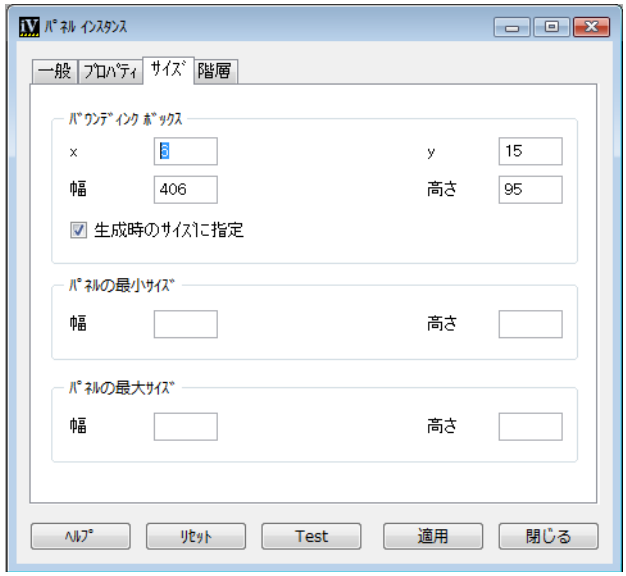


図3.16　パネル・インスタンスの詳細設定にある [サイズ] ページ

x、y、幅、高さ　これらのフィールドを使って、パネルの初期位置を指定します。デフォルトのパネル・サイズは、バッファのサイズに設定されます。パネルに新しいサイズを割り当てる場合、[生成時のサイズに指定] トグル・ボタンをオンにして、[幅] および [高さ] フィールドに目的の値を入力します。

パネルの最大および最小サイズ　これらのフィールドで、[幅] および [高さ] フィールドに目的の値を入力して、パネルの最大または最小フィールド・サイズを指定します。

パネル・インスタンス・ボタン

適用　パネル・オプションを確認するためにクリックします。

リセット　パネル・インスタンスの詳細設定を最後に確認した値に戻します。

テスト　パネルをテストするためにクリックします。アプリケーションのグローバル・テストとは異なり、この操作では、詳細設定を行っているパネルのみが表示されます。入力されているが [適用] ボタンで有効になっていないオプションは、テスト・パネルの作成に使われます。パネルが表示されるように設定されていない場合でも、テストを行うことはできます。

閉じる　パネル・インスタンスの詳細設定を閉じるためにクリックします。

ヘルプ　オンライン・ヘルプにアクセスするためにクリックします。

サブパネルの編集

アプリケーション・バッファ・ウィンドウを使用して、IlvContainerRectangle オブジェクトあるいは IlvNotebook オブジェクトに含まれているパネル・インスタンスを作成することができます。つまり、コンテナ矩形およびノートブックは、サブパネルを保持することができます。

コンテナ矩形あるいはノートブックのサブパネル、パネル・インスタンスを作成するには、次の手順に従います。

1. 対応するパレットを表示するために、パレット・パネルの上側ペインで [表示用矩形] をクリックします。
2. コンテナ矩形 (たとえば、IlvGadgetContainerRectangle) をガジェット・バッファ・ウィンドウまでドラッグします。
3. バッファを mymainpanel.ilv として保存します。
4. メイン・ウィンドウのツールバーの [パネル・クラス・エディタ] アイコン  をクリックして、クラス・パレットを表示させます。
5. クラス・パレットに Mymainpanel クラスを追加するために、[新規パネル・クラス] アイコン  をクリックします。
6. [ファイル] メニューから [新規] を選択します。表示されるサブメニューで [ガジェット] を選択します。

   新規ガジェット・バッファ・ウィンドウが開きます。
7. パレット・パネルの上側ペインで、[ガジェット] をクリックします。
8. IlvNotebook オブジェクトを現在のガジェット・バッファ・ウィンドウへドラッグします。
9. バッファを notebook.ilv として保存します。
10. クラス・パレットに Notebook クラスを追加するために、[新規パネル・クラス] アイコン  をクリックします。
11. アプリケーション・バッファ・ウィンドウをアクティブにするために、[ウィンドウ] メニューから <Application> を選択します。
12. Mymainpanel クラス・アイコンをクラス・パレットからアプリケーション・バッファ・ウィンドウにドラッグします。
13. Notebook クラス・アイコンをクラス・パレットから Mymainpanel 内のコンテナ矩形にドラッグします。

    コンテナ矩形は、この上にサブパネルをドロップすると強調表示されます。
14. [ファイル] メニューから [新規] を選択し、表示されたサブメニューで [ガジェット] を選択します。

新規ガジェット・バッファ・ウィンドウが開きます。

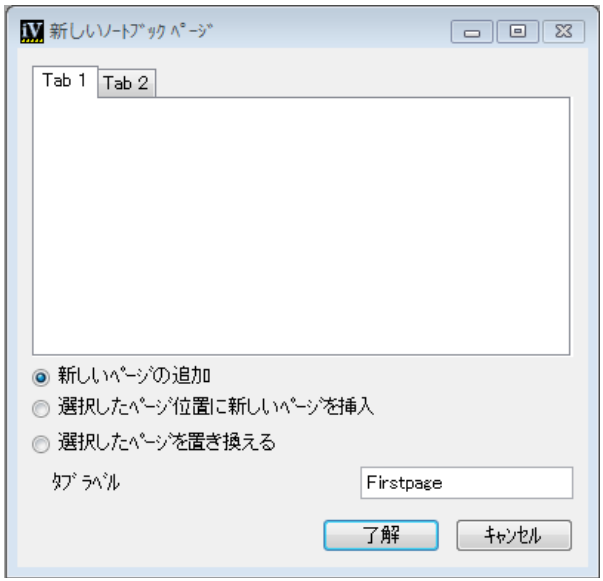
15. オブジェクトをバッファ・ウィンドウ (たとえば、テキスト・フィールドおよびメッセージ・ラベル) にドラッグし、`firstpage.ilv` として保存します。

16. クラス・パレットに `Firstpage` クラスを追加するために、[新規パネル・クラス] アイコン  をクリックします。

17. アプリケーション・バッファ・ウィンドウをアクティブにするために、[ウィンドウ] メニューから <Application> を選択します。

18. `Firstpage` クラス・アイコンをクラス・パレットからドラッグして、Mymainpanel パネルの内側のノートブック上でドロップします。

    次のダイアログボックスが表示されます。



このダイアログ・ボックスで、サブパネル・インスタンスを新規ノートブック・ページとして追加したり、既存ページを新規ページと置き換えることができます。ノートブックのサンプルがダイアログ・ボックス内に表示され、そこで新規ページを挿入する場所を示すページを選択できます。[新しいページを追加] トグル・ボタンをオンにすると、新規ページが最後のノートブック・ページの後ろに挿入され、選択しているページは無視されます。

サブパネル・インスタンスの詳細設定

パネル・インスタンスの詳細設定を行うには、次の処理を行います。

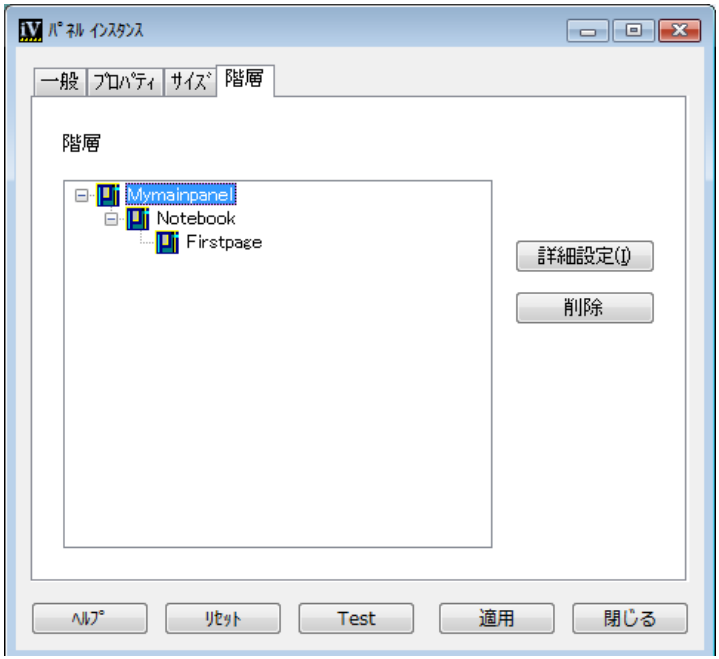
1. アプリケーション・バッファ・ウィンドウ内の `Mymainpanel` の左上隅のボックスをクリックします。

2. 表示されたメニューから [詳細設定パネル] を選択します。

   パネル・インスタンスの詳細設定が表示されます。

3. [階層] ページをクリックします。

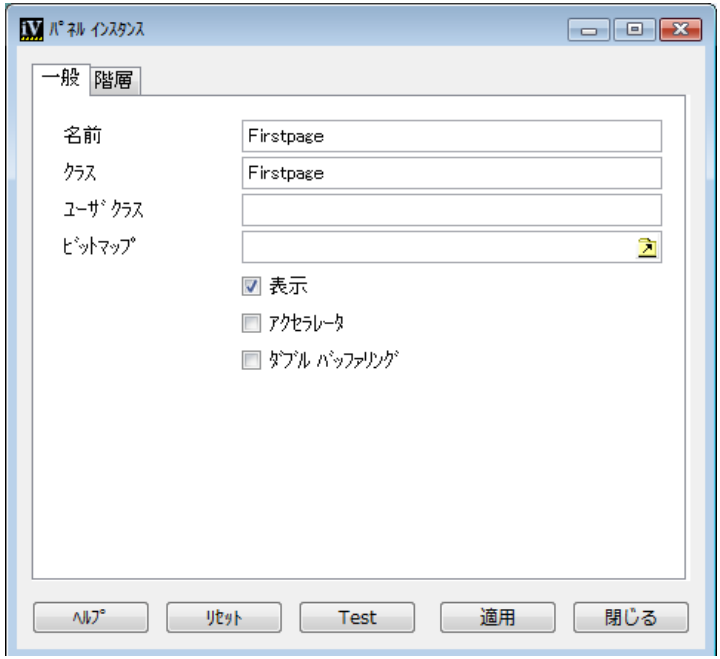
   このページで、パネルの階層がツリー・ガジェットとして表示されます。



   Firstpage パネルは Notebook パネルのサブパネルであり、Notebook パネルは Mymainpanel のサブパネルであることが確認できます。

4. ツリー・ガジェットで詳細設定したいサブパネルをダブルクリックするか、あるいはサブパネルを選択して [詳細設定] をクリックします。

   サブパネルのパネル・インスタンスの詳細設定が表示されます。



5. サブパネルを削除するには、階層ページのツリー・ガジェットでサブパネルを選択し、[削除] をクリックします。

アプリケーションのテスト

メイン・ウィンドウのツールバーの [テスト] アイコン  を使用して、アプリケーション・バッファがカレント・バッファであるときにアプリケーションをテストすることができます。[テスト] アイコンをクリックすると、アプリケーション内に表示された各パネル・インスタンス用のウィンドウが開きます。パネルを閉じるには、[テスト] アイコンを再度クリックします。

パネル・インスタンスを個別にテストしたい場合は、パネル・インスタンスの詳細設定にある [テスト] ボタンを使用します。

章

4

生成されたコードの使用

この章では、生成された C++ コードの使用方法を例を使って説明します。この章は、以下のトピックに分かれています。

- *アプリケーションの構築*
- *C++ のコード生成*
- *生成したコードの拡張*

アプリケーションの構築

次の 3 つのパネルから構成されるアプリケーションを作成します。

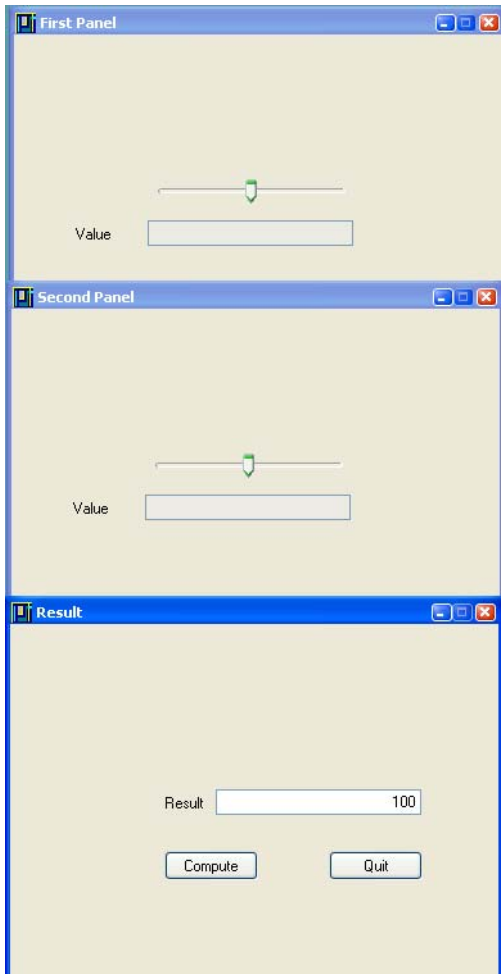


図 4.1 *パネル例*

アプリケーションを作成する手順は次のとおりです。

- *アプリケーション・クラスの設定*
- *最初のパネル・クラスを作成する*
- *2 番目のパネル・クラスを作成する*
- *C++ のコード生成*

アプリケーション・クラスの設定

最初の手順は、アプリケーション・プロパティの編集および保存です。

ここでは、デフォルト・アプリケーション testapp を編集すると想定します。

1. メイン・ウィンドウで、[コード] メニューから [アプリケーションの詳細設定] を選択して、アプリケーションの詳細設定を開きます。
2. クラス名を MyApplication に変更します。[適用]、続いて [閉じる] をクリックします。
3. アプリケーション・バッファ・ウィンドウで、[ファイル] メニューから [名前を付けて保存] を選択し、任意のディレクトリにアプリケーションを myappli.iva として保存します。

   この操作は、アプリケーション・ファイルすべてに対してファイル・ベース名およびデフォルト・パスを設定します。
4. パネルをもう一度調べて、アプリケーションのデフォルト・ディレクトリを確認します。

   ヘッダー、ソースとオブジェクト・ディレクトリは、デフォルトではデータ・ディレクトリになっています。これらの生成したファイルを、アプリケーションの詳細設定の対応するテキスト・フィールドに任意のディレクトリを指定して、別のディレクトリに保存することができます。

最初のパネル・クラスを作成する

最初のパネルと 2 番目のパネル (図 4.1 を参照) は、同じパネル・クラス (FirstPanelClass) の 2 つのインスタンスです。つまり、同じ内容ですが、名前、タイトル、プロパティが異なっています。

このパネル・クラスの 2 つのインスタンスにパネル・クラスを構築します。

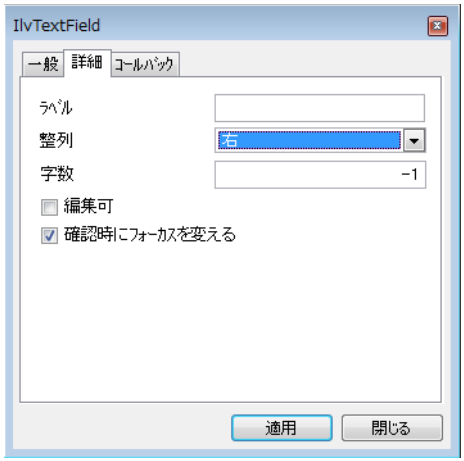
パネル・データ・ファイルの作成

1. 必要に応じて、新しい Gadget バッファ・ウィンドウを開きます。メイン・ウィンドウで、ファイル・メニューから [新規] を選択し、表示されるサブメニューの [ガジェット] を選択します。
2. パレット・パネルの上側ペインで [ガジェット] をクリックします。
3. 次のガジェットをパレット・パネルの下側ペインからドラッグして、Gadgets バッファ・ウィンドウへドロップします。
   - メッセージ・ラベル (クラス名：IlvMessageLabel)
   - テキスト・フィールド (クラス名：IlvTextField)

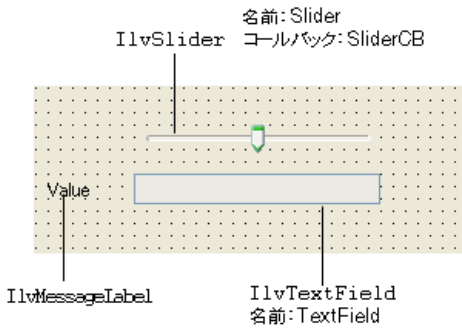
   ガジェットが選択されているとき、そのクラス名がメイン・ウィンドウの下部にあるメッセージ領域に表示されます。
4. メッセージ・ラベルをダブルクリックして、その詳細設定パネルを開きます。

5. メッセージ・ラベルの詳細設定にある [詳細] ページで次の処理を行います。
   - [ラベル] フィールドのテキストを削除します。
   - [ラベル] フィールドに `Value` と入力します。
6. [適用]、続いて [閉じる] をクリックします。
7. テキスト・フィールドをダブルクリックして、その詳細設定パネルを開きます。
8. テキスト・フィールドの詳細設定にある [詳細] ページで次の処理を行います。
   - [ラベル] フィールドのテキストを削除します。
   - [アライメント] オプション・メニューで [右] を選択します。
   - [編集可] トグル・ボタンをオフにします。

     テキスト・フィールドの詳細設定は、次のように表示されます。

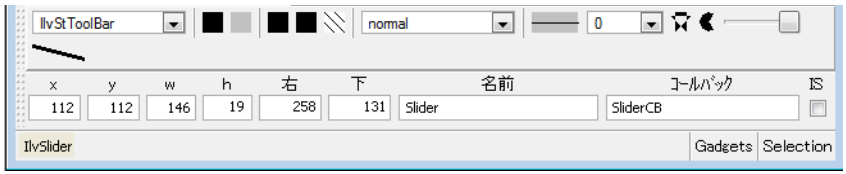


9. [適用]、続いて [閉じる] をクリックします。
10. パレット・パネルの上側ペインで、[その他] をクリックします。
11. 下側ペインから、水平スライダをドラッグし (クラス名： `IlvSlider`)、Gadgets バッファ・ウィンドウにドロップします。
12. オブジェクトを移動してリサイズしてから、下図のようにパネルの大きさを変更します。



13. スライダを選択します。
14. オブジェクト詳細情報の [名前] フィールドに `Slider` を入力します。
15. オブジェクト詳細情報の [コールバック] フィールドに `SliderCB` を入力します。

    オブジェクト詳細情報は、次のように表示されます。



16. テキスト・フィールドを選択します。
17. オブジェクト詳細情報の [名前] フィールドに `TextField` を入力します。
18. ファイル・メニューから [名前を付けて保存] を選択して、パネルを任意のディレクトリに `class1.ilv` として保存します。

パネル・クラスの設定

パネル・クラスを設定するには、次の処理を行います。

1. `class1` がカレント・バッファであることを確認します。
2. メイン・ウィンドウのツールバーの [パネル・クラス・エディタ] アイコン  をクリックして、クラス・パレットを開きます。
3. クラス・パレットの [新規パネル・クラス] アイコン  をクリックして、新規パネル・クラスを作成します。

   新規パネル・クラスを表すアイコンが、`Class1` というタイトルで、パレットに表示されます。

4. Class1 パネル・クラスを選択し、クラス・パレットにあるツールバーで [パネル・クラスの詳細設定] アイコン  をクリックして、その詳細設定を開きます。
5. 詳細設定の [一般] ページで、[クラス] フィールドに FirstPanelClass を入力します。
6. [適用]、続いて [閉じる] をクリックします。

最初のパネルを作成する

FirstPanelClass クラスの最初のインスタンスを作成するには、次の処理を行います。

1. アプリケーションを編集するために、メイン・ウィンドウのツールバーから [アプリケーションの編集] アイコン  を選択します。

   アプリケーション・バッファ・ウィンドウがアクティブになり、クラス・パレットが表示されます (まだ表示されていない場合)。
2. 最初のパネルのインスタンスを作成するには、クラス・パレットからアイコンをアプリケーション・バッファ・ウィンドウにドラッグします。

   アプリケーション・バッファ・ウィンドウに FirstPanelClass のインスタンスが表示されます。
3. パネル・インスタンスを詳細設定するには、パネルのタイトル・バーをダブルクリックします。

   パネル・インスタンスの詳細設定が表示されます。
4. パネル・インスタンスの詳細設定で、[名前] フィールドに FirstPanel を入力します。

   この名前はこのパネルを取得するために使用されます。
5. さらに、タイトルを First Panel に変更し、位置 (x,y) を 200,200 に移動できます。

   [一般] ページでタイトルを指定します。

   [サイズ] ページでパネルの位置を指定します。

   パネル・インスタンスの詳細設定は、次のように表示されます。

6. `FirstPanel` オプションを有効にするために [適用] をクリックし、続いて [閉じる] をクリックします。

2 番目のパネル・インスタンスを作成する

`FirstPanelClass` クラスの 2 番目のインスタンスを作成するには、次の処理を行います。

1. FirstPanelClass アイコンを再びクラス・パレットからアプリケーション・バッファ・ウィンドウにドラッグします。
2. `FirstPanelClass` の 2 番目のインスタンスのパネル・インスタンスの詳細設定にある [一般] ページで、フィールドを次のように変更します。
   - [名前] : `SecondPanel`
   - [タイトル]: `Second Panel`
3. 同じ詳細設定の [サイズ] ページで、フィールドを次のように変更します。
   - x: `200`
   - y: `400`

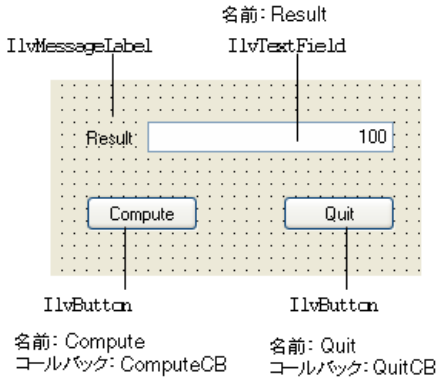
   パネル・インスタンスの詳細設定は、次のように表示されます。

4. [適用]、続いて [閉じる] をクリックします。

---

2 番目のパネル・クラスを作成する

下記に示すように、2 番目のパネル・クラスのインスタンスにパネル・クラスを構築します (これは、図 4.1 にある [結果] パネルです)。



パネル・データ・ファイルの作成

1. 新規 Gadgets バッファ・ウィンドウを開きます。メイン・ウィンドウで、ファイル・メニューから [新規] を選択し、表示されるサブメニューの [ガジェット] を選択します。
2. 上記のようなパネルを作成するために、新規バッファを編集します。
3. ファイル・メニューから [名前を付けて保存] を選択して、これを任意のディレクトリに `class2.ilv` として保存します。

パネル・クラスの設定

パネル・クラスを設定するには、次の処理を行います。

1. `class2` がカレント・バッファであることを確認します。
2. [コード] メニューのクラス・パレットを選択して、クラス・パレットを開きます (まだ開かれていない場合)。
3. クラス・パレットにあるツールバーの [新規パネル・クラス] アイコン  をクリックして、パネル・クラスを作成します。

   新規パネル・クラスを表すアイコンが、`Class2` というタイトルで、パレットに表示されます。
4. `class2` パネル・クラスを選択し、クラス・パレットで [パネル・クラスの詳細設定] アイコン  をクリックして、それをチェックします。

   詳細設定パネルが表示されます。
5. 詳細設定パネルにある [一般] ページで、[クラス] フィールドに `SecondPanelClass` を入力します。
6. [適用]、続いて [閉じる] をクリックします。

2 番目のパネルを作成する

1. アプリケーションを編集するために、メイン・ウィンドウ・ツールバーの [アプリケーションの編集] アイコン  をクリックします。

   現在のアプリケーション・バッファ・ウィンドウがアクティブになり、クラス・パレットが表示されます (まだ表示されていない場合)。
2. 2 番目のパネルのインスタンスを作成するには、クラス・パレットから `SecondPanelClass` アイコンをアプリケーション・バッファ・ウィンドウにドラッグします。

   アプリケーション・バッファ・ウィンドウに `SecondPanelClass` のインスタンスが表示されます。
3. `SecondPanelClass` のタイトル・バーをダブルクリックします。

パネル・インスタンスの詳細設定が表示されます。

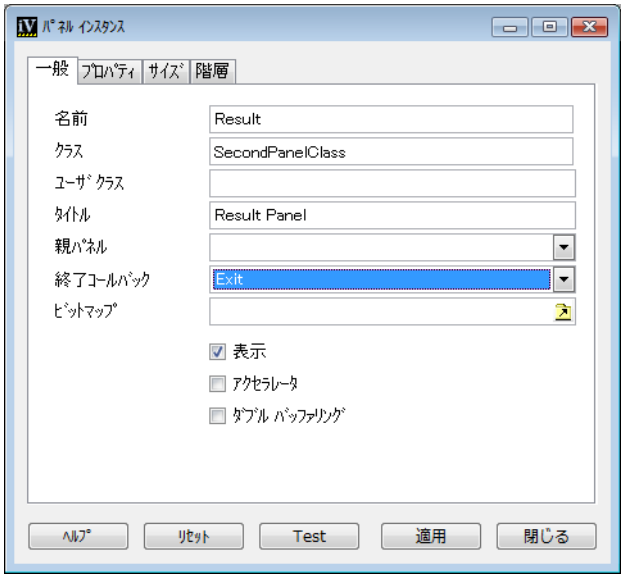
4. パネル・インスタンスの詳細設定にある [一般] ページで、[名前] フィールドに `Result` を入力します。

   この名前はこのパネルを取得するために使用できます。

5. 該当する詳細設定ページでその他のフィールドを次のように変更します。

   - [タイトル]: `Result Panel`
   - 終了コールバック：`Exit`
   - x: `200`
   - y: `650`

   詳細設定は、次のように表示されます。



6. オプションを有効にするために [適用] をクリックし、[閉じる] をクリックして詳細設定を閉じます。

C++ のコード生成

C++ コードを初めて生成する場合は、[コード] メニューから [すべて生成] を選択します。このアプリケーションの場合、IBM® ILOG® Views Studio が次のファイルを生成します。

- FirstPanelClass
  - class1.ilv　FirstPanelClass パネルのデータを含む。
  - class1.h　FirstPanelClass のヘッダー・ファイル。
  - class1.cc　FirstPanelClass のソース・ファイル (C++ ソース・ファイル拡張子がプラットフォーム上の .cc である場合)。
- SecondPanelClass
  - class2.ilv　SecondPanelClass パネルのデータを含む。
  - class2.h　SecondPanelClass のヘッダー・ファイル。
  - class2.cc　SecondPanelClass のヘッダー・ファイル。
- MyApplication
  - myappli.iva　アプリケーションの記述を含む。
  - myappli.h　アプリケーション・クラスのヘッダー・ファイル。
  - myappli.cc　main 関数も含むアプリケーションのソース・ファイル。
  - myappli.mak　アプリケーションのコンパイルおよびテストに使うシンプルな makefile。

次のセクションでは、class1.h、class1.cc、myappli.h、および myappli.cc ファイルについて説明します。

FirstPanelClass ヘッダー・ファイル

class1.h ヘッダー・ファイルが次のように生成されます。

```
// ------------------------------------------------- -*- C++ -*-
// File: /tmp/test/class1.h
// IlogViews 4.0 generated header file
// File generated Wed May 03 16:56:53 2000
//      by IBM ILOG Views Studio
// ------------------------------------------------------------------
#ifndef __class1__header__
#define __class1__header__

#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/textfd.h>
#include <ilviews/gadgets/msglabel.h>
#include <ilviews/gadgets/slider.h>

// ------------------------------------------------------------------
class FirstPanelClass
: public IlvGadgetContainer {
public:
    FirstPanelClass(IlvDisplay* display,
                              const char* name,
                    const char* title,
```

```
                    IlvRect*    size            = 0,
                    IlBoolean  useAccelerators = IlFalse,
                    IlBoolean  visible         = IlFalse,
                    IlUInt     properties      = 0,
                    IlvSystemView transientFor  = 0)
      : IlvGadgetContainer(display,
                           name,
                           title,
                           size ? *size : IlvRect(0, 0, 219, 58),
                           properties,
                           useAccelerators,
                           visible,
                           transientFor)
        { initialize(); }
    FirstPanelClass(IlvAbstractView* parent,
                    IlvRect*    size            = 0,
                    IlBoolean useacc = IlFalse,
                    IlBoolean visible = IlTrue)
      : IlvGadgetContainer(parent,
                           size ? *size : IlvRect(0, 0, 219, 58),
                           useacc,
                           visible)
        { initialize(); }
    // ____________________________________________________________
     virtual void SliderCB(IlvGraphic*);
    IlvSlider* getSlider() const
     { return (IlvSlider*)getObject("Slider"); }
    IlvTextField* getTextField() const
     { return (IlvTextField*)getObject("TextField"); }
protected:
    void initialize();
};

#endif /* !__class1__header__*/
```

ヘッダー

class1.h パネル・クラス・ヘッダー・ファイルの最初の行は、生成されたファイルの日付およびファイルの位置を示します。また、使用中の IBM ILOG Views バージョンもわかります。

含まれるヘッダー・ファイル

次の行は、生成されたクラスに必要なヘッダー・ファイルを表しています。

```
#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/textfd.h>
#include <ilviews/gadgets/msglabel.h>
#include <ilviews/gadgets/slider.h>
```

一般パネルに含まれている各オブジェクトについて、IBM ILOG Views Studio はその関連するヘッダー・ファイルを探します。この例のクラス FirstPanelClass は、そのテキスト・フィールド、メッセージ・ラベル、およびスライダ・オブジェクトのそれぞれについて、ファイル <ilviews/gadgets/textfd.h><ilviews/gadgets/msglabel.h>、および <ilviews/gadgets/slider.h> を含む必要があります。

ベース・クラス

FirstPanelClass のベース・クラス名は変更されていないため、生成クラスは、IlvGadgetContainer から派生します。

コンストラクタ

2 つのパブリック・コンストラクタが生成されます。

```
    FirstPanelClass(IlvDisplay* display,
                              const char* name,
                    const char* title,
                    IlvRect*    size            = 0,
                    IlBoolean  useAccelerators = IlFalse,
                    IlBoolean  visible         = IlFalse,
                    IlUInt     properties      = 0,
                    IlvSystemView transientFor  = 0)
      : IlvGadgetContainer(display,
                           name,
                           title,
                           size ? *size : IlvRect(0, 0, 219, 58),
                           properties,
                           useAccelerators,
                           visible,
                           transientFor)
        { initialize(); }
    FirstPanelClass(IlvAbstractView* parent,
                    IlvRect*    size            = 0,
                    IlBoolean useacc = IlFalse,
                    IlBoolean visible = IlTrue)
      : IlvGadgetContainer(parent,
                           size ? *size : IlvRect(0, 0, 219, 58),
                           useacc,
                           visible)
        { initialize(); }
```

最初のコンストラクタは、メイン・ウィンドウとしてパネルを構築します。2 番目のコンストラクタは、IlvAbstractView である親ビューの一部としてパネルを構築します。

コールバック

コールバック SliderCB はスライダに割り当てられているので、IBM ILOG Views Studio はその関連する仮想メンバ関数を生成します。

```
virtual void SliderCB(IlvGraphic*);
```

名前の付いたオブジェクト

2 つの名前の付いたオブジェクト、Slider および TextField は、次の生成したメンバ関数によってアクセスすることができます。

```
IlvSlider* getSlider() const
    { return (IlvSlider*)getObject("Slider"); }
IlvTextField* getTextField() const
    { return (IlvTextField*)getObject("TextField"); }
```

IBM ILOG Views Studio でこれらの関数を生成しない場合は、パネル・クラスの詳細設定 ([オプション] ページ) で [名前] トグル・ボタンをオフにします。

FirstPanelClass ソース・ファイル

ヘッダー

FirstPanelClass ソース・ファイルは、次のヘッダー行で始まります。

```
// ------------------------------------------------- -*- C++ -*-
// File: /tmp/test/class1.cc
// IlogViews 4.0 generated source file
// File generated Wed May 03 16:56:53 2000
//      by IBM ILOG Views Studio
// --------------------------------------------------------------
```

クラス・ヘッダー・ファイル

パネル・クラス・ヘッダー・ファイルは、最初に含まれたファイルです。

```
#include <class1.h>
```

パネル・データ

#include ステートメントおよびコールバック定義 _SliderCB の間の行は、パネル・クラス・コンストラクタが呼び出された時のパネル・データの読み込まれる方法を定義します。パネル・クラスの詳細設定 ([オプション] ページ) でデータ・トグル・ボタンがオンである場合、パネル・データは定数文字列で生成されます。この場合、ファイルからデータを読み込む代わりに、パネルはその詳細をコンパイラ制限がある場合を除いて、生成文字列 (istrstream を通して) から読み込むことができます。

コールバック

コールバック名 SliderCB は、パネル・オブジェクトに割り当てられているので、IBM ILOG Views Studio は次のコールバックを生成します。

```
static void ILVCALLBACK
_SliderCB(IlvGraphic* g, IlvAny)
{
    FirstPanelClass* o = (FirstPanelClass*)
                         g->GetCurrentCallbackHolder()->getContainer();
    if (o) o->SliderCB(g);
}
```

この関数は、グラフィック・オブジェクトからパネル・クラスを取得し、クラス宣言で宣言されている関連するメソッドを呼び出します。

コールバック・メソッド定義

パネル・クラスの詳細設定 ([オプション] ページ) でコールバック定義トグル・ボタンがオンになっているので、コールバック・メソッド SliderCB のデフォルト定

義は、ソース・ファイルで生成されます。これにより実際の関数定義を書く前に、アプリケーションのコンパイル、リンク、そしてテストを行うことができます。

生成されたコールバック・メソッドは、次のようになります。

```
void
FirstPanelClass::SliderCB(IlvGraphic* g)
{
    const char* className = g->className();
    IlvPrint(" %s : SliderCB method ...",className);
}
```

スライダによって呼び出された場合、この関数は次のメッセージを出力します。

```
IlvSlider : SliderCB method ...
```

IBM ILOG Views Studio でこのコールバック・メソッド定義を生成しない場合は、パネル・クラスの詳細設定 ([オプション] ページ) で [コールバック定義] トグル・ボタンをオフにします。この場合、別のファイルに `FirstPanelClass::SliderCB` の独自のバージョンを書いて、このファイル・オブジェクトをアプリケーションにリンクさせなくてはなりません。

initialize メンバ関数

生成されたクラス `FirstPanelClass` のコンストラクタは、`initialize` メンバ関数を呼び出して、パネルを初期化します。

`initialize` 関数は、アプリケーションをコンパイルするのに使用するプラットフォームに応じて、ファイルあるいは `istrstream` からパネル内容を読み込みます。

```
void
FirstPanelClass::initialize()
{
#if defined(ILVNOSTATICDATA)
    readFile(FILENAME);
#else  /* !ILVNOSTATICDATA */
    istrstream str((char*)_data);
    read(str);
#endif /* !ILVNOSTATICDATA */
    registerCallback("SliderCB", _SliderCB);
}
```

MyApplication ヘッダー・ファイル

ヘッダー

すべての生成されたファイル同様に、アプリケーションのヘッダー・ファイルの最初の行、`myappli.h` は、日付および生成ファイル用のディレクトリ・パスと IBM® ILOG® Views のバージョンを示します。

```
// -------------------------------------------------- -*- C++ -*-
// File: /tmp/test/myappli.h
// IlogViews 4.0 generated application header file
```

```
// File generated Wed May 03 16:56:53 2000
//      by IBM ILOG Views Studio
// ---------------------------------------------------------------
```

含まれるヘッダー・ファイル

アプリケーション・ヘッダー・ファイルは、デフォルト・ベース・クラス・ヘッダー・ファイルおよびそのパネル・クラス・ヘッダー・ファイルを含みます。

```
#include <ilviews/gadgets/appli.h>
#include <class1.h>
#include <class2.h>
```

MyApplication クラス

IBM ILOG Views Studio は、次のアプリケーション・クラスを生成します。

```
class MyApplication: public IlvApplication {
public:
    MyApplication(
        const char* appName,
        const char* displayName = 0,
        int argc = 0,
        char** argv = 0
    );
    MyApplication(
        IlvDisplay* display,
        const char* appName
    );
    ~MyApplication();
    virtual void makePanels();
    virtual void beforeRunning();
    FirstPanelClass* getFirstPanelClass() const
        { return (FirstPanelClass*) getPanel("FirstPanelClass"); }
    FirstPanelClass* getSecondPanel() const
        { return (FirstPanelClass*) getPanel("SecondPanel"); }
    SecondPanelClass* getResult() const
        { return (SecondPanelClass*) getPanel("Result"); }
};
```

ベース・クラス

生成したクラスは、`IlvApplication` から派生します。このクラスの説明に関しては、IBM ILOG Views ライブラリの一部である *IBM ILOG Views リファレンス・マニュアル*の `IlvApplication` を参照してください。

コンストラクタ

生成されたコンストラクタは、関連するベース・クラス・コンストラクタを呼び出すだけです。

makePanels メソッド

各アプリケーションに、IBM ILOG Views Studio はアプリケーションが初期化されたときに呼び出される `makePanels` メソッドを生成します。このメソッドは、生成

されたアプリケーション・パネルの作成を処理します。*IBM ILOG Views リファレンス・マニュアル*のクラス IlvApplication を参照してください。

パネル・アクセサ

IBM ILOG Views Studio は、アプリケーションの各パネルにパネルを戻すパネル・アクセサを生成します。

MyApplication ソース・ファイル

ヘッダー

通常通り、アプリケーション・ソース・ファイルの最初の行、myappli.cc は、生成されたファイルのファイル、日付およびパス・ディレクトリと IBM® ILOG® Views のバージョン情報を示します。

```
// -------------------------------------------------- -*- C++ -*-
// File: /tmp/test/myappli.cc
// IlogViews 4.0 generated application source file
// File generated Wed May 03 16:56:53 2000
       by IBM ILOG Views Studio
// ------------------------------------------------------------
```

クラス・ヘッダー・ファイル

アプリケーション・ソース・ファイルは常に、生成されたアプリケーション・クラス・ヘッダー・ファイルを含みます。

```
#include <myappli.h>
```

makePanels 関数定義

生成された makePanels メンバ関数は、次のようになります。

```
void
MyApplication::makePanels()
{
    // --- parameters ---
    IlvDisplay*         display = getDisplay();
    IlvRect             bbox;
    IlvContainer*       cont;
    // --- FirstPanel ---
    bbox.moveResize(200, 200, 500, 500);
    cont = new FirstPanelClass(display,
                               "FirstPanel",
                               "First Panel",
                               &bbox,
                               IlFalse,
                               IlFalse, 0, 0);
    addPanel(cont);
    cont->show();
    // --- SecondPanel ---
    bbox.moveResize(200, 300, 500, 500);
    cont = new FirstPanelClass(display,
                               "SecondPanel",
                               "Second Panel",
```

```
                                &bbox
                                IlFalse,
                                IlFalse, 0, 0);
    addPanel(cont);
    cont->show();
    // --- Result ---
    bbox.moveResize(200, 400, 500, 500);
    cont = new SecondPanelClass(display,
                                "Result",
                                "Result Panel",
                                &bbox
                                IlFalse,
                                IlFalse, 0, 0);
    addPanel(cont);
    cont->setDestroyCallback(IlvAppExit, this);
    cont->show();
    // --- The Exit panel is not wanted ---
    setUsingExitPanel(IlFalse);
}
```

このアプリケーションには 3 つのパネルが含まれています。`FirstPanel` および `SecondPanel` は、クラス `FirstPanelClass` の一部です。次の点に注意してください。

- 各パネルは、パネル・インスタンスの詳細設定 ([サイズ] ページ) で x および y フィールドによって指定された位置に作成されます。
- パネル・コンストラクタに渡される矩形のサイズは、それらのデータが読み込まれるときにリサイズされるため、実際にはパネルの大きさに影響を与えません。パネル・インスタンスの詳細設定 ([サイズ] ページ) で [生成時のサイズに指定] トグル・ボタンがオンになっている場合、パネル・インスタンスの詳細設定で指定されたバウンディング・ボックスの幅および高さの値が、作成された後のパネルのリサイズに使用されます。
- 各パネルは、作成された後、アプリケーションに追加されます。

  ```
  addPanel(cont);
  ```

- パネル・インスタンスの詳細設定 ([一般] ページ) で [表示] トグル・ボタンがオンになっている場合、パネルは `show()` メンバ関数によって表示されます。

  ```
  cont->show();
  ```

- [結果] パネルの終了コールバックがパネル・インスタンスの詳細設定 ([一般] ページ) で [終了] に設定されているので、次のコードが生成されます。

  ```
  cont->setDestroyCallback(IlvAppExit, this);
  ```

- [終了] パネルは必要とされていないので、次のコードが生成されます。

  ```
  setUsingExitPanel(IlFalse);
  ```

メイン関数

main() トグル・ボタンがアプリケーションの詳細設定にある [オプション] ページでチェックされているので、main 関数がアプリケーション・ソース・ファイルに生成されます。

```
main(int argc, char* argv[])
{
    // IlvSetCurrentCharSet(<YourCharSet>);
    IlvSetLanguage();
    MyApplication* appli = new MyApplication(?myappli?, 0, argc,
    argv);
    if (!appli->getDisplay())
        return -1;
    appli->run();
    return 0;
}
```

この関数は、クラス MyApplication のアプリケーションを作成します。アプリケーションを実行する前に、関数は作成したアプリケーションでディスプレイの作成が正しく行われたかどうかを確認します。

main 関数を生成しない場合は、main() トグル・ボタンをオフにします。

生成したアプリケーションのテスト

生成したアプリケーションをテストするには、アプリケーション・オブジェクト・ディレクトリの make ユーティリティを生成した Makefile を使って実行し、その後 myappli を起動します。以下は、コマンドの例です。オブジェクト・ディレクトリが /tmp/test である場合：

```
cd /tmp/test
make -f myappli.mak
myappli
```

アプリケーションを終了させるには、ウィンドウ・マネージャで [結果] パネルを閉じます。

生成したコードの拡張

メンバ関数を追加して、SliderCB の適切なバージョンを書くために、MyFirstPanelClass を FirstPanelClass から派生させます。

派生クラスの定義

ファイル myclass1.h で、次のように MyFirstPanelClass を宣言します。

```
#include <class1.h>
```

```
class MyFirstPanelClass
: public FirstPanelClass {
public:
    MyFirstPanelClass(IlvDisplay* display,
                                const char* name,
                      const char* title,
                      IlvRect*    size            = 0,
                      IlBoolean  useAccelerators = IlFalse,
                      IlBoolean  visible         = IlFalse,
                      IlvUInt     properties      = 0,
                      IlvSystemView transientFor  = 0):
    FirstPanelClass(display,
                    name,
                    title,
                    size,
                    useAccelerators,
                    visible)
    {}
    virtual void SliderCB(IlvGraphic*);
    IlvInt getValue() const { return getTextField()->getIntValue(); }
};
```

ベース・クラス

MyFirstPanelClass は、FirstPanelClass から派生しています。

コンストラクタ

1 つのコンストラクタを定義するだけです。このコンストラクタは、パネルをメイン・ウィンドウとして構築し、そのベース・クラス・コンストラクタを呼び出します。

コールバック・メソッド

SliderCB 仮想メンバ関数は、テキスト・フィールドにスライダ値を表示するために派生クラスで再定義されます。次に、そのような関数の定義の例を示します。

```
void
MyFirstPanelClass::SliderCB(IlvGraphic*)
{
    getTextField()->setValue(getSlider()->getValue(), IlTrue);
}
```

getValue メンバ関数

パネルによって表示される値を取得するには、このインライン・メンバ関数を定義する必要があります。

```
IlvInt getValue() const { return getTextField()->getIntValue(); }
```

派生クラスの使用

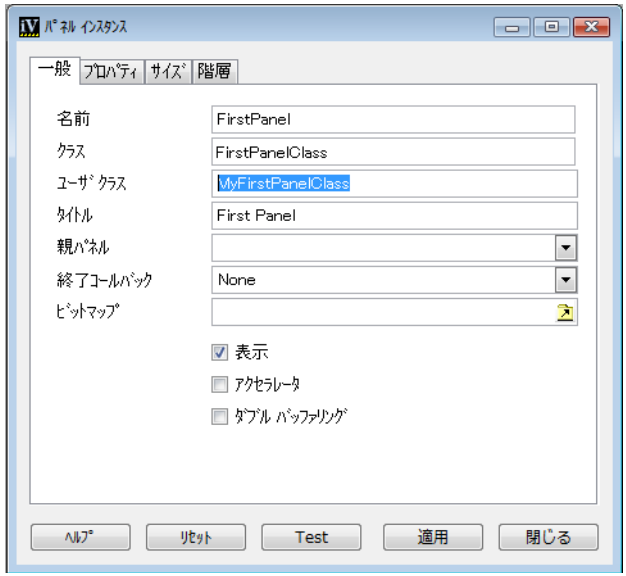
FirstPanel を作成するために必要な MyFirstPanelClass を FirstPanelClass の代わりに使用するには、次の処理を行います。

- パネル・インスタンスの詳細設定でユーザ・クラス・フィールドを設定します。
- #include ステートメントを生成したアプリケーション・ヘッダー・ファイルに挿入します。

ユーザ・クラスの設定

ユーザ・クラスを設定するには、次の処理を行います。

1. アプリケーション・バッファ・ウィンドウで、FirstPanelClass の最初のインスタンスをダブルクリックします。
2. パネル・インスタンスの詳細設定で、ユーザ・クラス・フィールドに MyFirstPanelClass を入力します。



3. [適用]、続いて [閉じる] をクリックします。

> ***メモ:*** SecondPanel *の場合は、手順2 および3 を繰り返します。*

次のコードを makePanels 関数に生成する代わりに、

```
cont = new FirstPanelClass(display,
                           "FirstPanel",
                           "First Panel",
                           &bbox,
                           IlFalse,
                           IlFalse, 0, 0);
);
```

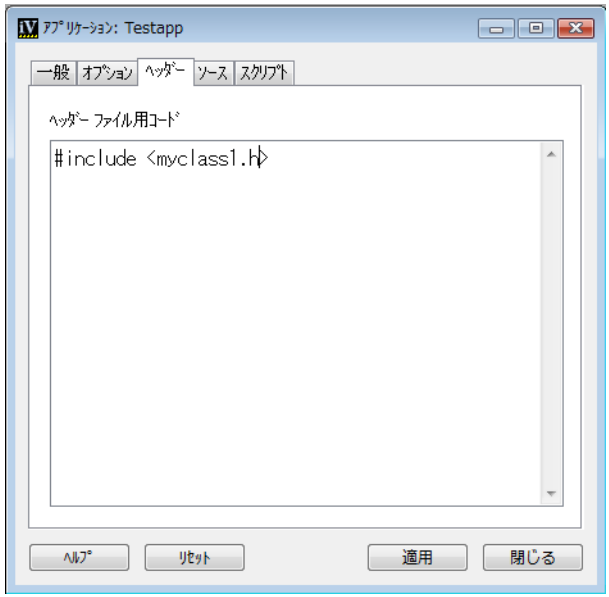
IBM ILOG Views Studio は、アプリケーション・ソース・ファイル myappli.cc の下にコードを生成します。

```
cont = new MyFirstPanelClass(display,
                             "FirstPanel",
                             "First Panel",
                             &bbox,
                             IlFalse,
                             IlFalse, 0, 0);
);
```

MyFirstPanelClass は、myclass1.h の中で宣言されているのでこのファイルを myappli.h に含める必要があります。

生成したアプリケーション・ヘッダー・ファイルにコードを挿入する

1. [コード] メニューから [アプリケーションの詳細設定] を選択します。[アプリケーションの詳細設定] が表示されます。
2. [アプリケーションの詳細設定] で、[ヘッダ] ページを開きます。
3. [ヘッダ] ページで、#include <myclass1.h> を入力します。



4. [適用]、続いて [閉じる] をクリックします。

アプリケーション・ソース・ファイルを再生成するとき、IBM ILOG Views Studio は次の式をアプリケーション・ヘッダー・ファイルに挿入します。

```
// --------------------------------------------------------------
// --- Inserted code
```

```
#include <myclass1.h>

// --- End of Inserted code
```

追加オブジェクト・ファイルのリンク

IBM ILOG Views Studio によって生成された Makefile は、生成したファイルのコンパイルおよびアプリケーションのリンクを処理します。さらに、独自のオブジェクト・ファイルを、USEROBJS 変数作成を通じてアプリケーションにリンクさせることができます。独自のオブジェクト・ファイルについての責任はユーザにあります。ただし、IBM ILOG Views Studio によって生成された make オプションをコピーして、追加オブジェクト・ファイルを維持するために独自の Makefile を書くことができます。

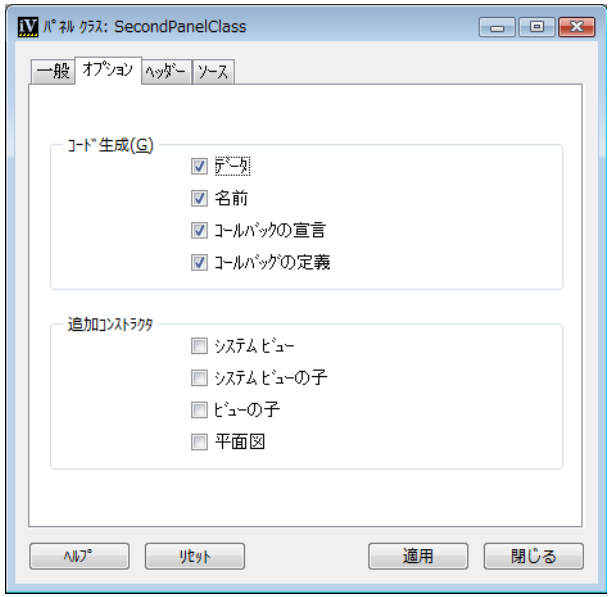
たとえば、MyFirstPanelClass::SliderCB の定義がユーザのオブジェクト・ファイル myclass1.o にある場合、生成した Makefile myappli.mak を次のように使用することができます。

```
make -f myappli.mak USEROBJS=myclass1.o
```

クラスを派生させずにコールバックを定義する

前のセクションでは、パネル・クラスを派生し、コールバック・メソッドを派生クラスで定義しました。ここでは、SecondPanelClass について、サブクラスを派生させずに生成したソース・ファイルにコールバック・メソッドを挿入します。

1. クラス・パレットで、SecondPanelClass を選択し、クラス・パレットのツールバーにある [パネル・クラスの詳細設定] アイコンをクリックします。
2. パネル・クラスの詳細設定で、[オプション] ページへ移動し、コールバックの定義トグル・ボタンをオフにします。



3. [ソース] ページを開き、次のコードをセクション「ソース・ファイル用コード」に入力します。

```
#include <ilviews/gadgets/appli.h>
#include <myclass1.h>

void
SecondPanelClass::ComputeCB(IlvGraphic*)
{
    IlvApplication* appli = IlvApplication::GetApplication(this);
    MyFirstPanelClass* pan1 =
        (MyFirstPanelClass*)appli->getPanel("FirstPanel");
    MyFirstPanelClass* pan2 =
        (MyFirstPanelClass*)appli->getPanel("SecondPanel");
    getResult()->setValue(pan1->getValue() + pan2->getValue(), IlTrue);
}
void
SecondPanelClass::QuitCB(IlvGraphic*)
{
    delete IlvApplication::GetApplication(this);
    IlvExit(0);
}
```

4. 操作を有効にするために [適用] をクリックし、続いて [閉じる] をクリックして詳細設定を閉じます。

5. パネル・クラス・ユーザ・コード・パネルで、[適用] をクリックします。

再生成するとき、IBM ILOG Views Studio は、2 つのコールバック・メソッドを `class2.cc` ファイルに挿入します。

章

5

IBM ILOG Views Studio の Gadgets 拡張機能をカスタマイズする

この章では、IBM® ILOG® Views Stuido の Gadgets 拡張機能設定オプションのリストを提供します。これらのオプションを使用して Studio をカスタマイズすることができます。

- *Gadgets 拡張機能設定オプション*

Gadgets 拡張機能設定オプション

IBM® ILOG® Views は、Gadgets 拡張機能用に以下の設定オプションを提供します。

- `additionalLibraries "<library list>"` で、生成されたアプリケーションにリンクさせるために IBM ILOG Views ライブラリのリストを指定します。

  例 :

  ```
  studio {
      additionalLibraries "ilvadvgadmgr ilvgadmgr ilvmgr";
  }
  ```

- `applicationBaseClass <className>` で、生成されたアプリケーション・クラスのベース・クラス名を指定します。デフォルトの値は `IlvApplication` です。

- applicationBufferBackground "<colorName>" で、アプリケーション・バッファ・ウィンドウの背景色を指定します。デフォルトの値は "Cadet Blue" です。
- applicationFileExtension "<extension>" で、アプリケーション・ファイルの拡張子を指定します。デフォルトの値は ".iva" です。
- applicationHeaderFile "<header>" で、生成されたアプリケーション・ヘッダー・ファイルに含まれるヘッダー・ファイルを指定します。デフォルトの値は "<ilviews/appli.h>" です。
- defaultApplicationName <name> で、新しいアプリケーションの名前を指定します。デフォルトの値は testapp です。

  例 :

  ```
  studio {
      defaultApplicationName newappli;
  }
  ```

- defaultCallbackLanguage <language> で、コールバック関数がオブジェクトに付加されるとき、デフォルトで使用されるコールバック言語を指定します。このオプションは、IBM ILOG Views Studio Script 拡張機能 (jsstudio) にのみ適用されます。デフォルトの値は JvScript です。jsstudio で記述したコールバック関数のデフォルトを JvScript にしない場合、このオプションを none に設定してください。

  例 :

  ```
  studio {
      defaultCallbackLanguage none;
  }
  ```

- defaultHeaderDir "<dir>" で、新しいアプリケーションにデフォルト設定されているヘッダー・ファイル・ディレクトリを指定します。指定したディレクトリは、アプリケーション・ディレクトリに関連しています。
  アプリケーション作成後は、このディレクトリをアプリケーションの詳細設定経由で変更できます。

  例 :

  ```
  studio {
      defaultHeaderDir "include";
  }
  ```

- defaultHeaderFileScope "<dir>" で、#include ステートメントで生成されるサブディレクトリを指定します。アプリケーション作成後は、このディレクトリをアプリケーションの詳細設定経由で変更できます。

  例 :

  ```
  studio {
      defaultHeaderFileScope "myinclude/";
  ```

```
}
```

- `defaultObjDir "<dir>"` で、新しいアプリケーションにデフォルト設定されている Makefile ディレクトリを指定します。指定したディレクトリは、アプリケーション・ディレクトリに関連しています。アプリケーション作成後は、このディレクトリをアプリケーションの詳細設定経由で変更できます。

  例 :

```
studio {
    defaultObjDir "obj";
}
```

- `defaultSrcDir "<dir>"` で、新しいアプリケーションにデフォルト設定されているソース・ファイル・ディレクトリを指定します。指定したディレクトリは、アプリケーション・ディレクトリに関連しています。アプリケーション作成後は、このディレクトリをアプリケーションの詳細設定経由で変更できます。

  例 :

```
studio {
    defaultObjDir "obj";
}
```

- `defaultSystemName "<name>"` で、アプリケーションの Makefile を生成するターゲット・システムを指定します。これは、アプリケーション特有の情報ではありません。生成されたアプリケーションをコンパイルするために使用するシステムのみに関係します。デフォルトでは、Makefile を、IBM ILOG Views Studio を実行中のシステム用に生成します。デフォルトで Makefile が生成されるプラットフォームを変更したい場合に、このオプションを使用します。

  例 :

```
studio {
    defaultSystemName "sparc_5_4.0";
}
```

- `headerFileExtension "<extension>"` で、生成されたヘッダー・ファイルの拡張子を指定します。デフォルトの値は `".h"` です。

  例 :

```
studio {
    headerFileExtension ".hxx";
}
```

- `JvScriptApplication <true/false>` で、生成された C++ アプリケーションが IBM ILOG Script for IBM ILOG Views を使用するかどうか指定します。このオプションは、GUI アプリケーション・プラグインを、`jsstudio` 拡張機能と併用する場合のみ該当します。

  例 :

```
studio {
    JvScriptApplication false;
}
```

- makeFileExtension "<extension>" で、生成された Makefile の拡張子を指定します。デフォルトの値は ".mak" です。例 :

```
studio {
    makeFileExtension ".mk";
}
```

- noPanelContents <true/false> で、パネル・インスタンスの内容を、アプリケーション・ファイルを開いたときに読み込むかどうかを指定します。このオプションのデフォルトは false です。多数のパネルを含むアプリケーションを頻繁に編集する場合は、読み込み時間を短縮するためにこのオプションを使用します。その後、アプリケーション・バッファ・ウィンドウのパネル・インスタンス・メニューから [コンテンツの読み込み] を選択して、パネルの内容を読み込むことができます。

- panelBaseClass <className> で、バッファのタイプを問わず、新規に作成されたパネル・クラスに自動的に与えられるベース・クラス名を指定します。このオプションを指定しない場合、ベース・クラス名はバッファのタイプに依存します。

  例 :

```
studio {
    panelBaseClass MyGadgetContainer;
}
```

- sourceFileExtension "<extension>" で、選択されたターゲット・プラットフォームに関係なく、生成された C++ ソース・ファイルの拡張子を指定します。デフォルト値は、アプリケーションの詳細設定パネルで選択されたターゲット・プラットフォームに依存します。

- system <systemDescription> は、アプリケーション・ファイルを生成するためにエディタが必要とするターゲット・プラットフォームに関する情報を宣言します。このオプションは繰り返すことができます。フォーマットは次のとおりです。

```
system "<system-name>" {
    <option-1> <value-1>;
    ...
    <option-n> <value-n>;
}
```

  system-name は、msvc5 や sparc_5_4.0 のような IBM ILOG Views プラットフォーム名です。これらのオプションは IBM ILOG Views が利用できるすべてのプラットフォーム上で適用されるので、これらを変更する必要はありません。次は、システムの記述に使用できるオプションのリストです。

- compiler "<command>" は、コンパイラをこのプラットフォーム上で実行させるコマンドを指定します。
- compilerOptions "<options>" は、オブジェクト・ファイルを作成するために、コンパイラに渡されるオプションを指定します。
- linker "<command>" は、リンカをこのプラットフォーム上で実行するコマンドを指定します。
- linkerOptions "<options>" は、実行可能ファイルを作成するためにリンカに渡されるオプションを指定します。
- libraries "<libraries>" は、実行可能ファイルを作成するためにリンクする IBM ILOG Views ライブラリのリストです。
- systemLibraries "<libraries>" は、実行可能ファイルを作成するためにリンクするシステム・ライブラリのリストです。
- motif : プラットフォームが Motif を使用できるかどうかを示す true あるいは false。

◆ toolBarItem <commandName> <toolBarName> [-before <refCommandName>] によって、コマンド <commandName> をツールバー <toolBarName > に追加できます。このオプションは繰り返すことができます。オプションで、キーワード -before を使用して、新しいコマンドをコマンド <refCommandName> の直後に挿入することを指定できます。

例 :

```
studio {
    toolBarItem SelectLabelMode IlvStGadgetBuffer -before
SelectFocusMode;
}
```

◆ userSubClassPrefix "<prefix>" によって、生成されたパネル・サブクラス用クラス名のプレフィックスをカスタマイズします。デフォルトでは、プレフィックスは "My" です。

◆ userSubClassSuffix "<suffix>" によって、生成されたパネル・サブクラス用クラス名のサフィックスを指定します。

章

6

IBM ILOG Views Studio の拡張

この章では、Gadgets 拡張機能インストール時に IBM® ILOG® Views Studio を拡張する追加の方法について説明します。説明内容は以下のとおりです。

- *IBM ILOG Views Studio の拡張*
- *独自のグラフィック・オブジェクトの統合*
- *IBM ILOG Views Studio の拡張*

> ***メモ:*** *IBM ILOG Views Studio を拡張するには、IBM ILOG Views Gadgets、および IBM ILOG Views Manager パッケージが必要です。*

IBM ILOG Views Studio コンポーネントの拡張

このセクションでは、拡張可能な IBM® ILOG® Views Studio コンポーネントについて説明します。

- *新規コマンドの定義*
- *新規パネルの定義*
- *IBM ILOG Views Studio メッセージ*

- *新規バッファの定義*
- *新規編集モードの定義*
- *IlvStExtension クラス*

新規コマンドの定義

コマンドとは、エディタを使用してユーザが実行できるアクションです。コマンドは、C++ クラス `IlvStCommand` で、ファイル `<ivstudio/command.h>` に記述されています。これは、以下で定義されます。

- メニュー、アイコン、あるいはヘルプ・メッセージに表示されるべき情報、およびコマンドが正常に実行された時に送信されるメッセージ名のリストを含む宣言。定義済みコマンドは、コマンド記述ファイル `studio.cmd` で宣言されています。このファイルに関する詳細情報は、*IBM ILOG Views Studio ユーザ・マニュアル*の「*IBM ILOG Views* Studio コマンド・ファイル」を参照してください。
- 仮想メンバ関数 `doIt` で定義されているアクション。エラーがない場合に 0 を返します。そうでない場合は、対応するエラーを返します。

コマンド・エラー

IBM ILOG Views Studio エラーは `IlvStError` クラスのサブクラスで、ファイル `<ivstudio/error.h>` で宣言されています。エラーは、コマンドの `doIt` メンバ関数によって返されます。エラーは文字列メッセージおよびタイプで定義されています。エラーには、3 つのタイプがあります。

- `IlvStInformation`
- `IlvStWarning`
- `IlvStFatal`

新規コマンドを追加する場合は、以下の手順に従います。

1. 仮想メンバ関数 `doIt` を定義するために `IlvStCommand` のサブクラスを定義します。
2. コマンド宣言ファイルあるいはオプション・ファイルに、直接記述子を追加します。
3. 新規コマンド宣言ファイルを使用したい場合は、`commandFile` オプションを使用してオプション・ファイルにそれを宣言します。
4. メンバ関数 `IlvStudio::registerCommand` を使用してコマンドをエディタに登録し、コマンドのインスタンスを構築するためにコマンド名および関数を与えます。

定義済みのコマンド・クラス

一般的な必要性のために 2 つのサブクラスがあります。

- カレント・バッファに新規オブジェクトを追加するための `IlvStClickAddObject`。
- 画面にパネルを表示させるための `IlvStShowPanel`。インスタンスの構築は、表示するためにパネルを使います。パネルが既に表示されている場合は、このコマンドによってパネルが非表示になります。

コマンドの実行

コマンド実行をエディタに要求し、これに失敗した場合、相応するエラーがコマンド実行プロシージャによって返され、エディタによって管理されます。成功した場合、コマンドに関連するメッセージのリストがエディタにより送られます。つまり、各メッセージに付加されているサブスクリプトが実行されるということです。

新規パネルの定義

IBM® ILOG® Views Studio インターフェースは複数のパネルで構成されています。パネルは、`IlvStPanelHandler` のサブクラスであり、これはファイル `<ivstudio/panel.h >` に記述されています。このクラスは、ガジェット・コンテナ・クラスではなく、`IlvGadgetContainer` のインスタンスである実際のグラフィック・パネルへのハンドルです。これにより、IBM ILOG Views Studio 内での振る舞いから完全に切り離してパネルのグラフィック側面を維持することができます。以下は、再定義できる仮想メンバ関数です。

- `connect` は、パネルを初期化します。このメソッドは通常、パネルが作成された後に呼び出されます。初期化からコンストラクタを切り離すためです。
- `apply` は、あらゆるオブジェクトに付加できる `apply` コールバックに関連付けられています。
- `cancel` は、あらゆるオブジェクトに付加できる `cancel` コールバックに関連付けられています。
- `reset` は、あらゆるオブジェクトに付加できる `reset` コールバックに関連付けられています。

`show` および `hide` メソッドは、IBM ILOG Views Studio パネルを表示 / 非表示にするために使用します。ハンドルのあるガジェット・コンテナを直接、表示 / 非表示にしないでください。

サブクラス `IlvStDialog` は、`IlvDialog` のインスタンス用ハンドルです。

IBM ILOG Views Studio メッセージ

IBM® ILOG® Views Studio メッセージは、発生したイベントを記述する情報を含んでいます。メッセージはサブスクリプトを集めます。サブスクリプトは、メッセージが送られたときに実行されるアクションです。メッセージはユーザが作成することはありませんが、名前を使用してエディタからアクセスすることができます。

サブスクリプトは、IlvStSubscription クラスのサブクラスで、ファイル <ivstudio/message.h> で宣言されています。レシーバに関連付けられており、doIt 仮想メンバ関数があります。たとえば、パネルが、カレント・バッファでオブジェクトの選択を変更するたびに生成される ObjectSelected メッセージに反応する場合、サブスクリプト・インスタンスを使用してこのメッセージにサブスクリプトします。これは、メッセージ・インスタンス上のメンバ関数 subscribe を呼び出してパネル・コンストラクタで行われます。メッセージ・インスタンスは、その名前を介してエディタによって与えられます。次に、メッセージ ObjectSelected が送られると、このサブスクリプトの doIt メンバ関数が呼び出されます。

新規バッファの定義

バッファは、IBM® ILOG® Views Studio で編集されるドキュメントです。その内容を表示、編集、保存そして読み込むために IlvManager を使用します。オブジェクトに関する更なる情報を保存するためにマネージャをサブクラス化する必要がある場合など、対応するバッファをサブクラス化しなくてはなりません。IlvStBuffer クラスは、IlvManager をカプセル化するために定義され、IlvStGadgetBuffer クラスは、IlvGadgetManager をカプセル化するために定義されます。これらのクラスは、<ivstudio/stbuffer.h> ファイルおよび <ivstudio/gadgets/gadbuf.h > で宣言されています。グラフィック・オブジェクトを編集および保存するために特殊マネージャ・クラスを定義する必要がある場合、対応するバッファ・クラスを定義しなくてはなりません。

バッファ・タイプの登録

.ilv ファイルを読み込むとき、IBM ILOG Views Studio は、まずファイル・*クリエータ・クラス*情報を読み込んで、このファイルを編集するために作成するバッファのタイプを決定します。たとえば、IlvGadgetManager によって保存されたファイルを読み込むとき、IBM ILOG Views Studio は、その .ilv ファイルのクリエータ・クラスが IlvGadgetManagerOutputFile であることを確認してから、IlvStGadgetBuffer を作成します。これは、バッファ・コンストラクタ関数をファイル・クリエータ・クラスに関連付ける IlvStBuffers::registerType 関数によって可能になります。この関数を使用してバッファ・タイプを登録します。IBM ILOG Views Studio は、すべてのバッファを管理するために IlvStBuffers オブジェクトを使用します。このオブジェクトへの参照を、IlvStudio::buffers を呼び出して取得できます。

```
static IlvStBuffer*
```

```
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
{
    // MyGadgetBuffer is a subtype of IlvStBuffer.
    return new MyGadgetBuffer(editor, name);
}

...
    editor->buffers().registerType("MyGadgetManagerOutput",
                                   MakeMyBuffer);
...
```

パネル・クラス

IlvStPanelClass オブジェクトは、バッファに生成する C++ パネル・クラスを記述する IBM ILOG Views Studio オブジェクトです。IBM ILOG Views Studio が、バッファで編集されたデータを使用して IlvContainer のサブクラスを生成するために必要とするすべての情報を含んでいます。

IlvStPanelClass オブジェクトは、クラス名、ベース・クラス、ファイルのベース名、ファイルが生成されるディレクトリなどを含んでいます。そのプロパティのいくつかは、ベース・クラスなど、対応するバッファのタイプに関連付けられています。Gadgets バッファ (IlvStGadgetBuffer) が IlvGadgetContainer のサブクラスを生成するために使用され、2D バッファ (IlvStBuffer) が IlvContainer のサブクラスを生成するために使用されます。

IlvGadgetContainer のサブクラスであるクラス MyContainer を定義して、マネージャが保存した追加情報を読み込めるようにするとしましょう。さらに、生成クラスを MyContainer から派生させたいとします。バッファを使用してパネル・クラスを作成するたびにパネル・クラスの詳細設定でベース・クラスを指定することもできますが、最良の方法は、ベース・クラスのデフォルトを MyContainer となるようにパネル・クラスを自動設定することです。

バッファに対して setUpPanelClass 仮想メンバ関数を定義して、これを行うことができます。この関数は、IBM ILOG Views Studio がバッファからパネル・クラスを作成するときに呼び出されます。

```
void
MyBuffer::setUpPanelClass(IlvStPanelClass* pclass) const
{
    IlvStGadgetBuffer::setUpPanelClass(pclass);
    pclass->setBaseClass("MyContainer");
}
```

カスタマイズしたコンテナ・クラスの統合

多くの場合、IBM ILOG Views Studio は、コンテナのインスタンスを作成します。たとえば、パネルをテストする場合や、これをアプリケーション・バッファ・ウィンドウに追加するときなどです。適切なクラスを選択するために、IBM ILOG Views Studio は、コンテナ情報オブジェクトのセットを使用します。IlvStContainerInfo オブジェクトは、IBM ILOG Views Studio がコンテナ・サブ

クラスについて必要な情報を提供し、そのサブクラスのインスタンスを作成します。

コンテナのクラスを統合するために IlvStContainerInfo のサブクラスを定義して、このクラスのインスタンスを IBM ILOG Views Studio コンテナ情報セットに次のように追加する必要があります。

```
studio->addContainerInfo(myContainerInfo);
```

新規編集モードの定義

編集モードとは、タイプ IlvManagerViewInteractor のオブジェクトをカプセル化する IBM® ILOG® Views Studio オブジェクトです。新規編集モードを追加するには、次の処理を行います。

1. クラス IlvStMode のオブジェクトを作成します。
2. このオブジェクトを IBM ILOG Views Studio モード・デリゲート IlvStModes に追加します。
3. IlvStSetMode クラスのインスタンスを返すコマンド・コンストラクタを定義します。
4. このコマンド・コンストラクタを登録します。
5. コマンド宣言ファイルにコマンド記述子を宣言します。

編集モードが作成されると、これをビットマップに関連付け、これをメイン・ウィンドウの左のツールバーに追加することができます。

このオブジェクトを IBM ILOG Views Studio モード・デリゲートに追加する

IlvStudio は、特定のサービスを処理する複数の「デリゲート」を持っています。IlvStudio には、モードを管理するために IlvStModes クラスのメンバがあります。その参照は次によってアクセスできます。

```
IlvStModes& IlvStudio::modes();
```

次の関数を使用して編集モードを追加することができます。

```
void IlvStModes::add(IlvStMode* mode)
```

IlvStSetMode クラスのインスタンスを返すコマンド・コンストラクタを定義する

定義するコマンド・コンストラクタは、簡単な関数にすることができます。たとえば、メニュー・モードは次のようにコーディングできます。

- 新規モードを追加します。

```
editor->modes().add(new IlvStMode(editor,
                                  "Menu",
                                  "SelectMenuMode",
                                  new IlvMakeMBLinkInteractor));
```

◆ コマンド・コンストラクタ関数

```
static IlvStCommand*
MkSelectMenuMode(IlvStudio*)
{
    return new IlvStSetMode("Menu");
}
```

IlvStExtension クラス

新規拡張子を初期化して IBM® ILOG® Views Studio に追加するには、定義済みのシーケンスで呼び出されるメソッドのセットを定義する IlvStExtension からクラスを派生させる必要があります。IlvStExtension のコンストラクタは、次の 2 つのパラメータを取ります。

◆ name は、拡張子の名前です。

◆ editor は、拡張されるエディタのインスタンスです。

IlvStExtension クラスのコンストラクタは、エディタの拡張子リストに新しいインスタンスを追加します。IlvStudio インスタンスを初期化する前に、拡張子のインスタンスを作成する必要があります。エディタが削除されるとき、このインスタンスも削除されます。拡張子は削除しないでください。

以下に例を示します。

```
#include <ivstudio/studext.h>

class MyStudioExtension
: public IlvStExtension {
public:
    MyStudioExtension(IlvStudio* editor);
    virtual IlBoolean preInitialize();
    virtual IlBoolean initializePanels();
    virtual IlBoolean initializeCommandDescriptors();
    virtual IlBoolean initializeBuffers();
    virtual IlBoolean initializeInspectors();
};

int
main(int argc, char* argv[])
{
    IlvSetLanguage();
    // --- Display ---
    IlvDisplay* display = new IlvDisplay("ivstudio", "", argc, argv);
    if (display->isBad()) {
        IlvFatalError("Couldn't open display");
        delete display;
        return 1;
    }

    // ---- Create and initialize the editor ---
    IlvStudio* editor = new IlvStudio(display, argc, argv);
    if (editor->isBad()) {
        IlvFatalError("Could not initialize the editor");
```

```
        delete display;
        return 2;
    }
    new MyStudioExtension(editor); // added line
    editor->initialize();
    editor->parseArguments();
    editor->mainLoop();
    return 0;
}
```

最初の初期化手順

preInitialize メソッドは、エディタが初期化されるときに最初に呼び出されます。この段階では、設定ファイルはまだ読み込まれていません。

このメソッドでは、次を実行します。

- 設定およびデータ・ファイルが保存されるディレクトリを含むように、表示パスを完成させます。
- 設定ファイルを追加します。
- バッファ・タイプを定義してエディタが初期化されるときにこれをデフォルト・バッファにしたい場合、デフォルト・コンストラクタをこのメソッドに設定しなくてはなりません。

例:

```
// A buffer constructor.
static IlvStBuffer* ILVCALLBACK
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
{
   return new MyGadgetBuffer(editor, name);
}

static const char* UserData = "../data";

IlBoolean
MyStudioExtension::preInitialize()
{
   IlvStudio* editor = getEditor();
   // Add the path.
   editor->getDisplay()->prependToPath(UserData);
   // Add an option file.
   editor->addOptionFile("mystudio.opt");
   // Must be done here so
   // the first default buffer will be a MyGadgetBuffer.
   editor->buffers().setDefaultConstructor(MakeMyBuffer);
   return IlTrue;
}
```

バッファの初期化

initializeBuffers メソッドは、定義済みバッファが初期化された後に呼び出されます。バッファの初期化および関連する初期化をこのメソッドで終了させることができます。

例：

```
IlBoolean
MyStudioExtension::initializeBuffers()
{
   IlvStudio* editor = getEditor();
   editor->buffers().registerType("MyGadgetManagerOutput", MakeMyBuffer);
   return IlTrue;
}
```

コマンド記述子の初期化

initializeCommandDescriptors メソッドは、コマンド記述子が読み込まれた後、および定義済みコマンド・コンストラクタが登録された後に呼び出されます。このメソッドでコマンド・コンストラクタを登録することができます。

例：

```
static IlvStCommand*
MkMyShowPanel(IlvStudio* editor)
{
    return new IlvStShowPanel(editor->getPanel("MyPanel"));
}

static IlvStCommand*
MkMyAddClass(IlvStudio*)
{
    return new MyAddClass;
}

static IlvStCommand*
MkMyNewBuffer(IlvStudio*)
{
    return new MyNewBuffer;
}

IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    // Register my commands.
    IlvStudio* editor = getEditor();
    editor->registerCommand("MyShowPanel", MkMyShowPanel);
    editor->registerCommand("AddMyClass", MkMyAddClass);
    editor->registerCommand("MyNewBuffer", MkMyNewBuffer);
    return IlTrue;
}
```

パネルの初期化

initializePanels メソッドが呼び出されたとき、パネル・プロパティが読み込まれて定義済みパネルが作成されますが、パネル・プロパティはまだパネルに適用されていません。このメソッドで独自のパネルを次のように作成します。

```
IlBoolean
MyStudioExtension::initializePanels()
{
    IlvStudio* editor = getEditor();
```

```
    // Create MyGadgetPalette.
    MyGadgetPalette* pal = new MyGadgetPalette(editor);
    pal->connect();
    // Create MyPanel.
    MyPanelHandler* pan = new MyPanelHandler(editor, "MyPanel");
    pan->connect();
    return IlTrue;
}
```

詳細設定の登録

詳細設定パネルを登録するには、詳細設定パネルを作成可能なオブジェクトを詳細設定したオブジェクトのクラス名にマッピングしなくてはなりません。詳細設定パネル・ビルダは、クラス `IlvStInspectorPanelBuilder` から派生させます。詳細設定パネル・ビルダ・クラスを宣言するには、マクロ `IlvStDefineInspectorPanelBuilder` を次のように使用します。

```
IlvStDefineInspectorPanelBuilder(MyClassInspector,\
                                 MyClassInspectorBuilder)
```

詳細設定パネルを登録するために使用したマッピングは、`initializeInspectors` メソッド内で行われ、これは定義済み詳細設定が初期化された後に呼び出されます。

以下は、詳細設定パネル・ビルダを追加する方法です。

```
IlBoolean
MyStudioExtension::initializeInspectors()
{
    IlvStudio* editor = getEditor();
    editor->inspector().registerBuilder("MyClass",
                                        new MyClassInspectorBuilder);
    return IlTrue;
}
```

編集モードの初期化

`initializeModes` メソッドは、定義済み編集モードが初期化された後に呼び出されます。編集モードを提供する場合は、ここで初期化することができます。

最後の初期化手順

`postInitialize` メソッドは、呼び出される最後のメソッドです。

独自のグラフィック・オブジェクトの統合

このセクションでは、グラフィック・オブジェクト・サブクラスを IBM® ILOG® Views Studio に統合させる方法について説明します。

新規クラスを統合するにはこれらの手順に従います。

1. クラスのインスタンスをカレント・バッファに追加するために使用するコマンドを定義します。
2. 必要な #include ステートメントを宣言して、オブジェクトを使用するパネル・クラスで生成します。
3. 既存パレット・パネルにクラスのインスタンスを置くか、あるいは独自のパレットを提供します。
4. オブジェクトのプロパティを編集するために詳細設定パネルを提供します。

オブジェクトを追加するための新規コマンドを定義します

ユーザ定義クラスのインスタンスを IBM® ILOG® Views Studio に追加するには、新規コマンドを記述する必要があります。クラス IlvStClickAddObject は、マウス・クリックで指示された位置にオブジェクトを追加するときに使用するコマンドを定義します。ユーザ定義グラフィック・クラスを作成するには、派生クラスにあるその仮想メンバ関数 makeObject を、下の例で示すように再定義する必要があります。

```
#include <ivstudio/edit.h>

class MyAddClass: public IlvStClickAddObject {
protected:
   virtual IlvStError* makeObject(IlvGraphic*& obj, IlvStudio* ed, IlvAny) {
      MyClass* mc = new MyClass(ed->getDisplay(), IlvRect(0, 0, 40, 40));
      obj = mc;
      return 0;
   }
};
```

以下のコマンド・コンストラクタはユーザ定義クラスの新規インスタンスを作成し、返します。

```
static IlvStCommand*
MkMyAddClass(IlvStudio*)
{
  return new MyAddClass;
}
```

IlvStExtension のサブクラスを作成し、initializeCommandDescriptors メソッドを定義して、次のようにコマンド・コンストラクタを登録することができます。

```
IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    getEditor()->registerCommand("AddMyClass", MkMyAddClass);
    return IlTrue;
}
```

オプション・ファイルで、次のようにコマンド宣言を書くことができます。

```
studio {
  // ...
  command AddMyClass {
    label "MyClass";
    prompt "Add an object of my class";
    category add;
  }
  // ...
}
```

コマンドに与えられた名前は、エディタで登録したものと同じです。この例では、コマンドはコマンド・パネルの追加カテゴリに表示されます。

必要に応じて、オプション・ファイルを ILVSTOPTIONFILE 環境変数を使用して宣言することができます。

生成コードに新規クラスの [インクルード・ファイル] および [ライブラリ・ファイル] を追加する

新しくユーザ定義されたクラスのインスタンスを含めパネルを定義する C++ コードは、新規クラスに対応する #include ステートメントを含んでいなくてはなりません。このインストラクションを追加するには、下記のコードを拡張機能クラスの初期化メソッドに挿入します。

例 :

```
#include <ivstudio/appcode.h>

IlBoolean
MyStudioExtension::initializeBuffers()
{
    // If the IlvRegisterClass is not already done.
    This macro must be called only once.
    IlvRegisterClass (MyClass, TheSuperClass);
    IlvRegisterClassCodeInformation (MyClass,"<myclass.h>","mylib");
    // ...
    return IlTrue;
}
```

パレット・パネルのカスタマイズ

独自のパレットをパレット・パネルに追加するには、.opt ファイルを使用します。新規パレットを追加するには、ツリー・ガジェットの新規パレットに対応するノードを定義して、そのパレットのオブジェクトを含めたデータ・ファイルを提供します。オプション・ファイルで、データ・ファイルの読み込みと表示に使用するコンテナのクラスを指定することができます。定義済みパレットを削除して、デフォルトで選択されるパレットを指定することができます。

パレット・パネルは 2 つの領域に分けられています。パネル上部の領域はツリー・ガジェットを表示し、下部領域はスクロール・ビューを表示します。ツリー・ガジェットは選択することができる利用可能なパレットの階層を表しています。スク

ロール・ビューのコンテナは、選択したパレットの内容を表示します。ツリーの各ノードはパレット記述子に対応しており、ツリーのノードの名前、データ・ファイル、ラベル、および位置で定義されます。パレット記述子は独自のコンテナを持っています。

パレットのコンテナは、パレットが初めて選択されたときに作成されます。コンテナ・クラスが指定された場合、IBM® ILOG® Views Studio は対応するコンテナ情報 (`IlvStContainerInfo`) を使用して、指定したクラスのインスタンスを作成します。デフォルトでは、`IlvGadgetContainer` オブジェクトを使用します。パレットが指定したデータ・ファイルを持つ場合、作成したコンテナはそのデータ・ファイルを読み込みます。すべてのパレット・コンテナは、選択されたパレットに付加されたものを除きすべて非表示です。

オプションのカスタマイズ

オプション・ファイルを使用して新規パレットをパレット・パネルに追加し、定義済みパレットを削除したり、デフォルト・パレットを指定することができます。

新規パレットを記述するには、以下のように `dragDropPalette` オプションを使用します。

```
dragDropPalette "<palette name>" {
    <option-1 <value-1>;
    ...
    <option-n <value-n>;
}
```

パレット・パネルのツリー・ガジェットから定義済みパレットを削除するには、次のように `removeDragDropPalette` オプションを使用します。

```
removeDragDropPalette "<palette name>"
```

デフォルトで選択されるパレットを指定するには、`defaultDragDropPalette` オプションを使用します。

```
defaultDragDropPalette "<palette name>"
```

メッセージのブロードキャスト

以下は、パレット・コンテナが初期化される、あるいはパレットが選択されるときにブロードキャストされるメッセージです。

- `PaletteContainerInitialized` 引数は、選択したパレットの記述子です。このメッセージがブロードキャストされると、コンテナが作成され、そのデータ・ファイルが読み込まれ、パレットのオブジェクトが初期化されます。
- `PaletteSelected` 引数は、選択したパレットの記述子です。

例

下記の例は、IBM ILOG Views Studio パレットに関連するオプションの使用方法を示しています。

```
// mystudio.opt
studio {
   dragDropPalette "MyRootPalette" {
       dataFileName "myfile1.ilv";
       path -before "Gadgets";
   }
   dragDropPalette "MyPalette" {
       label "My Palette";
       bitmap "myicon.gif";
       dataFileName "myfile2.ilv";
       path "Gadgets" "Miscellaneous";
   }
   removeDragDropPalette "ViewRectangles";
   defaultDragDropPalette "MyRootPalette";
}
```

詳細設定パネルの定義および統合

ユーザ定義のグラフィック・オブジェクトのプロパティを詳細設定するには、下記の手順に従います。

1. このオブジェクト・パネル用に新規詳細設定パネル・クラスを定義します。
2. この詳細設定クラスを IBM® ILOG® Views Studio に統合します。

詳細設定パネル・クラスの定義

新規詳細設定パネル・クラスは、`IlvStInspectorPanel` クラスから派生する必要があり、これはファイル `$ILVHOME/studio/ivstudio/inspectors/insppnl.h` で宣言されています。`IlvGraphic` オブジェクトに共通のプロパティの検査機能から自動的に継承させるために、`IlvStIGraphicInspectorPanel` からこのクラスを派生させることもできます。これらのプロパティは 2 つのノートブック・ページ、一般およびコールバックに表示されます。このクラスは、ファイル `$ILVHOME/studio/ivstudio/inspectors/gadpnl.h` にあります。`IlvGraphic` クラスのサブクラスを編集する詳細設定パネルを定義します。

このクラスを派生させるには、次の関数を定義します。

- コンストラクタは表示、パネルのタイトル、データ・ファイル名、およびアップデート・モードを提供するパネル・クラス・コンストラクタを呼び出します。最後の 2 つのパラメータはオプションです。
- `initializeEditors` は、詳細設定パネルのアクセサおよびエディタを登録するために呼び出されます。メソッドの始めに、詳細設定パネルに表示されるべきノートブック・ページを宣言することを忘れないでください。
- `initFrom` は、新規オブジェクトで詳細設定パネルが初期化されるたびに呼び出され、与えられたオブジェクトに応じてパネルを初期化します。たとえば、ライン・エディタでオブジェクトの x 位置を編集する場合、この関数を使用し

てライン・エディタのラベルを x オブジェクトの位置に設定します。デフォルトでは、このメソッドはアクセサおよびエディタを初期化します。これはオーバーライドできません。

- applyChange は、ユーザが [適用] ボタンをクリックして詳細設定オブジェクトに変更を適用させるときに呼び出されます。デフォルトでは、このメソッドはアクセサおよびエディタへのその役割を代理します。これはオーバーライドできません。

下記は、これらの関数定義の例です。

```
class MyClassInspector
: public IlvStIGraphicInspectorPanel
{
public:
    // Constructor
    MyClassInspector(IlvDisplay* display,
                     const char* title,
                     const char*   filename = 0,
                     IlvSystemView transientFor = 0,
                     IlvStIAccessor::UpdateMode mode
                           = IlvStIAccessor::OnApply):
    IlvStIGraphicInspectorPanel(display, title, filename,
                                transientFor, mode) {}
    virtual void initializeEditors();
};

IlvStDefineInspectorPanelBuilder(MyClassInspector,\
                                 MyClassInspectorPanel);

MyClassInspector::MyClassInspector(IlvDisplay* display,
                                   const char* title,
                                   const char*   filename,
                                   IlvSystemView transientFor,
                                   IlvStIAccessor::UpdateMode mode)
:IlvInspectorPanel(display, title, filename, transientFor, mode)
{}

void
MyClassInspector::initializeEditors()
{
    IlvStIGraphicInspectorPanel::initializeEditors();
    // Add notebook pages.
    addPage("&Specific", "../data/myclinsp.ilv";
    // Add editors.
    link("xfield", IlvGraphic::_xValue);
    link("yfield", IlvGraphic::_yValue);
}
```

詳細設定パネルを IBM ILOG Views Studio へ統合する

新規詳細設定をエディタに統合するには、拡張子クラスを変更して、下記のように initializeInspectors を定義します。

```
IlBoolean
MyStudioExtension::initializeInspectors()
{
```

```
    IlvStudio* editor = getEditor();
    editor->inspector().registerBuilder("MyClass",
                                        new MyClassInspectorPanel);
    return IlTrue;
}
```

IBM ILOG Views Studio の拡張 : 例

このセクションの例は、あらゆるグラフィック・オブジェクトで定義済みのコールバックと関連しているエディタの作成方法を示します。loadilv は、パラメータとして読み込まれるファイルの名前を取る定義済みのコールバックです。この名前は、グラフィック・オブジェクトにプロパティとして保存されます。

最初のタスクは、新規プロパティを保存および回復するために、IlvGadgetManager からクラスを派生して read および write を再定義することです。この部分については、ここでは説明しません。記述子 MyGadgetManagerOutput でオブジェクトを保存するクラス MyManager および MyManager によって保存されるオブジェクトを回復するクラス MyContainer があるとします。

エディタを拡張するには以下の手順に従います。

1. 新規マネージャとコンテナに関連付ける *新規バッファの定義*。
2. 新規タイプのバッファを作成するための *新規コマンドの定義*。
3. ファイル名をオブジェクトに関連付ける *新規パネルの定義*。

新規バッファ・クラスの定義

サブクラス MyGadgetBuffer を IlvStGadgetBuffer から定義します。下記は、ヘッダー例です。

```
class MyGadgetBuffer
: public IlvStGadgetBuffer {
public:
    MyGadgetBuffer(IlvStudio*, const char* name, IlvManager* = 0);
    virtual const char* getType () const;
    virtual const char* getTypeLabel() const;
    virtual void setUpPanelClass(IlvStPanelClass*) const;
};
```

ユーザは次を提供します。

- コンストラクタ、これは IlvStGadgetBuffer コンストラクタを呼び出します。マネージャのパラメータがまだ作成されていない場合、これは MyManager インスタンスを作成します。
- 仮想メンバ関数 getType、これはクラス名 MyGadgetBuffer を返します。

- 仮想メンバ関数 getTypeLabel、これはクラス名ラベルを返します。このラベルは、IBM ILOG Views Studio に使用され、メイン・ウィンドウでバッファのタイプを表示します。バッファ・タイプの識別子として使用される getType メンバ関数によって返されるラベルと異なる場合があります。
- 仮想メンバ関数 setUpPanelClass、これは IBM ILOG Views Studio パネル・クラスがバッファ用に作成されるときに呼び出されます。

MyGadgetBuffer クラスは、次のように定義されます。

```
#include <ivstudio/studio.h>
#include <ivstudio/stdesc.h>

#include <mybuf.h>
#include <myman.h>
#include <mycont.h>

MyGadgetBuffer::MyGadgetBuffer(IlvStudio* editor,
                                        const char* name,
                               IlvManager* mgr)
: IlvStGadgetBuffer(editor,
                    name,
                    mgr ? mgr : new MyManager(editor->getDisplay()))
{
}

const char*
MyGadgetBuffer::getType () const
{
    return "MyGadgetBuffer";
}

const char*
MyGadgetBuffer::getTypeLabel () const
{
    return "Mine";
}

void
MyGadgetBuffer::setUpPanelClass(IlvStPanelClass* pclass) const
{
    IlvStGadgetBuffer::setUpPanelClass(pclass);
    pclass->setBaseClass("MyContainer");
}
```

新規クラスがある場合、これをエディタに統合させる必要があります。つまり、新規記述子で保存されたファイルを新規バッファ・タイプに読み込まなくてはならないことをエディタに伝えます。

これを行うには、拡張機能クラスの initializeBuffers メソッドの registerType への呼び出しを追加します。次に例を示します。

```
#include <mybuf.h>

static IlvStBuffer*
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
```

```
{
    return new MyGadgetBuffer(editor, name);
}

IlBoolean
MyStudioExtension::initializeBuffers()
{
     IlvStudio* editor = getEditor();
     // ...
     editor->buffers().registerType("MyGadgetManagerOutput",
                                    MakeMyBuffer);
     // ...
     return IlTrue;
}
```

新規コマンドの定義

エディタは、MyManager で生成されたものを認識するようになりました。しかし、新規バッファ・インスタンスを作成する必要があります。

これを行うには、MyBuffer のインスタンスを作成するために新規コマンドを提供します。仮想メンバ関数 doIt を定義するために IlvStCommand のサブクラスを定義します。次に例を示します。

```
const char* NameNewBuffer = ?MyNewBuffer?;

class MyNewBuffer: public IlvStCommand {
public:
    virtual IlvStError* doIt(IlvStudio*, IlvAny);
};

IlvStError*
MyNewBuffer::doIt(IlvStudio* editor, IlvAny arg)
{
    if (arg) {
        editor->buffers().setCurrent((IlvStBuffer*)arg);
        return 0;
    }
    const char* name = editor->options().getDefaultBufferName();
    IlvStBuffer* buffer = new MyGadgetBuffer(editor, name);
    if (editor->buffers().get(name))
    buffer->newName(name); // uniq name
    return editor->execute(IlvNmNewBuffer, 0, 0, buffer);
}
```

ここでコマンドをエディタに統合させなくてはなりません。そのためには、次の手順に従います。

1. 新規コマンドの初期化関数への登録を追加して、これを構築する関数を提供します。
2. 新規コマンドを、mystudio.cmd と名付けられた新規コマンド宣言ファイルに記述します。このコマンド宣言ファイルを、commandFile オプションを使用してオプション・ファイルに指定しなくてはなりません。

下記は、initialize 関数の例です。

```
static IlvStCommand*
MkMyNewBuffer(IlvStudio*)
{
    return new MyNewBuffer;
}

IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    IlvStudio* editor = getEditor();
    // ...
    editor->registerCommand("MyNewBuffer", MkMyNewBuffer);
    // ...
    return IlTrue;
}
```

下記は、コマンド宣言の例です。

```
command MyNewBuffer {
    label "MyBuffer";
    prompt "Open my buffer";
    category buffer;
}
```

新規パネルの定義

新規プロパティを取得するため、新規パネルを作成する必要があります。下記の手順に従います。

1. ファイル名と名付けられたライン・エディタおよびコールバック [適用] および [取消し] の 2 つのボタンを IBM® ILOG® Views Studio と使用してパネルを作成します。
2. メンバ関数 [適用] および [取消し] の付いたコンストラクタを提供するために IlvStDialog のサブクラスを記述します。
3. 新規パネルをエディタに統合させます。

下記は、ヘッダー例です。

```
#include <ivstudio/panel.h>
class MyPanelHandler
: public IlvStDialog {
public:
    MyPanelHandler(IlvStudio* ed, const char* name,
                   IlvDialog* dlg = 0);
    virtual void apply();
    virtual void reset();
};
```

ユーザは次を提供します。

- コンストラクタ、これは作成したデータ・ファイルを与える IlvStDialog コンストラクタを呼び出します。これは、パネルを初期化して、メンバ関数 resetOnMessage によってメッセージ ObjectSelected にサブスクリプトします。サブスクリプトに渡されたコールバックは、パネルのリセット・メンバ関数を呼び出します。
- 仮想メンバ関数 apply、これはファイル名オブジェクト内容を読み込み、オブジェクト・プロパティに関連付けます。
- 仮想メンバ関数 reset、これは現在選択されているオブジェクトのプロパティのためにファイル名オブジェクト内容を初期化します。

下記は、コーディングの例です。

```
#include <ivstudio/studio.h>
#include <mypan.h>
#include <myutil.h>
#define DATAFILE "../data/mypanel.ilv"

MyPanelHandler::MyPanelHandler(IlvStudio* ed, const char* name,
                               IlvDialog* dlg)
: IlvStDialog(ed, name, DATAFILE, IlvRect(0, 0, 254, 71))
{
    IlvTextField* tf =
       (IlvTextField*)getDialog()->getObject("filename");
    tf->setLabel("", IlTrue);
    resetOnMessage("ObjectSelected");
}
void
MyPanelHandler::apply()
{
   IlvGraphic* obj = getEditor()->getSelection();
   if (obj) {
       const char* name =
   ((IlvTextField*)getDialog()->getObject("filename"))->getLabel();
       if (name && name[0]) {
         MySetParameter(obj, IlvGetSymbol(name));
         obj->setCallbackName(IlvGetSymbol("loadilv"));
         }
   }
}

void
MyPanelHandler::reset()
{
    IlvTextField* tf =
          (IlvTextField*)getDialog()->getObject("filename");
    IlvGraphic* obj = getEditor()->getSelection();
    IlvSymbol* fi = 0;
    if (obj)
        fi = MyGetParameter(obj);
    tf->setLabel(fi ? fi->name() : "", IlTrue);
}
```

パネル・クラスを作成したら、エディタに統合する必要があります。そのためには、次の手順に従います。

1. パネル構築をエディタ初期化関数に追加します。

2. その記述をファイル mystudio.pnl に提供します。

下記はパネルを表示するコマンドのあるコーディング例です。

```
static IlvStCommand*
MkMyShowPanel(IlvStudio* editor)
{
    return new IlvStShowPanel(editor->getPanel("MyPanel"));
}

IlBoolean
MyStudioExtension::initializePanels()
{
    // ...
    // Create MyPanel.
    MyPanelHandler* pan = new MyPanelHandler(getEditor(), "MyPanel");
    pan->connect();
    // ...
    return IlTrue;
}
```

コンテナ情報の提供

コンテナ・クラスを統合するには、IlvStContainerInfo のサブクラスを定義し、下記のように IBM® ILOG® Views Studio 情報セットに追加する必要があります。

```
class MyContainerInfo
: public IlvStContainerInfo {
public:
   MyContainerInfo() : IlvStContainerInfo("MyContainer") {}
   IlvContainer* createContainer(IlvAbstractView* parent,
                                 const IlvRect& bbox,
                                 IlBoolean       useacc,
                                 IlBoolean       visible) {
      return new MyContainer(parent, bbox, useacc, visible);
   }
   IlvContainer* createContainer(IlvDisplay*    display,
                                       const char* name,
                                 const char* title,
                                 const IlvRect& bbox,
                                 IlUInt        properties,
                                 IlBoolean       useacc,
                                 IlBoolean       visible,
                                 IlvSystemView    transientFor) {
       return new MyContainer(display,
                              name,
                              title,
                              bbox,
                              properties,
                              useacc,
                              visible,
                              transientFor);
   }
   const char* getFileCreatorClass() const {
     return "MyGadgetManagerOutput";
```

```
    }
};


    // ...
    editor->addContainerInfo(new MyContainerInfo());
```

コールバックの登録

IBM® ILOG® Views Studio で、`IlvStudio::registerCallback` 関数を呼び出してパネルあるいはアプリケーションをテストするとき、独自のコールバックを使用することができます。

```
static void ILVCALLBACK
MyCallback(IlvGraphic* obj, IlvAny)
{
    IlvPrint("MyCallback is called");
}

 ....
    IlvStudio* editor = ...
....
    editor->registerCallback("MyCallback", MyCallback);
    editor->registerCallback("myCallback", MyCallback);
 ....
```

章

7

詳細設定クラスの使用

この章では、IBM® ILOG® Views Studio 詳細設定クラスを紹介します。以下のトピックに関する情報が記載されています。

- *詳細設定とは？*
- *詳細設定パネルのコンポーネント*
- *新規詳細設定パネルの定義*

この章で参照されているクラスの詳細な説明に関しては、*IBM ILOG Views Studio リファレンス・マニュアル*を参照してください。

> ***メモ:*** *IBM ILOG Views Studio の詳細設定クラスを使用するには、IBM ILOG Views Gadgets および IBM ILOG Views Manager パッケージが必要です。*

詳細設定とは？

IBM® ILOG® Views Studio では、詳細設定は `IlvStInspector` クラスのインスタンスのことです。IBM ILOG Views Studio は、このクラスのインスタンスが 1 つ含まれており、アクティブ・バッファ内の選択したグラフィック・オブジェクトの詳細設定を行うために使用されます。詳細設定の役割は、最後に選択されたグラフィック・オブジェクトに対応する詳細設定パネルを表示することです。

適切な詳細設定パネルを表示するために、詳細設定は、グラフィック・オブジェクト・クラスをマッピングする表を維持してパネル・クラスの詳細設定を行います。グラフィック・オブジェクト・クラスに関連する詳細設定パネルがない場合、詳細設定は、関連する詳細設定パネルを有する継承パス内の最初のスーパークラスの詳細設定パネルにこれを付加します。仮に詳細設定するオブジェクトを、`IlvTextField` から派生したクラス、`IlvMyTextField` タイプであると想定します。このクラスに対して詳細設定パネルが定義されていない場合、IBM ILOG Views Studio 詳細設定は、`IlvTextField` 詳細設定パネルを表示します。

詳細設定パネルは、複数のコンポーネントから構成されています。これらについては次のセクションで説明します。

詳細設定パネルのコンポーネント

オブジェクトの詳細設定は、そのプロパティを調べるということです。一般的に、プロパティの詳細設定を行う場合、プロパティ・パネルは一組のコンポーネント、*アクセサ*および*エディタ*を使用します。

アクセサは詳細設定を行ったプロパティとのインターフェースとなり、エディタは詳細設定パネルでこれをグラフィカルに表すガジェットとのインターフェースとなります (例：`IlvTextField`)。アクセサがエディタと対になっているコンテキストでは、アクセサはプロパティ値を取得し、これをエディタを通じて表示します。この部分のエディタは、その内容が変更されたときにアクセサに通知します。つまり、プロパティの詳細設定は、詳細設定が初期化されたときにアクセサを初期化すること、そしてエディタの内容に行われた変更を有効にするようにアクセサに要求することを意味します。このコンテキストでは、オブジェクトの詳細設定を行うためにアクセサのリストのみが要求されます。

ただし、ある種のエディタについては、作業を行うためにアクセサにリンクされる必要がありません。たとえば、ガジェットのセットを表示 / 非表示にするために使用されるコンボ・ボックスは、初期化するためにデータにアクセスする必要がありません。同様に、コンボ・ボックス内の選択したアイテムの変更は、データに影響しません。これらのスタンドアロン・エディタはアクセサによって初期化されないため、明示的に初期化する必要があります。

アクセサ / エディタの対、およびスタンドアロン・エディタの両方を扱うのに、詳細設定パネルはクラス `IlvStIMainEditor` で定義されるメイン・エディタを使用します。各詳細設定パネルに表示される [適用] ボタンの管理も含めて、詳細設定パネルによって行われる詳細設定操作は、メイン・エディタによって処理されます。

以下の図には、次の項目がそれぞれ示されています。

- 詳細設定パネルの各種コンポーネントおよび相互関連の方法
- 詳細設定パネルの初期化ステップ

- 詳細設定パネルでプロパティが変更されたらどうなるか
- 詳細設定パネル経由で行われたプロパティへ変更を適用する際のステップ

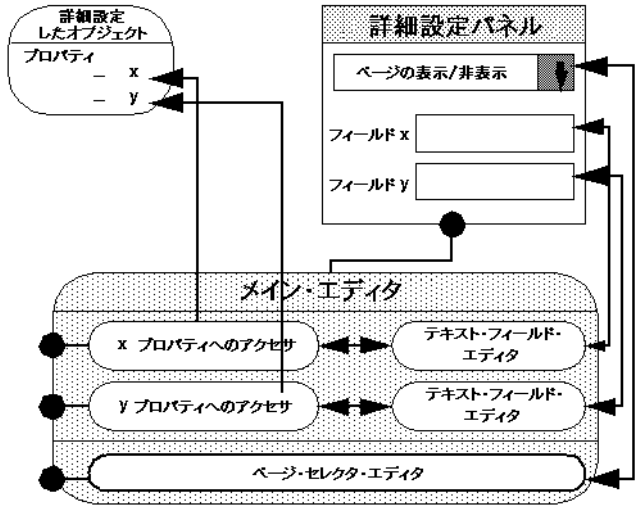


図7.1 *詳細設定パネルのコンポーネント*

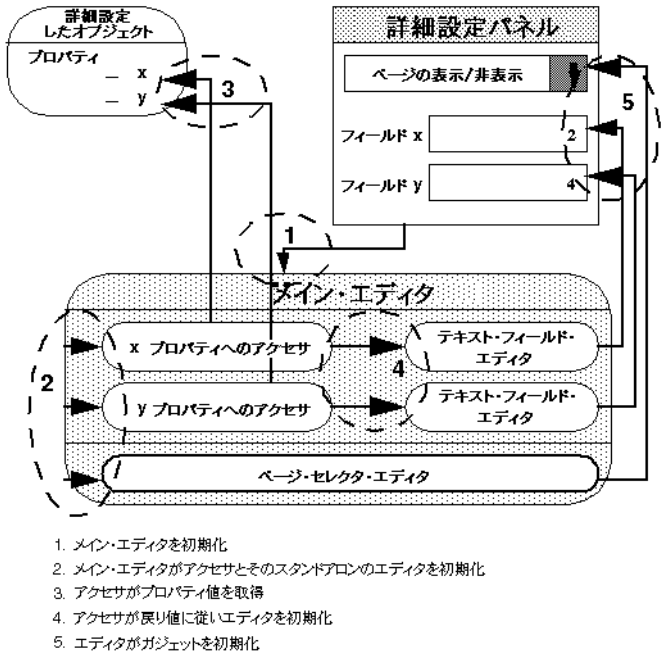


図7.2 詳細設定パネルの初期化ステップ

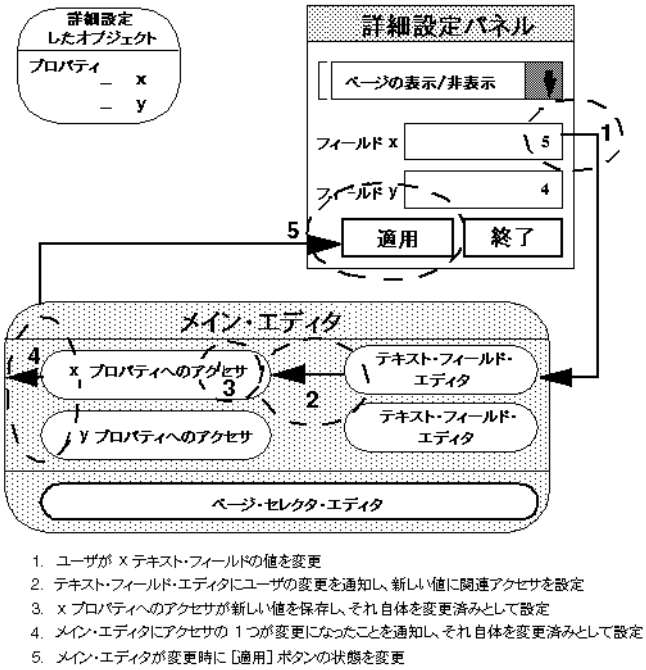
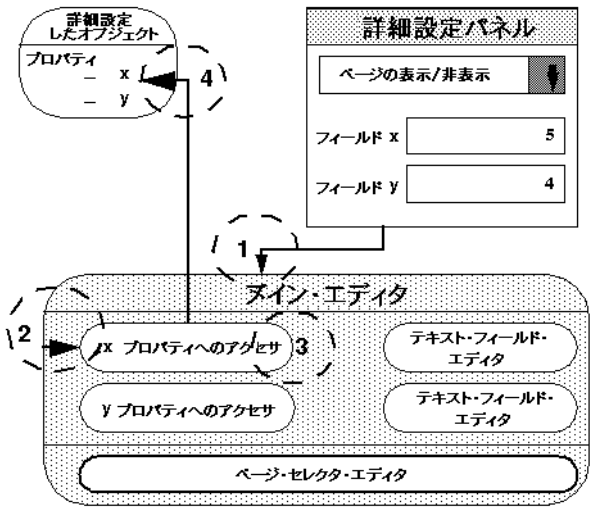


図7.3 *詳細設定パネルで変更が行われたらどうなるか*



1. ユーザーが [適用] ボタンをクリックする
2. メイン・エディタが apply() 関数を呼び出す
3. メイン・エディタが変更になったアクセサについてのみ apply() 関数を呼び出すよう指示する
4. 変更になったアクセサが、新しい値を詳細設定されたオブジェクトに適用する
5. 変更が適用されたら、アクセサは変更状態ではなくなる

図7.4　詳細設定パネルで行われた変更の適用

アクセサ

詳細設定パネルは、すべてのアクセサ・クラスのベース・クラスであるクラス `IlvStIAccessor` のアクセサを扱います。メソッド `initialize` および `apply` を呼び出して、アクセサ上で 2 つのアクションを実行します。最初のメソッドを呼び出して、呼び出しアクセサを初期化し、`apply` は詳細設定を行ったオブジェクトに行われた変更を有効にします。

プロパティ・アクセサ

ほとんどの場合、アクセサはオブジェクト・プロパティの詳細設定を行うために使用されます。下図に示すアクセサ・クラス階層を見れば、その `IlvStIPropertyAccessor`、`IlvStIAccessor` のサブクラスはライブラリのすべてのタイプのアクセサのベース・クラスであることがわかるでしょう。

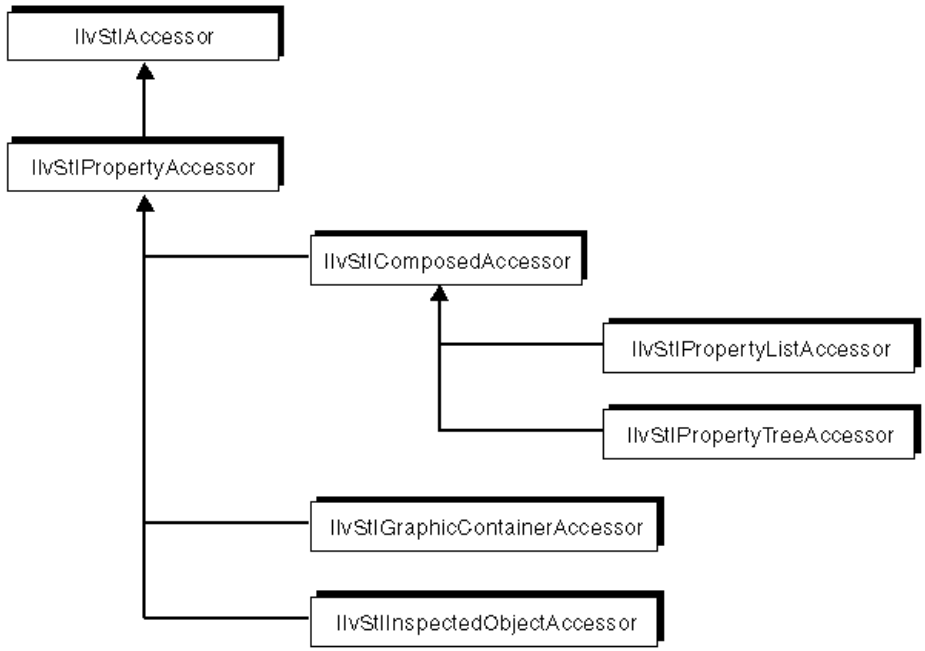


***図7.5** アクセサ階層*

プロパティ・アクセサは、プロパティがカプセル化されているクラス `IlvStIProperty` を経由してプロパティを操作します。たとえば、タイプ `IlvValue` のプロパティを操作するには、アクセサは、クラス `IlvStIProperty` から派生するタイプ `IlvStIValueProperty` を使用します。ここで `IlvValue` オブジェクトはカプセル化されています。

アクセサは、2 つの異なるモードを使用してプロパティの詳細設定を行います。

- 更新モード ― プロパティ・アクセサは変更を直ちに適用すべきか、あるいはユーザが [適用] ボタンをクリックしたときに適用すべきかを指定します。
- 構築モード ― プロパティが見つからない場合に、プロパティを作成するかどうか、および / あるいはコピーするかどうかを指定します。

プロパティ・アクセサの `initialize` および `apply` メソッドがこれらのパラメータを使用するため、`IlvStIPropertyAccessor` をサブクラス化するときにこれらを再定義する必要がありません。代わりに `initialize` および `apply` にそれぞれ呼び出されるメソッド `getOriginalValue` と `applyValue` を再定義します。

次の例は、ガジェット・アイテムのラベルにアクセスするために `IlvStIPropertyAccessor` をサブクラス化する方法を示しています。

```
class IlvLabelAccessor
```

```
: public IlvStIPropertyAccessor
{
public:
    IlvLabelAccessor(IlvGadgetItem* gadgetItem,
                     const char* name = 0,
                     UpdateMode updateMode = NoUpdate,
                     BuildMode buildMode = None):
          IlvStIPropertyAccessor(name, updateMode, buildMode),
          _gadgetItem(gadgetItem)
    {}
protected:
    IlvGadgetItem* _gadgetItem;
    IlvGadgetItem* getGadgetItem()const;
    virtual IlvStProperty* getOriginalValue() const;
    virtual void applyValue(IlvStProperty* property);
};

IlvGadgetItem*
IlvLabelAccessor::getGadgetItem() const
{
    return _gadgetItem;
}

IlvStIProperty*
IlvLabelAccessor::getOriginalValue()
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    return new IlvStIValueProperty(gadgetItem->getLabel(), ?label?);
}
void
IlvLabelAccessor::applyValue(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    IlvValue value;
    property->getValue(value);
    const char* label = (const char*)value;
    gadgetItem->setLabel(label);
}
```

依存アクセサ

詳細設定した特定のプロパティは直接、他の詳細設定したプロパティに依存します。たとえば、中間モードがそのトグル・ボタンに設定されていないのに、ユーザがそのトグル・ボタンの中間状態を詳細設定できるようであってはなりません。つまり、「中間状態」プロパティへのアクセサは常に、「中間モード」プロパティへのアクセサに設定されている値を認識していなくてはなりません。関連付けられたエディタは灰色表示されるか、その値に依存せずに表示される必要があります。「中間モード」プロパティへのアクセサが初期化または変更された場合、これに対応して「中間状態」へのアクセサが再初期化されなくてはなりません。この初期化を行う優先順序のために、「中間状態」プロパティへのアクセサは、メソッド `IlvStIAccessor::addDependentAccessor` を使用して、「中間モード」プロパティへのアクセサに依存する必要があります。

この依存機構は、複合アクセサによっても使用されます。これについては次のセクションで説明します。

複合アクセサ

複合アクセサとは、クラス IlvStICombinedAccessor のインスタンスであり、他のアクセサによって返されるオブジェクトのプロパティの詳細設定に使用される IlvStIPropertyAccessor のサブクラスです。たとえば、ガジェット・アイテムの名前の詳細設定に使用される名前アクセサを考えてみましょう。このアクセサをガジェット・アイテム・アクセサに組み合わせることで、ガジェット・アイテム・リストあるはメッセージ・ラベルのガジェット・アイテムの両方で選択された要素の詳細設定に使用することができます。他のアクセサを通して複合アクセサがアクセスするプロパティが再初期化されるとき、複合アクセサは再初期化されなくてはならないため、通常複合アクセサは依存アクセサとして実装されています。この例では、ガジェット・アイテム・リストの現在の選択の変更により名前アクセサを再初期化する必要があります。

175 ページの*アクセサ*セクションの例を、複合アクセサを表すために下に再び記します。次の例は、ガジェット・アイテムのラベルにアクセスするためにどのように IlvStICombinedAccessor をサブクラス化するかを示しています。

```
class IlvLabelAccessor
: public IlvStICombinedAccessor
{
public:
protected:
    IlvGadgetItem* getGadgetItem()const;
    virtual IlvStProperty* getOriginalValue();
    virtual void applyValue(IlvStProperty* property);
};

IlvGadgetItem*
IlvLabelAccessor::getGadgetItem()
{
    if (!getObjectAccessor())
         return 0;
    IlvStIProperty* property = getObjectAccessor()->get();
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

// The implementation of the getOriginalValue and applyValue methods
// are the same as in previous the sample.
...
```

リスト・アクセサ

リスト・アクセサは、IlvStICombinedAccessor から派生するクラス IlvStIPropertyListAccessor のインスタンスです。リスト・アクセサは、プロパティのリストの詳細設定に使用されます。これによりリスト内でプロパティを追加、削除、変更することができます。このタイプのアクセサは、クラス IlvStIPropertyListEditor のインスタンスともに機能します。この種のエディタは、リストの編集に使用されるガジェットを処理します。つまり、リスト・ガジェットと 4 つのボタン、[挿入]、[追加]、[削除]、[クリーン] です。

次のコード・サンプルは、ガジェット・アイテム・ホルダに含まれているガジェット・アイテムのリストにどのようにアクセスするかを表しています。

```
class IlvStIGadgetItemListAccessor
: public IlvStIPropertyListAccessor {
public:
    // ----------------------------------------------------------------------
    // Constructor / destructor
    IlvStIGadgetItemListAccessor(IlvStIPropertyAccessor* accessor = 0,
                                 IlvStIAccessor::UpdateMode updateMode=
                                                IlvStIAccessor::Inherited,
                                 const char* name = 0);
    ~IlvStIListGadgetItemAccessor();

   IlvListGadgetItemHolder* getListGadgetItemHolder() const;

protected:
    IlvGadgetItem* getGadgetItem(const IlvStIProperty*) const;
    virtual IlvStIProperty** getInitialProperties(IlUInt& count);
    virtual IlvStIProperty*  createDefaultProperty() const;
    virtual IlvGadgetItem* createGadgetItem(
                                      const IlvStIProperty* prop) const;

    virtual void addProperty(IlvStIProperty* property, IlUInt index);
    virtual void replaceProperty(IlvStIProperty* origProperty,
                                 IlvStIProperty* newProperty,
                                 IlUInt index);
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property, IlUInt index);
    virtual void moveProperty(IlvStIProperty* property,
                              IlUInt previousIndex,
                              IlUInt newIndex);
};

IlvStIGadgetItemListAccessor::
          IlvStIGadgetItemListAccessor(IlvStIPropertyAccessor* accessor,
                                       IlvStIAccessor::UpdateMode updateMode,
                                       const char* name):
   IlvStICombinedAccessor(accessor, update, name)
{
}

IlvStIGadgetItemListAccessor::~IlvStIGadgetItemListAccessor()
{
}
IlvListGadgetItemHolder*
IlvStIGadgetItemListAccessor::getListGadgetItemHolder() const
{
    if (!getObjectAccessor())
        return 0;
    IlvStIProperty* property = getObjectAccessor()->get();
    return (property? (IlvListGadgetItemHolder*)property->get() : 0);
}

IlvStIProperty**
IlvStIListGadgetItemAccessor::getInitialProperties(IlUInt& count)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
```

```
    if (!listHolder)
        return 0;
    count = (IlUInt)listHolder->getCardinal();
    if (!count)
        return 0;
    IlvStIProperty** properties = new IlvStIProperty*[count];
    for(IlUInt i = 0; i < count; i++)
        properties[i] = new IlvStIValueProperty(
                            (IlvAny)listHolder->getItem((IlvUShort)i));
    return properties;
}

IlvGadgetItem*
IlvStIListGadgetItemAccessor::getGadgetItem(
                                const IlvStIProperty* property)const
{
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

IlvStIProperty*
IlvStIListGadgetItemAccessor::createDefaultProperty()const
{
    return new IlvStIValueProperty(
                (IlvAny)new IlvGadgetItem("&Item", (IlvBitmap*)0));
}

IlvGadgetItem*
IlvStIListGadgetItemAccessor::createGadgetItem(
                                        const IlvStIProperty* prop) const
{
    const IlvStIGadgetItemValue* value =
                        ILVI_CONSTDOWNCAST(IlvStIGadgetItemValue, prop);
    if (!value)
        return 0;
    IlvGadgetItem* newGadgetItem =
              (value->getGadgetItem()? value->getGadgetItem()->copy() : 0);
    if (!newGadgetItem)
        return 0;
    newGadgetItem->setSensitive(IlTrue);
    newGadgetItem->showLabel(IlTrue);
    newGadgetItem->showPicture(IlTrue);
    newGadgetItem->setEditable(IlFalse);
    return newGadgetItem;
}

void
IlvStIListGadgetItemAccessor::addProperty(IlvStIProperty* property,
                                          IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (listHolder) {
        listHolder->insertItem(getGadgetItem(property), (IlvShort)index);
    }
}

void
IlvStIListGadgetItemAccessor::replaceProperty(IlvStIProperty* origProperty,
                                              IlvStIProperty* newProperty,
```

```
                                                IlUInt position)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvUShort)position);
    listHolder->insertItem(getGadgetItem(newProperty), (IlvUShort)position);
}

void
IlvStIListGadgetItemAccessor::deleteNewProperty(IlvStIProperty* property)
{
    delete getGadgetItem(property);
}

void
IlvStIListGadgetItemAccessor::deleteProperty(IlvStIProperty* property,
                                             IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder
        return;
    listHolder->removeItem((IlvShort)(IlvUShort)index);
}

void
IlvStIListGadgetItemAccessor::moveProperty(IlvStIProperty* property,
                                           IlUInt previousIndex,
                                           IlUInt newIndex)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvUShort)previousIndex, IlFalse);
    listHolder->insertItem(getGadgetItem(property),
                           (IlvShort)(IlvUShort)(newIndex -
                                      (newIndex > previousIndex? 1 : 0)));
}
```

ツリー・アクセサ

ツリー・アクセサは、IlvStICombinedAccessor から派生するクラス IlvStIPropertyTreeAccessor のインスタンスです。ツリー・アクセサは、プロパティのツリーの詳細設定に使用されます。これによりツリー内でプロパティを追加、削除、変更することができます。このタイプのアクセサは、クラス IlvStIPropertyTreeEditor のインスタンスともに機能します。この種のエディタは、ツリーの」編集に使用されるガジェットを処理します。つまり、ツリー・ガジェットおよび次の 5 つのボタン、[挿入]、[追加]、[Child を追加]、[削除]、[クリーン] です。

次のコード・サンプルは、ツリー・ガジェットに含まれているガジェット・アイテムのツリーにどのようにアクセスするかを表しています。

```
class IlvStIGadgetItemTreeAccessor
: public IlvStIPropertyTreeAccessor {
public:
```

```
    IlvStIGadgetItemTreeAccessor(IlvStIPropertyAccessor* accessor = 0,
                                 IlvStIAccessor::UpdateMode updateMode=
                                                  IlvStIAccessor::Inherited,
                                 const char* name = 0,
                                 IlvStIAccessor::BuildMode buildMode =
                                       IlvStIAccessor::Copy);
    ~IlvStIGadgetItemTreeAccessor();

    // --------------------------------------------------------------------
    IlvTreeGadgetItemHolder* getTreeGadgetItemHolder() const;

protected:

    IlvTreeGadgetItem* getGadgetItem(const IlvStIProperty*) const;
    IlvTreeGadgetItem* getParentGadgetItem(const IlvStIProperty*) const;

   // Applying.

    virtual IlUInt getChildPosition(const IlvStIProperty* parentProperty,
                                    const IlvStIProperty* property) const;
    virtual void addProperty(IlvStIProperty* property,
                             const IlvStIProperty* parent,
                             IlUInt childPosition);
    virtual void replaceProperty(IlvStIProperty* origProperty,
                                 IlvStIProperty* newProperty,
                                 const IlvStIProperty* parent,
                                 IlUInt childPosition);
     // Array of properties.
    virtual IlvStIProperty** getInitialChildrenProperties(
                                    IlUInt& count,
                                    const IlvStIProperty* parent = 0) const;
    // Insertion of properties.
    virtual IlvStIProperty* createProperty(const IlvStIProperty* parent,
                                           IlUInt childPosition,
                                           IlvAny param = 0) const;

     // Destruction of properties.
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property);
};

IlvStIGadgetItemTreeAccessor::IlvStIGadgetItemTreeAccessor(
                                    IlvStIPropertyAccessor* accessor,
                                    IlvStIAccessor::UpdateMode updateMode,
                                              const char* name,
                                    IlvStIAccessor::BuildMode buildMode):
    IlvStIPropertyTreeAccessor(accessor,
                               updateMode,
                               buildMode,
                               (name? name : "GadgetItemTreeAccessor"))
{
}

IlvStIGadgetItemTreeAccessor::~IlvStIGadgetItemTreeAccessor()
{
}

IlvTreeGadgetItemHolder*
```

```
IlvStIGadgetItemTreeAccessor::getTreeGadgetItemHolder()const
{
    IlvStIProperty* property = (_accessor? _accessor->get() : 0);
    return (property?  (IlvTreeGadget*)property->getPointer() : 0);
}

// --------------------------------------------------------------------------
IlvTreeGadgetItem*
IlvStIGadgetItemTreeAccessor::getGadgetItem(
                               const IlvStIProperty* property)const
{
    return (property? (IlvTreeGadgetItem*)property->getPointer() : 0);
}

IlvTreeGadgetItem*
IlvStIGadgetItemTreeAccessor::getParentGadgetItem(
                              const IlvStIProperty* property)const
{
    if (!property) {
         // Returns root.
          IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
           if(!holder)
                return 0;
          return holder->getRoot();
    }
    return (property? (IlvTreeGadgetItem*)property->getPointer() : 0);
}

IlUInt
IlvStIGadgetItemTreeAccessor::getChildPosition(
                              const IlvStIProperty* parentProperty,
                              const IlvStIProperty* property)const
{
    // Get parentItem.
    IlvTreeGadgetItem* parentItem = getParentGadgetItem(parentProperty);
    if (!parentItem)
          return (IlUInt)-1;

    IlvTreeGadgetItem* findItem = getGadgetItem(property);
    IlUInt position = 0;
    for(IlvTreeGadgetItem* item = parentItem->getFirstChild();
                           item;
                           item = item->getNextSibling(), position++) {
        if (item == findItem)
             return position;
    }
    return (IlUInt)-1;
}

void
IlvStIGadgetItemTreeAccessor::addProperty(IlvStIProperty* property,
                                          const IlvStIProperty* parent,
                                          IlUInt index)
{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if(!holder)
         return;
     holder->addItem(getParentGadgetItem(parent),
```

```
                    getGadgetItem(property), (IlvInt)index);
}

void
IlvStIGadgetItemTreeAccessor::replaceProperty(IlvStIProperty* origProperty,
                                              IlvStIProperty* newProperty,
                                              const IlvStIProperty* property,
                                              IlUInt index)
{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if(!holder)
        return;
    // Instead of removing the old gadget item and adding the new one, the
    // following line copies the attributes of the new created gadget item
    // to the old one.
    *(getGadgetItem(origProperty)) = *getGadgetItem(newProperty);
    // After this method is called, newProperty becomes the new
    // original property and should therefore be updated.
    // As we have copied attributes from the new created gadget item
    // to the initial one, the inspected gadget item
    // keeps being the one contained in origProperty.
    newProperty->setPointer(origProperty->getPointer());
}

// Array of properties.
IlvStIProperty**
IlvStIGadgetItemTreeAccessor::getInitialChildrenProperties(
                                        IlUInt& count,
                                        const IlvStIProperty* parent) const
{
    IlvTreeGadgetItem* parentItem = getParentGadgetItem(parent);
    if (!parentItem)
        return 0;
    IlvArray properties;
    for(IlvTreeGadgetItem* item = parentItem->getFirstChild();
                           item;
                           item = item->getNextSibling()) {
         properties.add(new IlvStIValueProperty((IlvAny)item));
    }
    count = properties.getLength();
    if (!count)
         return 0;
    IlvStIProperty** props = new IlvStIProperty*[count];
    ::memcpy(props,
             properties.getArray(),
             (size_t)(sizeof(IlvStIProperty*) * (IlvInt)count));
    return props;
}

// Inserting properties.
IlvStIProperty*
IlvStIGadgetItemTreeAccessor::createProperty(const IlvStIProperty*,
                                             IlUInt,
                                             IlvAny) const
{
    return new IlvStIValueProperty((IlvAny)new IlvTreeGadgetItem("&Item"));
}
```

```
// Destruction of properties.
void
IlvStIGadgetItemTreeAccessor::deleteNewProperty(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = (IlvGadgetItem*)property->getPointer();
    if (gadgetItem)
         delete gadgetItem;
}

void
IlvStIGadgetItemTreeAccessor::deleteProperty(IlvStIProperty* property)
{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if(!holder)
         return;
    holder->removeItem(getGadgetItem(property));
}
```

必須条件とバリデータ

アクセサはいくつもの検証を行うために、クラス IlvStIPrecondition および IlvStIValidator、およびそれらの派生クラスを内部で使用します。

必須条件

必須条件は、詳細設定を行ったプロパティにアクセスできるかどうかを決定するためにアクセサが実行するテストです。これらのテストは、クラス IlvStIPrecondition のメソッド isAccessible を呼び出して行われます。必須条件テストに成功すれば、アクセスが許可されます。そうでない場合、アクセスは拒否され、関連するエディタが無効になります。

クラス IlvStIPreconditionValue および IlvStICallbackPrecondition は、ほとんどのケースで必須条件テストを実行するのに十分な IlvStIPrecondition から派生した 2 つのクラスです。

IlvStIPreconditionValue オブジェクトは、アクセサが返した値と所定の値を比較します。

数値フィールドの scientific モード・プロパティへのアクセサを考えてみます。下記のとおり、アクセスは、浮動小数点モードが数値フィールドに設定されているときにのみ許可されます。

```
//This code extract is part of the code of an inspector panel.
IlvStIEditor* editor = link("NumFieldFloat", IlvNumberField::_floatModeValue);
IlvStIPropertyAccessor* floatAccessor = editor->getAccessor();
floatAccessor->setPreviewValueAccessor(previewAccessor,
                                       IlvNumberField::_floatModeValue);

// Scientific value.
IlvStIEditor* editor = link(?ScientificField?,
                            IlvNumberField::_scientificModeValue);
editor->getAccessor()->setPrecondition(
            new IlvStIPreconditionValue(floatAccessor,
                                        (IlBoolean)IlTrue,
```

```
                                        (IlBoolean)IlFalse));
```

IlvStICallbackPrecondition クラスは、isAccessible 関数のコードをコールバックに含めることができる場合に提供されます。このクラスを使用すると、IlvStIPrecondition クラスの派生を避けることができます。

次のコード・サンプルは、ガジェット・アイテム・ラベルが 1 つ以上の「行末」文字を含まない場合に、ガジェット・アイテム・ラベルの整列の変更を避けるために使用する必須条件を実装しています。

```
IlBoolean
IlvStIsMultiLineText(IlvStIProperty* property,
                     IlvAny,
                     IlvStIProperty**,
                     IlvStIPropertyAccessor::PropertyStatus*)
{
    if (!property)
        return IlFalse;
    IlvValue value;
    const char* label = (const char*)property->getValue(value);
    if (!label)
        return IlFalse;
    while (*label)
        if (*label++ == ?\n?)
             return IlTrue;
    return IlFalse;
}
```

コールバック必須条件は、次のように使用されます。

```
 // Define an accessor to the label of the inspected gadget item.
 IlvStIPropertyAccessor* labelAcc;
  ...
 // Define the accessor to the alignment of the
 // the inspected gadget item label.
 IlvStIPropertyAccessor* labelAlignAcc;
...
 labelAlignAcc->setPrecondition(
           new IlvStICallbackPrecondition(labelAcc,
                                          IlvStIsMultiLineText));
```

バリデータ

IBM ILOG Views Studio Inspector API は、ユーザが入力した値が正しいかどうかをテストするために使用するバリデータ・クラス、IlvStIValidator を含みます。テストは、クラスの isValid メソッドで行われます。このメソッドは、そのパラメータとしてパスされた値をテストして、値が有効でない場合に、タイプ IlvStIError のエラーを返します。テストを、ユーザの変更が入力されたときか、あるいはユーザが詳細設定パネルで [適用] ボタンをクリックしたときにのみに実行するかを定義できます。

IlvStIValidator には、値が最小値と最大値の間にあるかどうかをテストする派生クラス IlvStIRangeValidator があります。値範囲に加え、このクラスは、パラメータとしてメッセージ文字列を取ります。このメッセージ文字列は、1 つ以上

の %1、%2 および %3 サブ文字列を含むエラー・メッセージを指定します。これらのサブ文字列は、最小値、最大値、およびテストされた値にそれぞれ置換されます。

次の例は、ユーザが入力した月が 1 と 12 の間かどうかをチェックするためにどのようにバリデータを使用するかを表しています。

```
// Define an accessor to the month property, called monthAccessor.
IlvStIPropertyAccessor* monthAccessor;
...
// Add the month validator to the month accessor.
IlvStIRangeValidator* monthValidator =
    new IlvStIRangeValidator((IlvInt)1, (IlvInt)12, "&MonthNotInRange");
monthAccessor->setValidator(monthValidator);
```

メッセージ文字列 "&MonthNotInRange" は次のように変換されます。

```
"You must specify a month between %1 and %2".
```

エディタ

エディタは、詳細設定した値を編集するために使用されるタイプ IlvStIEditor のオブジェクトです。

アクセサに関連付けられているエディタ

ほとんどの場合、詳細設定する値は直接そのアクセサによって取得され、ガジェットによってグラフィカルに表示される関連エディタを経由して変更されます。

次は、エディタに関連付けることのできるガジェット・クラスのリストです。

- IlvTextField
- IlvNumberField
- IlvToggle
- IlvStringList
- IlvOptionMenu
- IlvScrolledComboBox
- IlvSelector
- IlvSpinBox

列挙されたガジェット・クラスのそれぞれに 1 クラスのエディタがあります。これらのエディタ・クラスは、IlvStIEditor のサブクラス、クラス IlvStIDefaultEditorBuilder にカプセル化されており、そのためユーザに対しては透過的です。エディタを作成してそれをガジェットに関連付けたい場合、クラス IlvStIDefaultEditorBuilder のインスタンスを作成して、これに付加するガジェット名を提供します。このインスタンスが初期化されるとき、それは指定した

ガジェットのタイプに対応するエディタを作成します。作成されたエディタは、IlvStIDefaultEditorBuilder インスタンスの子エディタとして管理されます。

次の例では、数値フィールドの浮動小数点モードの詳細設定に使用する floatAccessor 変数によって表されるアクセサがあります。次は、エディタの作成方法と、詳細設定パネルで "floatToggle" と名付けられたトグルを扱うためにアクセサと関連付ける方法を示しています。

```
IlvStIEditor* editor =
     new IlvStIDefaultEditorBuilder("floatToggle", floatAccessor);
addEditor(editor);
```

アクセサに関連付けられていないエディタ

まれなケースですが、詳細設定した値が複雑すぎて、IlvStIPropertyAccessor オブジェクトがこれを扱えないことがあります。たとえば、詳細設定したオブジェクトから直接値の配列を取得してこれをマトリックスに配置することの方が、IlvStIPropertyAccessor を実行し、値の配列を IlvStIProperty オブジェクトに適応させるよりも簡単です。この場合、クラスを IlvStIEditor から派生させ、次の仮想メンバを再定義しなくてはなりません。

- `virtual IlBoolean initialize() = 0;`
- `virtual IlBoolean apply() = 0;`
- `virtual IlBoolean connectHolder(IlvGraphicHolder* holder);`
- `virtual IlBoolean isModified() const;`
- `virtual void setModified(IlBoolean = IlTrue);`

これらのメソッドに関する詳細は、*IBM ILOG Views Studio リファレンス・マニュアル*を参照してください。

これらの派生したエディタのインスタンスは、他のエディタのように、メソッド IlvStIEditorSet::addEditor を呼び出して詳細設定パネルに追加されます。

新規詳細設定パネルの定義

次のセクションでは、新規詳細設定パネルの定義方法について説明します。新規詳細設定パネルの定義は、次に説明する 2 つの主要手順を含みます。詳細は、以下のとおりです。

1. 新規詳細設定クラスを作成する
2. IBM® ILOG® Views Studio に作成された詳細設定クラスを組み込む この手順はこの章では扱いません。IBM ILOG Views Studio に詳細設定を組み込む方法に関しては、157 ページの*詳細設定の登録*を参照してください。

このセクションの説明は、次のセクションで紹介する 1 つの例に沿って進めます。

例

この例は、ユーザが選択できる色のセットを表示するコンボ・ボックスに詳細設定パネルを作成することから構成されます。この詳細設定パネルは、コンボ・ボックスに表示される色を定義し、これらの色の表示方法を設定するために使用されます。

図 7.6 はコンボ・ボックス、図 7.7 は関連する詳細設定パネルを表しています。

図 7.6 *小さな色付き四角形 (左) と全体に色が付いた四角形 (右) のある色コンボ・ボックス*

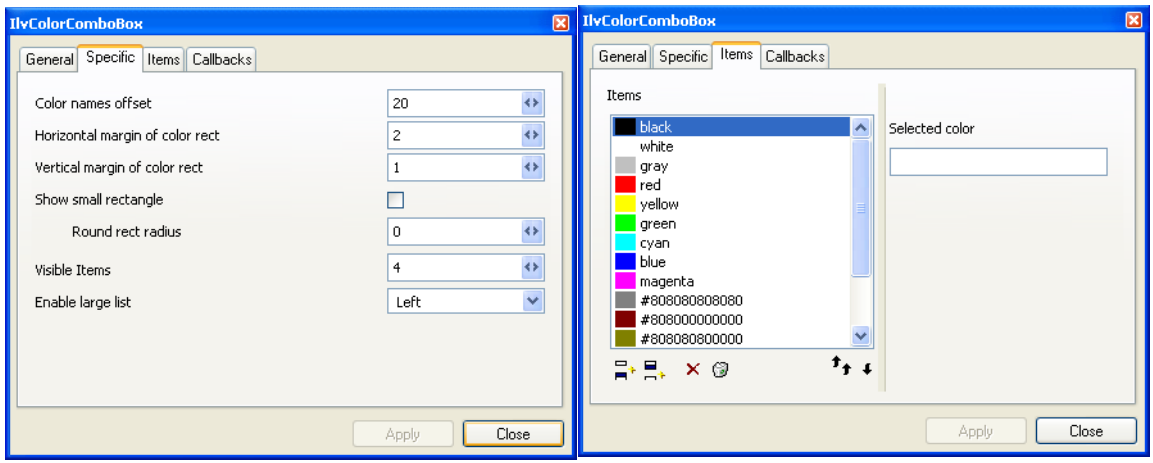


図 7.7 *色コンボ・ボックス詳細設定パネル・ページ*

この例のコード一式は、以下のディレクトリにあります。

```
$(ILVHOME)/samples/studio/colorbox
```

色コンボ・ボックス詳細設定パネルの作成

グラフィック・オブジェクトの詳細設定は、クラス `IlvStInspectorPanel` から派生します。この例では、詳細設定されたグラフィック・オブジェクトはガジェットでもあるため、これから作成する詳細設定パネルは `IlvStInspectorPanel` のサブクラスである、`IlvStIGadgetInspectorPanel` から派生します。

```
class IlvColorComboBoxInspectorPanel
: public IlvStIGadgetInspectorPanel {
public:
    IlvColorComboBoxInspectorPanel(IlvManager* manager,
                                   IlvSystemView transientFor = 0,
                                   IlvStIAccessor::UpdateMode =
                                                  IlvStIAccessor::OnApply);
    virtual void initializeEditors();
};
```

171 ページの*詳細設定パネルのコンポーネント*セクションで言及したように、詳細設定パネルにはアクセサおよびエディタが実装されています。これらはメソッド `initializeEditors` で宣言されていなくてはなりません。この例では、詳細設定パネルを構成する 2 つのノートブック・ページの実装に対応させて、このメソッドの定義を 2 つの部分に分けます (図 7.7 を参照してください)。説明は、以下の 2 セクションに分かれています。

メモ: *[一般] および [コールバック] ページが自動的に作成されます。*

[詳細] ページの実装

[詳細] ページは、次のプロパティの詳細設定を行うためのものです。

- **色名オフセット** 色名を表示するのに使用されるオフセットを指定します。このプロパティは、以下の値によって定義されます。

  `IlvListGadgetItemHolder::_labelOffsetValue`

- **色付き四角形水平マージン** 色付き四角形の垂直境界およびガジェット・アイテムの境界との間のマージンを指定します。このプロパティは、以下の値によって定義されます。

  `IlvColorDrawInfo::_HColorRectMarginValue`

- **色付き四角形水平マージン** 色付き四角形の垂直境界およびガジェット・アイテムの境界との間のマージンを指定します。このプロパティは、以下の値によって定義されます。

  `IlvColorDrawInfo::_VColorRectMarginValue`

- **小さな四角形** ガジェット・アイテムの色付き四角形が色名オフセット・プロパティで指定されたマージンで表示されるべきか、あるいはガジェット・アイテム全体を占めるべきかを指定します。このプロパティは、以下の値によって定義されます。

  `IlvColorDrawInfo::_SmallColorRectValue`

- **角丸四角形半径** 四角の角に適用される半径を指定します。この例の目的のために小さな四角形プロパティが設定されている場合、このプロパティは無視されます。このプロパティは、以下の値によって定義されます。

```
IlvColorDrawInfo::_ColorRoundRectRadius
```

- **表示アイテム**　どのアイテムがコンボ・ボックスのドロップ・ダウン・リストに表示されるかを指定します。このプロパティは、以下の値によって定義されます。

```
IlvScrolledComboBox::_nbVisibleItemsValue
```

- **大きなリスト可**　コンボ・ボックスで「大きなリスト可」オプションを指定します。このオプションの詳細については、*IBM ILOG Views リファレンス・マニュアル*のクラス IlvScrolledComboBox を参照してください。このプロパティは、以下の値によって定義されます。

```
IlvScrolledComboBox::_largeListValue
```

上のプロパティを詳細設定するには、initializeEditors メンバ関数を次のように定義しなくてはなりません。

```
void
IlvColorComboBoxInspectorPanel::initializeEditors()
{

   // Color name offset.
   link("ColorNameOffset", IlvListGadgetItemHolder::_labelOffsetValue);

   // Horizontal margin.
   link("XColorMargin", IlvColorDrawInfo::_HColorRectMarginValue);

   // Vertical margin.
   link("YColorMargin", IlvColorDrawInfo::_VColorRectMarginValue);

   // Small rectangle editor.
   link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);

   // Rounded rectangle editor.
   link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);

   // Visible items.
   link("ComboVisibleItems", IlvScrolledComboBox::_nbVisibleItemsValue);
}
```

link メソッドは、自動的にエディタを作成して最初のパラメータとして渡される名前を持つガジェットに関連付けます。これはその 2 番目のパラメータとして提供されるプロパティへのアクセサも作成します。さらに、プロパティの詳細設定を行うのに併用するエディタとアクセサをリンクします。この関数の詳細は、リファレンス・マニュアルのクラス IlvStInspectorPanel を参照してください。

変更のプレビュー

詳細設定パネル経由で色コンボ・ボックス・プロパティに行われた変更は、プレビュー・ガジェットに反映されます。これを行うには、プレビュー・ガジェットとして使用するガジェットへのアクセサを作成しなくてはなりません。このアクセサを作成するには、クラス IlvStIGraphicContainerAccessor を使用することを推奨します。これを行ったら、アクセサをプレビュー・アクセサとして、

setPreviewAccessor あるいは setPreviewValueAccessor メンバ関数 (IlvStIEditor および IlvStIPropertyAccessor) を使用して詳細設定を行ったプロパティに登録する必要があります。このセクションの前の方で言及したプロパティ用のプレビュー・ガジェットを実装する処理手順を次に挙げます。

```
void
IlvColorComboBoxInspectorPanel::initializeEditors()
{
    IlvStIPropertyAccessor* previewGadgetAcc =
          new IlvStIGraphicContainerAccessor(getHolder(), "ColorItemsList");

    IlvStIEditor* editor;

    // Color name offset.
    editor = link("ColorNameOffset",
                  IlvListGadgetItemHolder::_labelOffsetValue);
    editor->setPreviewValueAccessor(
                               previewGadgetAcc,
                               IlvListGadgetItemHolder::_labelOffsetValue);

    // Horizontal margin.
    editor = link("XColorMargin", IlvColorDrawInfo::_HColorRectMarginValue);
    editor->setPreviewValueAccessor(previewGadgetAcc,
                                    IlvColorDrawInfo::_HColorRectMarginValue);

    // Vertical margin.
    editor = link("YColorMargin", IlvColorDrawInfo::_VColorRectMarginValue);
    editor->setPreviewValueAccessor(previewGadgetAcc,
                                    IlvColorDrawInfo::_VColorRectMarginValue);

    // Small rectangle editor.
    editor = link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);
    editor->setPreviewValueAccessor(previewGadgetAcc,
                                    IlvColorDrawInfo::_SmallColorRectValue);

    // Rounded rectangle editor.
    editor = link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);
    editor->setPreviewValueAccessor(previewGadgetAcc,
                                    IlvColorDrawInfo::_ColorRoundRectRadius);
}
```

必須条件の使用

このセクションの前の方で、角丸四角形半径プロパティは、小さな四角形プロパティが設定されているときは無視されると説明しました。次のコードは、この条件をどのように実装するかを表しています。

```
    // Small rectangle editor.
    editor = link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);
    IlvStIPropertyAccessor* smallRectAcc =
                            (IlvStIPropertyAccessor*)editor->getAccessor();

    // Rounded rectangle editor.
    editor = link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);
    editor->getAccessor()->setPrecondition(
              new IlvStIPreconditionValue(smallRectAcc,
                                          IlFalse, (IlvInt)0));
```

```
    ...
```

[アイテム] ページの実装

[アイテム] ページで、ユーザはコンボ・ボックスに表示される色のリストを編集することができます。色のリストを扱うには、まず、下図のように、クラス IlvStIPropertyListAccessor を派生させてリスト・アクセサを定義しなくてはなりません。リスト・アクセサについては、178 ページの *リスト・アクセサ*を参照してください。

```
class IlvColorItemsAccessor
: public IlvStIPropertyListAccessor {
public:
    // ------------------------------------------------------------------------
    // Constructor / destructor
    ....
    // ------------------------------------------------------------------------
    IlvListGadgetItemHolder* getListGadgetItemHolder() const;

protected:
    IlvGadgetItem* getGadgetItem(const IlvStIProperty* property) const;

    virtual IlvStIProperty** getInitialProperties(IlUInt& count);
    virtual IlvStIProperty*  createDefaultProperty() const;

    virtual void addProperty(IlvStIProperty* property, IlUInt index);
    virtual void replaceProperty(IlvStIProperty* origProperty,
                                 IlvStIProperty* newProperty,
                                 IlUInt index);
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property, IlUInt index);
};
```

getListGadgetItemHolder メソッドは、表示する色を含むガジェット・アイテム・ホルダを返します。この値は、コンストラクタに渡された IlvStIPropertyAccessor によって返されたものです。

```
IlvListGadgetItemHolder*
IlvColorItemsAccessor::getListGadgetItemHolder()const
{
    IlvStIProperty* property = (_accessor? _accessor->get() : 0);
    return (property? (IlvListGadgetItemHolder*)property->getPointer() : 0);
}
```

getGadgetItem メソッドは、そのパラメータとして提供されたプロパティに保存されたガジェット・アイテムを返します。

```
IlvGadgetItem*
IlvColorItemsAccessor::getGadgetItem(const IlvStIProperty* property)const
{
    return (IlvGadgetItem*)(property? property->getPointer() : 0);
}
```

getInitialProperties メソッドは、コンボ・ボックスに含まれる初期色に対応するプロパティの配列を返します。

```
IlvStIProperty**
IlvColorItemsAccessor::getInitialProperties(IlUInt& count)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return 0;

    count = (IlUInt)listHolder->getCardinal();
    if (!count)
        return 0;
    IlvStIProperty** properties = new IlvStIProperty*[count];
    for(IlUInt i = 0; i < count; i++) {
         properties[i] =
             new IlvStIValueProperty(
                  (IlvAny)listHolder->getItem((IlvUShort)i), "Item");
    }
    return properties;
}
```

createDefaultProperty メソッドは、ユーザが新しい色を作成するために [追加] ボタンを押したときに呼び出されます。デフォルトでは、この色は黒です。

```
IlvStIProperty*
IlvColorItemsAccessor::createDefaultProperty() const
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return 0;
    IlvValue valueInfo(IlvColorDrawInfo::_ColorInfosValue->name());
    IlvColorDrawInfo* colorInfo = (IlvColorDrawInfo*)(IlvAny)
                          listHolder->getGadget()->queryValue(valueInfo);
    return new IlvStIValueProperty(
                    new IlvColorGadgetItem(listHolder->getGadget()->
                                           getDisplay()->getColor("Black"),
                                         colorInfo),
                    "Item");
}
```

addProperty メソッドは、index パラメータに指定された位置に最初のパラメータとして与えられたプロパティに含まれるガジェット・アイテムに変更が行われたときに呼び出されます。ガジェット・アイテムはコンボ・ボックスに追加されます。

```
void
IlvColorItemsAccessor::addProperty(IlvStIProperty* property, IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (listHolder)
        listHolder->insertItem(getGadgetItem(property),
                               (IlvShort)(IlvUShort)index);
}
```

replaceProperty メソッドは、2 番目のパラメータとして与えられたプロパティに含まれるガジェット・アイテムによって最初のパラメータとして与えられたプロパティ内にあるガジェット・アイテムを置き換える変更が行われるときに呼び

出されます。3 番目のパラメータは、コンボ・ボックスの置き換えられたガジェット・アイテムの位置を示します。

```
void
IlvColorItemsAccessor::replaceProperty(IlvStIProperty* origProperty,
                                       IlvStIProperty* newProperty,
                                       IlUInt)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    IlvGadgetItem* origGadgetItem = getGadgetItem(origProperty);
    IlvGadgetItem* newGadgetItem = getGadgetItem(newProperty);
    *(origGadgetItem) = *newGadgetItem;
    newProperty->setPointer(origGadgetItem);
}
```

deleteNewProperty メソッドは、そのパラメータとしてパスされたプロパティに含まれるガジェット・アイテムを削除します。このメソッドは、ユーザによって行われた変更がキャンセルされたときに、[追加] ボタンを押して作成されたガジェット・アイテムを破壊するために呼び出されます。このガジェット・アイテムは実際にはコンボ・ボックスに追加されていないので、削除する必要はありません。

```
void
IlvColorItemsAccessor::deleteNewProperty(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem(property);
    if (gadgetItem)
        delete gadgetItem;
}
```

deleteProperty メソッドは、色コンボ・ボックスからパラメータとして与えられたプロパティに含まれるガジェット・アイテムを削除する変更が行われるときに呼び出されます。

```
void
IlvColorItemsAccessor::deleteProperty(IlvStIProperty*, IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvShort)(IlvUShort)index);
}
```

色リスト・アクセサの再利用

これまで見てきたように、リスト・アクセサはコンボ・ボックスに直接アクセスするのではなく、そのガジェット・アイテム・ホルダを経由します。そのため、同じリスト・アクセサを色文字列リストの詳細設定に再利用することができます。詳細設定を行ったコンボ・ボックスのガジェット・アイテム・ホルダにアクセスするには、次のコード・サンプルに示すように、複合アクセサを作成する必要があります。複合アクセサの詳細は、178 ページの *複合アクセサ* を参照してください。

この複合アクセサは、パラメータとして IlvColorItemsAccessor コンストラクタに提供されます。

```
class IlvColorGadgetItemHolderAccessor
: public IlvStICombinedAccessor
{
public:
    IlvColorGadgetItemHolderAccessor(IlvStIPropertyAccessor* accessor = 0,
                                     UpdateMode updateMode = NoUpdate,
                                     BuildMode buildMode = None,
                                     const char* name = 0);
    // ---------------------------------------------------------------------
protected:
    virtual IlvStIProperty* getOriginalValue();
};

IlvStIProperty*
IlvColorGadgetItemHolderAccessor::getOriginalValue()
{
    IlvStIProperty* property =
                  (getObjectAccessor()? getObjectAccessor()->get() : 0);
    if (!property)
         return 0;
    IlvColorComboBox* combo = (IlvColorComboBox*)property->getPointer();
    if ((!combo) || (!combo->getStringList()))
        return 0;
    return new IlvStIValueProperty((IlvListGadgetItemHolder*)combo,
                                   "ColorHolder");
}
```

クラス IlvColorItemsAccessor は、次の方法で詳細設定パネルで使用されます。

```
IlvColorItemsAccessor* lstAccessor =
    new IlvColorItemsAccessor(
         new IlvColorGadgetItemHolderAccessor(getInspectedGraphicAccessor()));
```

リストの色を変更する

前のセクションで、色のリストにアクセスする方法を説明しました。ここでは、リストで選択された色を変更してみましょう。これを行うには、IlvColorGadgetItem ガジェット・アイテムの色の詳細設定を可能にするクラスを定義します。色はガジェット・アイテム・ラベルによって定義されるため、ラベルの変更は色の変更を意味します。

```
class IlvGadgetItemColorAccessor
: public IlvStICombinedAccessor
{
public:
      ...
      // ---------------------------------------------------------------------
protected:
    IlvGadgetItem* getGadgetItem()const;
    virtual IlvStIProperty* getOriginalValue();
    virtual void            applyValue(IlvStIProperty*);
};

IlvGadgetItem*
```

```
IlvGadgetItemColorAccessor::getGadgetItem() const
{
    IlvStIProperty* property =
                (getObjectAccessor()? getObjectAccessor()->get() : 0);
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

IlvStIProperty*
IlvGadgetItemColorAccessor::getOriginalValue()
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    if (!gadgetItem
        return 0;
    return new IlvStIValueProperty(gadgetItem->getLabel(), "Color");
}

void
IlvGadgetItemColorAccessor::applyValue(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    if (!gadgetItem)
        return;
    IlvValue value;
    gadgetItem->setLabel((const char*)property->getValue(value));
}
```

このクラスは、次のように詳細設定パネル・コードで使用されます。

```
editor = new IlvStIPropertyColorEditor("EditColorItem",
       new IlvGadgetItemColorAccessor(lstAccessor->getSelectionAccessor()));
addEditor(editor);
```

クラス IlvStIPropertyColorEditor は、色の選択を可能にする選択フィールドとのインターフェースです。選択フィールドに表示されるラベルは、選択した色の名前です。

色リスト・エディタの作成

アイテムのリストを表示するには、IlvStIPropertyListEditor によって処理される IlvStringList を使用するのが一般的です。しかしこの例では、文字列ではなくリストに色ガジェット・アイテムを表示させます。IlvStringList のサブクラスであるクラス IlvColorStringList を使用して、このようなリストを実装しました。この新しいクラスとのインターフェースするために、次のエディタ・クラスを定義しました。

```
class IlvColorListEditor
: public IlvStIPropertyListEditor {
public:
    // ----------------------------------------------------------------------
    // Constructor / destructor
    ...
    // ----------------------------------------------------------------------
    // Overridables.
    virtual IlBoolean connectHolder(IlvGraphicHolder* holder);
protected:
    virtual IlvGadgetItem* createGadgetItem(
```

```
                                        const IlvStIProperty* property) const;
};

IlBoolean
IlvColorListEditor::connectHolder(IlvGraphicHolder* holder)
{
    // Replaces string list of colors by an IlvColorStringList.
    IlvGraphicHolder* subHolder;
    IlvGadget* oldList =
          (IlvGadget*)IlvStIFindGraphic(holder, getName(), &subHolder);
    if (!oldList)
        return IlvStIPropertyListEditor::connectHolder(holder);
    IlvRect bbox;
    oldList->boundingBox(bbox);
    IlvColorStringList* colorList =
            new IlvColorStringList(oldList->getDisplay(),
                                   bbox,
                                   oldList->getThickness(),
                                   oldList->getPalette());
    colorList->useFullSelection(IlTrue, IlFalse);
    colorList->setSelectionMode(IlvStringListSingleSelection);
    colorList->setExclusive(IlTrue);
    colorList->scrollBarShowAsNeeded(IlTrue, IlTrue, IlFalse);
    subHolder->getContainer()->replace(oldList, colorList, IlTrue);

    return IlvStIPropertyListEditor::connectHolder(holder);
}

IlvGadgetItem*
IlvColorListEditor::createGadgetItem(const IlvStIProperty* property) const
{
    IlvGadgetItem* gadgetItem = (IlvGadgetItem*)property->getPointer();
    if (!gadgetItem)
        return 0;
    IlvValue valueInfo(IlvColorDrawInfo::_ColorInfosValue->name());
    IlvColorDrawInfo* colorInfo =
         (IlvColorDrawInfo*)(IlvAny)getListGadget()->queryValue(valueInfo);
    IlvGadgetItem* newGadgetItem =
         new  IlvColorGadgetItem(getDisplay()->getColor(gadgetItem-
>getLabel()),
                                 colorInfo);
    newGadgetItem->setEditable(IlFalse);
    return newGadgetItem;
}
```

色アイテムの詳細設定のためのアクセサおよびエディタを詳細設定パネルに宣言する

色アイテムの詳細設定用アクセサおよびエディタは、次のように initializeEditors メソッドで宣言されます。

```
IlvColorItemsAccessor* lstAccessor =
     new IlvColorItemsAccessor(
         new IlvColorGadgetItemHolderAccessor(getInspectedGraphicAccessor()));
addEditor(new IlvColorListEditor(lstAccessor, "ColorItemsList"));

IlvStIEditor* editor =
    new IlvStIPropertyColorEditor("EditColorItem",
        new IlvGadgetItemColorAccessor(lstAccessor->getSelectionAccessor()));
addEditor(editor);
```

II 部

IBM ILOG Views Gadgets

第 2 部では、IBM® ILOG® Views Gadgets を組み込んだアプリケーション開発に関する情報を提供します。

章

8

IBM ILOG Views Gadgets の概要

IBM® ILOG® Views Gadgets パッケージは、対話的なグラフィカル・ユーザ・インターフェースを作成するための C++ 型ライブラリです。このパッケージは、IBM ILOG Views Foundation パッケージの頂点に構築され、ガジェットと呼ばれる特殊なグラフィック・オブジェクト作成のためのクラスから構成されており、これをグラフィック・パネルやインターフェースを作成するためにコンテナ・オブジェクトに追加することができます。ボタン、ツールバー、およびメニューなどは、IBM ILOG Views Gadgets で作成できる多くの対話的なグラフィック・オブジェクトの一部です。

この概要の章は、以下のセクションで構成されています。

- *Gadgets の主な機能*
- *Gadgets ライブラリ*
- *Gadgets のスナップショット*

Gadgets の主な機能

IBM® ILOG® Views Gadgets ライブラリは、次を提供します。

- ボタン、テキスト・フィールド、メニュー、ツールバーなど多数の軽量グラフィック・オブジェクト。

- 定義済みダイアログ・ボックスなど多数のガジェット・コンテナ。
- 4 種類の定義済み表現スタイル：Motif®、Microsoft® Windows® 3.11、Microsoft Windows 95、Microsoft Windows XP。
- ユーザ独自の表示スタイルを簡単に作成する方法。
- UNIX® ワークステーションおよび Microsoft Windows を実行する PC に移植可能なライブラリ。
- Motif および Microsoft Windows など標準のウィジェット・ツールキットでプログラミングしたアプリケーションと、IBM ILOG Views ガジェットを使用した新しいアプリケーションとを組み合わせる簡単な方法。

Gadgets のスナップショット

すべてのガジェットのベース・クラスは、`IlvGadget` です。このクラスは、IBM® ILOG® Views Foundation ライブラリのクラスである `IlvGraphic` から派生します。

下記の図は、Gadgets ライブラリが提供する数々のガジェットを示したものです。

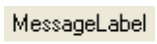
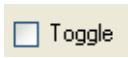
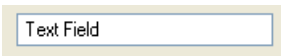
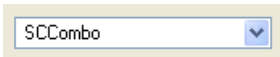
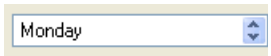
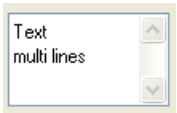
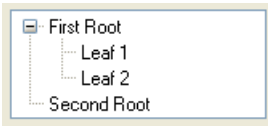
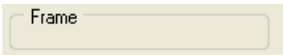
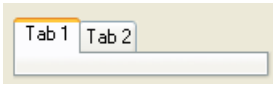
メニュー

<table>
<tr>
<td> 

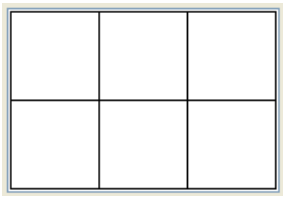
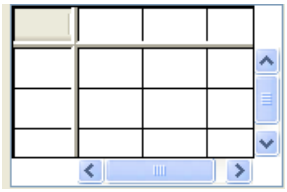
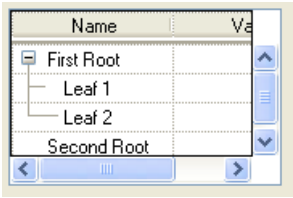
IlvMenuBar</td>
<td> 
IlvToolBar</td>
<td> 

IlvPopupMenu</td>
</tr>
</table>

共通ガジェット

 IlvMessageLabel	 IlvButton	 IlvToggle
 IlvTextField	 IlvComboBox および IlvScrolledComboBox	 IlvSpinBox
 IlvStringList	 IlvText	 IlvTreeGadget
 IlvFrame	 IlvNotebook	

マトリックス

 IlvMatrix	 IlvSheet	 IlvHierarchicalSheet

Gadgets ライブラリ

アプリケーションで使用する各ガジェットに、適切なヘッダー・ファイルが含まれている必要があります。ガジェットのヘッダー・ファイルは、次のディレクトリにあります。

```
ILVHOME/include/ilviews/gadgets
```

アプリケーションを次のガジェット・ライブラリにリンクする必要もあります。

- Microsoft® Windows® プラットフォームの場合は、`ilvgadgt.lib`
- UNIX® プラットフォームの場合は、`libilvgadgt`

高度なガジェットを使用している場合は、アプリケーションを次のガジェット・ライブラリにリンクする必要があります。

- Microsoft Windows プラットフォームの場合は、`ilvadvgdt.lib`
- UNIX プラットフォームの場合は、`libilvadvgdt`

ガジェット・クラスが標準あるいは高度なガジェット・ライブラリのどちらにあるかについては、リファレンス・マニュアルを参照してください。

> ***メモ:*** *ガジェット・ライブラリは、*`ILVHOME` *ディレクトリに位置するリソースを使用します。*`ILVHOME` *を設定したくない、あるいは IBM ILOG Views がターゲットのコンピュータにインストールされていない場合、アプリケーションにこれらのリソースを追加する必要があります。*

また、アプリケーションが使用するルック・アンド・フィールによっては、アプリケーションをルック・アンド・フィール・ガジェット・ライブラリにリンクさせる必要もあります。デフォルトでは、UNIX 上で起動するアプリケーションは Motif® ルックを使用し、Windows 上で起動するアプリケーションは Windows が提供するルックの 1 つを使用します。詳細については、224 ページの *ガジェットのルック・アンド・フィール* を参照してください。

コード・サンプル

ボタンとコンテナを表示する基本的なプログラムです。ボタンをクリックするとプログラムが終了します。

```
#include <ilviews/gadgets/button.h>
#include <ilviews/gadgets/gadcont.h>

void Quit(IlvGraphic*, IlAny arg)
{
    IlvDisplay* display = (IlvDisplay*)arg;
    delete display;
    IlvExit(0);
}

int main(int argc, char* argv[])
{
    // Create the display.
    IlvDisplay* display = new IlvDisplay("Hello", "", argc, argv);
    if(!display)
       return 0;
    if (display->isBad()) {
       delete display;
       return 1;
    }

    // Create the container.
    IlvGadgetContainer* cont =
       new IlvGadgetContainer(display, "Hello", "Hello", IlvRect(0,0,100,100));
    cont->moveToScreen(IlvCenter);

    // Add the button.
    IlvButton* button = new IlvButton(display, IlvPoint(30, 30), "Click Me !");
    button->addCallback(Quit, display);
    cont->addObject(button);

   // Show the container and run the event loop.
    cont->show();
    IlvMainLoop();

    return 0;
}
```

章

9

ガジェットの理解

この章では、ライブラリのすべてのガジェットに共通するプロパティを紹介します。以下のトピックから構成されています。

- *ガジェット・ホルダ*
- *共通ガジェット・プロパティ*
- *ガジェットのルック・アンド・フィール*

ガジェット・ホルダ

ガジェット・ホルダは、ガジェットの保存、表示、および処理用のオブジェクトです。ガジェット・ホルダのメイン・クラスは、`IlvGadgetContainer` クラスです。このクラスは `IlvContainer` から派生しているので、オブジェクトを追加したり、削除するメンバ関数などは、このスーパークラスが提供するすべての機能を継承しています。キーボード・フォーカス管理、アタッチメント、およびツールチップなど基本的な機能も提供しています。

このセクションには、以下のトピックに関する情報が記載されています。

- *利用可能なガジェット・ホルダのリスト*
- *イベントの処理*

- *フォーカス管理*
- *ガジェット・アタッチメント*

利用可能なガジェット・ホルダのリスト

`IlvGadgetContainer` オブジェクトが、IBM® ILOG® Views Gadgets が提供する唯一のガジェット・ホルダというわけではありません。このセクションでは、その他の利用可能なガジェット・ホルダを紹介します。

- *ガジェット・マネージャ*
- *ノートブック*
- *マトリックス*
- *ツールバー*
- *ペイン・コンテナ*

次に関する情報もあります。

- ガジェット・ホルダ使用における *制約*

ガジェット・マネージャ

`IlvGadgetManager` クラスは、ガジェットを扱う `IlvManager` クラスのサブクラスです。マネージャの詳細については、関連する*ユーザ・マニュアル*を参照してください。`IlvManager` オブジェクトと違って、`IlvGadgetManager` のインスタンスは、同時に複数のビューに表示することができないため、関連付けられたビューを 1 つだけ持ちます。ベーシック・グラフィック・オブジェクトでは、複数ビューの表示が可能です。

原則的に、ガジェットを `.ilv` ファイル (IBM ILOG Views フォーマット) に保存する場合を除き、ガジェットの保存にはガジェット・マネージャよりもガジェット・コンテナを使用することをお勧めします。

ノートブック

ガジェットをノートブック・ページ内に表示させることができます。デフォルトのノートブック・ページは、ガジェット・コンテナを使用して実装されています。詳細は 253 ページの *ノートブック・ページの処理* を参照してください。

マトリックス

マトリックスとは、行および列から構成される特殊なガジェットです。各マトリックス・アイテムは、マトリックス内で独自の振る舞いを持つガジェットを含んでいます。詳細については、327 ページの *マトリックスでガジェットを使用する* を参照してください。

ツールバー

ガジェットをツールバーの中に表示させることができます。詳細については、312 ページの *ツールバーでガジェットを管理する* を参照してください。

ペイン・コンテナ

`IlvPanedContainer` クラスは、コンテナをペインに分割する `IlvGadgetContainer` のサブクラスです。各ペインは、ガジェットをカプセル化することができます。詳細については、337 ページの *グラフィック・ペインの作成* を参照してください。

制約

ガジェットを保存するのに単純なコンテナを使用することはお勧めしません。これらのオブジェクトは、キーボード・フォーカス管理のような機能を実装しないためです。ガジェットをこれらのコンテナに追加すると、予期せぬ結果を招くことがあります。

また、ガジェットをズームインあるいはズームアウトすることはできません。このため、ガジェット・ホルダのスケール係数を変更することはお勧めしません。

イベントの処理

ガジェット・ホルダは、イベントをガジェットにディスパッチする役目を果たします。`IlvGadget` クラスは、マウス・クリックや、キーボードの使用などのユーザ・イベントを処理する `handleEvent` メンバ関数を持ちます。他のグラフィック・オブジェクトとは異なり、ガジェットを使用可能にするため、これにインタラクタを設定する必要はありません。

> ***メモ:*** *ただし、希望する場合はガジェットにインタラクタを設定することができます。*

`handleEvent` メンバ関数は仮想であり、追加イベントを処理するためにサブクラス内で再定義することができます。

ガジェット・ホルダ・イベント

マウスが `IlvGadget` オブジェクト内に入る、あるいはそこから離れるとき、関連付けられているガジェット・ホルダは、`IlvMouseEnter` および `IlvMouseLeave` イベント (これら 2 つのイベントは enum `IlvEventType` で定義されています) を生成します。これらのイベントはガジェット、またはその関連付けられているインタラクタがある場合はそれに送られ、`handleEvent` メンバ関数によって処理されます。次に、仮想メンバ関数 `IlvGadget::enterGadget` あるいは `IlvGadget::leaveGadget` が呼び出されます。デフォルトでは、これらのメンバ関数は Enter Gadget および Leave Gadget コールバックをそれぞれ呼び出します。217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

この 1 つの結果として、アクセラレータが IlvView オブジェクトに付加されているため、IlvMouseEnter および IlvMouseLeave イベントでアクセラレータをトリガできないという点があります。IlvView オブジェクトは、これらのイベントを認識できません。

> ***メモ:*** *これは* IlvGadgetContainer *および* IlvGadgetManager *クラスにのみ該当します。*

フォーカス管理

ガジェット・ホルダは、キーボード・フォーカスを管理します。ガジェットの場合、フォーカスを持つということは、キーボード・イベントを受け取ることができるということを意味します。ユーザがマウスでその上をクリックするときにフォーカスを取得します。

Tab キーを押すと、フォーカスが次のガジェットに移動します。Shift-Tab を押すと、フォーカスが前のガジェットに移動します。デフォルトでは、Tab キーは左から右のガジェットへ、そして上から下のガジェットへとフォーカスを移動させます。ただし、Tab キーが押されたときにどの順番でガジェットがフォーカスを得るかを、フォーカス・チェーンを定義して指定することができます。

このセクションでは、以下のトピックを取り上げます。

- *フォーカス・チェーンの定義*
- *ガジェット・ホルダ間にフォーカス・チェーンを設定する*
- *フォーカス変更の通知*

フォーカス・チェーンの定義

同じガジェット・ホルダに保存されているガジェットのみをフォーカス・チェーンでリンクすることができます。フォーカス・チェーンを定義するには、次の IlvGraphic クラスのメンバ関数を使用します。

- IlvGraphic::setNextFocusGraphic
- IlvGraphic::setPreviousFocusGraphic
- IlvGraphic::setLastFocusGraphic
- IlvGraphic::setFirstFocusGraphic

メンバ関数 setNextFocusGraphic および setPreviousFocusGraphic で提供されている name パラメータは、ターゲット・ガジェットの名前から作成する必要があるシンボル名です。たとえば、gadget のフォーカス・チェーンの次のオブジェクトを [ボタン] という名前のガジェットにしたい場合は、次を呼び出します。

```
gadget->setNextFocusGraphic(IlvGetSymbol("Button"));
```

ガジェット・ホルダ間にフォーカス・チェーンを設定する

デフォルトでは、ユーザがフォーカス・チェーンの最後のガジェットに達したとき、フォーカス・ループはチェーンの最初のガジェットに戻ります。しかし、次のメンバ関数を使用すると、フォーカスを指定する別のガジェット・ホルダに移動させることができます。

- `IlvGraphicHolder::getNextFocusHolder`
- `IlvGraphicHolder::setNextFocusHolder`
- `IlvGraphicHolder::getPreviousFocusHolder`

フォーカス変更の通知

キーボード・フォーカスが `IlvGadget` オブジェクトに入る、あるいはそこから離れるとき、その関連付けられたガジェット・ホルダは `IlvKeyboardFocusIn` および `IlvKeyboardFocusOut` イベントを生成します。これらのイベントはガジェット、またはその関連付けられているインタラクタがある場合はそれに送られ、`handleEvent` メンバ関数によって処理されます。次に、仮想メンバ関数 `IlvGadget::focusIn` および `IlvGadget::focusOut` が呼び出されます。デフォルトでは、これらのメンバ関数は Focus In および Focus Out コールバックをそれぞれ呼び出します。217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

ガジェット・アタッチメント

ガジェット・ホルダは、ホルダがリサイズされたときにガジェットのジオメトリを管理するアタッチメント・モデルを提供します。このアタッチメント・モデルは、`IlvGraphicHolder` インターフェースによって定義されます。

`getHolder` メンバ関数を使用して `IlvGraphicHolder` インターフェースへのポインタを得ることができます。

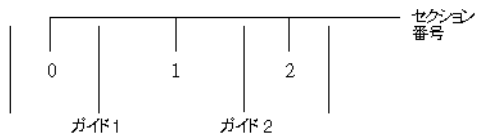
ガジェットをそのガジェット・ホルダに付加するには、ガイドを定義しなくてはなりません。

このセクションでは、以下のトピックを取り上げます。

- *ガイドおよびセクションの概要*
- *ガジェットをガイドに付加する*
- *ガジェットの重みを設定する*

ガイドおよびセクションの概要

ガイドは、ガジェット・ホルダをいくつかのセクションに、水平あるいは垂直に分割します。



ガイドが区切るセクションには番号が振られますが、ガイドに番号は振られていません。新規ガイドが追加されると、その結果生じる新しいセクションを含めるためにセクションに番号が振られます。

デフォルトでは、ウィンドウの境界に沿ったものしかありません。

ホルダがリサイズされると、それぞれのそのセクションは重みに応じてサイズ変更されます。セクションの重みとは、ウィンドウがリサイズされたときに他のセクションに相対的に (ガイドで区切られ) 割り当てられているウィンドウの部分のことです。次の式は、ウィンドウがリサイズされたときに各セクションに適用されます。

$$\text{新しいセクションのサイズ} = \text{セクションの初期サイズ} + \text{デルタ} \times \frac{\text{セクションを限定するガイドの重み}}{\text{すべてのガイドの重み合計}}$$

ここで、デルタは、ウィンドウの新しいサイズからその初期サイズを差し引いたものと等しくなります。

下のリストにある IlvGraphicHolder メンバ関数を使用してガイドを操作することができます。

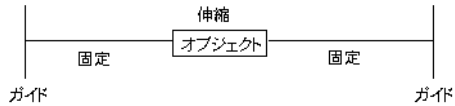
- addGuide
- removeGuide
- getGuideCardinal
- getGuidePosition
- getGuideSize
- getGuideWeight
- getGuideLimit

ガジェットをガイドに付加する

ガイドが定義されると、メンバ関数 IlvGraphicHolder::attach を使用して、ガジェットをこれらに付加することができます。

```
holder->attach(object);
```

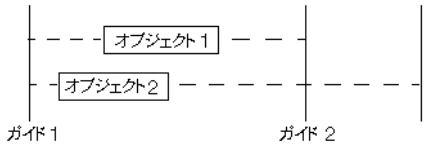
このコードは、下の図で示すようにガジェットをガイドに付加します。



この例では、ウィンドウの境界に沿って配置されているデフォルトのガイドを使用します。しかし、attach メンバ関数の最後の 3 つのパラメータを使用すると、他のガイドを指定することができます。

```
holder->attach(obj1, IlvHorizontal, 0, 1, 0, 1, 1);
holder->attach(obj2, IlvHorizontal, 0, 1, 0, 1, 2);
```

これにより、次のような結果になります。

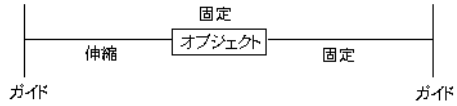


ガジェットの重みを設定する

attach メンバ関数の 3、4、5 番目のパラメータは、ガジェット前の重み、ガジェットの重み、そしてガジェットの後ろの重みをそれぞれ定義します。これらの重みは、同じ式を使用してガイドの重みと同じ方法で使用します。たとえば、以下の呼び出しの場合、

```
holder->attach(object, IlvHorizontal, 1, 0, 0);
```

次のアタッチメントが生成されます。



共通ガジェット・プロパティ

このセクションには、以下のトピックに関する情報が記載されています。

- *ガジェットの外観*
- *ガジェットにコールバックを関連付ける*
- *ガジェットのローカライズ*

- *ニーモニックをガジェット・ラベルへ関連付ける*
- *ツールチップの設定*
- *ガジェット・リソース*

ガジェットの外観

ガジェットの外観を以下によって定義することができます。

- *ガジェットをセンシティブとして設定する*
- *ガジェットの幅を設定する*
- *ガジェットを透明に設定する*
- *ガジェット・フレームの表示/非表示*

ガジェットをセンシティブとして設定する

ガジェットは、イベントに反応する場合、センシティブであるといいます。つまり、ユーザがこれをクリックするときにイベントが発生します。センシティブ・ガジェットの外観は、下図に示すように、非センシティブ・ガジェットの外観とは異なっています。

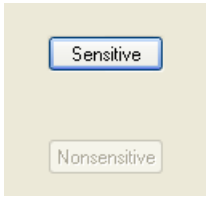


図9.1 *センシティブなガジェットと非センシティブなガジェット*

ガジェットのセンシティビティを変更するには、メンバ関数 `IlvGraphic::setSensitive` を使用します。

ガジェットがイベントに反応すべきかどうかを指定するために、メンバ関数 `IlvGadget::setActive` を、そのパラメータとして `IlFalse` とともに使用することもできます。

このメソッドと `setSensitivity` メンバ関数の違いは、ガジェットの描画が変更されないということ、およびガジェットの `handleEvent` メンバ関数が呼び出されないということです。

ガジェットの幅を設定する

幅を変更してガジェットの外観をカスタマイズすることができます。幅は、境界、修飾などを描くのに使用する陰影の大きさを定義します。ガジェットの幅を変更するには、メンバ関数 `IlvGadget::setThickness` を使用します。

> ***メモ:*** *使用されているルック・アンド・フィールによっては、ガジェットの幅さを変更すると外観に影響することがあります。特に、Microsoft® Windows® と Microsoft Windows 95 では、大部分のガジェットは幅を考慮しません。*

次の図は、Motif で異なる幅のある 2 つのボタンを表しています。

図 9.2 *異なる幅の値があるボタン*

ガジェットを透明に設定する

デフォルトでは、透明なメッセージ・ラベルを除いて、すべてのガジェットは不透明です。248 ページの *IlvMessageLabel の使用* を参照してください。メンバ関数 `IlvGadget::setTransparent` を、パラメータとして `IlTrue` とともに呼び出してガジェットを透明にすることができます。下の図のすべてのガジェットは透明です。透明に設定すると、背景テクスチャが見えるようになります。ここでは、Windows 95 のルック・アンド・フィールのガジェットが表示されています。

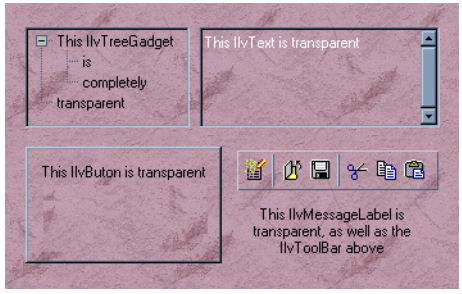
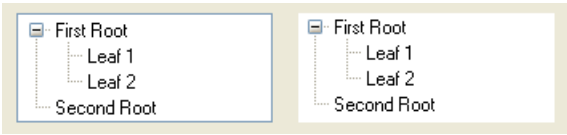


図 9.3 *透明ガジェット*

> ***メモ:*** *これらのガジェットを透明に設定すると、スクロール可能ガジェットのスクロール速度が遅くなることがあります。*

ガジェット・フレームの表示 / 非表示

ほとんどのガジェットでは、立体表示するときにフレームを使用します。ガジェットのフレームは、ガジェットが描画される最後の部分です。これは、`IlvGadget::drawFrame` メンバ関数を呼び出して描画されます。`IlvGadget::showFrame` メンバ関数を呼び出して、フレームの可視性を変更することができます。下の図は、フレーム付きとフレームなしの 2 つのガジェットを表しています。



ガジェットにコールバックを関連付ける

コールバック関数を、`IlvGraphic::addCallback` メソッドを使用してガジェットに関連付けることができます。コールバック関数は通常、ユーザがその上でアクションを行ったときに、その関連付けられているガジェットの `handleEvent` メンバ関数によって呼び出されます。コールバック関数のプロトタイプは、`IlvGraphicCallback` タイプによって定義されています。

ガジェットでは、複数のコールバック・タイプを、それぞれ特定のユーザ・アクションに対応させて定義することができます。各コールバック・タイプは、関連するアクションが実行されたときに呼び出されるコールバック関数のリストを保存しています。

定義済みコールバック・タイプ

ガジェットには、定義済みのコールバック・タイプがあります。

- Main - このコールバック・タイプは、ガジェットに付加されているメイン・アクションを実行するコールバックを定義します。
- Focus In - このコールバック・タイプは、ガジェットにフォーカスが与えられたときにアクションを実行するコールバックを定義します。このコールバック・タイプに対応するシンボルは、`IlvGadget::FocusInSymbol` によって取得されます。211 ページの *フォーカス管理* を参照してください。
- Focus Out - このコールバック・タイプは、ガジェットにフォーカスが与えられたときにアクションを実行するコールバックを定義します。このコールバック・タイプに対応するシンボルは、`IlvGadget::FocusOutSymbol` によって取得されます。211 ページの *フォーカス管理* を参照してください。

- Enter Gadget - このコールバック・タイプは、マウスがガジェットに入るときにアクションを実行するコールバックを定義します。このコールバック・タイプに対応するシンボルは、`IlvGadget::EnterGadgetSymbol` によって取得されます。210 ページの*ガジェット・ホルダ・イベント*を参照してください。
- Leave Gadget - このコールバック・タイプは、マウスがガジェットを離れるときにアクションを実行するコールバックを定義します。このコールバック・タイプに対応するシンボルは、`IlvGadget::LeaveGadgetSymbol` によって取得されます。210 ページの*ガジェット・ホルダ・イベント*を参照してください。

たとえば、Focus In コールバックをガジェットに追加したい場合は、次のようにコーディングします。

```
gadget->addCallback(IlvGadget::FocusInSymbol(), callback);
```

ここで、`callback` は次のように宣言されています。

```
void callback(IlvGraphic* g, IlAny arg) { ... }
```

これらの一般定義済みコールバック・タイプに加えて、各ガジェット・タイプには特殊定義済みコールバック・タイプが付加されています。詳細は、各ガジェットの説明のセクションを参照してください。

ガジェットのローカライズ

テキストを含むガジェットはローカライズできます。ガジェットのローカライズとは、最終アプリケーションで使用される言語にテキストを翻訳することです。このプロパティにより、現在の言語を動的に簡単に変更できる多言語アプリケーションを作成することができます。

IBM® ILOG® Views では、メッセージ・データベースをファイルとして作成し、最終アプリケーションで表示されるすべてのテキストを翻訳とともに保存することができます。メッセージ・データベース・ファイルは、`.dbm` 拡張子を持ちます。IBM ILOG Views *Foundation ユーザ・マニュアル*の「国際化」にある IlvMessageDatabase を参照してください。

最終アプリケーションで使用される言語に応じて動的にガジェットのテキストを変更させるには、ハードコード化する代わりにテキストを保存するメッセージ・データベースへの参照を提供しなくてはなりません。

次のメッセージ・データベースを作成したとしましょう。

```
Message: &MenuPrinterSetup
en_US: Printer Setup
fr_Fr: Configuration imprimante
```

次のように、`&MenuPrinterSetup` によって定義される文字列をガジェット・ラベルに割り当てることができます。

```
gadget->setLabel("&MenuPrinterSetup");
```

getLabel メンバ関数を呼び出すと、文字列 &MenuPrinterSetup を返し、getMessage メンバ関数を呼び出すと、現在の言語への翻訳を提供します (たとえば、英語の「Print Setup」およびフランス語の「Configuration imprimante」)。

ニーモニックをガジェット・ラベルへ関連付ける

ガジェットをニーモニック文字に関連付けることができます。ニーモニックとは、ガジェットをアクティブにするためにキーボート・ショートカットとして使用する、ガジェット・ラベルの下線付きの文字です。

ニーモニックをガジェット・ラベルに関連付けるには、カレット (^) をニーモニックとして使用したい文字の前にタイプします。

```
gadget->setLabel("^File");
char mnemo = gadget->getMnemonic();
```

> ***メモ:*** *ガジェット・ラベル内にカレット (^) をタイプするには、次のエスケープ・シーケンスを使用します。*\^.

使用する言語に応じて、異なるニーモニックにすることができます。たとえば、言語データベース (.dbm ファイル) に次のような特殊エントリを設定できます。

```
Message: &MenuPrinterSetup
en_US: Printer ^Setup
en_Fr: ^Configuration imprimante
```

英語ではニーモニック文字として「S」が使用されているところで、フランス語では「C」を使用します。

ツールチップの設定

ガジェットをツールチップに関連付けることができます。ツールチップは、ガジェットにユーザがマウス・ポインタを置くと表示される短い説明テキストです。デフォルトでは、ツールチップはガジェット・ホルダによりサポートされています。ツールチップをガジェット・ホルダの外で使用したい場合、クラス IlvToolTipHandler を使用します。

このセクションでは、以下のトピックを取り上げます。

- *ツールチップの作成*
- *ガジェットへツールチップを付加する*
- *ツールチップを有効または無効にする*
- *特殊ツールチップ*

ツールチップの作成

ツールチップは、IlvToolTip クラスのインスタンスです。ツールチップを作成するには、次を呼び出します。

```
IlvToolTip* tooltip = new IlvToolTip("This is a test");
```

ガジェットへツールチップを付加する

IlvToolTip は IlvNamedProperty クラスのサブクラスであるため、ツールチップをメンバ関数 IlvGraphic::setNamedProperty を使用してガジェットに付加できます。

```
gadget->setNamedProperty(new IlvTooltip("This is a test"));
```

ツールチップを有効または無効にする

静的メンバ関数 IlvToolTip::Enable を使用して、アプリケーション・レベルでツールチップ管理を有効、または無効にすることができます。

特殊ツールチップ

いくつかのガジェットは、IlvToolBar、IlvTreeGadget、IlvMatrix、IlvStringList、および IlvPopupMenu など、独特のツールチップ機構を持っています。

詳細は、これらガジェットのセクションを参照してください。

ガジェット・リソース

システム・リソース機構により、実行時にグラフィック・オブジェクトをカスタマイズできます。オブジェクト・リソースは、メンバ関数 addObject を使用してオブジェクトがガジェット・ホルダに追加されたときに変換されます。

1 つのリソース設定を個別オブジェクトに、あるいはオブジェクト・クラスに適用させることができます。スコープを個別の*保存*オブジェクトまたは*保存*クラスに制限することもできます。ここで、*保存*は IlvGadgetContainer または IlvGadgetManager を表します。

各グラフィック・オブジェクト・クラスは、リソースとして重要なパラメータのセットを定義することができます。

定義済みオブジェクト・リソース

IlvGraphic は、次のオブジェクト・リソースを実装します。

リソース名	説明	値
x	X 位置	整数文字列
y	Y 位置	整数文字列

リソース名	説明	値
w または width	水平サイズ	整数文字列
h または height	垂直サイズ	整数文字列

IlvSimpleGraphic は次のリソースを実装します。

リソース名	説明	値
background	パレット背景色	色名
foreground	パレット前景色	色名
font	パレット・フォント	フォント名
pattern	パレット・パターン	パターン名
colorPattern	パレット色パターン	パターン名
lineStyle	パレット線の種類	線の種類名
lineWidth	パレット線の幅	整数文字列
fillStyle	パレット塗りつぶし種類	FillPattern FillMaskPattern FillColorPattern
arcMode	パレット円弧モード	ArcPie ArcChord
fillRule	パレット塗りつぶしルール	EvenOddRule WindingRule
alpha	パレット・アルファ値	整数文字列
antialiasingMode	パレット・アンチエイリアシング・モード	DefaultAntialiasing UseAntialiasing NoAntialiasing

警告: IlvSimpleGraphic *リソースは、デフォルト・パレットを持つグラフィック・オブジェクトにのみ適用されます。*

オブジェクト・リソースの設定

グラフィック・リソースの*表示システム・リソース: getResource* に説明されているのと同じ方法でこれらのリソースに値を定義します。構文がシステム依存であっても、リソース設定のグローバル構造は同じです。デフォルトの値は key value です。リソース設定の左部分の key は、*表示システム・リソース: getResource* に説

明されているリソース設定よりも複雑です。そのため、この設定の対象となるオブジェクトは簡単に識別できます。キー設定は次のように定義されます。

```
Program.Storage.GraphicObject.Resource
```

次は、これらの 4 つのフィールドの記述です。

- Program は、アプリケーション名あるいは文字列 IlogViews のいずれかです。
- Storage は、ガジェット・コンテナまたはガジェット・マネージャの名前、あるいは文字列 IlvGadgetContainer または IlvGadgetManager のいずれかになります。
- GraphicObject は、グラフィック・オブジェクトの名前 (IlvGraphic::getName によって返される名前) であるか、グラフィック・オブジェクト・クラスの名前 (IlvGraphic::className によって返される名前) になります。
- Resource は、このリソースを定義するクラスのドキュメントに表示されるときの、オブジェクト・リソースの名前です。

フィールド Program、Storage および GraphicObject は、ワイルド・カード '*' で置き換えることができます。

オブジェクト、ガジェット・コンテナ、およびガジェット・マネージャの名前を文書化しておくことはアプリケーション開発者の責任です。

このクラスによって定義されるリソースの名前を文書化しておくことはグラフィック・オブジェクト設計者の責任です。

例: オブジェクト・リソースの指定

ここでは、偶数 / 奇数ルールを使用して、IlvPolygon クラスのすべてのインスタンスを赤くし、塗りつぶすように指定する方法を紹介します。

- X ウィンドウで、以下を ~/.Xdefaults ファイルに追加します。
  - IlogViews*IlvPolygon.foreground: red
  - IlogViews*IlvPolygon.fillRule: EvenOddRule
- Microsoft Windows では、以下を .INI ファイルに追加します。
  Section [IlogViews] or [<ApplicationName>]:
  - *IlvPolygon.foreground=red
  - *IlvPolygon.fillRule=EvenOddRule

優先度と競合

複数のリソース設定が同じターゲットに適用される場合、IBM ILOG Views は、もっとも精密な設定に優先度を与えます。

- つまり、文字列は '*' に対して優先されます。

- さらに、アプリケーション名は IlogViews に対して優先されます。
- ガジェット・コンテナあるいはガジェット・マネージャ名は IlvGadgetContainer あるいは IlvGadgetManager に対して優先されます。
- オブジェクト名はオブジェクト・クラスに対して優先されます。

これらの優先度にもかかわらず競合が残る場合、その結果は定義されません。

例: リソースの優先度

X ウィンドウ構文の使用

```
1. IlogViews.*.*.foreground: blue
2. myApp.*.*.foreground: green
3. myApp.*.IlvButton.foreground: red
4. myApp.myPanel.*.foreground: yellow
5. myApp.myPanel.myButton.foreground: cyan
```

ライン 5 はその他すべてに対して優先されます。
ライン 4 はライン 1 および 2 に対して優先されます。

ライン 3 および 4 の間に解決されていない競合があります。myPanel と呼ばれるガジェット・コンテナの IlvButton の色は予測不可能です。

新規リソースの追加

新しいリソースをグラフィック・オブジェクトに追加する場合、仮想メンバ関数 IlvGraphic::applyResources をオーバーロードしなくてはなりません。このメソッドは、オブジェクト・リソースを読み込み、IlvGadgetContainer および IlvGadgetManager の addObject メンバ関数によって呼び出されます。

このメソッドをオーバーロードするとき、サブクラスはスーパークラスの applyResources メソッドを呼び出し、その後 2 番目の IlvDisplay::getResource メンバ関数を使用して定義した新しいリソースに可能な値を取得します。

```
const char* getResource(const char* resourceName,
                        const char* objectName,
                        const char* objectClassName,
                        const char* storageName = 0,
                        const char* storageClassName = 0) const;
```

例: リソースの追加

```
// Assuming class MyObjectClass: public MyObjectSuperClass
// definig a method setLabel.
// The following defines a resource called "labelString".

void MyObjectClass::applyResources(const char* storageName,
                                   const char* storageClassName,
                                   const char* objectName,
                                   const char* objectClassName,
                                   IlvDisplay* display)
{
    if(!display)
       display = getDisplay();
    MyObjectSuperClass::applyResources(storageName,
                                       storageClassName,
                                       objectName,
                                       objectClassName,
                                       display);
    const char* resource = display->getResource("labelString",
                                                objectName,
                                                objectClassName,
                                                storageName,
                                                storageClassName);
    if (resource)
       setLabel(resource);
}
```

ガジェットのルック・アンド・フィール

いくつかのガジェットの概観および振る舞いは、使用されるグラフィック環境に一致するよう変更することができます。現在、IBM® ILOG® Views では、4 種類のグラフィック環境、Motif®、Microsoft® Windows® 3.11、Microsoft Windows 95、Microsoft Windows XP に対応しています。どのルック・アンド・フィールをガジェットが使用するかを決定することができます。既存項目から継承させてカスタム・ルック・アンド・フィールを定義することも、独自のルック・アンド・フィールをデザインすることもできます。

このセクションには、以下のトピックに関する情報が記載されています。

- *デフォルト・ルック・アンド・フィールの使用*
- *複数のルック・アンド・フィールを使用する*
- *ルック・アンド・フィールを動的にロードする*
- *ルック・アンド・フィールを動的に変更する*
- *Windows XP ルック・アンド・フィールの使用*

デフォルト・ルック・アンド・フィールの使用

デフォルトでは、1 つのスタイルが IBM® ILOG® Views プログラムに用意されています。これはコンピュータ・システムの標準スタイルです。

- UNIX® の Motif® スタイル
- Windows 3.x および Microsoft Windows NT 3.x 上の Microsoft® Windows® 3.11 スタイル
- Windows 95 Windows NT 4, and Windows 2000 上の Microsoft Windows 95 スタイル
- Windows XP 上の Microsoft Windows XP スタイル

> ***メモ:*** *このデフォルト設定を* `ILVLOOK` *環境変数、あるいは LOOK リソースを使用してオーバーライドすることができます。この場合、アプリケーションで指定したルックへのアクセスを提供しているか確認してください。提供していない場合は、使用されません。227 ページの複数のルック・アンド・フィールを使用するおよび228 ページのルック・アンド・フィールを動的にロードするを参照してください。*

> ***メモ:*** *Microsoft Windows XP スタイルは、Microsoft Windows XP を実行しているコンピュータ上でのみ利用できます。Microsoft Windows XP プラットフォームに構築された IBM ILOG Views アプリケーションは、Microsoft Windows の古いバージョン (Windows 2000、NT など) では実行できないことがあります。詳細は、230 ページの Windows XP ルック・アンド・フィールの使用を参照してください。*

アプリケーションを構築するプラットフォームに応じて、対応するルック・アンド・フィール・ライブラリにリンクさせなくてはなりません。次の表は、利用可能なライブラリをまとめたものです。

表 9.1 *Windows プラットフォーム用ルック・ライブラリ*

ルック	標準ガジェット・ライブラリ	高度なガジェット・ライブラリ
Motif	`ilvmlook.lib`	`ilvamlook.lib`
Windows 3.11	`ilvwlook.lib`	`ilvawlook.lib`

表 9.1 Windows プラットフォーム用ルック・ライブラリ

ルック	標準ガジェット・ライブラリ	高度なガジェット・ライブラリ
Windows 95	ilvw95look.lib、ilvwlook.lib	ilvaw95look.lib、ilvawlook.lib
Windows XP	ilvwxplook.lib、ilvw95look.lib、ilvwlook.lib, uxtheme.lib	ilvawxplook.lib、ilvaw95look.lib、ilvawlook.lib

uxtheme.lib は、Microsoft ライブラリであることに注意してください。このライブラリがコンピュータ上にない場合、Microsoft Platform SDK で入手できます。SDK は次のサイトで取得できます。http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en

表 9.2 UNIX プラットフォーム用ルック・ライブラリ

ルック	標準ガジェット・ライブラリ	高度なガジェット・ライブラリ
Motif	libilvmlook	libilvamlook
Windows 3.11	libilvwlook	libilvawlook
Windows 95	libilvw95look、libilvwlook	libilvaw95look、libilvawlook

> ***メモ:*** *Windows XP ルックは、上記の表では言及されていません。このルックは、Microsoft Windows XP オペレーティング・システムを実行しているプラットフォームでのみ利用できます。詳細は、230 ページの Windows XP ルック・アンド・フィールの使用を参照してください。*

たとえば、IBM ILOG Views 標準ガジェット・ライブラリ (詳細は 205 ページの *Gadgets ライブラリ* を参照してください) を Microsoft Windows 95 上で使用してプログラムを構築しているのであれば、ilvw95look.libÅAilvwlook.lib とリンクする必要があります。

同様に、UNIX 上で IBM ILOG Views 高度ガジェット・ライブラリを使用してプログラムを構築している場合は、libilvmlook および libilvamlook とリンクしなくてはなりません。

しかし、共有ライブラリを使用している場合は、ルック・アンド・フィール・ライブラリとのリンクを避けることができます。228 ページの*ルック・アンド・フィールを動的にロードする* を参照してください。

> ***メモ:*** *Windows プラットフォームでは、静的ライブラリを使用しているときでもルック・アンド・フィール・ライブラリとのリンクは必要ありません。アプリケーションが必要とするライブラリは、そのヘッダー・ファイルにある特定指示によって自動的にリンクされます。*

複数のルック・アンド・フィールを使用する

プログラムで複数のスタイルを使用したい場合は、どの追加スタイルを使用するかを指示するためにコンパイラ・オプション、あるいは `include` ファイルを追加する必要があります。

- コンパイラ・フラグで、使用するスタイルにシンボル名を定義します。
  - Windows® アプリケーション上の Motif® ルック用 `ILVMOTIFLOOK`
  - X Window アプリケーション上の Microsoft® Windows® 3.11 ルック用 `ILVWINDOWSLOOK`
  - X Window アプリケーション上の Microsoft Windows 95 ルック用 `ILVWINDOWS95LOOK`
  - Windows アプリケーション上の Microsoft Windows XP 用 `ILVWINDOWSXPLOOK`
- 実装ファイルで、他の `#include` 指示の前にスタイル・ヘッダー・ファイルを含めます。ガジェット・クラスを宣言するヘッダー・ファイルがこれらのヘッダー・ファイルの 1 つより前に置かれている場合、対応する仮想スタイルがプログラムに読み込まれないようにしてください。読み込まれると、アプリケーションのスタイルを変更するときクラッシュする可能性があります。

  以下は、含まれるファイルです。

  - `<ilviews/motif.h>`

    Microsoft Windows アプリケーション用 Motif ルックへのアクセスを追加する。

  - `<ilviews/windows.h>`

    X Window アプリケーション用 Microsoft Windows 3.11 ルックへのアクセスを追加する。

  - `<ilviews/win95.h>`

    X Window アプリケーション用 Microsoft Windows 95 ルックへのアクセスを追加する。

- `<ilviews/winxp.h>`

  Microsoft Windows XP ルックへのアクセスを追加する。このルックは、Windows XP プラットフォームでのみ使用可能です。

アプリケーションを、使用するルックに対応するルック・アンド・フィール・ガジェット・ライブラリにリンクさせる必要もあります。225 ページの ï 9.1 および 226 ページの ï 9.2 を参照してください。

しかし、共有ライブラリを使用している場合はルック・アンド・フィール・ライブラリとのリンクを避けることができます。228 ページの*ルック・アンド・フィールを動的にロードする* を参照してください。

> ***メモ:*** *デフォルト・ルック・アンド・フィールを使用したくない場合は、*`ILVNODEFAULTLOOK` *フラグでコンパイルしなくてはなりません。このフラグでコンパイルすると、デフォルト・ルック・アンド・フィール・ライブラリとリンクされません。*

ルック・アンド・フィールを動的にロードする

ルック・アンド・フィールを動的にロードする場合、アプリケーションがどのルック・アンド・フィールを使用するのかを配慮する必要はありません。アプリケーションが何に依存するかに応じて、ルックは実行時にロードされるからです。つまり、アプリケーションをルック・アンド・フィール特殊ライブラリのいずれにもリンクさせる必要がないということです。

ロードのメカニズム

- ロードには、共有ライブラリ (Microsoft® Windows® の場合は DLL) を使用します。これがないとモジュールの動的ロードはできません。
- ルックの特殊ヘッダー・ファイルを含める必要はありません。含めると対応するライブラリにリンクさせなくてはならなくなります。
- 定義された ILVNODEFAULTLOOK シンボルでコンパイルします。コンパイルしないと、デフォルト・ルック・ライブラリとリンクさせる必要が生じます。

> ***メモ:*** *ルック・アンド・フィールの動的ロードを使用することを強くお勧めします。これにより、アプリケーションを実行時に使用されるスタイルから完全に独立させることができます。*

ルック・アンド・フィールを動的に変更する

グラフィック・オブジェクトの外観は異なるレベルで管理されます。

- オブジェクト・レベル

  メソッド `IlvGraphic::getLookFeelHandler()` は、オブジェクトにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルトの定義では、オブジェクト・ホルダによって定義されたルック・アンド・フィール・ハンドラを使用します。

- ホルダ・レベル

  メソッド `IlvGraphicHolder::getLookFeelHandler()` は、ホルダにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルトの定義では、ホルダ表示インスタンスによって定義されたルック・アンド・フィール・ハンドラを使用します。

- 表示レベル

  メソッド `IlvDisplay::getLookFeelHandler()` は、表示インスタンスにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルト値は、アプリケーションが作成されたプラットフォームによって定義されます。詳細については、225 ページの *デフォルト・ルック・アンド・フィールの使用* を参照してください。

コンテナ全体またはアプリケーション全体の単一ガジェットのルックを、それぞれメソッド `IlvGadget::setLookFeelHandler`、`IlvGadgetContainer::setLookFeelHandler`、および `IlvDisplay::setLookFeelHandler` を使用して変更することができます。

ルック・アンド・フィール・ハンドラは、`IlvLookFeelHandler` クラスのサブクラスです。各ハンドラは、これを識別する固有の名前を持っています。以下は、4 つの定義済みルック・アンド・フィール・スタイルの名前です。

Motif	motif
Windows 3.11	windows
Windows 95	win95
Windows XP	winxp

ルック・アンド・フィール・ハンドラは表示インスタンスに関連付けられています。ルック・アンド・フィール・ハンドラのインスタンスを取得するには、名前をパラメータとして取得するメソッド `IlvDisplay::getLookFeelHandler` を使用します。

```
IlvLookFeelHandler* lfh = display->getLookFeelHandler(IlGetSymbol("motif"));
```

必要なルック・アンド・フィールが既にこの表示に作成されている場合、これが返されます。そうでない場合は新規作成されます。ルック・アンド・フィールが作成できない場合、メソッドは `0` を返します。

表示全体のルック・アンド・フィールを変更する

上記で説明したとおり、メソッド IlvDisplay::setLookFeelHandler を、表示全体のルック・アンド・フィールを変更するために使用します。ただし、定義済みのルックを記述するために enum (IlvLookStyle) を定義するには、ILOG Views 4.0 API を使用する必要があります。この enum は、次のように使用することもできます。

```
#include <ilviews/ilv.h>

typedef enum IlvLookStyle {
                IlvOtherLook,
                IlvMotifLook,
                IlvWindowsLook,
                IlvWindows95Look,
                IlvWindowsXPLook
};
```

IlvDisplay クラスの次のメンバ関数により、この enum を使用して look 表示リソースの設定を操作することができます。

- IlvDisplay::getCurrentLook は、この表示インスタンスに使用される現在のスタイル識別子を返します。表示の現在のルック・アンド・フィールが定義済みのルック・アンド・フィールではない場合、IlvOtherLook が返されます。
- IlvDisplay::setCurrentLook は、この表示インスタンスによって使用されるスタイル識別子を style に設定します。

次のメソッドを使用して、表示のルック・アンド・フィールの変更の通知を受けることができます。

- IlvDisplay::addChangeLookCallback で、スタイルが動的に変更されたときに呼び出されるユーザ定義関数を追加することができます。
- IlvDisplay::removeChangeLookCallback によって、スタイルが動的に変更されたときに呼び出されるユーザ定義関数を削除することができます。

Windows XP ルック・アンド・フィールの使用

Microsoft® Windows® 3.11、Microsoft Windows 95、および Motif スタイルは、プラットフォームから独立していますが、Microsoft Windows XP スタイルは、コンポーネントの描画にシステム (Microsoft Windows XP) を使用します。つまり、このスタイルは、Microsoft Windows XP を実行するプラットフォーム上でのみ使用する

ことができます。また、新規テーマが Microsoft Windows XP で利用可能になると、IBM ILOG Views アプリケーションでも利用できるようになります。

> ***メモ:*** *他の Microsoft Windows プラットフォーム上で、Microsoft Windows XP スタイルを使用するアプリケーションを構築することができます。この場合、Microsoft Platform SDK をインストールする必要があるかもしれません。SDK は次のサイトで取得できます。*`http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en`

IBM ILOG Views ダイナミック・リンク・ライブラリ (DLL) の使用

IBM ILOG Views DLL (`dll_md` あるいは `dll_mda`) を使用すると、Microsoft Windows XP スタイルを必要なときに動的にロードできます。つまり、どの Microsoft Windows プラットフォームでも実行できる IBM ILOG Views アプリケーションを構築できます。アプリケーションは、必要に応じて Microsoft Windows XP スタイルをロードします。詳細は、228 ページの*ルック・アンド・フィールを動的にロードする*を参照してください。

IBM ILOG Views 静的ライブラリの使用

IBM ILOG Views 静的ライブラリ (`stat_st`、`stat_sta`、`stat_md`、`stat_mda`、`stat_mt`、または `stat_mta`) を使用するときは、次の点に注意してください。

- IBM ILOG Views アプリケーションを Microsoft Windows XP プラットフォームのデフォルト・スタイルを使用して構築している場合、アプリケーションをコンパイル中はフラグ `WINVER` を `0x501` に設定する必要があります。これ以外に設定すると、Microsoft Windows 95 スタイルのみが登録されます。225 ページの*デフォルト・ルック・アンド・フィールの使用*を参照してください。
- Microsoft Windows XP および IBM ILOG Views 静的ライブラリを使用する IBM ILOG Views アプリケーションは Microsoft Windows XP を実行しているプラットフォームでのみ実行することができます。IBM ILOG Views アプリケーションを Microsoft Windows XP プラットフォーム上でコンパイルし、このアプリケーションをあらゆる Windows プラットフォーム上で実行する場合、`WINVER` フラグを定義しない、あるいは `ILVNODEFAULTLOOK` フラグを定義するかのいずれかを行うことができます。後者の場合、アプリケーションを XP ライブラリ以外のルック・アンド・フィール・ライブラリにリンクさせる必要があります。詳細は、227 ページの*複数のルック・アンド・フィールを使用する*を参照してください。
- モジュールの動的ロードは、静的ライブラリの使用中は無効になるので、アプリケーションを正しいライブラリにリンクする必要があります。225 ページの*Windows プラットフォーム用ルック・ライブラリ*を参照してください。

章

10

ダイアログ

Gadgets ライブラリは、ダイアログ・ボックス作成に使用できる `IlvDialog` クラスを提供します。このクラスは `IlvGadgetContainer` から継承しているため、そのインスタンスはガジェットを含むことができます。`IlvDialog` は、標準ダイアログ・ボックスを実装するサブクラスを多数備えています。

この章では、以下のトピックを取り上げます。

- *定義済みダイアログ・ボックス*
- *独自のダイアログ・ボックスの作成*
- *ダイアログ・ボックスの表示 / 非表示*
- *デフォルト・ボタンの設定*

次の図は、ダイアログ・クラス階層を示しています。

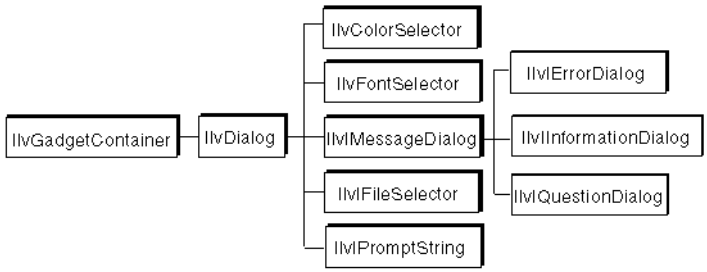


図10.1 *ダイアログ・ボックスのクラス階層*

定義済みダイアログ・ボックス

Gadgets ライブラリは標準ダイアログ・ボックスを定義するために、次のクラスを提供しています。

- *IlvIMessageDialog*
- *IlvIQuestionDialog*
- *IlvIErrorDialog*
- *IlvIWarner*
- *IlvIInformationDialog*
- *IlvIFileSelector*
- *IlvIPromptString*
- *IlvFontSelector*
- *IlvColorSelector*

IlvIMessageDialog

メッセージ・ダイアログ・ボックス (`IlvIMessageDialog` クラス) には、メッセージ・テキスト・フィールド、ビットマップおよび 2 つのボタンがあります。

図 10.2 *メッセージ・ダイアログ・ボックス*

> ***メモ:*** *デフォルトでは、ダイアログはビットマップを含みません。そのため、ビットマップを提供する必要があります。*

IlvIQuestionDialog

質問ダイアログ・ボックス (`IlvIQuestionDialog` クラス) は質問を表示し、「はい」あるいは「いいえ」で返答します。

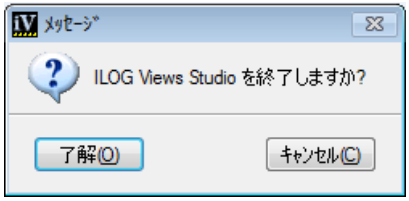


図 10.3 *質問ダイアログ・ボックス*

質問ダイアログ・ボックスのコード・サンプルを次に示します。

```
{
    IlvIQuestionDialog dlg(getDisplay(), msg, 0,
                           IlvDialogOkCancel, transientFor);
    dlg.setString("dialog message");
    if (dlg.get()) ...
}
```

このコードは、使用後に破壊される `dlg` という名前のダイアログ・ボックスを作成します。これは、`transientFor` で指定されるビューの一時的なダイアログ・ボックスです。これには [OK] および [取消し] の 2 つのボタンがあります。メソッドは、`get` ダイアログ・ボックスを開き、その結果を待ちます。このメソッドは、[OK] が選択された場合には `IlTrue` を返し、それ以外の場合には `IlFalse` を返します。

IlvIErrorDialog

エラー・ダイアログ・ボックス (`IlvIErrorDialog` クラス) は、エラー・メッセージを表示します。

図 10.4　エラー・ダイアログ・ボックス

IlvIWarner

警告ダイアログ・ボックス (`IlvIWarner` クラス) は、警告メッセージを表示します。

図 10.5　警告ダイアログ・ボックス

IlvIInformationDialog

情報ダイアログ・ボックス (`IlvIInformationDialog` クラス) は、情報メッセージを表示します。

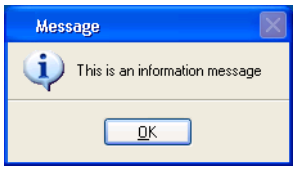


図 10.6　情報ダイアログ・ボックス

IlvIFileSelector

ファイルのセレクタ (`IlvIFileSelector` クラス) は、ユーザにファイル名を選択するよう要求します。

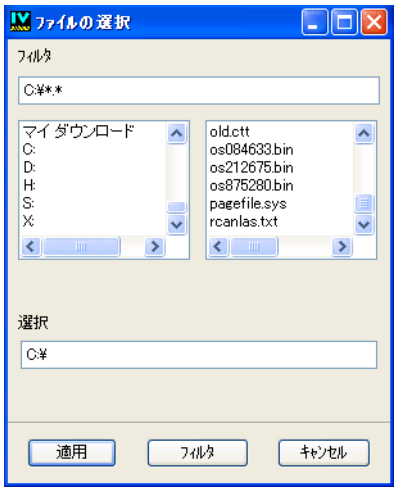


図 10.7 *ファイルのセレクタ*

次は、ファイルのセレクタの使用例です。

```
filesel = new IlvIFileSelector(display, 0, "*.cc");
filesel->setName("File Chooser");
filename = filesel->get();
if (filename && filename[0] && IlvFileExists(filename)) ...
```

> ***メモ:*** *作業中のプラットフォーム専用のファイルのセレクタを使用したい場合は、*`IlvFileSelector` *あるいは* `IlvFileBrowser` *クラスを使用します。*

IlvIPromptString

入力文字列 (`IlvIPromptString` クラス) は、ユーザに文字列を選択あるいは入力するように要求します。

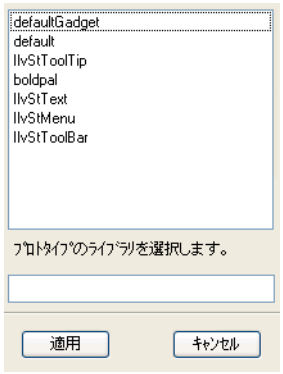


図 10.8　プロンプト文字列

IlvFontSelector

フォントのセレクタ (`IlvFontSelector` クラス) は、ユーザにフォントを選択するよう要求します。

図 10.9　フォントのセレクタ

IlvColorSelector

色のセレクタ (`IlvColorSelector` クラス) は、ユーザに色を選択するよう要求します。

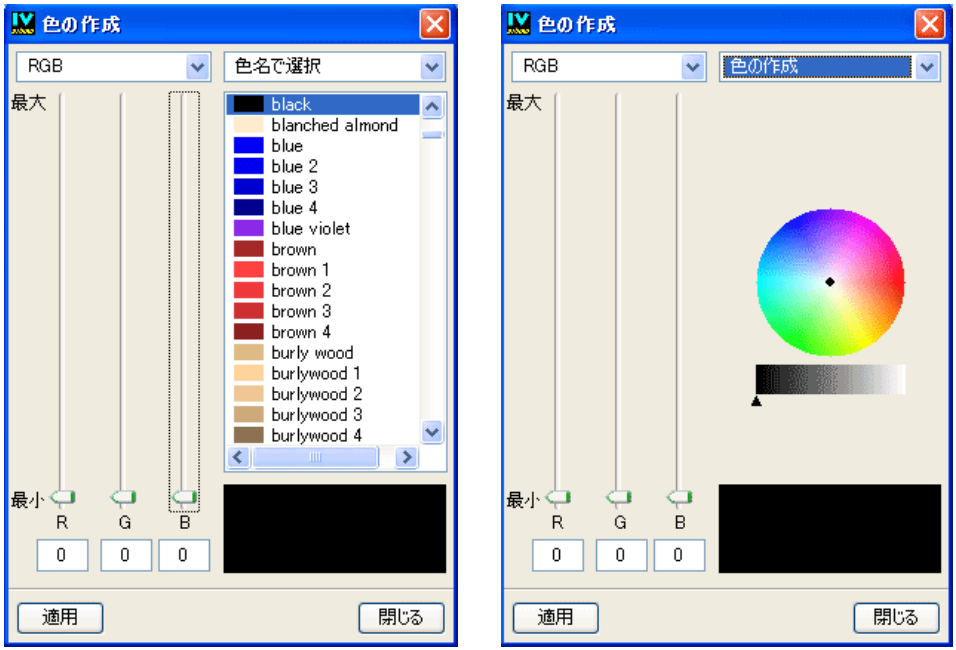


図 10.10 *色名 (左) およびカラー・ホイール (左) のある色のセレクタ*

独自のダイアログ・ボックスの作成

独自のダイアログ・ボックスを作成するには、次の手順で行います。

1. パネルの視覚的表現をデザインします。

   この手順には、どのガジェットを使用するか、どのようにフォーカスを管理するか、あるいはダイアログのサイズが変更された時にガジェットはどのように振る舞うかの選択など、いくつかの局面が含まれます。この段階は、IBM® ILOG® Views Studio を使用して行うことができます。

2. ダイアログ・ボックスにパネルを表示させる。

   IBM ILOG Views Studio の生成されたコード (詳細は、4 章*生成されたコードの使用*を参照してください)、あるいは直接 `IlvDialog` クラスを使用することができます。`IlvDialog` のコンストラクタは、ファイル名をパラメータとして渡す機能を提供します。

`IlvDialog` クラスには、既に登録されたコールバック関数が 2 つあります。

◆ 適用 : コールバック関数は仮想 `IlvDialog::apply` メソッドを呼び出します。

- 取消し：コールバック関数は仮想 IlvDialog::cancel メソッドを呼び出します。

たとえば、ダイアログ・ボックスにボタンを 2 つ設定し、1 つは「適用」に設定するコールバック、もう 1 つは「取消し」に設定するコールバック関数にすることができます。

ダイアログ・ボックスの表示 / 非表示

IlvDialog は、ダイアログ・ボックスを管理するメソッドを提供します。

ユーザが [OK] あるいは [取消し] (適用あるいは取消しのコールバックを呼び出す) をクリックするまで待機する場合は、IlvDialog メソッド wait を使用します。このメソッドはモーダル・ダイアログ・ボックスを表示します。wasCanceled メソッドは、ユーザが取消しをクリックしたかどうかを通知します。

```
dialog.wait();
if (!dialog.wasCanceled())  {
  ...
 }
```

この場合は、show および hide メソッドを使用することもできます。標準ダイアログ・ボックスには特殊な独自のメソッドが備わっており、これらを表示し、値が返されるまで待機します。

デフォルト・ボタンの設定

ダイアログ・ボックスに、デフォルト・ボタンを設定することができます。デフォルト・ボタンは、ダイアログ・ボックスにキーボード・フォーカスがある時にユーザが Enter キーを押すと、アクティブになるボタンです。

デフォルト・ボタンは他から区別するために、特別な外観をしています。デフォルト・ボタンを設定するには、メンバ関数 setDefaultButton を使用します。

デフォルト・ボタンが定義されている場合、Enter キーを押すとこのボタンにのみ適用されます。場合によっては、この振る舞いを無効にしたいこともあるでしょう。たとえば、マトリックスを編集している時、変更を有効にするために Enter キーを使用する場合があります。この振る舞いを変更するには、メンバ関数 IlvGadget::usesDefaultButtonKeys を使用します。

章

11

共通ガジェットの使用

この章では、Gadgets ライブラリで提供されているさまざまなガジェットの使用方法について説明します。以下のトピックから構成されています。

- *IlvArrowButton の使用*
- *IlvButton の使用*
- *IlvComboBox および IlvScrolledComboBox の使用*
- *IlvDateField の使用*
- *IlvFrame の使用*
- *IlvMessageLabel の使用*
- *IlvNotebook の使用*
- *IlvNumberField の使用*
- *IlvOptionMenu の使用*
- *IlvPasswordTextField の使用*
- *IlvScrollBar の使用*
- *IlvSlider の使用*
- *IlvSpinBox の使用*
- *IlvStringList の使用*

- *IlvText の使用*
- *IlvTextField の使用*
- *IlvToggle の使用*
- *IlvTreeGadget の使用*

IlvArrowButton の使用

クラス `IlvArrowButton` は、矢印を表示するボタンを定義します。矢印は上向き、下向き、右向き、左向きにすることができます。`IlvArrowButton` は、`IlvButton` のサブクラスです。

図 11.1 *矢印ボタン*

矢印の方向を `IlvArrowButton::setDirection` を使用して指定し、`IlvArrowButton::getDirection` で取得することができます。

241 ページの *IlvButton の使用* を参照してください。

IlvButton の使用

クラス `IlvButton` は、ユーザがクリックできる矩形領域を定義します。`IlvButton` は、`IlvMessageLabel` のサブクラスで、この周りに立体の矩形を追加します。

図 11.2 *ボタン*

ボタンの内側に表示されるラベルを、さまざまな設定で整列させることができ、さらにローカライズもできます。これらのプロパティの詳細については、248 ページの *IlvMessageLabel の使用*を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *ボタンにビットマップを表示する*
- *ボタン・フレームの表示*
- *ニーモニックをボタンに関連付ける*

◆ *イベント処理およびコールバック*

ボタンにビットマップを表示する

ボタンを使ってビットマップを表示することができます。表示できるビットマップは、センシティブ、非センシティブ、選択、強調の 4 種類です。

センシティブ・ビットマップに関する説明は、249 ページの *ビットマップをメッセージ・ラベルに関連付ける*を参照してください。

選択されたビットマップは、ボタンがクリックされた場合に表示されます。選択したビットマップを設定するには、`IlvButton::setSelectedBitmap` を使用します。

強調されたビットマップは、ボタン上にマウスがある場合に表示されます。強調されたビットマップを設定するには、`IlvButton::setHighlightedBitmap` を使用します。

ボタン・フレームの表示

`IlvButton::showFrame` メンバ関数を使用して、ボタンが強調されたときに周りのフレームを表示させるかどうかを指定することができます。次の図は、Windows® 95 ルック・アンド・フィールのボタンを示しています。

図 11.3 *非表示のフレーム付きボタン (左) および表示されたフレーム付きボタン (右)*

ニーモニックをボタンに関連付ける

ボタン・ラベルをニーモニック文字に関連付けることができます。ニーモニック文字に対応するキーを押すと、`IlvButton::activate` メンバ関数が呼び出されます。ボタンにキーボード・フォーカスがない場合、その文字のモディファイア・キー (PC では Alt、UNIX では Meta) を押します。

218 ページの *ガジェットのローカライズ*を参照してください。

イベント処理およびコールバック

ユーザがボタンをクリックするか、それに対応するニーモニック文字を押す、または Enter キーかスペース・バーを押すと、`IlvButton::activate` メンバ関数が呼び出されます。この仮想メンバ関数は、ボタンのメイン・コールバックを呼び出します。

210 ページの*イベントの処理*および 217 ページの*ガジェットにコールバックを関連付ける*を参照してください。

IlvComboBox および IlvScrolledComboBox の使用

クラス `IlvComboBox` は、テキスト・フィールドをユーザが選択できる定義済み文字列のリストを組み合わせます。`IlvScrolledComboBox` は、リストが一定数の選択肢を超えるとスクロールバーを表示します。`IlvComboBox` クラスは `IlvTextField` および `IlvListGadgetItemHolder` のサブクラスで、`IlvScrolledComboBox` クラスは `IlvComboBox` のサブクラスです。

図 11.4 *コンボ・ボックス*

274 ページの *IlvTextField の使用* を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *コンボ・ボックスを編集不可に設定する*
- *アイテムの設定および取得*
- *選択の変更および取得*
- *大型リストの使用*
- *表示アイテム数の設定*
- *コンボ・ボックスのローカライズ*
- *イベント処理およびコールバック*

コンボ・ボックスを編集不可に設定する

デフォルトで、コンボ・ボックスのテキスト・フィールド部分は編集することができます。つまり、新しいテキストを入力したり、クリップボードからテキストを貼り付けて変更できます。メンバ関数 `IlvTextField::setEditable` で、読み取り専用モードに切り替えることができます。このモードでは、メニューからのみ値を選択することができます。

読み取りモードに切り替えると、コンボ・ボックスの外観も変わります。そのため、コンボ・ボックスの編集モードを変更するときには `IlvGadget::reDraw` メンバ関数を呼び出す必要があります。

アイテムの設定および取得

IlvComboBox は、IlvListGadgetItemHolder のサブクラスであるため、コンボ・ボックスのアイテムを変更するにはこのクラスのメンバ関数を使用しなくてはなりません。

IlvListGadgetItemHolder を参照してください。

選択の変更および取得

IlvComboBox は IlvTextField のサブクラスであるため、コンボ・ボックスへのテキストの設定や取得は、このクラスのメンバ関数を使用する必要があります。

275 ページの *テキストの設定および取得* を参照してください。

インデックス番号と IlvComboBox::setSelected を使用してコンボ・ボックスの選択したアイテムを設定し、IlvComboBox::whichSelected を使用して取得することができます。

大型リストの使用

IlvComboBox と異なり、IlvScrolledComboBox によって表示されるアイテム・リストは、コンボ・ボックスの幅に対応する固定幅を持ちます。リストに大型アイテムが含まれる場合、テキスト・フィールドに適合しないため大型リストとなることがあります。この振る舞いを変更するには、メンバ関数 IlvScrolledComboBox::enableLargeList を使用します。

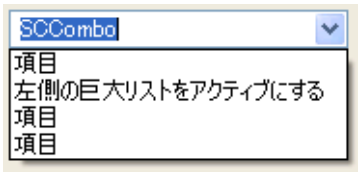


図 11.5 *大型リストが有効になったコンボ・ボックス*

表示アイテム数の設定

スクロール・ボックスでは、リストの表示アイテムの数を固定することができます。全アイテムが表示される場合、スクロールバーはありません。

表示アイテムの数を設定するには、メンバ関数 IlvScrolledComboBox::setVisibleItems を使用します。

コンボ・ボックスのローカライズ

コンボ・ボックスに表示されるテキストはローカライズできます。ローカライズできるのは、編集不可コンボ・ボックスのみです。

218 ページの*ガジェットのローカライズ*を参照してください。

イベント処理およびコールバック

マウスあるいは矢印キーを使用したり、新規テキストを入力した上で Enter キーを押して新規アイテムを選択すると、コンボ・ボックスの Main コールバックが呼び出されます。

Open List コールバックは、ユーザがコンボ・ボックス・リストを開くことで呼び出されます。Open List コールバック・メソッドを設定するには、`IlvComboBox::OpenListSymbol`() によって返されるコールバック記号 () を使用します。

217 ページの*ガジェットにコールバックを関連付ける* を参照してください。

IlvDateField の使用

クラス `IlvDateField` は、各種形式で日付を編集するために特殊なテキスト・フィールド・ガジェットを定義します。`IlvDateField` は、`IlvTextField` のサブクラスです。

図 11.6 *日付フィールド*

このセクションでは、以下のトピックを取り上げます。

- *日付の書式設定*
- *日付値の設定および取得*

日付の書式設定

`IlvDateField` は、日付をさまざまな形式で表示することができます。日付形式を指定するには、メンバ関数 `IlvDateField::setFormat` を使用します。

日付は、3 つの要素から構成されます。日、月、そして年です。これらの要素は区切り文字で分けられます。`setFormat` メンバ関数で、これらの要素およびどのセパレータを使用するかを指定できます。

デフォルトの値は次のとおりです。`12/31/1995` (`df_Month`, `df_Day`, `df_Year`)。

形式は以下のように定義されます。

```
enum format
    {
     df_day,           // 1
     df_Day,           // 01

     df_month,         // 3
     df_Month,         // 03
     df_month_text,    // March
     df_abbrev_month, // Mar

     df_year,          // 95
     df_Year           // 1995
   };
```

- df_day　先行ゼロを付けないで日を指定します。
- df_Day　必要な場合は先行ゼロを付けて日を指定します。
- df_month　先行ゼロを付けないで月を指定します。
- df_Month　必要な場合は先行ゼロを付けて月を指定します。
- df_month_text　月の名前を記述します。下記の月の名前が言語データベースに表示される場合は、そこから対応する名前を選択します。その他の場合は、& 記号のない月の名前が選択されます。

  月の名前は次のとおりです。&January, &February, &March, &April, &May, &June, &July, &August, &September, &October, &November, &December。
- df_abbrev_month　省略された月の名前を記述します。下記の月の名前が言語データベースに表示される場合は、そこから対応する名前を選択します。その他の場合は、& 記号のない省略月の名前が選択されます。

  省略された月の名前は次のとおりです。&january, &february, &march, &april, &may, &june, &july, &august, &september, &october, &november, &decemberÅB
- df_year　年の下 2 桁を記述します。
- df_Year　年を記述します。

フィールドに値が含まれている場合に書式を変更すると、この値に新しい書式が適用されます。

> ***メモ:*** setFormat *メンバ関数には日、月または年の書式を 1 つだけ渡すことができます。それ以外の場合には関数が* IlFalse *を返し、書式は変更されません。形式は埋め込まれた* enum *宣言で定義されています。次のように設定されています。*
> ```
> obj->setFormat(IlvDateField::df_day);
> ```

形式の例

April,2,1995 (`df_month_text, df_day, df_Year with separator ,`)

2/4/95 (`df_day, df_month, df_year with separator /`)

02/04/1995 (`df_Day, df_Month, df_Year with separator /`)

日付値の設定および取得

`IlvDateField` の日付を設定する、あるいは取得するには、メンバ関数 `IlvDateField::setValue` および `IlvDateField::getValue` を使用します。

2000 年問題の管理

2000 年問題を回避する正しい方法は、年を表すときに 4 桁の数字を使用することです。これは、`setFormat` メンバ関数を使用して `IlvDateField` クラスで実行できます。

しかし、日付の一部として年を表すときに 2 桁の値を使用しなくてはならない場合、`IlvDateField` API がこの問題を解決するいくつかのメソッドを用意しています。

1. `SetBaseCentury` で、4 桁の年の再計算に使用されるベース・センチュリを指定することができます。
2. `GetBaseCentury` は、`SetBaseCentury` で設定されたベース・センチュリを返します。デフォルト値は 1900 です。
3. `SetCenturyThreshold` を使うと、閾値を指定することができ、それを超えると、ベース・センチュリの値は、`GetBaseCentury()` によって返される値プラス 1 となります。
4. `GetCenturyThreshold` は、`SetCenturyThreshold` によって設定された値を返します。デフォルト値は 30 です。

たとえば、ベース・センチュリが 1900 で、閾値が 30 の場合、10 の値は 2010 に変換され、40 の場合は 1940 に変換されます。

IlvFrame の使用

クラス `IlvFrame` は、ラベルの周りの矩形を表示します。パネルのセクションでガジェットのグループ化に使用されます。`IlvFrame` は、クラス `IlvMessageLabel` から派生します。

図 11.7 *フレーム*

248 ページの *IlvMessageLabel の使用* を参照してください。

ニーモニックをフレームに関連付ける

フレーム・ラベルをニーモニック文字に関連付けることができます。ニーモニック文字に対応するキーとモディファイア・キー (PC では Alt、UNIX では Meta) を押すと、キーボード・フォーカスがフォーカスを持つことのできるフレームの最初のガジェットに与えられます。

218 ページの *ガジェットのローカライズ* および 211 ページの *フォーカス管理* を参照してください。

IlvMessageLabel の使用

クラス `IlvMessageLabel` は、メッセージを表示し、これにビットマップを付けることができます。メッセージは `IlvDisplay` の現在のインスタンスに関連付けることのできるデータベースに記録されます。

ビットマップに関連するメッセージの整列は、どこにでも設定できます。さらに、ブロック全体 (メッセージ + ビットマップ) の整列を、そのバウンディング・ボックスに対して `IlvCenter`、`IlvLeft`、または `IlvRight` に変更することも可能です。

図 11.8 は、`IlvMessageLabel` の例を示します。ピクチャに関連するメッセージの配列は、`IlvBottom` であり、`IlvMessageLabel` のグローバル配列は `IlvCenter` です。

図 11.8 *メッセージ・ラベル*

このセクションでは、以下のトピックを取り上げます。

- *ビットマップをメッセージ・ラベルに関連付ける*
- *メッセージ・ラベルを不透明にする*
- *メッセージ・ラベルのレイアウト*
- *メッセージ・ラベルのローカライズ*
- *ニーモニックの関連付け*

ビットマップをメッセージ・ラベルに関連付ける

ビットマップは、メンバ関数 `IlvMessageLabel::setBitmap` および `IlvMessageLabel::setInsensitiveBitmap` を使用してメッセージ・ラベルに関連付けることができます。

`setBitmap` は、メイン・ビットマップをメッセージ・ラベルに関連付けます。`setInsensitiveBitmap` は、メッセージ・ラベルが非センシティブに設定されたときに表示するビットマップを設定します。非センシティブ・ビットマップを提供しない場合は、メッセージ・ラベルが非センシティブに設定されると、自動的にデフォルトがセンシティブ・ビットマップから計算されます。

図 11.9 *センシティブおよび非センシティブ・ビットマップの付いたメッセージ・ラベル*

218 ページの *ガジェットのローカライズ* を参照してください。

メッセージ・ラベルを不透明にする

他のガジェットと異なり、メッセージ・ラベルはデフォルトでは透明です。これを不透明にするには、メンバ関数 `IlvGadget::setTransparent` を、そのパラメータとしての `IlFalse` とともに呼び出します。不透明メッセージ・ボックスのバウンディング・ボックスは、オブジェクト・パレットで設定された背景色で塗りつぶされています。

図 11.10 *不透明のメッセージ・ラベル*

216 ページの *ガジェットを透明に設定する* を参照してください。

メッセージ・ラベルのレイアウト

メッセージ・ラベルがラベルおよびビットマップの両方を表示するとき、メンバ関数 `IlvMessageLabel::setLabelPosition` を使用してビットマップに関連するラベルの位置を変更することができます。

たとえば、ラベルをビットマップの左に表示したい場合、以下を呼び出します。

```
message->setLabelPosition(IlvLeft);
```

ラベルとビットマップの間のスペースを 20 ピクセルに設定するには、以下を呼び出します。

```
message->setSpacing(20);
```

メッセージ・ラベルのバウンディング・ボックス内のグループ化されたラベルおよびビットマップを中央揃えにするには、以下を呼び出します。

```
message->setAlignment(IlvCenter);
```

図 11.11 *1 つのブロックのように整列されたラベルとビットマップ*

メッセージ・ラベルのローカライズ

メッセージ・ラベルはローカライズすることができます。

218 ページの *ガジェットのローカライズ* を参照してください。

ニーモニックの関連付け

メッセージ・ラベルにニーモニック文字を含ませることができます。ニーモニック文字に対応するキーとモディファイア・キー (PC では Alt、UNIX では Meta) を押す、あるいはメッセージ・ラベルをクリックすると、フォーカス・チェーンの次のガジェットにフォーカスが与えられます。

218 ページの *ガジェットのローカライズ* を参照してください。

IlvNotebook の使用

クラス IlvNotebook は、実際のノートブックをシミュレートします。ノートブックは、そのタブをクリックすることで選択して前面に移動させることができるページから構成されています。これらのページは、クラス IlvNotebookPage で実装されています。

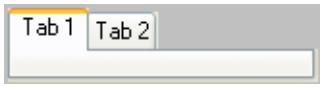


図 11.12 *ノートブックとページ*

このセクションでは、以下のトピックを取り上げます。

- *ノートブック・タブのカスタマイズ*
- *ノートブック・ページの処理*
- *イベント処理およびコールバック*

ノートブック・タブのカスタマイズ

ノートブックのタブは、さまざまな方法でカスタマイズできます。

- *タブの位置の設定*
- *タブの向きを設定する*
- *タブ・マージンの設定*
- *ページ・マージンの設定*

タブの位置の設定

ノートブックのタブは、その境界のどこにでも (上、下、左、右) 表示可能です。タブの位置をメンバ関数 IlvNotebook::setTabsPosition で変更し、IlvNotebook::getTabsPosition で取得することができます。

タブの向きを設定する

タブの中で、ラベルを水平あるいは垂直に描画することができます。ラベルの向きは、メンバ関数 IlvNotebook::setLabelsVertical で変更することができます。ラベルが水平か垂直であるかを調べるには、IlvNotebook::areLabelsVertical を使用します。

タブ・ラベルが垂直である場合、ラベルは上から下へ、あるいは下から上へと書くことができます。

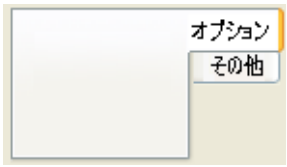


図 11.13　垂直タブ付きのノートブック

下から上へ描かれているラベルは、反転されているといいます。垂直ラベルの描き方を変更するには、これらのメンバ関数 `IlvNotebook::mustFlipLabels` および `IlvNotebook::flipLabels` を使用します。

タブ・マージンの設定

タブの境界とそのラベルの間のマージンを、次のメンバ関数で変更することができます。`IlvNotebook::getXMargin`、`IlvNotebook::setXMargin`、`IlvNotebook::getYMargin`、および `IlvNotebook::setYMargin`。

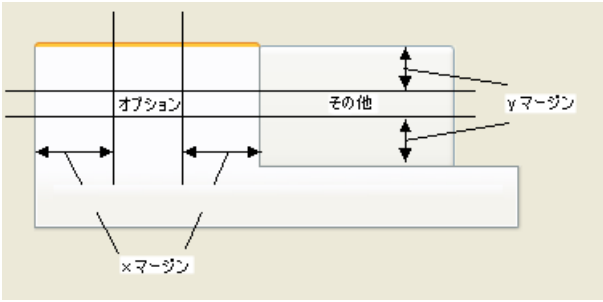


図 11.13　タブ・マージン(x および y)

ページ・マージンの設定

ノートブックの境界とそのページ・ビューの間のマージンを変更することもできます。メンバ関数 `IlvNotebook::getPageArea` に返される値は、ページ・マージンに設定された値に依存します。

以下は、ページ・マージンを設定するためのメンバ関数です。`IlvNotebook::setPageTopMargin`、`IlvNotebook::setPageBottomMargin`、`IlvNotebook::setPageLeftMargin`、`IlvNotebook::setPageRightMargin`。対応する get メンバ関数でマージンを取得することができます。

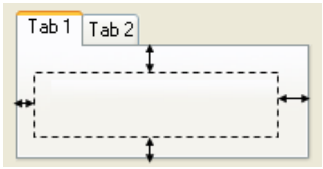


図 11.14　ページ・マージン

ノートブック・ページの処理

ノートブックのページは、クラス IlvNotebookPage によって実装されています。これは特定の要件に合わせるためにサブクラス化することができます。IlvNotebookPage のインスタンスは、IlvGadgetContainer あるいはその他のタイプのビューをカプセル化することができます。254 ページの *ページ内容の表示* を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *ノートブックの追加および削除*
- *ページ内容の表示*
- *ノートブック・ページのカスタマイズ*
- *ノートブック・ページの色を変更する*
- *タブ内容の設定*

ノートブックの追加および削除

作成時のノートブックにはページがありません。ノートブックには少なくとも 1 つのページが必要です

ノートブックにページを追加するには、IlvNotebook::addPage メンバ関数の 1 つを使用します。

```
IlvNotebookPage* addPage(IlvNotebookPage* page,
                         IlvUShort idx = IlvNotebookLastPage);

IlvNotebookPage* addPage(const char* label,
                         IlvBitmap*  bitmap = 0,
                         IlBoolean  transparent = IlTrue,
                         const char*   filename = 0,
                         IlvUShort idx = IlvNotebookLastPage);
```

最初の addPage メンバ関数で、IlvNotebookPage のサブクラスを追加することができます。2 番目のメンバ関数は、IlvNotebookPage の新しいインスタンスを作成します。idx パラメータは、ページが追加される位置を指定します。

IlvNotebookPage は、常に特定のノートブックに関連しています。つまり、IlvNotebookPage をノートブック間で共有することはできません。IlvNotebookPage::getNotebook を使用して、ページに関連するノートブックを取得することができます。

ノートブックのページ数を調べるには、IlvNotebook::getPagesCardinal を使用します。

内部ページの配列を取得するには、IlvNotebook::getPages を使用します。最初のページを取得するには、以下を呼び出します。

```
page = notebook->getPages()[0];
```

特定のページを削除するには、IlvNotebook::removePage を使用します。

ページ内容の表示

メンバ関数 IlvNotebookPage::createView は、IlvGadgetContainer タイプのビューを作成し、IlvNotebookPage::getView で取得することができるページの内容を表示します。

.ilv ファイルをノートブック・ページに、メンバ関数 IlvNotebookPage::setFileName で内容を読み込み、このファイルを IlvNotebookPage::getFileName で取得することができます。

メンバ関数 setFileName は、ビューが IlvGadgetContainer あるいはそのサブクラスであると想定します。他のタイプのビューを使用している場合は、これをオーバーライドする必要があります。

ノートブック・ページのカスタマイズ

メンバ関数 IlvNotebookPage::setView を使用して IlvNotebookPage によって保持されているビューを変更することができます。IlvNotebookPage のサブクラスで、メンバ関数 IlvNotebookPage::createView を再定義することもできます。254 ページの *ページ内容の表示* を参照してください。

このメンバ関数は、パラメータとして与えられたサイズで非表示ビューをインスタンス化します。これはまた、.ilv ファイル (getFileName の結果) を新規ビューに読み込みます。

たとえば、ノートブック・ページに IlvScrolledView インスタンスをカプセル化させるには、以下のように IlvNotebookPage をサブクラス化します。

```
class myNotebookPage : public IlvNotebookPage
{
public:
    myNotebookPage(IlvNotebook* gadget,
                   const char*    label,
                   Ilvbitmap*     bitmap,
                   IlBoolean     transparent,
                   const char*    filename)
   : IlvNotebookPage(gadget, label, bitmap, transparent, filename) {}
       virtual IlvView* createView(IlvAbstractView* parent,
   const IlvRect& size);
};

IlvView* myNotebookPage::createView(IlvAbstractView* parent,
                                    const IlvRect& size)
{
  IlvScrolledView* scview = new IlvScrolledView(parent,size);
  IlvGadgetContainer* child = new IlvGadgetContainer(scview->getClipView(),
                                                     IlvRect(0,0,100,100));
  if (_filename && _filename[0])
    child->readFile(_filename);
  child->fitToContents();
  return scview;
}
```

その後、新規ノートブック・ページを作成してノートブックに追加します。

```
myNotebookPage* np5=
   new myNotebookPage(nb,"Page5",0,IlFalse,"../snbook.ilv");
nb->addPage(np5);
```

ビューで、.ilv ファイルを読み込める場合、メンバ関数 IlvNotebookPage::setFileName をオーバーロードすることができます。

ページにさらに描画を追加する必要がある場合、draw メソッドをオーバーロードすることができます。

```
void draw(IlvPort* dst,
          const IlvRect& pageRect,
          const IlvTransformer* t,
          const IlvRegion* clip) const;
```

ページに以下のコンストラクタを作成する必要があります。

```
MyNotebookPage::MyNotebookPage(IlvNotebook* notebook);
MyNotebookPage::MyNotebookPage(IlvNotebook* notebook,
                               const char*    label,
                               IlvBitmap*   bitmap,
                               IlBoolean     transparent,
                               const char*  filename);
MyNotebookPage::MyNotebookPage(const MyNotebookPage& source);
MyNotebookPage::MyNotebookPage(IlvNotebook* notebook,
                               IlvInputFile&);
```

メンバ関数 IlvNotebookPage::write および IlvInputFile をパラメータとして取得するコンストラクタにより、ページの .ilv 形式を拡張できます。

ノートブック・ページの色を変更する

ノートブックの各ページは、異なる背景色を持つことができます。この色を変更するには、メンバ関数 IlvNotebookPage::setBackground を使用します。ノートブック・ページの背景色を変更すると、この色はそのビューの背景に適用されます。

タブ内容の設定

ノートブック・タブは、ラベルおよびビットマップを含むことができます。ノートブック・タブに表示するラベルを設定するには、メンバ関数 IlvNotebookPage::setLabel を使用します。IlvNotebookPage::getLabel を使用してこのラベルを取得します。

このラベルは、ニーモニックを持つことができます。218 ページの *ガジェットのローカライズ*を参照してください。

ノートブック・タブに表示するビットマップを設定するには、メンバ関数 IlvNotebookPage::setBitmap を使用します。IlvNotebookPage::getBitmap を使用してこのビットマップを取得します。

イベント処理およびコールバック

ユーザがノートブック・ページをマウスでクリックして、矢印キーを押して、あるいはそれに関連付けられているニーモニック文字を押して選択するとき、メンバ関数 `IlvNotebook::changeSelection` が呼び出されます。このメンバ関数は、パラメータとして以前に選択されたページとともに `IlvNotebook::pageDeselected` を、パラメータとして新しく選択されたページとともに `IlvNotebook::pageSelected` を呼び出します。`IlvNotebook::pageDeselected` は、`IlvNotebookPage::deSelect` を呼び出し、Page Deselected コールバックをトリガします。`IlvNotebook::pageSelected` は、`IlvNotebookPage::select` を呼び出し、Page Deselected コールバックをトリガします。これらのタイプを、`IlvNotebook::PageSelectedCallbackType` および `IlvNotebook::PageDeselectedCallbackType` で取得することができます。ノートブック・ページをリサイズするには、Page Resize コールバックを呼び出します。このタイプを `IlvNotebook::PageResizedCallbackType` で取得することができます。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

IlvNumberField の使用

クラス `IlvNumberField` は、各種形式で数値を編集するために特殊なテキスト・フィールドを定義します。`IlvNumberField` は、`IlvTextField` のサブクラスです。

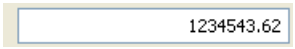


図 11.15 *数値フィールド*

274 ページの *IlvTextField の使用* を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *編集モードの選択*
- *形式の選択*
- *値範囲の定義*
- *値の設定および取得*
- *「桁」区切りの指定*
- *小数点文字の指定*
- *イベント処理およびコールバック*

編集モードの選択

IlvNumberField クラスには 2 つメイン編集モードがあります。ひとつは整数用 (IlvInt) で、もうひとつは浮動小数点数 (IlvDouble) 用です。有効な編集モードは使用されているコンストラクタによって異なります。

整数の編集用の数値フィールドを作成するには、次の 2 つのコンストラクタのうち 1 つを使用します

```
IlvNumberField* field  = new IlvNumberField(display,0,
                                            IlvRect(10,10, 100, 30));
IlvNumberField* field  = new IlvNumberField(display,
                                            IlvPoint(10,10), 0);
```

浮動小数点数の編集用の数値フィールドを作成するには、次の 2 つのコンストラクタのうち 1 つを使用します

```
IlvNumberField* field  = new IlvNumberField(display, 0.0,
                                            IlvRect(10,10, 100, 30));
IlvNumberField* field  = new IlvNumberField(display,
                                            IlvPoint(10,10), 0.0);
```

形式の選択

数値フィールドは、形式に割り当てることができます。現在の形式実行時に IlvNumberField::setFormat で変更することができます。

フォーマットは、この enum 宣言で定義されています。

```
enum {  thousands  = 1,
        scientific = 2,
        padright   = 4,
        showpoint  = 8,
        floatmode  = 16};
```

次のように設定されています。

```
obj->setFormat(IlvNumberField::floatmode|IlvNumberField::scientific);
```

以下は、各種日付形式の記述です。

- floatmode - 浮動値を編集するにはこのモードを使用します。このモードは自動的に、コンストラクタがタイプ IlvDouble の値とともに使用されているときに設定されます。257 ページの *編集モードの選択* を参照してください。
- scientific - このモードは、ガジェットで IlvDouble 値の出力を制御するために使用します。scientific が設定されていると、小数点の前は 1 桁、小数点の後ろの桁数が指定された数値と同じである場合 (デフォルトでは 6) に、値は科学的表記で変換されます。文字 e は、指数を示します。scientific が設定されていない場合、値は小数点後の精度桁数 (デフォルトでは 6 桁) 付きの 10 進法に変換されます。このモードは、floatmode が設定されている時にのみ機能します。

- padright - 小数点の後ろにゼロを追加するのにこのモードを使用します。このモードは、floatmode が設定されている時にのみ機能します。
- showpoint - このモードは、小数点の後ろのゼロを削除するときに小数点を残すために padright モードとともに使用します。このモードは、floatmode が設定されている時にのみ機能します。
- thousands - 「桁」区切り文字を表示するためにこのモードを使用します。デフォルトの「桁」区切りは、「,」です。258 ページの *「桁」区切りの指定*を参照してください。

値範囲の定義

数値フィールドで編集できる最小値および最大値を指定することができます。編集するのが整数か浮動小数点値かに応じて 2 組のメンバ関数があります。

整数には、IlvNumberField::setMaxInt および IlvNumberField::setMinInt を使用します。

浮動小数点数には、IlvNumberField::setMaxFloat および IlvNumberField::setMinFloat を使用します。

値の設定および取得

IlvNumberField は、値の設定および取得用に 2 組のメンバ関数を提供します。

値が整数の場合、以下を使用します。

```
IlvInt     getIntValue(IlBoolean& error) const;
IlBoolean setValue(IlvInt, IlBoolean redraw = IlFalse);
```

値が浮動小数点数の場合

```
IlvDouble  getFloatValue(IlBoolean& error) const;
IlBoolean setValue(IlvDouble, IlBoolean redraw = IlFalse);
```

「桁」区切りの指定

メンバ関数 thousands および float が設定されているとき、桁区切りが表示されます。デフォルトの「桁」区切りは、コンマ文字「,」です。この文字を、メンバ関数 IlvNumberField::setThousandSeparator を使用して変更することができます。このメンバ関数を呼び出しても、数値フィールドのテキストは直接変更されません。フィールドに値が既に含まれている場合、まずその値を取得して区切り文字を変更し、値を再び設定しなくてはなりません。

小数点文字の指定

浮動小数点数用デフォルトの小数点文字は、ピリオド文字です (.)。この文字を、メンバ関数 `IlvNumberField::setDecimalPointChar` を使用して変更することができます。

このメンバ関数を呼び出しても、数値フィールドのテキストは直接変更されません。フィールドに値が既に含まれている場合、まずその値を取得して小数点文字を変更し、値を再び設定しなくてはなりません。

イベント処理およびコールバック

ユーザが数値フィールドで Enter キーを押すと、`IlvNumberField::validate` メンバ関数が呼び出されます。この仮想メンバ関数は、数値フィールドに関連付けられている Main コールバックを呼び出し、キーボード・フォーカスをフォーカス・チェーンの次のガジェットへ移動します。これはフィールド内容が数値に変換されている場合にのみ起こり、この数値は `IlvNumberField::setMaxFloat`、`IlvNumberField::setMinFloat`、`IlvNumberField::setMaxInt`、および `IlvNumberField::setMinInt` で指定された範囲内になります。

258 ページの *値範囲の定義*、211 ページの *フォーカス管理*、および 217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

IlvOptionMenu の使用

クラス `IlvOptionMenu` は、ユーザがその中から選択できるアイテムのドロップダウン・リストを定義します。

図 11.16 *オプション・メニュー*

> ***メモ :*** *Microsoft® Windows® ルック・アンド・フィールはオプション・メニューを提供していないため、クラス* `IlvOptionMenu` *は、使用スタイルが Microsoft Windows の場合、コンボ・ボックスとして表示されます。*

このセクションでは、以下のトピックを取り上げます。

- *アイテムの設定および取得*
- *選択したアイテムの変更および取得*

- *オプション・メニューのローカライズ*
- *イベント処理およびコールバック*

アイテムの設定および取得

IlvOptionMenu は、IlvListGadgetItemHolder のサブクラスであるため、オプション・メニューのアイテムを変更する場合は、このクラスのメンバ関数を使用しなくてはなりません。

IlvListGadgetItemHolder を参照してください。

選択したアイテムの変更および取得

オプション・メニューで選択したアイテムを変更するには、メンバ関数 IlvOptionMenu::setSelected を使用します。選択したアイテムのインデックスを取得するには、IlvOptionMenu::whichSelected を使用します。

オプション・メニューのローカライズ

オプション・メニューはローカライズすることができます。

218 ページの *ガジェットのローカライズ* を参照してください。

イベント処理およびコールバック

ユーザがメニューから新しいアイテムをマウスでポイントする、あるいは矢印キーを使用して選択すると、仮想メンバ関数 IlvOptionMenu::doIt が呼び出されます。必要に応じて、オプションのサブクラスでオーバーライドすることができます。

デフォルト定義では、オプション・メニューの Main コールバックを呼び出します。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

IlvPasswordTextField の使用

IlvPasswordTextField クラスは、パスワード入力用の特殊なテキスト・フィールドです。パスワードの機密性を守るため、このフィールドに入力した文字が特殊な文字に置き換わります。IlvPasswordTextField は、IlvTextField のサブクラスです。

ユーザによって入力されたテキストを取得するには、IlvTextField::getLabel メンバ関数を使用します。実際のテキストの場所に入力された文字を変更するには、IlvPasswordTextField::setMaskChar を呼び出します。

IlvScrollBar の使用

クラス IlvScrollBar は、ウィンドウをスクロールするために使用される 2 つの矢印とスライダ付きの矩形領域を定義します。この矩形領域は、スクロールバーと呼ばれます。

図 11.17 *スクロールバー*

このセクションでは、以下のトピックを取り上げます。

- *スクロールバー値の設定*
- *スクロールバーの向きを設定する*
- *イベント処理およびコールバック*

スクロールバー値の設定

スクロールバーは、以下の値によって定義されます。

- 現在の値
- その最小値と最大値
- スライダのサイズ
- インクリメントとは、スクロールバーの矢印をクリック、あるいは左、右、上、下のキーを押したときに、スクロールバーの現在の値に加えられる、あるいはそこから引かれる値です。
- ページ追加とは、スライダと矢印の間の領域をクリックするか Page-Up キーまたは Page-Down キーを押したときに現在のスクロールバー値に加えられる、あるいはそこから引かれる値のことです。

スクロールバーの現在の値は、最小値および (最大 - スライダ・サイズ) 値の間で変更することができます。

IlvScrollBar::setValues メソッドを使用して、スクロールバーの現在の値、およびその最小値、最大値を設定します。

IlvScrollBar::setIncrement および IlvScrollBar::setPageIncrement メソッドを使用して、インクリメントおよびページ追加を設定します。

スクロールバーの向きを設定する

スライダには、4 つの向きを設定できます。これはコンストラクタで指定します。その向きを IlvScrollBar::setOrientation を使用して変更することもできます。スライダの向きを、以下のように設定します。

- IlvLeft　左側を最小値にした水平スライダ
- IlvRight　右側を最小値にした水平スライダ
- IlvTop　最上部を最小値にした水平スライダ
- IlvBottom　最下部を最小値にした水平スライダ

イベント処理およびコールバック

ユーザがスライダをドラッグすると、スクロールバーの値が変更され、仮想メンバ関数 IlvScrollBar::drag が呼び出されます。このメンバ関数は、サブクラスでオーバーライドすることができます。デフォルト定義が、メンバ関数 IlvScrollBar::valueChanged を呼び出します。

IlvScrollBar::valueChanged は、ユーザがスクロールバー矢印、あるいはスライダと矢印の間の領域をクリックしたり、矢印キーまたは Home キーおよび End キーを押したときに呼び出されます。この仮想メンバ関数は、サブクラスでオーバーライドすることができます。デフォルト定義では、スクロールバーに関連する Main コールバックが呼び出されます。

スライダをドラッグした後に放すと、仮想メンバ関数 IlvScrollBar::dragged が呼び出されます。このメンバ関数は、サブクラスでオーバーライドすることができます。デフォルト定義では、スクロールバーに関連する Secondary コールバックが呼び出されます。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

IlvSlider の使用

クラス IlvSlider は、スライダを含む矩形領域を定義します。ユーザがスライダを移動させると、その値が変化します。

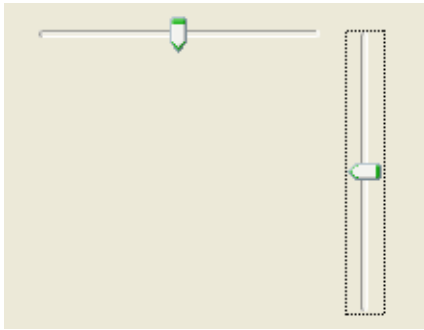


図 11.18 *水平および垂直スライダ*

このセクションでは、以下のトピックを取り上げます。

- *スライダ値の設定*
- *スライダの向きを設定する*
- *サムの向きを設定する*
- *イベント処理およびコールバック*

スライダ値の設定

クラス `IlvSlider` は、範囲内で値を変更する簡単な方法を提供します。

スライダは、以下の値によって定義されます。

- 現在の値
- その最小値と最大値
- スライダのサイズ
- ページ追加とは、スライダの外側の領域をクリックするか、Page-Up キーまたは Page-Down キーを押したときに現在のスライダ値に加えられる、あるいはそこから引かれる値のことです。

メンバ関数 `IlvSlider::setValues` を使用してスライダの値およびその範囲を設定することができます。ページ追加を `IlvSlider::setPageIncrement` を使用して設定することができます。

スライダの向きを設定する

スライダには、4 つの向きを設定できます。これはコンストラクタで指定します。その向きを `IlvSlider::setOrientation` を使用して変更することもできます。スライダの向きを、以下のように設定します。

- `IlvLeft`　左側を最小値にした水平スライダ
- `IlvRight`　右側を最小値にした水平スライダ
- `IlvTop`　最上部を最小値にした水平スライダ
- `IlvBottom`　最下部を最小値にした水平スライダ

サムの向きを設定する

サムの向きも、`IlvSlider::setThumbOrientation` メソッドを使用して設定することができます。ただし、この設定は、すべてのルック・アンド・フィール・スタイルでサポートされているわけではありません。たとえば、Motif ルック・アンド・フィールを使用しているときは、サムの向きを設定しても何も変化はありません。

次の図は、サムの向きが異なる 2 つのスライダを示しています。

図 11.19　*スライダのサムの向き*

イベント処理およびコールバック

ユーザがスライダをドラッグし、その外をクリックするか、矢印キー、Home キーあるいは End キーを押すと、スライダ値が変更され、仮想メンバ関数 `IlvSlider::valueChanged` が呼び出されます。このメンバ関数は、特殊なアクションを実行させるためにサブクラスでオーバーライドすることができます。デフォルト定義では、スライダに関連する Main コールバックが呼び出されます。スライダに変更が加えられると、slider コールバックが呼び出されます。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

IlvSpinBox の使用

クラス `IlvSpinBox` は、2 つのボタンおよびタイプ `IlvTextField` あるいは `IlvNumberField` の複数のフィールドから構成される組み合わせガジェットを定義します。

テキスト・フィールドについては、ボタンを使用している間、ユーザがスピンさせることのできる定義済み文字列値のリストを定義することができます。数値

フィールドに対しては、ボタンを使って、指定した値範囲の中でユーザが加算または減算できる数値のセットを定義することができます。

また、スピン・ボックスに任意のグラフィック・オブジェクトを追加することもできます。

図 11.20 *スピン・ボックス*

256 ページの *IlvNumberField の使用*および 274 ページの *IlvTextField の使用*を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *スピン・ボックスへのフィールドを追加 / 削除する*
- *テキスト・フィールドの処理*
- *数値フィールドの処理*
- *イベント処理およびコールバック*

スピン・ボックスへのフィールドを追加 / 削除する

作成時のスピン・ボックスに、フィールドはありません。2 つの矢印ボタンのみで構成されています。1 つ以上の `IlvTextField` あるいは `IlvNumberField` をスピン・ボックスに追加することができます。しかし、アプリケーションでフィールドのないスピン・ボックスで値を加算、または減算させることができます。

フィールドの追加

スピン・ボックスにフィールドを追加するには、表示させたい値のタイプ (文字列か数値) に応じて、2 つのメンバ関数のうち 1 つを使用することができます。

```
void addField(IlvTextField* field,
              const char**  values,
              IlvUShort     count,
              IlvUShort     pos,
              IlBoolean    loop,
              IlvUShort     at     = 0,
              IlBoolean    redraw = IlFalse);
```

`values` パラメータは、スピンさせる文字列を保持します。`count` パラメータは、`values` に文字列の数を指定します。

```
void addField(IlvNumberField* field,
              IlvDouble       value,
              IlvDouble       increment,
              IlBoolean    loop,
              IlvUShort       at     = 0,
              IlBoolean    redraw = IlFalse);
```

数値フィールドをスピン・ボックスに追加する時、数値フィールドによって指定された値の範囲内で、ボタンを使って数値フィールドの値を変更します (256 ページの *IlvNumberField の使用*を参照してください)。

value パラメータは、フィールドの初期値です。increment パラメータは、[加算] あるいは [減算] ボタンをクリックしたときに数値フィールドの値に加えられる、または差し引かれる値を指定します。

loop パラメータが IlTrue に設定されている場合、スピン・ボックスは、ユーザが最後の値を加算しようとした時に最初の値を返し、ユーザが最初の値を減算しようとした時に最後の値を返します。

at パラメータで、スピン・ボックスの特定の場所にフィールドを挿入できます。

以下は簡単な例です (spinbox は、IlvSpinBox オブジェクトへのポインタです)。

```
const char* values[7] = {"Monday", "Tuesday", "Wednesday",
                         "Thursday","Friday","Saturday","Sunday"};
spinbox->addField(new IlvTextField(display,"",IlvRect(0,0,10,10)),
                  values, 7, 0, IlTrue);
```

> ***メモ:*** *IlvTextField を作成するために使用する矩形はここでは何の意味もありません。また、IlvTextField をコンテナに追加する必要もありません。ここではスピン・ボックスに管理されているからです。*

フィールドの削除

フィールドを IlvSpinBox から削除するには、IlvSpinBox::removeObject を使用します。このメンバ関数は、修飾としてスピン・ボックスに加えられたグラフィック・オブジェクトも削除します。

グラフィック・オブジェクトの追加

スピン・ボックスにグラフィック・オブジェクトをメンバ関数 IlvSpinBox::addObject で追加することができます。スピン・ボックスに表示されるグラフィック・オブジェクトは、修飾として機能し、特定の振る舞いはありません。

テキスト・フィールドの処理

スピン・ボックスのフィールドがタイプ IlvTextField である場合、定義済み文字列の配列を IlvSpinBox::getLabels および IlvSpinBox::getLabelsCount で取得することができます。

定義済み文字列をテキスト・フィールドに IlvSpinBox::addLabel で追加し、IlvSpinBox::removeLabel で削除することができます。

次のメンバ関数で、テキスト・フィールドの内容を設定あるいは取得することができます。

```
const char* getLabel(IlvTextField*) const;
void setLabel(IlvTextField* field,
              const char*   label,
              IlBoolean    redraw = IlFalse);
void setLabel(IlvTextField* field,
              IlvUShort     index,
              IlBoolean    redraw = IlFalse);
```

数値フィールドの処理

スピン・ボックスのフィールドがタイプ IlvNumberField である場合、指定した加算を IlvSpinBox::setIncrement で設定し、IlvSpinBox::getIncrement で取得することができます。

加算とは、ユーザがスピン・ボックス・ボタンをクリックしたときにフィールド値に追加されたり、これから取得される値です。

次のメンバ関数でフィールドの数値を設定あるいは取得することができます。

```
IlvDouble getValue(IlvNumberField* field,
                   IlBoolean& error)const;
IlBoolean setValue(IlvNumberField* field,
                    IlvDouble value);
```

イベント処理およびコールバック

クラス IlvSpinBox は、2 つのコールバック・タイプを定義します。以下を使用してアクセスできる加算および減算です。

```
static IlvSymbol* IncrementCallbackType();
static IlvSymbol* DecrementCallbackType();
```

これらのコールバックは、ユーザが [加算] および [減算] ボタンをクリックした時に呼び出されます。アクティブなフィールドがある場合には、コールバックが呼び出される直前に加算 / 減算します。Main コールバックは、どちらの場合も呼び出されます。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

IlvStringList の使用

クラス IlvStringList は、クラス IlvGadgetItem あるいはサブクラスのガジェット・アイテムのリストを表示します。IlvStringList は、IlvScrolledGadget および IlvListGadgetItemHolder のサブクラスです。

文字列リストは、最大 32767 アイテムまで保存でき、ラベル、ビットマップ、あるいはグラフィック・オブジェクト (クラス IlvGraphic) から構成され、スクロールバーをサポートしています。

図 11.21 *文字列リスト*

このセクションでは、以下のトピックを取り上げます。

- *文字列リスト・アイテムの操作*
- *文字列リスト・アイテムの外観をカスタマイズする*
- *ツールチップの表示*
- *文字列リスト・アイテムのローカライズ*
- *イベントの処理およびコールバック*

文字列リスト・アイテムの操作

文字列リスト・アイテムの操作用メンバ関数は、ベース・クラス IlvListGadgetItemHolder で定義されています。

文字列リスト・アイテムの外観をカスタマイズする

ガジェット・アイテムのグラフィック機能に加えて (クラス IlvGadgetItem を参照)、クラス IlvStringList は、そのアイテムのグローバル表示をカスタマイズする方法をいくつか提供しています。

- *アイテムの高さを定義する*
- *ラベルおよびピクチャの表示*
- *ラベルおよびピクチャ位置の設定*
- *ラベル整列の設定*
- *選択モードの選択*

アイテムの高さを定義する

デフォルトでは、文字列リストのアイテムはさまざまな高さにすることができます。ただし、メンバ関数 IlvStringList::setDefaultItemHeight を使用して、全アイテムを同じ高さで表示することもできます。

ラベルおよびピクチャの表示

文字列リストのピクチャは、メンバ関数 IlvStringList::showPicture で、表示にしたり非表示にしたりできます。

同様に、文字列リストのラベルは、メンバ関数 IlvStringList::showLabel で、表示にしたり非表示にしたりできます。

デフォルトでは、文字列リストはラベルおよびピクチャの両方を表示します。

> ***メモ:*** *IlvGadgetItem::showLabel および IlvGadgetItem::showPicture で特定のアイテムのこのグローバル設定をオーバーライドすることができます。*

ラベルおよびピクチャ位置の設定

ピクチャに関連するアイテム・ラベルの位置を、IlvStringList::setLabelPosition で変更することができます。

デフォルトでは、ラベルは、ピクチャの右に置かれます (IlvRight)。

> ***メモ:*** *IlvGadgetItem::setLabelPosition で特定のアイテムのこのグローバル設定をオーバーライドすることができます。*

ラベル整列の設定

ラベル位置が IlvRight (デフォルト値) であり、特定のアイテムのみがピクチャおよびラベルを使用しているとき、図 11.21 で示すように、すべてのラベルを左寄せにする場合があります。

デフォルトでは、アイテム・ラベルは自動的に整列されます。アイテムが変更されると、リストは新しいラベル整列を再計算します。この操作は時間がかかるため、メンバ関数 IlvStringList::autoLabelAlignment を使用して、ラベルの自動整列をオフにすることができます。

全ピクチャのサイズがわかっているため、自動ラベル整列モードをオフにしたい場合もあります。この場合、IlvStringList::setLabelOffset を呼び出します。

たとえば、次の呼び出しでは、各ラベル・アイテムを 30 ピクセルの左マージンを付けて表示させます。

```
slist->setLabelOffset(30);
```

選択モードの選択

文字列リスト・アイテムが選択されると、強調表示されます。IlvStringList クラスは、選択したアイテムを表示するために 2 つの異なるモードを提供します。

- **フル選択モード**　このモード (デフォルト) が設定されていると、選択は文字列リストの全幅に広がります。

- **部分選択モード**　このモードが選択されていると、選択は、アイテム・ラベルのみに広がります。

これらの 2 つのモードを下図に示します。

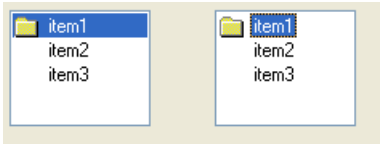


図 11.22　*フル選択モード (左) および部分選択モード (右)*

`IlvStringList::useFullSelection` を使うと、1 つのモードから別のモードへ切り替えることができます。

ツールチップの表示

文字列リストは、メンバ関数 `IlvStringList::useToolTips` で、ツールチップをオンにしている場合、マウス・ポインタが部分的に表示されたアイテムの上に置かれたときにツールチップを表示させることができます。

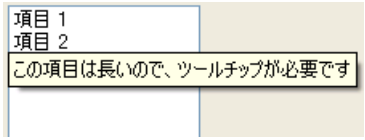


図 11.23　*表示されたツールチップ*

> ***メモ :*** *ツールチップは、部分選択モードが設定されているときのみ機能します。269 ページの選択モードの選択を参照してください。*

文字列リスト・アイテムのローカライズ

文字列リスト・ラベルはローカライズすることができます。

218 ページの *ガジェットのローカライズ*を参照してください。

イベントの処理およびコールバック

このセクションでは、以下のトピックを取り上げます。

- *選択モード*
- *文字列リスト・アイテムの選択およびダブルクリック*
- *文字列リスト・アイテムの編集*

◆ *文字列リスト・アイテムのドラッグ・アンド・ドロップ*

選択モード

文字列リストには、2 つの選択モードがあります。単一 (あるいは、排他的) 選択と複数選択です。

単一選択モードでは、1 度に 1 つのアイテムだけを選択できます。このモードには、2 つのサブモードがあります。

◆ `IlvStringListSingleSelection` - 同時に選択することのできるアイテムは つだけです。

◆ `IlvStringListSingleBrowseSelection` - このモードは前と似ていますが、選択したアイテムをクリックすると選択が解除される点が異なります。

複数選択モードでは、複数のアイテムを同時に選択することができます。このモードには、3 つのサブモードがあります。

◆ `IlvStringListBrowseSelection` - アイテムをクリックする、あるいはマウスをドラッグさせて、複数のアイテムを同時に選択することができます。同様に、複数のアイテムをクリックする、あるいは中央ボタンでマウスをドラッグさせて、複数アイテムの選択を解除することができます。

◆ `IlvStringListMultipleSelection` - アイテムをクリックするとアイテムを選択するか、または選択を解除します。

◆ `IlvStringListExtendedSelection` - Shift キーあるいは Control キーを使用して選択を広げることができます。

有効の選択モードを変更するには、メンバ関数 `IlvStringList::setExclusive` を使用します。サブモードを変更するには、`IlvStringList::setSelectionMode` を使用します。複数選択モードでは、メンバ関数 `setSelectionLimit` で、選択できるアイテムの数に制限を設定することができます。

文字列リスト・アイテムの選択およびダブルクリック

ユーザが文字列リスト・アイテムをダブルクリックすると、Main コールバックが呼び出されます。ユーザがアイテムを選択、あるいは選択を解除すると、Select コールバックが呼び出されます。このコールバックを設定するには、メンバ関数 `IlvStringList::setSelectCallback` を使用します。

217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

リストのアイテムの選択およびダブルクリックを制御するには、次のメンバ関数をサブクラスで再定義することができます。`IlvStringList::select` (アイテムの選択)、`IlvStringList::unSelect` (選択の解除)、あるいは `IlvStringList::doIt` (アイテムのダブルクリック)。

コーディングで選択を変更する場合は、`IlvStringList::setSelected` を使用することができます。

文字列リスト・アイテムの編集

文字列リストのアイテムは編集できます。295 ページの *ガジェット・アイテムの検索*を参照してください。

文字列リスト・アイテムのドラッグ・アンド・ドロップ

IlvStringList クラスは、簡単に使えるドラッグ・アンド・ドロップ機構を提供しています。297 ページの *ガジェット・アイテムのドラッグ・アンド・ドロップ*を参照してください。

IlvText の使用

クラス IlvText は、複数行エディタを定義します。IlvText は、IlvScrolledGadget のサブクラスであるため、テキスト・エディタにはスクロールバーが付いています。

図 11.24 *複数行テキスト・エディタ*

クラス IlvText は、テキストの設定と取得、および編集モードと読み取り専用モード間の切り替え用に、多くのメンバ関数を提供しています。

スクロールバーの処理に関する詳細は、ベース・クラス IlvScrolledGadget を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *テキストの設定および取得*
- *イベント処理*

テキストの設定および取得

IlvText オブジェクトの内容を仮想メンバ関数 IlvText::setText で指定し、IlvText::getText で取得することができます。

特定行のテキストを IlvText::setLine で設定し、IlvText::getLine で取得することができます。

IlvText::addLine および IlvText::removeLine で行を追加あるいは削除することができます。

クラス IlvText は、複数行を設定あるいは取得するための役に立つメソッドを多数備えています。

イベント処理

このセクションでは、以下のトピックを取り上げます。

- *チェック・メソッド*
- *キーボード・ショートカット*

チェック・メソッド

ユーザが一定の ASCII 文字をテキスト・ガジェットに入力するたびに、仮想メンバ関数 IlvText::check が呼び出されます。デフォルトの定義では、選択したテキストを削除し、現在のカーソルの位置に入力した文字を追加します。

キーボード・ショートカット

次の表は、テキスト・フィールドで使用できるキーボード・ショートカットのリストです。

キー	振る舞い
Home あるいは Ctrl+A	行の先頭にカーソルを移動します。
Meta <	テキストの頭にカーソルを移動します。
Meta >	テキストの終わりにカーソルを移動します。
End あるいは Ctrl+E	行の最後にカーソルを移動します。
左矢印キーあるいは Ctrl+B	カーソルを 1 文字左に移動させます。
右矢印キーあるいは Ctrl+F	カーソルを 1 文字右に移動させます。
↑ キーあるいは Ctrl+P	カーソルを 1 行上に移動させます。
↓ キーあるいは Ctrl+N	カーソルを 1 行下に移動させます。
Page Up	カーソルを ページ前に移動させます。
Page Down	カーソルを ページ後ろに移動させます。
Ctrl+K	カーソルより後のテキストを削除します。
Del あるいは Ctrl+D	カーソルより後の文字を 文字削除します。

キー	振る舞い
Back Space あるいは Ctrl+H	カーソルより前の文字を 1 文字削除します。
Ctrl+X	選択されたテキストをクリップボードに移動します。
Ctrl+C	選択されたテキストをクリップボードにコピーします。
Ctrl+V	クリップボードからテキストを貼り付けます。
Ctrl+Insert (Windows®)	選択されたテキストをクリップボードにコピーします。
Shift+Insert (Windows)	クリップボードからテキストを貼り付けます。
Ctrl+Left、Ctrl+Right	カーソルを、1 ワード左へあるいは右へ移動します。
Shift+Left、Shift+Right Shift+Up、Shift+Down	選択を 1 文字分上、下、左、右へ広げます。
Ctrl+Shift+Left、 Ctrl+Shift+Right	選択を 1 ワード分左または右へ広げます。
Shift+Home、Shift+End	選択を行の最初あるいは最後まで広げます。
Ctrl+Shift+Home Ctrl+Shift+End	選択をテキストの最初あるいは最後まで広げます。

IlvTextField の使用

クラス `IlvTextField` は、短い文字列を編集するために使用される 1 行テキスト・エディタを定義します。

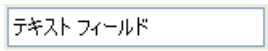


図 11.25 *テキスト・フィールド*

このセクションでは、以下のトピックを取り上げます。

- *テキストの整列*
- *テキストの設定および取得*
- *テキスト・フィールドのローカライズ*
- *文字数の制限*
- *イベントの処理とコールバック*

テキストの整列

IlvTextField のテキストは、左揃え (デフォルト)、右揃え、あるいは中央揃えにすることができます。テキスト整列を変更するには、IlvTextField::setAlignment を使用します。

テキストの設定および取得

メンバ関数 IlvTextField::setLabel および IlvTextField::getLabel を使用してテキストの設定および取得を行います。クラス IlvTextField は、整数あるいは浮動値のようなフォーマットされたテキストの設定および取得に便利なメソッドも含んでいます。

- getIntValue() は、整数値を取得します。
- getFloatValue() 浮動値を取得します。
- setValue(IlvInt) 整数値を設定します。
- setValue(IlvFloat,const char* format) 浮動値を設定します。

IlvTextField のサブクラスは整数値、浮動値、日付、およびパスワードを編集します。

245 ページの *IlvDateField の使用*、256 ページの *IlvNumberField の使用*を参照してください。

テキスト・フィールドのローカライズ

読み取り専用モードのテキスト・フィールドはローカライズすることができます。

218 ページの*ガジェットのローカライズ*を参照してください。

文字数の制限

テキスト・フィールドで編集できる文字列を IlvTextField::setMaxChar で制限することができます。パラメータが -1 に設定されているときは、好きなだけ文字を入力することができます。このメンバ関数は、テキスト・フィールドに入力できる文字数を制限しますが、IlvTextField::setLabel で指定できる文字数の制限はできません。275 ページの*テキストの設定および取得*を参照してください。

イベントの処理とコールバック

このセクションでは、以下のトピックを取り上げます。

- *Validate メソッドおよびMain コールバック*
- *チェック・メソッド*

◆ *labelChanged メソッド*

Validate メソッドおよび Main コールバック

ユーザがテキスト・フィールドで Enter キーを押すと、`IlvTextField::validate` メンバ関数が呼び出されます。この仮想メンバ関数は、テキスト・フィールドの Main コールバックを呼び出し、フォーカスをフォーカス・チェーンの次のガジェットへ移動します。

テキスト・フィールドに Main コールバックを設定すると、これを簡単に検査する方法が提供されます。Main コールバックの代わりに、テキスト・フィールドの検査する Focus Out コールバックを設定することができます。この場合、フィールドは、フォーカスを失ったときに検査されます。

217 ページの *ガジェットにコールバックを関連付ける*および 211 ページの *フォーカス管理*を参照してください。

チェック・メソッド

ユーザが一定の ASCII 文字をテキスト・フィールドに入力するたびに、仮想 `IlvTextField::check` メンバ関数が呼び出されます。デフォルトの定義では、選択したテキストを削除し、現在のカーソルの位置に入力した文字を追加します。

このメソッドは、許可されている最大文字数をチェックします (275 ページの *文字数の制限*を参照してください)。そのため、これを再定義するとき、この機構が機能するようにテストを追加してください (下の例を参考)。

labelChanged メソッド

ユーザがテキスト・フィールドの内容を変更するとき、メンバ関数 `IlvTextField::labelChanged` が呼び出されます。デフォルト定義では、Change コールバックを呼び出します。

このコールバックを設定するには、`IlvTextField::setChangeCallback` を使用します。

217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

キーボード・ショートカット

次の表は、テキスト・フィールドで使用できるキーボード・ショートカットのリストです。

キー	振る舞い
Home あるいは Ctrl+A	テキストの頭にカーソルを移動します。
End あるいは Ctrl+E	テキストの終わりにカーソルを移動します。
左矢印キーあるいは Ctrl+B	カーソルを 1 文字左に移動させます。

キー	振る舞い
右矢印キーあるいは Ctrl+F	カーソルを 1 文字右に移動させます。
Ctrl+K	カーソルより後のテキストを削除します。
Ctrl+U	カーソルより前のテキストを削除します。
Del あるいは Ctrl+D	カーソルより後の文字を 文字削除します。
Back Space あるいは Ctrl+H	カーソルより前の文字を 1 文字削除します。
Ctrl+X	選択されたテキストをクリップボードに移動します。
Ctrl+C	選択されたテキストをクリップボードにコピーします。
Ctrl+V	クリップボードからテキストを貼り付けます。
Ctrl+Insert (Windows)	選択されたテキストをクリップボードにコピーします。
Shift+Insert (Windows)	クリップボードからテキストを貼り付けます。
Ctrl+Left、Ctrl+Right	カーソルを、1 ワード左へあるいは右へ移動します。
Shift+Left、Shift+Right	選択を 1 文字分左、または右へ拡大します。
Ctrl+Shift+Left、 Ctrl+Shift+Right	選択を 1 ワード分左または右へ広げます。
Shift+Home、Shift+End	選択をテキストの最初あるいは最後まで広げます。

IlvToggle の使用

クラス `IlvToggle` はトグル・ボタンとラジオ・ボタンを定義します。トグル・ボタンとラジオ・ボタンは、状態を示すラベルとマーカーから成ります。状態マーカーは、矩形やひし形として表すことができます。クラス `IlvToggle` は、トグル・ボタンを実装し、そのマーカーに色を付けることができるサブクラス `IlvColoredToggle` があります。

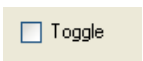


図 11.26 *トグル・ボタン*

このセクションでは、以下のトピックを取り上げます。

- *トグル・ボタンの状態および色の変更*
- *トグル・ボタンおよびラジオ・ボタンのスタイル*

- *トグル・ボタンでビットマップを表示する*
- *ラベルの整列および位置決め*
- *状態マーカーのサイズ変更*
- *トグル・ボタンのローカライズ*
- *ニーモニックをトグル・ボタンに関連付ける*
- *イベントの処理およびコールバック*
- *セレクタでトグル・ボタンをグループ化する*

トグル・ボタンの状態および色の変更

状態マーカーの外観は、関連するトグル・ボタンあるいはラジオ・ボタンの状態 (オンまたはオフ) に応じて変化します。トグル・ボタンの状態を設定するには、メンバ関数 `IlvToggle::setState` を使用し、`IlvToggle::getState` を使用して取得します。

色付きトグル・マーカーの色を設定するには、
`IlvColoredToggle::setCheckColor` を使用し、
`IlvColoredToggle::getCheckColor` to を使用して取得します。

トグル・ボタンおよびラジオ・ボタンのスタイル

クラス `IlvToggle` には、2 つの異なる形状、標準トグル・ボタンおよびラジオ・ボタンを設定できます。

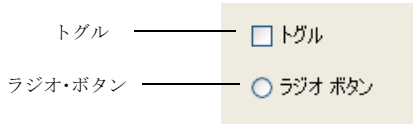


図 11.27 *トグル・ボタンのさまざまなスタイル*

トグル・ボタンのスタイルをラジオとして設定するには、`IlvToggle::setRadio` を使用します。

トグル・ボタンでビットマップを表示する

`IlvToggle` インスタンスは、ビットマップを表示するように要求されたときでも常にラベルを表示します。トグル・ボタンがビットマップ描画するように設定されている場合、およびそのラベルが空でない場合、`IlvToggle` インスタンスが、ビットマップ上にラベルとともに表示されます。

トグル・ボタン上にビットマップを表示するには、メンバ関数 `IlvToggle::setBitmap` を使用します。

ラベルの整列および位置決め

トグル・ボタンのラベルは、状態マーカーの右あるいは左に位置することができます。ラベルは、スペースの左、右、あるいは中央に揃えることができます。

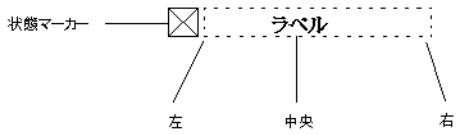


図 11.28 *トグル・ラベルのテキスト整列*

ラベルの位置を設定するには、メンバ関数 `IlvToggle::setPosition` を使用します。

ラベルの整列を設定するには、メソッド `IlvToggle::setTextAlignment` を使用します。

状態マーカーのサイズ変更

状態マーカーのサイズ (つまり、バウンディング・ボックスの高さおよび幅) をメンバ関数 `IlvToggle::setCheckSize` で変更することができます。

状態マーカーに `0` のサイズを設定し、状態マーカーのサイズをデフォルトのサイズに変更します。

> ***メモ :*** *Windows® ルック・アンド・フィールが選択されているとき、マーカー・サイズを変更しても何も変わりません。*

トグル・ボタンのローカライズ

トグル・ボタンのラベルはローカライズすることができます。

218 ページの *ガジェットのローカライズ* を参照してください。

ニーモニックをトグル・ボタンに関連付ける

トグル・ボタンをニーモニック文字に関連付けることができます。

218 ページの *ガジェットのローカライズ* を参照してください。

イベントの処理およびコールバック

ユーザがトグル・ボタンをクリックする、その関連するニーモニック文字を押す、あるいは Enter キーあるいはスペース・バーを押すと、ボタンの状態が変更にな

り、メンバ関数 activate が呼び出されます。この仮想メンバ関数は、トグル・ボタンの Main コールバックを呼び出します。

217 ページの*ガジェットにコールバックを関連付ける*を参照してください。

セレクタでトグル・ボタンをグループ化する

ラジオ・ボックスを作成するために、トグル・ボタンを IlvSelector にグループ化することができます。IlvSelector クラスは、それが保持するグラフィックの中で独特の選択を処理するグラフィック・セットの特別な種類 (IlvGraphicSet) です。

セレクタの 2 つの便利なメソッドで、何が選択されているかを調べることができます。

```
IlvShort    whichSelected() const;
IlvGraphic* whichGraphicSelected() const;
```

> ***メモ:*** *クラス* IlvSelector *は、*IlvGadget *のサブクラスではないので、「セレクタ」インタラクタが対話的セレクタを持つように明示的に設定する必要があります。*

ソース・プログラム

```
#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/toggle.h>
#include <ilviews/graphics/selector.h>

static void QuitCallback(IlvView* top, IlvAny)
{
    IlvDisplay* display = top->getDisplay();
    delete top;
    delete display;
    IlvExit(0);
}

int main(int argc, char* argv[])
{
    IlvDisplay* display = new IlvDisplay("Demo", "", argc, argv);
    if (!display || display->isBad()) {
        IlvFatalError("Couldn't open display");
        delete display;
        IlvExit(-1);
    }

    IlvGadgetContainer* container =
       new IlvGadgetContainer(display,
                              "Demo",
                              "Demo",
                              IlvRect(0, 0, 100, 150));
    container->setDestroyCallback(QuitCallback);
```

```
    IlvSelector* selector = new IlvSelector;
    IlvToggle* toggle;
    toggle = new IlvToggle(display, IlvPoint(10, 10), "Toggle 1");
    selector->addObject(toggle);
    toggle = new IlvToggle(display, IlvPoint(10, 50), "Toggle 2");
    selector->addObject(toggle);
    toggle = new IlvToggle(display, IlvPoint(10, 90), "Toggle 3");
    selector->addObject(toggle);

    container->addObject("Selector", selector);

    container->show();

    IlvMainLoop();
    return 0;
}
```

セレクタの作成

セレクタは、そのコンストラクタを呼び出すことで作成されます。

```
IlvSelector* selector = new IlvSelector;
```

次にインタラクタを設定します。

```
selector->setInteractor(IlvInteractor::Get("Selector"));
```

トグル・ボタンの追加

各トグル・ボタンが作成され、addObject メソッドでセレクタに追加されます。

```
IlvToggle* toggle;
toggle = new IlvToggle(display, IlvPoint(10, 10), "Toggle 1");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 50), "Toggle 2");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 90), "Toggle 3");
selector->addObject(toggle);
```

セレクタをそのコンテナに追加する

```
container->addObject("Selector", selector);
```

IlvTreeGadget の使用

IlvTreeGadget はアイテムの階層リスト表示するガジェットです。各アイテムは、IlvTreeGadgetItem クラスのインスタンスであり、IlvGadgetItem のサブクラスです。アイテムを展開または折りたたんで、サブアイテムを表示 / 非表示にすることができます。

図 11.29 *ツリー・ガジェット*

IlvTreeGadget クラスは、スクロールバーを処理します。スクロールバーの処理に関する詳細は、ベース・クラス IlvScrolledGadget を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *ツリー階層の変更*
- *ツリー階層でのナビゲーション*
- *アイテム特性の変更*
- *ガジェット・アイテムの展開および折りたたみ*
- *ツリー・ガジェット階層の外観を変更する*

ツリー階層の変更

ツリー・ガジェットには、IlvTreeGadgetItemHolder::getRoot メンバ関数を使用して取得できる非表示のルート・アイテムがあります。

階層の作成

アイテムの階層リストを作成するには、まず、そのリストの一部となるアイテムを作成する必要があります。次にいくつかの例を示します。

```
item1 = new IlvTreeGadgetItem("item1");    // Creates an item with a label.
item2 = new IlvTreeGadgetItem("item1",     // Creates an item with a label
                              bitmap);  // and a bitmap.
item3 = new IlvTreeGadgetItem(bitmap);     // Creates an item with a bitmap.
item4 = new IlvTreeGadgetItem(graphic);    // Creates an item with a graphic.
```

ツリー・ガジェット・アイテムを作成したら、以下の方法でこれらをツリー構造に配列できます。

- 上記のとおり、ツリー・ガジェット・アイテムを作成し、これらを 1 つずつ、メンバ関数 IlvTreeGadget::addItem を使用してツリー・ガジェットに追加します。

- 完全な新規階層を作成し、これを単一操作でツリー・ガジェットに追加します。このソリューションの方が、より効果的です。これを行うには、ツリー・ガジェット・アイテムを上述のように作成し、これらを子として、IlvTreeGadgetItem::insertChild を使用して追加します。それからルート・アイテムを IlvTreeGadget::addItem で追加します。

```
IlvTreeGadgetItem* item = new IlvTreeGadgetItem("New Item");
item->insertChild(new IlvTreeGadgetItem("Leaf 1"));
item->insertChild(new IlvTreeGadgetItem("Leaf2"));
tree->addItem(0 /* tree->getRoot() */, item);
```

ツリー・ガジェット・アイテムの削除

アイテムをツリー・ガジェットから削除するとき、その子もすべてツリーから削除されます。

アイテムを破壊せずに削除するには、IlvTreeGadgetItemHolder::detachItem を使用します。全アイテムを一度に削除するには、IlvTreeGadget::removeAllItems を呼び出します。

ツリー・ガジェット・アイテムの移動

アイテムおよびその子を現在の親アイテムから、新しい親アイテムに、IlvTreeGadgetItemHolder::moveItem で移動させることができます。

ツリー階層でのナビゲーション

ツリー階層を作成したら、メンバ関数 IlvTreeGadgetItem::getParent、IlvTreeGadgetItem::getFirstChild、IlvTreeGadgetItem::getNextSibling、IlvTreeGadgetItem::getPrevSibling を使用してツリー内をナビゲートすることができます。

アイテム特性の変更

IlvTreeGadgetItem の表示特性、そのラベルおよびビットマップなどを変更する場合は、ベース・クラス IlvGadgetItem を参照してください。

アイテムの子がいくつ既知であるかを IlvTreeGadgetItem::setUnknownChildCount で指定することができます。この場合、ツリー・ガジェットでアイテムを [拡大] ボタンで拡大することができます。これにより、アイテムにサブアイテムがないときでも呼び出される拡大コールバックを使えるようになります。こうして、アイテムを拡大コールバックに追加することができます。

ガジェット・アイテムの展開および折りたたみ

ガジェット・アイテムは、[拡大] ボタンをクリックする、あるいはそれをダブルクリックして展開、または折りたたむことができます。アイテムを展開するとは、

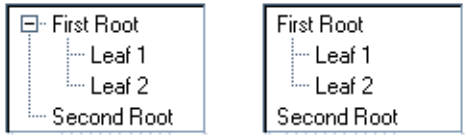
そのサブアイテムを全部表示することです。アイテムを折りたたむとは、そのサブアイテムを全部非表示にすることです。以下のメンバ関数 `IlvTreeGadget::shrinkItem` および `IlvTreeGadget::expandItem` を使用して、同じ操作を行うこともできます。

ツリー・ガジェット階層の外観を変更する

`IlvTreeGadget` 階層表示の方法は、アプリケーションの要件に合わせてカスタマイズできます。これを行うには、以下の `IlvTreeGadget` メンバ関数を使用します。

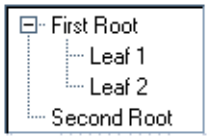
アイテムからこの親へリンクする線は、`IlvTreeGadget::showLines` メンバ関数で、表示にしたり非表示にしたりできます。

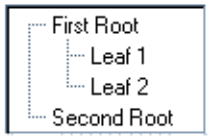
ルート・アイテムとその子を接続するために、`IlvTreeGadget::setLinesAtRoot` メンバ関数を使用して線を描くことができます。



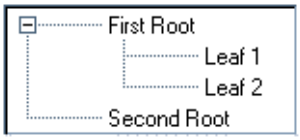
ルート・アイテムの子を接続する (あるいは切り離す) ために、`IlvTreeGadget::linkRoots` メンバ関数を使用して、線を描くことができます。

展開 / 折りたたみ用ボタンは、`IlvTreeGadget::showButtons` メンバ関数を使用して、表示あるいは非表示に設定することができます。





アイテムとその親との間のインデントを、IlvTreeGadget::setIndent メンバ関数を使用して定義することができます。



ラベルおよびピクチャの可視性

ツリー・ガジェットのすべてのピクチャの可視性をこのメソッドを呼び出して変更することができます。

```
void showPicture(IlBoolean value = IlTrue,
                 IlBoolean redraw = IlTrue);
```

同様に、ツリー・ガジェットのすべてのラベルの可視性をこのメソッドを呼び出して変更することができます。

```
void showLabel(IlBoolean value = IlTrue,
               IlBoolean redraw = IlTrue);
```

デフォルトでは、ツリー・ガジェットはラベルおよびピクチャの両方を表示します。

> ***メモ:*** IlvGadgetItem *クラスの API を通じて、特定のアイテムのグローバル設定をオーバーライドすることができます。詳細については、メソッド* IlvGadgetItem::showLabel *および* IlvGadgetItem::showPicture *を参照してください。*

ラベルおよびピクチャの位置

ピクチャに関連するアイテム・ラベルの位置を変更する場合があります。これを行うには、次のメソッドを使用します。

```
void setLabelPosition(IlvPosition position,
                      IlBoolean redraw = IlTrue);
```

デフォルトでは、ラベルはピクチャの右に置かれます (IlvRight)。

> ***メモ:*** IlvGadgetItem *クラスの API を通じて、特定のアイテムのグローバル設定をオーバーライドすることができます。詳細については、* IlvGadgetItem::setLabelPosition method *を参照してください。*

イベントの処理とコールバック

ツリー・ガジェットには、いくつかの定義済みコールバックがあります。コールバックは常に、特定のアイテムに関連しています。コードのコールバックに関連付けられているアイテムを取得するには、メンバ関数 IlvTreeGadget::getCallbackItem を使用します。

選択モード

ツリー・ガジェットには、2 つの異なる選択モードがあります。

- **単一選択モード**　一度に 1 つのアイテムのみ選択することができます。
- **拡張選択モード**　複数のアイテムを選択することができ、選択を拡張できます。

選択モードは、以下のタイプで定義されています。

```
enum IlvTreeSelectionMode
  {
   IlvTreeExtendedSelection = 0,
   IlvTreeSingleSelection   = 1
  }
```

選択モードを変更するには、次の IlvTreeGadget メンバ関数を使用します。

```
IlvTreeSelectionMode getSelectionMode() const;
void setSelectionMode(IlvTreeSelectionMode mode);
```

Select コールバック

ユーザがアイテムを選択、あるいは選択を解除すると、Select コールバックが呼び出されます。そのタイプは、メンバ関数 IlvTreeGadget::SelectCallbackType で取得できます。217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

Expand コールバック

ユーザがアイテムを展開すると、Expand コールバックが呼び出されます。そのタイプは、メンバ関数 IlvTreeGadget::ExpandCallbackType で取得できます。217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

Shrink コールバック

ユーザがアイテムを折りたたむと、Shrink コールバックが呼び出されます。そのタイプは、メンバ関数 `IlvTreeGadget::ShrinkCallbackType` で取得できます。217 ページの*ガジェットにコールバックを関連付ける*を参照してください。

Activate コールバック

ユーザがサブアイテムのないアイテムをダブルクリックすると、Activate コールバックが呼び出されます。そのタイプは、`IlvGadgetItemHolder::ActivateCallbackType` で取得することができます。217 ページの*ガジェットにコールバックを関連付ける*を参照してください。

ツリー・ガジェット・アイテムの編集

ツリー・ガジェット・アイテムを編集することができます。296 ページの*ガジェット・アイテムの編集*を参照してください。

アイテムのドラッグ・アンド・ドロップ

`IlvTreeGadget` クラスは、簡単に使用できるドラッグ・アンド・ドロップ機構を提供します。297 ページの*ガジェット・アイテムのドラッグ・アンド・ドロップ*を参照してください。

章

12

ガジェット・アイテム

ほとんどのガジェットは、IlvGadgetItem クラスで定義されたアイテムから構成されています。

この章では、ガジェット・アイテムを紹介し、その使用方法について説明します。以下のトピックから構成されています。

- *ガジェット・アイテムの概要*
- *ガジェット・アイテムの使用*
- *ガジェット・アイテム・ホルダ*
- *リスト・ガジェット・アイテム・ホルダ*

ガジェット・アイテムの概要

ガジェット・アイテムは IlvGadgetItem クラスのオブジェクトです。ガジェット・アイテムは、ラベル、ピクチャ、あるいはその両方で表されるガジェット要素です。これらをドラッグ・アンド・ドロップして対話的に編集することができます。ツールチップを表示させたり、ローカライズすることもできます。218 ページの*ガジェットのローカライズ*を参照してください。

ガジェット・アイテムは、振る舞いを実装しません。振る舞いは、これを管理するガジェットによって制御されます。

ガジェット・アイテムは、次のガジェット・クラスおよびその派生クラスによって処理されます。

- `IlvAbstractMenu`
- `IlvMatrix`
- `IlvMessageLabel`
- `IlvStringList`
- `IlvTreeGadget`
- `IlvNotebook`

図 12.1 は、アイテムから構成されるいくつかのガジェットを示したものです。左から右へ、ボタン、ツリー・ガジェット、文字列リスト、ポップアップ・メニュー、ツールバー、そしてオプション・メニューです。

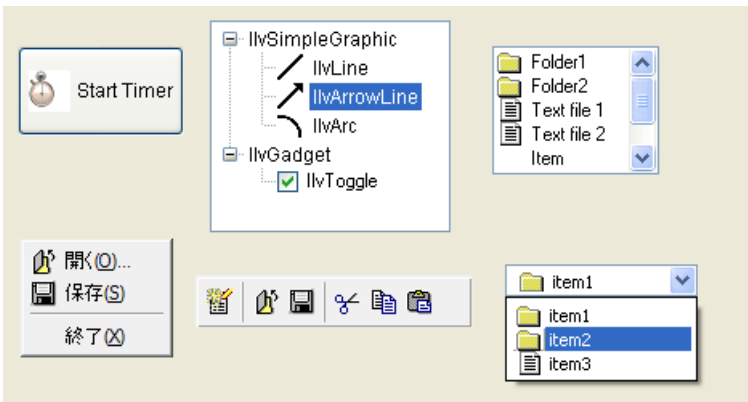


図 12.1 *ガジェット・アイテムで構成されるガジェット*

ガジェット・アイテムの使用

このセクションでは、以下のトピックを取り上げます。

- *ガジェット・アイテムの作成*
- *ラベルの設定*
- *ピクチャの設定*
- *ガジェット・アイテムのレイアウトを指定する*
- *非センシティブなガジェット・アイテム*
- *動的なタイプ*

- *ガジェット・アイテムのあるパレットの使用*
- *ガジェット・アイテムの描画*

ガジェット・アイテムの作成

ガジェット・アイテムは、ラベル、ピクチャ、あるいはその両方で表現されます。ピクチャは、ビットマップあるいはグラフィック・オブジェクトです。290 ページの *ラベルの設定*および 291 ページの *ピクチャの設定*を参照してください。

作成するときにガジェット・アイテムの表示方法を定義することができます。次にいくつかの例を示します。

```
item1 = new IlvGadgetItem("Item1"); // Creates an item with only a label.
item2 = new IlvGadgetItem("Item2",  // Creates an item with a label
                          bitmap);  // and a bitmap.
item3 = new IlvGadgetItem(bitmap);  // Creates an item with a bitmap.
item4 = new IlvGadgetItem("Item 4", // Creates an item with a label
                          graphic); // and an IlvGraphic.
item5 = new IlvGadgetItem(graphic); // Creates an item with an IlvGraphic.
```

ラベルの設定

ガジェット・アイテムはラベルで表現することができます。ラベルをガジェット・アイテムに関連付けるには、メンバ関数 `IlvGadgetItem::setLabel` を使用します。

ガジェット・アイテム・ラベルが複数行に渡る場合は、`IlvGadgetItem::setLabelAlignment` を使用して、テキストを右、左、あるいは中央のいずれに寄せるかを指定することができます。

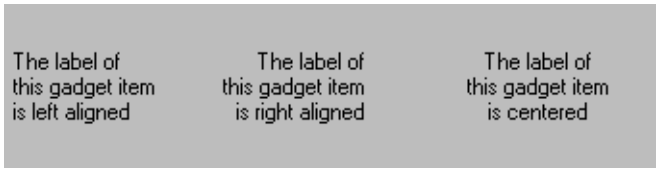


図 12.2 *さまざまな整列によるメッセージ・ラベル*

メンバ関数 `IlvGadgetItem::setLabelOrientation` を使用して、ガジェット・アイテム・ラベルを水平 (デフォルト)、垂直のどちらで描くかを指定することもできます。

図12.3 *垂直に表示されたメッセージ・ラベル*

ピクチャの設定

ガジェット・アイテムは、IlvBitmap あるいは IlvGraphic オブジェクトのいずれかを含むことができます。この後のセクションでは、ガジェット・アイテムがこれらのオブジェクトをどのように処理するかを説明します。

IlvBitmap をピクチャとして使用する

ガジェット・アイテムはビットマップの配列を管理します。配列の各ビットマップは、インデックスあるいはシンボル名によってアクセスできます。ガジェット・アイテムによって表示されているビットマップを、そのビットマップ配列から、IlvGadgetItem::getCurrentBitmap を使用して取得することができます。このメンバ関数は、表示されているビットマップをガジェット・アイテムの状態から決定します。たとえば、ガジェット・アイテムが選択されている場合、「選択された」状態に対応するビットマップが返されます。

次は、ビットマップで表現できるさまざまなガジェット・アイテムの状態と関連付けたシンボル名です。

```
IlvGadgetItem::BitmapSymbol();              Sensitive state
IlvGadgetItem::SelectedBitmapSymbol();      Selected state
IlvGadgetItem::InsensitiveBitmapSymbol();   Nonsensitive state
IlvGadgetItem::HighlightedBitmapSymbol();   Highlighted state
```

いくつのビットマップがガジェット・アイテムに関連付けられているかを調べるには、メソッド IlvGadgetItem::getBitmapCount を使用します。

選択時にガジェット・アイテムによって表示されるビットマップを設定するには、次を呼び出します。

```
item->setBitmap(IlvGadgetItem::SelectedBitmapSymbol(), bitmap);
```

ガジェット・アイテムが非センシティブに設定されたときに表示されるビットマップを取得するには次を呼び出します。

```
IlvBitmap* bitmap = item->getBitmap(IlvGadgetItem::InsensitiveBitmapSymbol());
```

IlvGraphic をピクチャとして使用する

ガジェット・アイテムは IlvGraphic オブジェクトで表現することができます。メンバ関数 IlvGadgetItem::setGraphic を使用してガジェット・アイテムをグラフィック・オブジェクトに関連付けます。

ガジェット・アイテムのレイアウトを指定する

メンバ関数 IlvGadgetItem::setLabelPosition を使用して、そのピクチャに関連するガジェット・アイテム・ラベルの位置を定義することができます。たとえば、ラベルをピクチャの下に置くには、次を呼び出します。

```
item->setLabelPosition(IlvBottom);
```

IlvGadgetItem::setSpacing で、ラベルとピクチャの間のスペースを固定することもできます。たとえば、スペースを 10 ピクセルに設定するには、次を呼び出します。

```
item->setSpacing(10);
```

ガジェット・アイテムは任意の大きさにすることができます。その寸法は自動的に、そのラベル、ピクチャ、ラベル位置、そしてラベルとピクチャの間のスペースから計算されます。ガジェット・アイテムのサイズを取得するには、次のメソッドを使用します。

```
IlvDim width = item->getWidth();
IlvDim height = item->getHeight();
```

> ***メモ:*** *ガジェット・アイテムの幅と高さは、65535 ピクセル以下にします。メンバ関数* getWidth *および* getHeight *は、アイテムがガジェットによって管理されていない場合、*0 *を返します。*

ガジェット・アイテム内のラベルおよびピクチャの位置を調べるには、次のメンバ関数を使用します。

```
item->labelRect(rect, itembbox);// Puts the label bounding box of the item
                                // in rect when the item is drawn in itembbox.
item->pictureRect(rect, itembbox); // Puts the picture bounding box of the item
                                   // in rect when the item is drawn in
itembbox.
```

ガジェット・アイテムを構成するラベルやピクチャは、表示または非表示にすることができます。ガジェット・アイテム・ラベルを非表示にするには、パラメータを IlFalse に設定した IlvGadgetItem::showLabel を使用します。ガジェット・アイテムにピクチャが含まれていない場合、非表示になります。

ピクチャを表示するには、パラメータを IlvTrue に設定した IlvGadgetItem::showPicture メソッドを使用します。

非センシティブなガジェット・アイテム

デフォルトでは、ガジェット・アイテムはセンシティブです。つまり、これらはユーザ・イベントに反応します。パラメータとして IlFalse のあるメンバ関数 IlvGadgetItem::setSensitive を呼び出すと、ガジェット・アイテムを非センシティブにすることができます。この場合、ガジェット・アイテムは画面上で灰色表示され、選択することができません。

センシティブ・ビットマップのみが提供される場合、非センシティブ・ビットマップが自動的に計算されます。非センシティブ・ビットマップが提供されると、このビットマップが使用されます。

動的なタイプ

ガジェット・アイテムは動的にタイプ化されるため、サブクラス化や保存、簡単な読み取りができます。

次のコードで、クラス情報にアクセスできます。

```
IlvClassInfo* classInfo = item->getClassInfo();
```

アイテムのタイプを確認するには、次を使用します。

```
if (item->isSubtypeOf(IlvTreeGadgetItem::ClassInfo())) {
// The item is an IlvTreeGadgetItem.
}
```

ガジェット・アイテムのあるパレットの使用

ガジェット・アイテムを描画するために、いくつかのパレットが使用されます。

- IlvGadgetItem::getOpaquePalette を使用して返されるパレットが、アイテムが不透明のときに使用されます。
- IlvGadgetItem::getSelectionPalette によって返されるパレットは、選択したガジェット・アイテムの背景を描画するために使用されます。
- IlvGadgetItem::getSelectionTextPalette によって返されるパレットは、選択したガジェット・アイテムのテキストを描画するために使用されます。
- IlvGadgetItem::getHighlightTextPalette によって返されるパレットは、強調表示されたガジェット・アイテムのテキストを描画するために使用されます。
- IlvGadgetItem::getInsensitivePalette によって返されるパレットは、非センシティブ・ガジェット・アイテムを描画するために使用されます。

- IlvGadgetItem::getNormalTextPalette によって返されるパレットは、選択されていないガジェット・アイテムのテキストを描画するために使用されます。

デフォルトでは、ガジェット・アイテムはそのホルダのパレットを使用します。ただし、ガジェット・アイテムに関連付けられているパレットを変更することができます。こうして同じガジェット内に異なるパレットのあるガジェット・アイテムを持つことができます。

任意のガジェット・アイテムに割り当てられているパレットを変更するには、以下のメンバ関数を使用します。

- IlvGadgetItem::setNormalTextPalette
- IlvGadgetItem::setSelectionTextPalette
- IlvGadgetItem::setHighlightTextPalette
- IlvGadgetItem::setOpaquePalette

ガジェット・アイテムの描画

ガジェット・アイテムを描画するには、仮想メンバ関数 IlvGadgetItem::draw が呼び出されます。ガジェット・アイテムの描画方法をカスタマイズするために、サブクラス内でこれをオーバーライドできます。

ガジェット・アイテム・ホルダ

ガジェット・アイテム・ホルダは、管理ガジェット・アイテムの抽象クラスであるクラス IlvGadgetItemHolder のオブジェクトです。ガジェット・アイテム・ホルダにリンクされていない場合、ガジェット・アイテムはそのサイズを計算することができず、描画されません。通常、ガジェット・アイテムをそのホルダにリンクさせる必要はありません。この操作は、自動的に管理ガジェットが行います。

ガジェット・アイテムに関する詳細は、289 ページの *ガジェット・アイテムの使用*を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *ガジェット・アイテムの機能*
- *ガジェット・アイテムの検索*
- *ガジェット・アイテムの再描画*
- *ガジェット・アイテムの作成*
- *ガジェット・アイテムの編集*

ガジェット・アイテムの機能

ガジェット・アイテム上でグローバル操作を実行する場合、ガジェット・アイテム上よりもホルダ上の対応する関数を呼び出す方が便利です。たとえば、非表示にしたいピクチャのあるリスト上の各アイテムについて

```
item->showPicture(IlFalse);
```

を呼び出すのは厄介です。

この理由から、ガジェット・アイテムは、そのホルダから特定の機能を継承します。任意の機能がガジェット・アイテム・レベルで再定義されていない場合は、ガジェット・アイテムはこれをそのホルダから受け取ります。編集可能な状態、ラベル、ピクチャの可視性、ラベルの位置、ラベルの向きなどがこれに該当します。

たとえば、ツールバーのすべてのピクチャを非表示にしたい場合 (`IlvToolBar` は、`IlvGadgetItemHolder` のサブクラス)、単に次を呼び出します。

```
toolbar->showPicture(IlFalse);
```

さらに、この選択をツールバーの 4 番目のアイテムにオーバーライドさせ、このピクチャを表示したい場合は、次を呼び出します。

```
toolbar->getItem(3)->showPicture(IlTrue);
```

ガジェット・アイテムの検索

メンバ関数 `IlvGadgetItemHolder::getItemByName` を使うと、アイテムをその名前から検索することができます。このメソッドは、ツリー構造の一部であるアイテムを検索するときに特に便利です (`IlvAbstractMenu` あるいは `IlvTreeGadget`)。

ガジェット・アイテムの再描画

ガジェット・アイテムのグラフィック表示を、`IlvGadgetItem` メンバ関数の 1 つを使用して変更するとき、ガジェット・アイテムは自動的に再描画されます。

次の例では、`setLabel` を呼び出して変更されたリストの領域を再描画します。

```
IlvStringList* list = ...
list->getItem(0)->setLabel("First Item");
```

同時に複数のグラフィック表示の変更を適用したい場合、次のようにガジェット・アイテム・ホルダの再描画メカニズムを使用することができます。

```
IlvStringList* list = ...
list->initReDrawItems();
list->getItem(0)->setLabel("First Item");
list->getItem(1)->setLabel("Second Item");
list->reDrawItems();
```

再描画操作は、IlvGadgetItemHolder::reDrawItems メソッドが呼び出されたときにのみ実行されます。

ガジェット・アイテムの作成

IlvGadgetItemHolder クラスは、指定したラベル、ビットマップ、あるいは IlvGraphic オブジェクトからアイテムを作成するメソッドを含みます。

```
virtual IlvGadgetItem* createItem(const char* label,
                                  IlvGraphic* g = 0,
                                  IlvBitmap* bitmap = 0,
                                  IlvBitmap* sbitmap = 0,
                                  IlBoolean copy = IlTrue)const;
```

このメソッドは、パラメータとして渡されたラベル、グラフィック、あるいはビットマップを使用して IlvGadgetItem オブジェクトを作成します。IlvGadgetItem のサブクラスを返すために IlvGadgetItemHolder のサブクラスでオーバーライドできます。これは、IlvTreeGadgetItem のインスタンスを返すために createItem が再定義されたツリー・ガジェットの場合です。

ガジェット・アイテムの編集

IlvGadgetItemHolder クラスは、次のガジェット・クラスのガジェット・アイテム編集をサポートしています。IlvMatrix、IlvStringList、および IlvTreeGadget。

編集を有効にする

ガジェット・アイテムを編集可能にするには、メンバ関数 IlvGadgetItem::setEditable を、そのパラメータとしての IlTrue と一緒に呼び出さなくてはなりません。どちらのクラスにガジェット・アイテムが属するかに応じて、IlvMatrix::allowEdit、IlvStringList::setEditable、あるいは IlvTreeGadget::setEditable のいずれかで、管理がジェットのレベルで編集可能にすることもできます。

たとえば、次のコードにより、文字列リストの 2 番目のアイテム以外の全ガジェット・アイテムの編集が可能になります (インデックス番号 1 で指定)。

```
slist->setEditable(IlTrue);
slist->getItem(1)->setEditable(IlFalse);
```

> ***メモ :*** *Start Edit コールバックから編集を可能にすることもできます。297 ページの編集の制御 を参照してください。*

ガジェット・アイテムの編集

編集が可能になると、ガジェット・アイテムをクリックする、あるいは F2 キーを選択しながら押す、のいずれの操作で対話的にガジェット・アイテムを編集することができます。メンバ関数 `IlvGadgetItem::edit` を使用すると、コードでアイテムを編集することもできます。

編集の制御

アイテムの編集時には、2 つのコールバックが呼び出されます。

- Start Edit Item が、編集プロセスの始めで呼び出されます。Start Edit Item コールバックを設定するには、メンバ関数 `IlvGadgetItemHolder::StartEditItemCallbackType` によって返されるシンボルを使用します。

  このコールバック内でアイテムを編集不可に設定して、操作をキャンセルすることができます。

- Endt Edit Item が、編集プロセスの終わりで呼び出されます。End Edit Item コールバックを設定するには、メンバ関数 `IlvGadgetItemHolder::EndEditItemCallbackType` によって返されるシンボルを使用します。

  Escape キーを押して、アイテムの編集をキャンセルすることができます。この場合、End Edit Item コールバックは呼び出されません。

グラフィック・オブジェクトの「コールバック」を参照してください。

ガジェット・アイテムのドラッグ・アンド・ドロップ

`IlvGadgetItemHolder` クラスは、ドラッグ・アンド・ドロップ機能を備えています。`IlvMatrix`、`IlvStringList`、および `IlvTreeGadget` のインスタンスのみが、ガジェット・アイテムのドラッグ・アンド・ドロップ機能をサポートしています。

ドラッグ・アンド・ドロップを有効にする

ドラッグ・アンド・ドロップ機能をガジェット・アイテムで有効にするには、次のメンバ関数の 1 つを、そのパラメータとしての `IlTrue` と一緒に呼び出さなくてはなりません。`IlvMatrix::allowDragDrop`、`IlvStringList::allowDragDrop`、あるいは `IlvTreeGadget::allowDragDrop`。

ドラッグ・アンド・ドロップの制御

ドラッグ・アンド・ドロップ機能が有効になると、ガジェット・アイテムを現在の位置からドラッグして任意の場所にドロップさせることができます。次のコールバックが呼び出されます。

- Start Drag Item が、ドラッグ・アンド・ドロップ・イベントの開始時に呼び出されます。このコールバックを設定するには、`IlvGadgetItemHolder::StartDragItemCallbackType` によって返されるシンボルを使用します。

  このコールバックから、メンバ関数 `IlvGadgetItemHolder::setDraggedItem` をそのパラメータとしての 0 とともに呼び出して操作をキャンセルすることができます。

- Drag Item は、マウスが動かされるたびに呼び出されます。このコールバックを設定するには、`IlvGadgetItemHolder::DragItemCallbackType` によって返されるシンボルを使用します。

- End Drag Item は、ドラッグされたアイテムが作業領域のどこかにドロップされたときに呼び出されます。
  このコールバックを設定するには、`IlvGadgetItemHolder::EndDragItemCallbackType` によって返されるシンボルを使用します。

ドラッグ・アンド・ドロップ操作の間、ドラッグしているアイテムを `IlvGadgetItemHolder::getDraggedItem` を使用して取得することができます。ドラッグされているアイテムのゴースト像を変更することもできます。デフォルトでは、ゴースト像は XOR モードで描画されたドラッグされたアイテムです。独自のゴースト像を使用するには、`IlvGadgetItemHolder::setDraggedImage` を Start Drag Item あるいは Drag Item コールバックから呼び出します。

リスト・ガジェット・アイテム・ホルダ

リスト・ガジェット・アイテム・ホルダは、ガジェット・アイテムのリストを管理する特殊なタイプのガジェット・アイテム・ホルダです。クラス `IlvListGadgetItemHolder` は、文字列リストおよびメニューなどのガジェット・アイテム・リストを処理するすべてのガジェットのベース・クラスです。

ガジェット・アイテム・ホルダについては、294 ページの *ガジェット・アイテム・ホルダ*を参照してください。

このセクションでは、以下のトピックを取り上げます。

- *リストの変更*
- *アイテムへのアクセス*
- *リストの並べ替え*

リストの変更

リストを変更するすべてのメンバ関数は、変更領域を自動的に再描画します。変更が行われるたびにその領域を再描画せずに、複数の変更をリストに加えたい場合、IlvGadgetItemHolder クラスの再描画メカニズムを使用することができます。

詳細については、295 ページの *ガジェット・アイテムの再描画* を参照してください。

アイテムをリストに追加する

アイテムをリストに追加するには、いくつかのメンバ関数が利用できます。もっとも重要なものは、IlvListGadgetItemHolder::insertItem です。このメンバ関数は、アイテムをリストの指定した位置に挿入します。addLabel および insertLabel などのその他のメソッドは、IlvGadgetItemHolder::createItem メソッドを使ってアイテムを作成後に insertItem メソッドを呼び出します。詳細については、296 ページの *ガジェット・アイテムの作成* を参照してください。

例：

```
IlvStringList* list = ....
list->insertLabel("Label 1");
```

これは次と等しくなります。

```
IlvStringList* list = ....
IlvGadgetItem* item = list->createItem("Label 1");
list->insertItem(item);
```

リストの全アイテムを変更する

一時にリストの全アイテムを変更したい場合もあります。これを行うには、IlvListGadgetItemHolder::setItems メソッドを使用します。このメソッドを使用すると、ひとつずつアイテムを追加するよりも効率的です。

次に、メソッド IlvListGadgetItemHolder::setItems の使用方法を説明します。

```
IlvUShort count = 3;
IlvGadgetItem** items = new IlvGadgetItem*[count];
items[0] = new IlvGadgetItem("item0");
items[1] = new IlvGadgetItem("item1");
items[2] = new IlvGadgetItem("item2");
IlvStringList* list = ...
list->setItems(items, count);
delete [] items;
```

アイテムはコピーされず、items 配列をホルダで使用しないため、削除する必要がある点に注意してください。

setLabels メソッドなど他のメンバ関数を使用して、リスト全体を変更することができます。これらすべての関数は、メンバ関数 IlvListGadgetItemHolder::setItems を呼び出します。

リストからアイテムを削除する

リストからアイテムを削除するには、IlvListGadgetItemHolder::removeItem メンバ関数を使用します。

全アイテムの削除

リストからすべてのアイテムを削除するには、
IlvListGadgetItemHolder::empty メンバ関数を使用します。

アイテムへのアクセス

リスト・ガジェット・アイテム・ホルダによって管理されているアイテムの数を調べるには、IlvListGadgetItemHolder::getCardinal メンバ関数を使用します。

リスト内での位置を使用してアイテムを取得するには、
IlvListGadgetItemHolder::getItem メンバ関数を使用します。

ホルダ内でのアイテムの位置を調べるには、
IlvListGadgetItemHolder::getIndex メンバ関数を使用します。

アイテムのラベルがわかっている場合は、
IlvListGadgetItemHolder::getPosition メンバ関数を使用して、そのアイテムを見つけることもできます。

リストの並べ替え

メンバ関数 IlvListGadgetItemHolder::sort を使用してリストを並べ替えることができます。これは、パラメータとして比較関数を取得します。独自の比較関数を提供しない場合は、仮想メンバ関数
IlvListGadgetItemHolder::compareItems が使用されます。このメソッドでは単に、strcmp 関数を使用して 2 つの文字列を比較し、その結果を返します。

他の関数を使用する場合は、sort の呼び出しで指定するか、compareItems メンバ関数を IlvListGadgetItemHolder のサブクラス内で再定義します。

次は、降順でアイテムを並べ替えるリスト比較関数の例です。

```
int MyCompareFunction(const char* string1,
                      const char* string2,
                      IlvAny,
                      IlvAny)
{
   return -strcmp(string1, string2);
}
```

章

13

メニュー、メニュー・バーとツールバー

IBM® ILOG® Views Gadgets ライブラリは、メニュー・アイテムを処理するメニューおよびツールバーを作成するためのクラスを提供しています。

この章では、以下のトピックを取り上げます。

- *メニュー、メニュー・バーとツールバーの概要*
- *メニューおよびメニュー・アイテム*
- *ポップアップ・メニュー*
- *メニュー・バーとツールバー*

メニュー、メニュー・バーとツールバーの概要

メニューは、ユーザにコマンドのセットを提供します。ユーザがメニューあるいはツールバーのアイテムを選択すると、特定のアクションが直ちに行われる、あるいはアクションが実行される前にダイアログ・ボックスが表示され、ユーザは追加情報を提供するよう求められます。メニューはメニュー・バーやツールバーに付加することも、スタンドアロンにすることもできます。

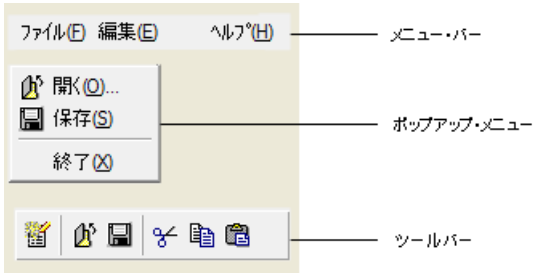


図13.1 *メニュー*

メニューおよびメニュー・アイテム

このセクションでは、メニューおよびメニュー・アイテムを定義するためのクラスについて説明します。以下のトピックから構成されています。

- *IlvAbstractMenu の使用*
- *IlvMenuItem の使用*

IlvAbstractMenu の使用

クラス `IlvAbstractMenu`、`IlvGadget` のサブクラスは、メニュー・バー、ツールバー、およびポップアップ・メニューに共通インターフェースを定義します。`IlvAbstractMenu` も `IlvListGadgetItemHolder` クラスから継承し、ガジェット・アイテムを処理します。`IlvAbstractMenu` は、`IlvGadgetItem` のサブクラスである `IlvMenuItem` オブジェクトのリストを処理します。

メニュー・アイテムの操作

メニュー・アイテムを操作するメンバ関数は、クラス `IlvListGadgetItemHolder` に定義されています。

コールバック

メニュー・アイテムを選択して強調表示すると、Highlight コールバックが呼び出されます。このコールバックにより、ユーザ選択に応じてアクションが実行されます。たとえば、Highlight コールバックを使って、ポップアップ・メニューのアイテムが強調表示された時に小さなヘルプ・メッセージを表示させることができます。

Highlight コールバックをメンバ関数 `IlvAbstractMenu::HighlightCBSymbol` によって返されるシンボルと一緒に設定することができます。

次は、強調表示されたアイテムのインデックスを簡単に記述する Highlight コールバックの例です。

```
static void
Highlight(IlvGraphic* g, IlvAny any)
{
   // Highlighted item position.
   IlvShort pos = *(IlvShort*)any;
   IlvAbstractMenu* menu = (IlvAbstractMenu*)g;
   if (pos != -1)
     IlvPrint("Item %d highlighted", pos);
   else
     IlvPrint("No item highlighted");
  }
```

> ***メモ:*** IlvShort *にキャストされると、*any *パラメータの値が、強調表示されたメニュー・アイテムの位置になります。どのアイテムも強調表示されていない場合は、*-1 *です。*

イベントの処理

クラス IlvAbstractMenu は、サブクラス内で再定義できる次の仮想メンバ関数を含んでいます。

- IlvAbstractMenu::isSelectable は、メニュー・アイテムが選択可能かどうかを指定します。
- IlvAbstractMenu::selectNext および IlvAbstractMenu::selectPrevious は、ユーザが矢印キーを使用してメニュー内を移動する時に、次あるいは前の選択可能なアイテムを返します。
- IlvAbstractMenu::select および IlvAbstractMenu::unSelect は、指定したアイテムが選択、あるいは選択解除されたときに呼び出されます。

IlvMenuItem の使用

メニュー・バー、ツールバー、およびポップアップ・メニューは、メニュー・アイテムと呼ばれるいくつかのアイテムから構成されます。メニュー・アイテムは、IlvGadgetItem のサブクラスである IlvMenuItem クラスによって実装されています。これらは、ラベル、ビットマップ、あるいは IlvGraphic オブジェクトを表示させることができます。12 章*ガジェット・アイテム*を参照してください。

メニュー・アイテムの作成

次のコード・サンプルは、3 つのメニュー・アイテムを作成します。ラベル、ビットマップ、そして IlvGraphic オブジェクトのあるメニュー・アイテムです。

```
item1 = new IlvMenutItem("item1");        // Creates an item with a label.
item2 = new IlvMenuItem(bitmap);          // Creates an item with a bitmap.
item3 = new IlvMenuItem(graphic);         // Creates an item with a graphic.
```

メニュー・アイテムはセパレータとしても使用できます。セパレータは、メニューでメニュー・アイテムによって表されるコマンドのグループを分ける線です。

```
item4 = new IlvMenuItem();          // Creates a separator.
```

アイテムがセパレータかどうかを IlvMenuItem::getType メンバ関数を使用して次のように調べることができます。

```
if (item->getType() == IlvSeparatorItem) {
...
}
```

サブメニューをメニュー・アイテムに付加する

セパレータでないメニュー・アイテムはサブメニューを表示できます。サブメニューをメニュー・アイテムに付加するには、メンバ関数 IlvMenuItem::setMenu を使用します。メニュー・アイテムがポップアップ・メニューに属している場合、その隣の小さな矢印がサブメニューへのアクセスがあることを示します。

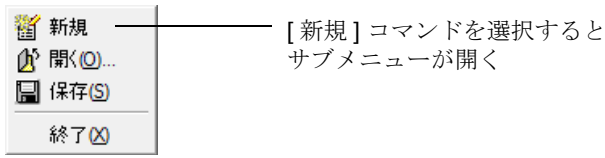


図 13.2　サブメニューのある新規メニュー・アイテム

コールバックをメニュー・アイテムに関連付ける

ユーザがメニュー・アイテムを選択すると、関連付けられているコールバックがそのアクション遂行のために呼び出されます。各メニュー・アイテムは特定のコールバックを有しています。

コールバックをメニュー・アイテムに付加するには、次のメンバ関数のいずれかを使用します。

- `item->setCallback(myCallback);`

  ここで、myCallback は、次のように記述される関数になります。

  ```
  static void
  myCallback(IlvGraphic* g, IlvAny data)
  {
  ....
  }
  ```

  g パラメータは、コールバックをトリガするアイテムのホルダ、つまり、IlvAbstractMenu のサブクラスのインスタンスです。data パラメータは、メンバ関数 IlvGadgetItem::setClientData でインストールすることができるメニュー・アイテムのクライアント・データです。

コールバックをメニュー・アイテム・セパレータ、あるいはサブメニューを有するメニュー・アイテムに設定しても、これらのコールバックは決して呼び出されることがないので無意味です。

- `item->setCallbackName("myCallback");`

  この場合、コールバック名 ÅumyCallbackÅv は、メニューを保持するコンテナに登録する必要があります。

メニュー・アイテムにコールバックがない場合、メニューに関連付けられているメイン・コールバックがあれば、これが呼び出されます。これにより、メニューの各アイテムに同じアクションを実行できます。217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

ニーモニックをメニュー・アイテムに関連付ける

ニーモニック文字をメニュー・アイテムに関連付けることができます。モディファイア・キー (PC では Alt、UNIX では Meta) およびメニューあるいはツールバー・アイテムに関連付けられたニーモニック文字を押すと、付加されているポップアップ・メニューが表示されます。メニューが開いているとき、ニーモニック文字を押すとメニューの該当するコマンドが選択されます。つまり、メニュー・アイテム・コールバックをトリガします。218 ページの *ガジェットのローカライズ*を参照してください。

アクセラレータをメニュー・アイテムに関連付ける

ポップアップ・メニュー・アイテムは、アクセラレータに関連付けることができます。アクセラレータは、モディファイア・キーと文字キーの組み合わせです。ユーザがキーの組み合わせを押すと、メニュー・アイテムコールバックに該当メニューを開くことなく直接アクセスします。

アクセラレータは 2 つの部分からなります。キーの組み合わせと、アクセラレータそのものです。キーの組み合わせは、関連するメニュー・アイテムの横に表示されます。

たとえば、キーの組み合わせ Ctl + A をメニュー・アイテムに割り当てる場合は、次のコードを使用します。

```
item->setAcceleratorText("Ctrl+A");
item->setAcceleratorModifiers(0);
item->setAcceleratorKey(IlvCtrlChar('A'));
```

ポップアップ・メニュー

ポップアップ・メニューは、垂直に配置されたメニューのリストから構成されます。ポップアップ・メニューは、`IlvAbstractMenu` のサブクラスであるクラス `IlvPopupMenu` で実装されています。303 ページの *IlvAbstractMenu の使用*および 304 ページの *IlvMenuItem の使用*を参照してください。

図 13.3 *ポップアップ・メニュー*

このセクションでは、以下のトピックを取り上げます。

- *ポップアップ・メニューでアイテム・ラベルを整列する*
- *切り離しメニューの使用*
- *Open Menu コールバックの使用*
- *チェックの付いたメニュー・アイテムの使用*
- *スタンドアロン・メニューの使用*
- *ポップアップ・メニューでツールチップを使用する*

ポップアップ・メニューでアイテム・ラベルを整列する

デフォルトでは、ポップアップ・メニューのラベルは自動的に、図 13.4 の左端のポップアップ・メニューのように整列されます。ただし、独自のラベル・オフセットをメンバ関数 `IlvPopupMenu::setLabelOffset` で指定することができます。

中央の図は、デフォルト整列モードが無効になり、特定ラベル・オフセットが定義されていないポップアップ・メニューを表しています。右端の図は、40 ピクセルのラベル・オフセットで整列されたポップアップ・メニューを表しています。

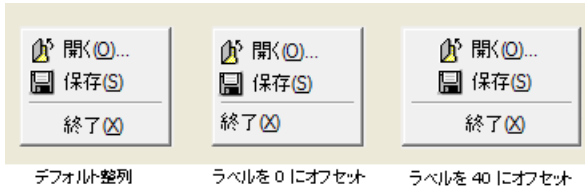


図 13.4 *メニュー・アイテムラベルの整列*

切り離しメニューの使用

ポップアップ・メニューは、切り離すことができます。つまり、メニュー・バーから切り離し、どこにもドッキングしないウィンドウに配置できます。切り離しメニューは、上部の境界を横切る破線で表されます。

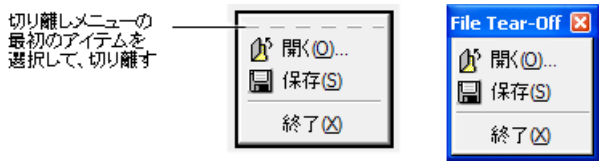


図 13.5 *切り離しメニュー*

最初のアイテムを選択することでポップアップ・メニューを切離すことができます。切り離しポップアップ・メニューの最初のアイテムは、`IlvTearOffItem` タイプである必要があります。メニュー・アイテムを切り離しアイテムとして設定するには、`IlvMenuItem::setTearOff` メンバ関数を使用します。

Open Menu コールバックの使用

ユーザがポップアップ・メニューを開くたびに、Open Menu コールバックが呼び出されます。このコールバックは、たとえば、保存するものがある場合に [保存 (不必要)] から [保存 (必要)] へと変えるなど、アプリケーションの状態に応じてメニューのアイテムを変更する時に特に便利です。これを行うもっとも簡単な方法は、アプリケーションの状態を確認する Open Menu コールバックを設定し、アイテム・ラベルをそれに従って変更することです。

Open Menu コールバックは、メンバ関数
`IlvPopupMenu::OpenMenuCallbackSymbol` で設定することができます。217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

チェックの付いたメニュー・アイテムの使用

ポップアップ・メニューのメニュー・アイテムには、その横に小さなチェック・マーク (Microsoft® Windows® の場合は ✔、Motif® の場合は小さなボタン) を付けることができます。チェック・マークは一般に、「オン / オフ」オプションを表すメニュー・アイテムとともに使用されます。

図 13.6 *チェックの付いたメニュー・アイテム*

メニュー・アイテムにチェック・マークを設定する最善の場所は、Open Menu コールバックです。このようにすると、チェック・マークは、メニューが開かれた時は常に正しい状態になります。308 ページの *Open Menu コールバックの使用*を参照してください。

チェックの付いたアイテムを選択しても、チェック・マークは自動的には消えません。必要に応じてアイテムのチェックを外す必要があります。チェック・マークををメニュー・アイテムに設定するには、メンバ関数 `IlvMenuItem::setChecked` を使用します。`IlvMenuItem::isChecked` を使用して、メニュー・アイテムにチェック・マークが付いているかどうかを調べます。

スタンドアロン・メニューの使用

ポップアップ・メニューは、サブメニューとして、あるいはスタンドアロン・メニューとして使用することができます。ほとんどのスタンドアロン・メニューは、ユーザが作業領域でクリックしたときに表示される (通常は、右マウス・ボタンで) 文脈依存型メニューとして使用されます。

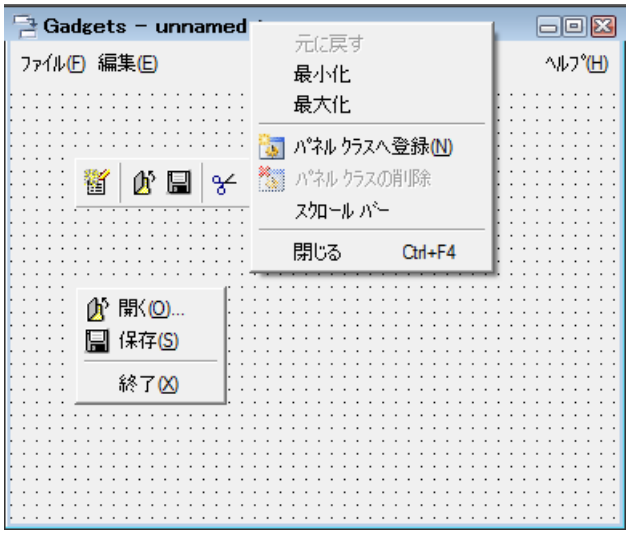


図 13.7 *文脈依存型メニュー*

ポップアップ・メニューをスタンドアロン・メニューとして使用するには、`IlvPopupMenu::get` メンバ関数を使用します。

ユーザが文脈依存型メニューからアイテムを選択すると、メンバ関数 `IlvPopupMenu::doIt` が呼び出されます。

> ***メモ:*** *文脈依存型メニューをサブメニューとすることはできません。*

ポップアップ・メニューでツールチップを使用する

ポップアップ・メニュー内のメニュー・アイテムはツールチップに関連付けることができます。ツールチップは、関連付けられたメニュー・アイテムにマウスを置くと表示される短い説明テキストです。

ツールチップをメニュー・アイテムに設定するには、`IlvMenuItem::setToolTip` を使用します。ツールチップをオフにするには、`IlvPopupMenu::useToolTips` のパラメータを `IlFalse` にして呼び出します。

メニュー・バーとツールバー

IBM® ILOG® Views Gadgets では、メニュー・バーおよびツールバーは、それぞれクラス `IlvMenuBar` および `IlvToolBar` によって実装されています。両方のクラスは、`IlvAbstractMenu` のサブクラスである `IlvAbstractBar` から派生します。

このセクションでは、以下のトピックを取り上げます。

- *IlvAbstractBar の使用*
- *IlvMenuBar および IlvToolBar の使用*

IlvAbstractBar の使用

`IlvAbstractBar` は、メニュー・バーあるいはツールバー・アイテムのサイズおよび位置を管理する抽象クラスです。303 ページの *IlvAbstractMenu の使用*および 304 ページの *IlvMenuItem の使用*を参照してください。

このセクションでは、次のトピックを取り上げます。

- *バーの向きを設定する*
- *バー・ジオメトリの制約*
- *ジオメトリの変更をバーに通知する*
- *デフォルト・アイテム・サイズの設定*
- *アイテムを右寄せに整列する*
- *ドッキング機能の使用*

バーの向きを設定する

バーの向きは、メンバ関数 `IlvAbstractBar::setOrientation` で指定し、`IlvAbstractBar::getOrientation` で取得することができます。

バーは垂直にすることができます。この場合、メニュー・アイテムは上から下へと配置されます。また、水平にすると、アイテムは左から右へと配置されます。

図 13.8　垂直および水平ツールバー

バー・ジオメトリの制約

メンバ関数 `IlvAbstractBar::setConstraintMode` で、バー・ジオメトリを制約すると、どのようなサイズであってもすべてのアイテムを表示することができます。このメンバ関数が `IlTrue` に設定されると、バーは自動的にすべてのアイテムを格納するようにリサイズされます。アイテムは、必要に応じて複数行に展開されることができます。制約モードがオンになっているかどうかを調べるには、`IlvAbstractBar::useConstraintMode` を呼び出します。

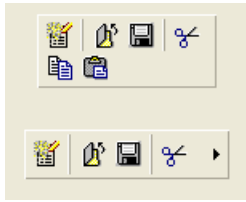


図 13.9　制約 (上) および非制約ツールバー (下)

ジオメトリの変更をバーに通知する

制約モードがオンになっていると、仮想メンバ関数 `IlvAbstractBar::geometryChanged` が、次の場合に呼び出されます。

- 垂直バーの高さを変更すると、幅も変更されます。
- 水平バーの幅を変更すると、高さも変更されます。

デフォルト・アイテム・サイズの設定

バーの全アイテムに対してデフォルト・サイズを、メンバ関数 `IlvAbstractBar::setDefaultItemSize` で設定し、`IlvAbstractBar::getDefaultItemSize` で取得することができます。バーのアイテム間のスペースを `IlvAbstractBar::setSpacing` で指定し、これを `IlvAbstractBar::getSpacing` で取得することができます。

アイテムを右寄せに整列する

メンバ関数 `IlvAbstractBar::setFlushingRight` を使うと、バーの最後のアイテムを右境界に整列させることができます。たとえば、ヘルプ・メニューはたいていの場合右寄せされています。

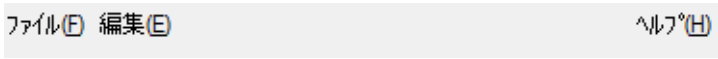


図 13.10 *右寄せに整列されたヘルプ・メニュー*

ドッキング機能の使用

抽象バー・オブジェクトは、ドッキングしたり、ドッキング解除することができます。354 ページの *ドッキング・バーの使用*を参照してください。

IlvMenuBar および IlvToolBar の使用

クラス `IlvMenuBar` および `IlvToolBar` は、メニュー・バーおよびツールバーを定義します。

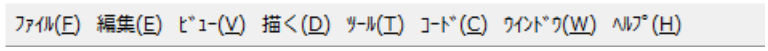


図 13.11 *メニュー・バー*

図 13.12 *ツールバー*

これらのクラスは非常に似ています。唯一の違いは、`IlvToolBar` はツールチップやガジェットなど、`IlvMenuBar` がサポートしていない対話的な機能を提供していることです。

ツールバーでガジェットを管理する

メンバ関数 `IlvGadgetItem::setGraphic` を使うと、ガジェットをツールバー・アイテムとして使用できます。これらのがジェットはアクティブで、ユーザ・イベントに反応します。

メンバ関数 `IlvListGadgetItemHolder::insertGraphic` を使うと、ガジェットをツールバーに追加できます。

図 13.13 は、コンボ・ボックスのあるツールバーを表しています。

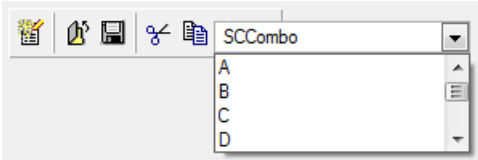


図 13.13　ガジェットのあるツールバー

ツールバーのガジェットをクリックすると、ガジェットにフォーカスが与えられ、すべてのキーボード・イベントが直接それに送られます。211 ページの *フォーカス管理* を参照してください。

`IlvToolBar::setFocusItem` で特定アイテムにフォーカスを強制的に与え、フォーカスのあるガジェットを `IlvToolBar::getFocusItem` で取得することができます。

ツールバーでツールチップを使用する

ツールバーのメニュー・アイテムはツールチップに関連付けることができます。ツールチップは、関連付けられたメニュー・アイテムにマウスを置くと表示される短い説明テキストです。

ツールチップをメニュー・アイテムに設定するには、`IlvMenuItem::setToolTip` を使用します。ツールチップをオフにするには、`IlvToolBar::useToolTips` をパラメータを `IlFalse` に設定して呼び出します。

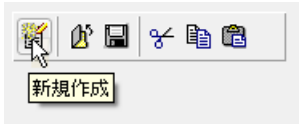


図 13.14　表示されたツールチップ s

章

14

マトリックス

IBM® ILOG® Views Gadgets ライブラリは、マトリックス作成のためにクラスを提供しています。

この章では、以下のトピックを取り上げます。

- *マトリックスの概要*
- *IlvAbstractMatrix の使用*
- *IlvMatrix の使用*
- *IlvSheet の使用*
- *IlvHierarchicalSheet の使用*

マトリックスの概要

マトリックスは、グリッドを形成する行および列から構成される矩形領域です。行と列の交点がセルを形成します。マトリックスは、各種マトリックス・アイテム、ラベル、数値、グラフィック・オブジェクト、ガジェット、あるいはガジェット・アイテムを含むことができます。マトリックスはスクロールバーを持つことができます。

マトリックスを実装する IBM® ILOG® Views Gadgets クラスは、`IlvAbstractMatrix`、`IlvMatrix`、`IlvSheet`、および `IlvHierarchicalSheet` です。

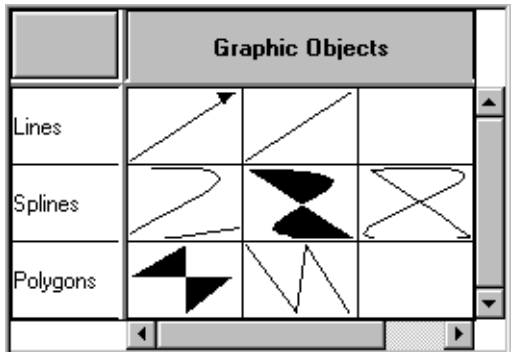


図 14.1 *マトリックス*

IlvAbstractMatrix の使用

クラス `IlvAbstractMatrix` は、マトリックスを描画するための抽象クラスです。そのメンバ関数のいくつかは仮想で、サブクラス内で再定義される必要があります。このクラスにより、アイテムが複数の行または列に渡って展開するべきかどうか、マトリックスがいくつまで固定の行または列を含むかを指定することができます。スクロールも管理します。

このセクションでは、以下のトピックを取り上げます。

- *IlvAbstractMatrix のサブクラス化*
- *複数セルに渡るアイテムの描画*
- *固定行および列の設定*
- *イベントの処理*

IlvAbstractMatrix のサブクラス化

クラス `IlvAbstractMatrix` に値はありません。サブクラスに実装されなくてはならない純粋なメンバ関数のセットを提供します。

- `IlvAbstractMatrix::rows` と `IlvAbstractMatrix::columns` は、マトリックス内の行と列の数を返す必要があります。

- IlvAbstractMatrix::rowSameHeight および IlvAbstractMatrix::columnSameWidth は、すべての行および列が同じ高さと幅になる必要がある場合、IlTrue を返さなくてはなりません。
- IlvAbstractMatrix::getRowHeight および IlvAbstractMatrix::getColumnWidth は、各行の高さおよび各列の幅を返さなくてはなりません。rowSameHeight が IlTrue を返す場合、getRowHeight(0) は行の高さを返し、getColumnWidth(0) は列の幅を返します。
- IlvAbstractMatrix::drawItem は、行および列番号によって定義された指定されたマトリックス内の位置にアイテムを描画します。このメンバ関数は、マトリックス・アイテムのバウンディング・ボックスおよびクリップ矩形も指定します。

複数セルに渡るアイテムの描画

複数の行および列に渡るアイテムを持つことができます。この機能を可能にするには、ブール型メンバ値 _allowCellMode を、IlvAbstractMatrix コンストラクタ内で IlTrue に設定しなくてはなりません。また IlvAbstractMatrix::cellInfo も再定義しなくてはなりません。このメンバ関数は、マトリックス・アイテムがいくつの行および列に渡るのか、およびその左上のセルの位置を指定します。

次の例では、マトリックス・アイテムは (10、10) の位置から始まり、5 行および 5 列に渡るように定義されています。

```
if ((colno >= 10) && (colno < 15) &&
       (rowno >= 10) && (rowno < 15))
   {
      startcol = 10;
      startrow = 10;
      nbcol = 5;
      nbrow = 5;
   }
   else
      IlvAbstractMatrix::cellInfo(colno, rowno,
                                  startcol, startrow,
                                  nbcol, nbrow);
```

メモ: *複数の行および列に渡るアイテムは重ねることはできません。*

このメンバ関数が再定義されると、左上のセルのみが描画されます (IlvAbstractMatrix::drawItem を参照してください)。drawItem メンバ関数に渡される矩形は、マトリックス・アイテムが対象とするすべての行および列を含みます。

固定行および列の設定

マトリックス内で固定のままに留まる行と列の数を指定することができます。固定行および列は、ユーザがマトリックスをスクロールするときでも常に表示されます。一番左にある列と一番上の行のみ固定することができます。

固定行および列を作成するには、IlvAbstractMatrix::setNbFixedRow および IlvAbstractMatrix::setNbFixedColumn を使用します。

イベントの処理

クラス IlvAbstractMatrix は、特定の振る舞いを定義しません。

メンバ関数 IlvAbstractMatrix::handleEvent は、マトリックスがスクロールバーを持つ場合、単にスクロールバーに関するイベントを処理し、IlvAbstractMatrix::handleMatrixEvent を呼び出します。

マトリックスに特定の振る舞いを実装させたい場合、サブクラスのこのメンバ関数を再定義しなくてはなりません。

次のメソッドは、クラスに振る舞いを記述する上で役に立ちます。

```
virtual IlBoolean pointToPosition(const IlvPoint& p,
                                  IlvUShort& colno,
                                  IlvUShort& rowno,
                                  const IlvTransformer* t = 0) const;
```

このメソッドは、colno および rowno で、マトリックスがトランスフォーマ t を使用して表示されるときに、p ポイントの下のアイテムの位置を返します。返される値は、この位置にアイテムがある場合は IlTrue 、ない場合は IlFalse です。

```
IlBoolean rowBBox(IlvUShort rowno,
                  IlvRect& rect,
                  const IlvTransformer* t = 0) const;
IlBoolean columnBBox(IlvUShort colno,
                     IlvRect& rect,
                     const IlvTransformer* t = 0) const;
IlBoolean cellBBox(IlvUShort colno,
                   IlvUShort rowno,
                   IlvRect& rect,
                   const IlvTransformer* t = 0) const;
```

上記のメソッドは、マトリックスがトランスフォーマ t で描画されているとき、rect で列、行、あるいはセルのバウンディング・ボックスを計算します。メソッドは、アイテムが一部でも表示されていると、IlTrue を返し、そうでない場合は IlFalse を返します。

列を再描画するには、IlvAbstractMatrix::invalidateColumn メソッドを使用します。行を再描画するには、IlvAbstractMatrix::invalidateRow メソッドを使用します。

IlvMatrix の使用

マトリックスは、IlvMatrix クラスのインスタンスであり、IlvAbstractMatrix のサブクラスです。マトリックスは行および列から成る矩形グリッドで、多種のオブジェクト (ラベル、グラフィック・オブジェクト、その他のガジェットなど) を含むことができます。これらのオブジェクトは、マトリックス・アイテムと呼ばれ、クラス IlvAbstractMatrixItem に属します。

このセクションでは、以下のトピックを取り上げます。

- *列および行の処理*
- *マトリックス・アイテムの処理*
- *イベントの処理*
- *マトリックスでガジェット・アイテムを使用する*

列および行の処理

このセクションでは、行および列に対して行うことができる各種操作を紹介します。

- *行および列の追加*
- *行および列のリサイズ*
- *自動サイズ調整モードの設定*

行および列の追加

IlvMatrix コンストラクタでマトリックスが含むことのできる行および列の数を指定することができます。

```
IlvMatrix(IlvDisplay* display,
          const IlvRect& rect,
          IlvUShort nbcol,
          IlvUShort nbrow,
          IlvDim xgrid = IlvDefaultMatrixWidth,
          IlvDim ygrid = IlvDefaultMatrixWidth,
          IlvDim thickness = IlvDefaultGadgetThickness,
          IlvPalette* palette = 0);
```

> ***メモ:*** *マトリックスには少なくとも つの行と つの列が含まれなくてはなりません。*

新しい列および行をマトリックスに、IlvMatrix::insertColumn および IlvMatrix::insertRow メンバ関数を使用して追加でき、これらを

IlvMatrix::removeColumn あるいは IlvMatrix::removeRow を使用して削除できます。

メンバ関数 IlvMatrix::reinitialize を使用して、1 回の操作でマトリックスの列および行の数を変更することができます。

行および列のリサイズ

初期設定した列の高さおよび行の幅は、IlvMatrix コンストラクタに提供されている xgrid および ygrid パラメータによって指定されます。マトリックスが作成されたとき、その行および列は既に、これらのパラメータで示されているのと同じ寸法を持っています。ただし、オリジナルの設定は、各列の幅および各行の高さをそれぞれ設定できる IlvMatrix::setXgrid および IlvMatrix::setYgrid メンバ関数で変更することができます。各列および行のサイズを、メンバ関数 IlvMatrix::resizeColumn および IlvMatrix::resizeRow で変更することもできます。この場合、マトリックス用に定義されたグローバル設定は使用されず、これらの値を変更しても他の行および列の寸法には影響しません。

マトリックスのすべての列および行が同じサイズになるように戻すには、IlvMatrix::sameHeight および IlvMatrix::sameWidth を使用します。

自動サイズ調整モードの設定

マトリックスがリサイズされたときに、メンバ関数 IlvMatrix::autoFitToSize で、マトリックスの列および行のサイズを自動的に調整するように要求できます。この機能は、マトリックスにスクロールバーがあるときは利用できません。「自動サイズ調整」モードが設定されると、マトリックスがリサイズされたときに最後の列の幅あるいは最後の行の高さのみが調整されるように、IlvMatrix::adjustLast で指定することができます。

IlvMatrix::fitToSize で、マトリックス・バウンディング・ボックスに合うようにすべての列および行のサイズを再計算することもできます。

マトリックス・アイテムの処理

マトリックス・アイテムは、クラス IlvAbstractMatrixItem のサブクラスのインスタンスです。マトリックス・アイテムは選択したり、編集することができます。マトリックス・アイテムとして使用されるガジェットはアクティブで、ユーザ入力に反応します。

このセクションでは、以下のトピックを取り上げます。

- *定義済みマトリックス・アイテム・クラス*
- *マトリックス・アイテムの新規サブクラスを作成する*
- *マトリックス・アイテムの追加および削除*
- *マトリックス・アイテムの再描画*

- *マトリックス・アイテムの整列*
- *立体マトリックス・アイテムの作成*
- *マトリックス・アイテム選択の設定*
- *マトリックス・アイテムのセンシティビティを変更する*

定義済みマトリックス・アイテム・クラス

次は、IlvAbstractMatrixItem のサブクラスのリストです。

- IlvLabelMatrixItem は、マトリックス・アイテムをラベルとして定義します。
- IlvFilledLabelMatrixItem は、マトリックス・アイテムを背景の塗りつぶされたラベルとして定義します。
- IlvBitmapMatrixItem は、マトリックス・アイテムをビットマップとして定義します。
- IlvIntMatrixItem は、マトリックス・アイテムを整数として定義します。
- IlvFilledIntMatrixItem は、マトリックス・アイテムを背景の塗りつぶされた整数として定義します。
- IlvFloatMatrixItem は、マトリックス・アイテムを浮動小数点値として定義します。
- IlvFilledFloatMatrixItem は、マトリックス・アイテムを背景の塗りつぶされた浮動小数点値として定義します。
- IlvDoubleMatrixItem は、マトリックス・アイテムを倍精度浮動小数点値として定義します。
- IlvFilledDoubleMatrixItem は、マトリックス・アイテムを背景の塗りつぶされた倍精度浮動小数点値として定義します。
- IlvGraphicMatrixItem は、マトリックス・アイテムをグラフィック・オブジェクトとして定義します。
- IlvGadgetMatrixItem は、マトリックス・アイテムをガジェットとして定義します。このタイプのマトリックスは、マトリックス内でアクティブ化できるという点で IlvGraphicMatrixItem オブジェクトとは異なります。
- IlvGadgetItemMatrixItem は、マトリックス・アイテムをガジェット・アイテムとして定義します。

マトリックス・アイテムの新規サブクラスを作成する

IlvAbstractMatrixItem クラス (320 ページの *定義済みマトリックス・アイテム・クラス*参照) の定義済みのサブクラスが、ニーズに合わない場合、独自のマトリックス・アイテム・サブクラスを作成することができます。このセクションでは、新

規マトリックス・アイテム・クラスを正しく登録する方法について説明します。通常は、これによりマトリックス・アイテム・クラスを永続的なものにすることができます。

下に示すコード・サンプルは、サンプル **edit** から抜粋したものです。このサンプルは、ILVHOME/samples/gadgets/table/src/edit.cpp にあります。ILVHOME は、IBM ILOG Views がインストールされたルート・ディレクトリです。

ここで記述されているクラスは、IlvFloatMatrixItem のサブクラスです。これは、表示フォーマットへのアクセスを与えるために、IlvFloatMatrixItem::getFormat メソッドをオーバーライドします。ただし、ここで重要な点はクラスそれ自体ではなく、マクロを使用したクラスの登録です。

```
class FormattedFloatItem : public IlvFloatMatrixItem
{
public:
   FormattedFloatItem(IlFloat value, const IlString& format)
     : IlvFloatMatrixItem(value),
       _format(format)
   {
   }
   void setFormat(const IlString& format)
   {
       _format = format;
   }
   virtual const char* getFormat() const
   {
       return (const char*)_format;
   }
    DeclareMatrixItemInfo();
    DeclareMatrixItemIOConstructors(FormattedFloatItem);
protected:
    IlString _format;
};
```

マクロ DeclareMatrixItemInfo は、クラス情報を処理するのに必要なメソッドおよびメンバを宣言します。

マクロ DeclareMatrixItemIOConstructors は、I/O およびコピー・コンストラクタを宣言します。これらのコンストラクタは、次の方法で定義されます。

```
FormattedFloatItem::FormattedFloatItem(const FormattedFloatItem& source)
    : IlvFloatMatrixItem(source),
      _format(source._format)
{
}
FormattedFloatItem::FormattedFloatItem(IlvDisplay* display,
                                       IlvInputFile& is)
  : IlvFloatMatrixItem(display, is),
    _format()
{
    _format.readQuoted(is.getStream());
}
```

マクロ DeclareMatrixItemInfo がクラス宣言で使用されたため、write マクロが必要です。これは単にスーパークラス write メソッドを呼び出し、_format メンバをストリームに書き出します。IlvAbstractMatrixItem の読み取り専用サブクラスを作成するには、マクロ DeclareMatrixItemInfoRO を使用します。

```
void
FormattedFloatItem::write(IlvOutputFile& os) const
{
    IlvFloatMatrixItem::write(os);
    os.getStream() << " ";
    _format.writeQuoted(os.getStream());
    os.getStream() << " ";
}
```

copy および readItem メソッドの実装は、次のマクロを使用して定義します。

```
IlvPredefinedMatrixItemIOMembers(FormattedFloatItem);
```

これでクラスが IlvFloatMatrixItem のサブクラスとして登録されました。

```
IlvRegisterMatrixItemClass(FormattedFloatItem, IlvFloatMatrixItem);
```

マトリックス・アイテムの追加および削除

アイテムをマトリックスの指定した位置に、メンバ関数 IlvMatrix::set で追加し、IlvMatrix::remove で削除することができます。IlvMatrix::getItem メンバ関数は、与えられたマトリックスの位置にあるアイテムを取得します。

マトリックス・アイテムの再描画

アイテムを追加または削除した、あるいは何らかの方法で変更した後、IlvMatrix::reDrawItem を呼び出してこれを再描画しなくてはなりません。すべての変更が行われるのを待ってから、マトリックス全体を再描画するために IlvGadget::reDraw を呼び出すこともできます。

マトリックス・アイテムの整列

マトリックス・アイテムはセルの中央、あるいはセルの右または左の境界に揃えることができます。

図 14.2 *セルでアイテムを整列する*

マトリックス・アイテムの整列を、メンバ関数 IlvMatrix::setItemAlignment を使用して変更することができます。変更を有効にするためにマトリックス・アイテムを再描画しなくてはなりません。322 ページの *マトリックス・アイテムの再描画*を参照してください。

> ***メモ:*** *ズーム不可グラフィックのある* IlvGadgetMatrixItem *および* IlvGraphicMatrixItem *オブジェクトは、セルの矩形全体を占めており、整列させることはできません。これらのクラスに関しては、320 ページの定義済みマトリックス・アイテム・クラスを参照してください。*

立体マトリックス・アイテムの作成

マトリックス・アイテムに特殊立体効果を作成することができます。立体のアイテムは、ボタンと同じ外観をしています。選択されると、立体マトリックス・アイテムはクリックしたボタンのように見えます。ボタンを立体的に見せるには、IlvMatrix::setItemRelief を使用します。変更を有効にするには、redrawItem メンバ関数を呼び出さなくてはなりません。322 ページの *マトリックス・アイテムの再描画*を参照してください。

マトリックス・アイテム選択の設定

マトリックス・アイテムは選択可能です。選択されたマトリックス・アイテムは、塗りつぶされた矩形で囲まれます。

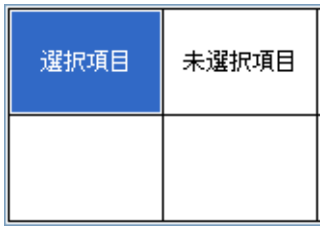


図 14.3 *選択された、および選択されていないマトリックス・アイテム*

単一のマトリックス・アイテムをメンバ関数 IlvMatrix::setItemSelected で選択することができます。行および列全体は、IlvMatrix::setColumnSelected および IlvMatrix::setRowSelected で選択することができます。単一選択モード setSelectedItem は、以前に選択されたアイテムの選択解除は行いません。325 ページの *選択モード*を参照してください。

アイテムを選択したら、それを再描画しなくてはなりません。322 ページの *マトリックス・アイテムの再描画*を参照してください。

選択が描画される方法を、サブクラスのメンバ関数 IlvMatrix::drawSelection をオーバーライドして変更することができます。

マトリックスで選択された最初のアイテムを取得するには、
`IlvMatrix::getFirstSelected` を呼び出します。

マトリックス・アイテムのセンシティビティを変更する

マトリックス・アイテムはセンシティブ、あるいは非センシティブにすることができます。非センシティブ・マトリックス・アイテムは、選択や編集ができず、デフォルトでは下に示すように灰色で表示されます。メンバ関数
`IlvMatrix::setItemGrayed` とパラメータとして `IlFalse` を使用して、これらをセンシティブ・アイテムのように (つまり、灰色ではなく) 表示できます。

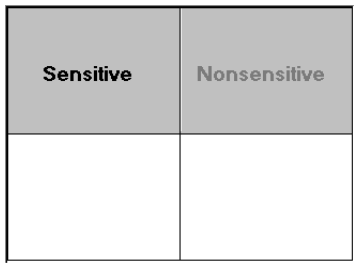


図 14.4 *センシティブおよび非センシティブなマトリックス・アイテム*

アイテムのセンシティビティを変更するには、メンバ関数
`IlvMatrix::setItemSensitive` を使用します。変更を有効にするには、アイテムを再描画しなくてはなりません。322 ページの *マトリックス・アイテムの再描画*を参照してください。

> ***メモ:*** *マトリックス・アイテムがグラフィック・オブジェクトあるいはガジェットである場合、センシティビティの変更は、アイテムの描画に影響しません。*

マトリックス・アイテムを `IlvMatrix::setItemReadOnly` で読み取り専用に設定することもできます。読み取り専用マトリックスは選択することはできますが、編集はできません。

イベントの処理

このセクションでは、メンバ関数 `IlvMatrix::handleMatrixEvent` で実装されている標準的なマトリックスの振る舞いについて説明します。本セクションは、次のような構成になっています。

- *選択モード*
- *マトリックス・アイテムの編集*
- *Item コールバック*

- *Activate コールバック*
- *マトリックスでガジェットを使用する*
- *handleEventMatrix の変更*

選択モード

マトリックスには 2 つの選択モードがあります。単一 (あるいは排他的) 選択または複数選択です。選択モードの設定を行うには、`IlvMatrix::setExclusive` とそのパラメータとして、単一選択モードを指定したい場合は `IlTrue` 、複数選択モードをしたい場合は `IlFalse` とともに使用します。

単一選択モードでは、1 度に 1 つのアイテムだけを選択できます。このモードには、2 つのサブモードがあります。

- 単一選択 - このモードでは、1 度にアイテムを つだけ選択できます。選択が変更されたとき、以前に選択されたアイテムは選択解除されます。
- 単一ブラウズ選択 - このモードは前のモードに似ていますが、真ん中のマウス・ボタンで選択したアイテムをクリックすると、選択が解除されます。

複数選択モードでは、複数のアイテムを同時に選択することができます。このモードには、2 つのサブモードがあります。

- ブラウズ選択 - このモードでは、複数のアイテムをクリックする、あるいはマウスをドラッグさせて複数のアイテムを同時に選択することができます。同様に、複数のアイテムをクリックする、あるいは中央ボタンでマウスをドラッグさせて、複数アイテムの選択を解除することができます。
- 拡張選択 - このモードでは、複数のアイテムをクリックする、あるいはマウスをドラッグさせて、複数のアイテムを同時に選択することができます。アイテムの選択中に、Shift および CTRL キーを押して選択を広げることができます。`IlvMatrix::setExtendedSelectionOrientation` を使用して選択を広げる方向を指定することもできます。

選択サブモードを指定するには、`IlvMatrix::setBrowseMode` を使用します。

マトリックス・アイテムを選択あるいは選択を解除したとき、マトリックスの Main コールバックが呼び出されます。

> ***メモ:*** *センシティブでないアイテムを選択することはできません。*

マトリックス・アイテムの編集

マトリックスの編集が許可されている場合 (`IlvMatrix::allowEdit` を参照)、マトリックス・アイテムを編集することができます。非センシティブあるいは読み取り専用のマトリックスは編集することができません。アイテムが編集されているとき、下の図で示すようにエディタがその上に表示されます。マトリックス・アイテム・エディタのベース・クラスは、`IlvMatrixItemEditor` クラスです。それは、

マトリックス・アイテムを表示し、編集するのに使用される IlvGraphic オブジェクトをカプセル化します。IlvMatrix によって使用されるデフォルト・エディタ・クラスは、IlvDefaultMatrixItemEditor クラスです。それは、マトリックス・アイテムを編集するために IlvTextField オブジェクトを使用します。この振る舞いを、IlvMatrixItemEditorFactory クラスを使用して変更することができます。

図 14.5 *マトリックス・アイテムの編集*

アイテムを編集するには、まずそれをマウスで選択してクリックします、あるいは最後に選択されたアイテムを編集するには、F2 キーを押します。変更を有効にするには、テキスト・フィールドで Enter キーを押す、あるいは他のセルをクリックします。仮想メンバ関数 IlvMatrix::validate が呼び出されます。デフォルト定義では、アイテムがある場合はそれに関連するコールバックが呼び出されます。下の「Item コールバック」を参照してください。アイテムがない場合は、マトリックスの Secondary コールバックを呼び出します。217 ページの *ガジェットにコールバックを関連付ける*を参照してください。

アイテムをコードで編集するには、IlvMatrix::editItem を呼び出します。

IlvMatrix::getEditedItem は、編集中のマトリックス・アイテムの位置を返します。

Item コールバック

各マトリックス・アイテムにコールバックを付加できます。アイテムの編集を有効にすると、これに関連するコールバックが呼び出されます。コールバックは次によって定義されます。

```
typedef void (*IlvMatrixItemCallback)(IlvMatrix* matrix,
                                      IlvUShort  column,
                                      IlvUShort  row,
                                      IlvAny     arg);
```

matrix はアイテムを含むマトリックスを、column および row はコールバックを呼び出すアイテムの位置、そして arg はコールバックをインストールするときに渡される引数を指定します。

コールバックをアイテムに付加するには、IlvMatrix::setItemCallback を使用します。

Activate コールバック

ユーザがアイテムをダブルクリック、あるいは Enter キーを押すと、メンバ関数 `IlvMatrix::activateMatrixItem` が呼び出されます。デフォルトでは、このメソッドは Active Item コールバックを呼び出します。このコールバックを設定するには、`IlvMatrix::ActivateMatrixItemCallbackType` を使用します。217 ページの*ガジェットにコールバックを関連付ける*を参照してください。

> ***メモ:*** *デフォルトでは、マトリックス・アイテムをダブルクリックすると、アイテムが編集可能状態になります。この場合、メンバ関数* `IlvMatrix::activateMatrixItem` *は呼び出されません。このデフォルトの振る舞いをオーバーライドしたい場合、* `IlFalse` *をパラメータとして* `IlvMatrix::allowEditOnDoubleClick` *を呼び出します。*

マトリックスでガジェットを使用する

ガジェット・マトリックス・アイテムは他のアイテムとは異なる振る舞いをします。これらは独自の `handleEvent` メンバ関数を使用してイベントに反応します。つまり、マトリックス内のガジェットが、マトリックス外のガジェットのように振る舞います。マトリックスは、キーボード・フォーカスを持つことができる特殊ガジェット・マトリックス・アイテムを定義します。このアイテムを `IlvMatrix::setFocus` で指定することができます。

矢印キーを使用してマトリックス内をナビゲートしていくと、ガジェット・マトリックス・アイテムを含むセルに到達できます。ここで、ナビゲーションを続けるか、ガジェット・マトリックス・アイテムにイベントを送信するかのいずれかの選択肢があります。ナビゲーションを続けるには、矢印キーを使用してセルを離れます。イベントをガジェット・マトリックス・アイテムに送信するには、ガジェットがキャッチするキーを押す (矢印キー以外のキー) か、あるいは CTRL+Enter を押します。ガジェット・マトリックス・アイテムは、別の CTRL+Enter、あるいはこれが処理しないキーを受け取るまで、すべてのキーボード入力を受け取ります。

handleEventMatrix の変更

メソッド `IlvMatrix::handleMatrixEvent` を再定義して `IlvMatrix` のサブクラス内のマトリックスのデフォルト振る舞いを変更する必要が生じるかもしれません。このような場合は、次のようないくつかのメソッドが役に立ちます。

このメソッドは、マウスがポイントする列および行を返します。

```
virtual IlBoolean pointToPosition(IlvPoint& p,
                                  IlvUShort& c,
                                  IlvUShort& r,
                                  const IlvTransformer* t = 0) const;
```

このメソッドは、マウスがポイントするマトリックス・アイテムを返します。

```
virtual IlvAbstractMatrixItem* pointToItem(IlvPoint&  p,
                                           IlvUShort& c,
                                           IlvUShort& r,
                                           const IlvTransformer* t = 0) const;
```

マトリックスでガジェット・アイテムを使用する

マトリックスは、ガジェット・アイテムを IlvGadgetItemMatrixItem クラス、IlvAbstractMatrixItem のサブクラスを経由して保持することができます。このクラスのインスタンスは、ガジェット・アイテムをカプセル化するので、そのすべての機能から利点が得られます。12 章*ガジェット・アイテム*を参照してください。

ピクチャおよびラベルの可視性

マトリックス内のすべてのピクチャを表示させるかどうかを、メソッド IlvMatrix::showPicture を呼び出して指定することができます。同様に、IlvMatrix::showLabel を使用してそのラベルすべての可視性を定義することができます。デフォルトでは、マトリックスはラベルおよびピクチャの両方を表示します。

> ***メモ:*** IlvGadgetItem *クラスの API を使用して、特定のアイテムのグローバル設定をオーバーライドすることができます。詳細については、メソッド* IlvGadgetItem::showLabel *および* IlvGadgetItem::showPicture *を参照してください。*

ラベルおよびピクチャの位置

ピクチャに関連するアイテム・ラベルの位置を変更する場合があります。これを行うには、メソッド IlvMatrix::setLabelPosition を使用します。デフォルトでは、ラベルはピクチャの右に置かれます (IlvRight)。

> ***メモ:*** IlvGadgetItem *クラスの API を使用して、特定のアイテムのグローバル設定をオーバーライドすることができます。詳細については、*
> IlvGadgetItem::setLabelPosition *を参照してください。*

アイテムの編集

マトリックスに配置されているガジェット・アイテムを編集することができます。296 ページの*ガジェット・アイテムの編集*を参照してください。

アイテムのドラッグ・アンド・ドロップ

IlvMatrix クラスは、簡単に使用できるドラッグ・アンド・ドロップ機構を提供します。297 ページの*ガジェット・アイテムのドラッグ・アンド・ドロップ*を参照してください。

ツールチップ

マトリックスは、マウス・ポインタが部分的に表示されたアイテムの上に置かれたときにツールチップを表示することができます。

IlvSheet の使用

シートは、クラス `IlvSheet` によって実装される特殊なタイプのマトリックスです。318 ページの *IlvMatrix の使用*を参照してください。シートでは、固定行および列は、立体線で区切られています。

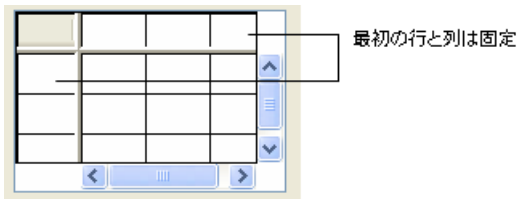


図 14.6 *シート*

`IlvSheet` には、クラス `IlvMatrix` のすべての振る舞いがあります。さらに、これにより動的に列あるいは行をリサイズできます。これを行うには次の 2 つの方法があります。

- 固定列および行のグリッド・ラインをクリックして、ドラッグし、列および行をリサイズできます。
- 固定列あるいは行のグリッド・ライン上をダブルクリックして、列あるいは行に、そのより大きいアイテムのサイズを与えます。

IlvHierarchicalSheet の使用

`IlvHierarchicalSheet` クラスは、その列の 1 つでツリー構造を表示する `IlvSheet` のサブクラスです。これは、複数の列を処理する特殊な `IlvTreeGadget` オブジェクトとして考えることができます。ツリー・アイテムは、タイプ `IlvTreeGadgetItem` であり、ツリー階層を処理するのに使用される API は `IlvTreeGadget` オブジェクトにとても近いということを意味します。281 ページの *IlvTreeGadget の使用*を参照してください。

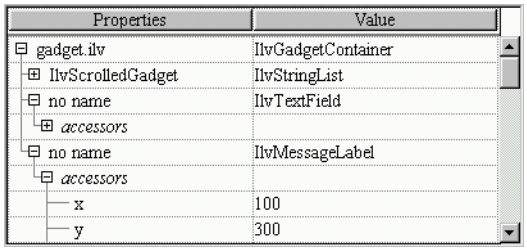


図 14.7 *階層シート*

このセクションでは、以下のトピックを取り上げます。

- *ツリー階層の変更*
- *ツリー階層でのナビゲーション*
- *ツリー・アイテムの特性を変更する*
- *ガジェット・アイテムの展開および折りたたみ*
- *ツリー・ガジェット階層の外観を変更する*
- *イベント処理およびコールバック*

ツリー階層の変更

階層シートには、`IlvHierarchicalSheet::getRoot` メンバ関数を使用して取得できる非表示のルート・アイテムがあります。

階層の変更

ツリー階層を変更したいときは、`set`、`removeRow` などの `IlvSheet` メンバ階層を使用しないでください。その代わりに、次に示すように `IlvHierarchicalSheet` メソッドを使います。

アイテムの階層リストを作成するには、次の手順に従います。

- 282 ページの*階層の作成*で説明されているとおりにツリー・ガジェット・アイテムを作成して、`IlvHierarchicalSheet::addItem` メンバ関数で、これらを 1 つずつ階層シートに追加します。
- 完全な新規階層を作成し、これを単一操作でツリー・ガジェットに追加します。これを行うには、ツリー・ガジェット・アイテムを上で説明したとおりに作成し、これらを `IlvTreeGadgetItem::insertChild` を使用して子として追加します。この解決法は、最初に説明したものに比べ、より効果的です。

その後、新規階層のルートを、`IlvHierarchicalSheet::addItem` を使って、次のようにツリーに追加します。

```
IlvTreeGadgetItem* item = new IlvTreeGadgetItem("New Item");
```

```
item->insertChild(new IlvTreeGadgetItem("Leaf 1"));
item->insertChild(new IlvTreeGadgetItem("Leaf2"));
hsheet->addItem(0 /* hsheet->getRoot() */, item);
```

アイテムの削除

アイテムを階層シートから削除するとき、その子すべてもツリーから削除されます。`IlvHierarchicalSheet::removeItem` を使用して単一アイテムをツリーから削除します。あるいは `IlvHierarchicalSheet::removeAllItems` を使用してそのアイテムすべてを一度に削除します。

> ***メモ:*** *新規アイテムをツリー・ガジェットに追加するとき、その対応する行が自動的に作成されます。同様に、アイテムを削除するときは、その行が削除されます。*

ツリー階層でのナビゲーション

階層ツリー内を移動するには、283 ページの *ツリー階層でのナビゲーション*に記述されているメンバ関数を使用します。`IlvHierarchicalTree::getTreeItem` および `IlvHierarchicalSheet::getItemRow` も使用することができます。

ツリー・アイテムの特性を変更する

283 ページの *アイテム特性の変更*を参照してください。

ガジェット・アイテムの展開および折りたたみ

ガジェット・アイテムの展開ボタンをクリックしてガジェット・アイテムを展開あるいは折りたたむことができます。アイテムを展開するとは、そのサブアイテムを全部表示することです。アイテムを折りたたむとは、そのサブアイテムを全部非表示にすることです。同じ操作を `IlvHierarchicalSheet::shrinkItem` および `IlvHierarchicalSheet::expandItem` を使って行うことができます。

アイテムの親が折りたたまれ、アイテムが非表示になるとき、シートのその対応する行は消えます。しかし、削除されたわけではありません。

ツリー・ガジェット階層の外観を変更する

`IlvHierarchicalSheet::showLines` メンバ関数を使用して、アイテムをその親とリンクする直線を表示したり、非表示にしたりすることができます。

アイテムとその親との間のインデントを、`IlvHierarchicalSheet::setIndent` メンバ関数を使用して定義することができます。

イベント処理およびコールバック

Expand コールバック

ツリー・ガジェット・アイテムを展開すると、Expand コールバックが呼び出されます。コールバック・タイプは、`IlvHierarchicalSheet::ExpandCallbackType` で取得することができます。217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

Shrink コールバック

ツリー・ガジェット・アイテムを折りたたむと、Shrink コールバックが呼び出されます。コールバック・タイプは、`IlvHierarchicalSheet::ShrinkCallbackType` で取得することができます。217 ページの *ガジェットにコールバックを関連付ける* を参照してください。

章

15

ペイン

高度に直観的かつカスタマイズ可能なアプリケーションを作成するために、グラフィカル・パネル、ツールバー、メニュー・バーなどのグラフィカル・ユーザ・インターフェースを構成する多数の要素をさまざまな大きさのペイン内でグループ化することができます。

この章では、ペインの概要とそれをグラフィカル・アプリケーションで使用する方法について説明します。以下のトピックから構成されています。

- *ペインの概要*
- *ペインの作成*
- *ペイン・コンテナにペインを追加する*
- *ペインのリサイズ*

ペインの概要

IBM® ILOG® Views Gadgets ライブラリは、ペインをサポートしています。ペインは、`IlvGraphic` あるいは `IlvView` オブジェクトなどのあらゆる種類の描画を表示するグラフィカル領域です。

ペインは、クラス `IlvPane` のペイン・コンテナに格納されているクラス `IlvPanedContainer` のオブジェクトです。ペイン・コンテナは垂直あるいは水平

にすることができます。ペインは、垂直ペイン・コンテナでは上から下へ、水平ペイン・コンテナでは左から右に配置されます。垂直コンテナ内のペインはすべて同じ幅ですが、高さは異なることがあります。同様に、垂直コンテナ内のペインはすべて同じ幅ですが、高さは異なる場合があります。

ペインは、ペイン・コンテナをカプセル化することができるため、次の図で示すような複雑な入れ子ペイン構造を構築できます。

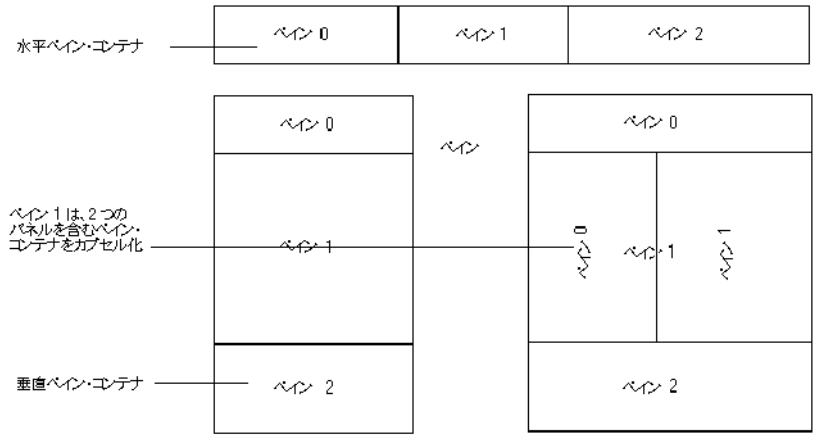


図 15.1 *水平および垂直ペイン・コンテナ、およびカプセル化されたペイン・コンテナ*

次の図は、ペインを実装するアプリケーション・メイン・ウィンドウを表しています。

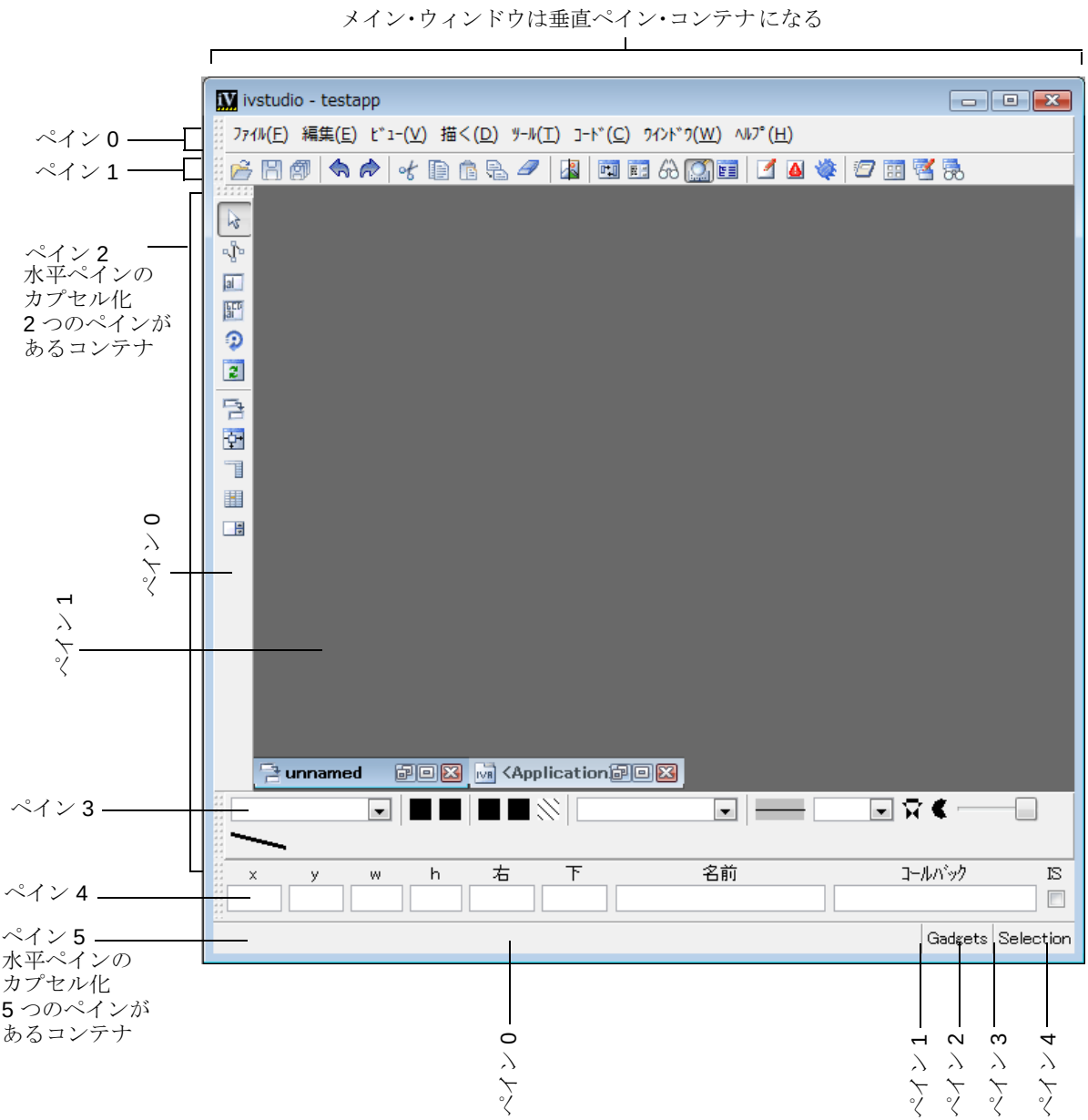


図15.2 *複数のペインから構成されるアプリケーション・メイン・ウィンドウ*

ペインの作成

ペインは、IlvPane クラスのインスタンスです。抽象クラスなので、これをサブクラス化するか、この定義済みのサブクラスの 1 つを使用しなくてはなりません。

- IlvViewPane、これはあらゆる IlvView オブジェクトをカプセル化します。
- IlvGraphicPane、これはあらゆる IlvGraphic オブジェクトをカプセル化します。

その定義済みサブクラスはほとんどすべてアプリケーション要件に適うため、通常は IlvPane クラスをサブクラス化する必要はありません。

このセクションでは、以下のトピックを取り上げます。

- *グラフィック・ペインの作成*
- *ビュー・ペインの作成*
- *ペインの表示 / 非表示*

グラフィック・ペインの作成

次の例は、グラフィック・ペインの作成方法を示しています (IlvGraphicPane)。

まず、グラフィック・ペインに追加したい IlvGraphic オブジェクトを作成します。ここで、IlvTreeGadget オブジェクトをカプセル化します。

```
IlvDisplay* display = ...
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
```

そして、グラフィック・ペインを作成します。

```
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree",tree);
```

コンストラクタに提供される最初の引数は、ペインの名前を表す文字列です。

ビュー・ペインの作成

次の例は、ビュー・ペインの作成方法を示しています (IlvViewPane)。

まず、ビュー・ペインに追加したい IlvView オブジェクトを作成します。

```
IlvView* view = new IlvView(parent, IlvRect(0, 0, 100, 100));
```

そして、ビュー・ペインを作成します。

```
IlvViewPane* viewPane = new IlvViewPane("View", view);
```

コンストラクタに提供される最初の引数は、ペインの名前を表す文字列です。

> ***メモ:*** *ビュー・ペインで使用されるビューは、サブビューである必要があります。カプセル化されたビューが、親ビューを最初の引数として受け取る* `IlvView` *コンストラクタで作成されたのは、この理由によります。*

ペインの表示 / 非表示

`IlvPane::show` および `IlvPane::hide` メンバ関数を用いてペインを表示あるいは非表示にすることができます。非表示のペインは、そのペイン・コンテナに表示されません。

> ***メモ:*** *ペインを追加あるいは削除して、あるいはペインを表示または非表示にしてペイン・コンテナのレイアウトを変更する場合は、変更を有効にするために、* `IlvPanedContainer::updatePanes` *メンバ関数を呼び出す必要があります。*

ペイン・コンテナにペインを追加する

ペイン・コンテナは、`IlvPanedContainer` クラスのインスタンスであると同時に、`IlvGadgetContainer` のサブクラスであり、これにペインを追加する必要があります。

このセクションでは、以下のトピックを取り上げます。

- *ペイン・コンテナの作成*
- *ペイン・コンテナのレイアウトを変更する*
- *ペインの取得*
- *ビュー・ペインでペイン・コンテナをカプセル化する*

ペイン・コンテナの作成

ペイン・コンテナを作成するとき、その方向 (水平あるいは垂直) を指定しなくてはなりません。ペインのコンテナが垂直の場合、ペインは上から下に配列されます。水平ペインでは、左から右に配列されます。

次のコード・サンプルでは、垂直コンテナをトップ・ビューとして垂直ペイン・コンテナを作成します。

```
IlvPanedContainer* container = new IlvPanedContainer(display,
                                                     "Paned Container",
                                                     "Paned Container",
                                                      IlvRect(0, 0, 500, 500),
                                                      IlvVertical);
```

メンバ関数 IlvPanedContainer::getDirection および IlvPanedContainer::setDirection を使用して指定した方向を取得し、これを変更することができます。

ペイン・コンテナを作成すれば、これにペインをメンバ関数 IlvPanedContainer::addPane で追加したり、あるいは IlvPanedContainer::removePane でペインをペイン・コンテナから削除したりできます。

ペイン・コンテナのレイアウトを変更する

ペイン・コンテナの現在のレイアウトを、ペインを追加、削除、表示あるいは非表示にして変更する場合、変更を有効にするために、IlvPanedContainer::updatePanes メンバ関数を呼び出す必要があります。

```
container->addPane(pane1);
container->addPane(pane2);
container->addPane(pane3);
container->updatePanes();
```

ペインの取得

任意のペイン・コンテナが処理するペイン数を調べるのに、メンバ関数 IlvPanedContainer::getCardinal を使用することができます。

IlvPanedContainer::getPane メンバ関数で、ペインのインデックスあるいはその名前を使用してペインを取得できます。

IlvPanedContainer::getIndex メンバ関数を使用して特定のペインのインデックスを取得できます。

> ***メモ:*** *ペイン・コンテナは、インデックスを使用して保持するペインを参照します。しかし、内部的な理由により変更する場合があるため、インデックスを使用してペインを参照することはお勧めしません。代わりに、メンバ関数* IlvPane::setName *を使用して、ペインを識別します。*

ビュー・ペインでペイン・コンテナをカプセル化する

クラス IlvPanedContainer は、IlvGadgetContainer から継承し、それ自体は IlvView のサブクラスであるため、ビュー・ペイン内でペイン・コンテナをカプセル化できます。

ビュー・ペイン内でのペイン・コンテナをカプセル化することにより、335 ページの図 15.1 に示すように複雑な入れ子ペイン構造を作成することができます。

次のコード・サンプルでは、ビュー・ペイン内で水平ペイン・コンテナをカプセル化します。

まず、メイン垂直ペイン・コンテナを作成します。

```
IlvPanedContainer* container = new IlvPanedContainer(display,
                                                     "Paned Container",
                                                     "Paned Container",
                                                      IlvRect(0, 0, 500, 500),
                                                      IlvVertical);
```

それから、水平ペイン・コンテナを作成し、ビュー・ペイン内でこれをカプセル化します。

```
IlvPanedContainer* innerContainer = new IlvPanedContainer(container,
                                                          IlvRect(0, 0, 500,200),
                                                          IlvHorizontal);
IlvViewPane* viewPane = new IlvViewPane("ViewPane", innerContainer);
```

> ***メモ:*** *この例では、*innerContainer *を* container *のサブビューとして作成しています。この練習問題は必須ではありませんが、独自のアプリケーションを作成するときにはこの方法に従うことをお勧めします。*innerContainer *の親として* container *を指定しない場合、*container *に追加されたときに再び親に指定されます。*

最後の手順では、メイン・ペイン・コンテナにビュー・ペインを追加します。

```
container->addPane(viewPane);
```

> ***メモ:*** IlvPanedContainer::getViewPane *メンバ関数を使用して任意のペイン・コンテナをカプセル化するビュー・ペインを取得できます。どのビュー・ペインもペイン・コンテナをカプセル化しない場合、このメンバ関数は* 0 *を返します。*

ペインのリサイズ

ペインはリサイズすることができます。

このセクションでは、以下のトピックを取り上げます。

- *リサイズ・モードおよびペインの最小サイズを設定する*
- *スライダ付きペインのリサイズ*

リサイズ・モードおよびペインの最小サイズを設定する

ペイン・コンテナをリサイズする場合、それが保持するペインは、そのリサイズ・モードに応じてサイズが変わります。ペインには、3 つのリサイズ・モードがあります。

- 固定 - 固定ペインはリサイズできません。
- 伸縮 - 伸縮ペインは常にリサイズできます。
- リサイズ可能 - リサイズ可能ペインは、そのペイン・コンテナが伸縮ペインを含まない場合のみサイズを変更できます。

ペインのリサイズ・モードを設定するには、`IlvPane::setResizeMode` メンバ関数を使用します。デフォルトでは、新規ペインのリサイズ・モードは固定になっています。

ペインに最小サイズを設定することもできます。ペインに最小サイズを設定するには、`IlvPane::setMinimumSize` メンバ関数を使用します。新規ペインの最小サイズは、デフォルトでは 1 です。指定した最小サイズより小さいペインを作成することはできません。

> ***メモ:*** *リサイズ・モードおよびペインの最小サイズは、水平および垂直方向の両方に定義することができます。*

スライダ付きペインのリサイズ

スライダ・ペインは、クラス `IlvSliderPane` のインスタンス、`IlvGraphicPane` のサブクラスで、隣接するペインのリサイズを行うためにドラッグすることができます。

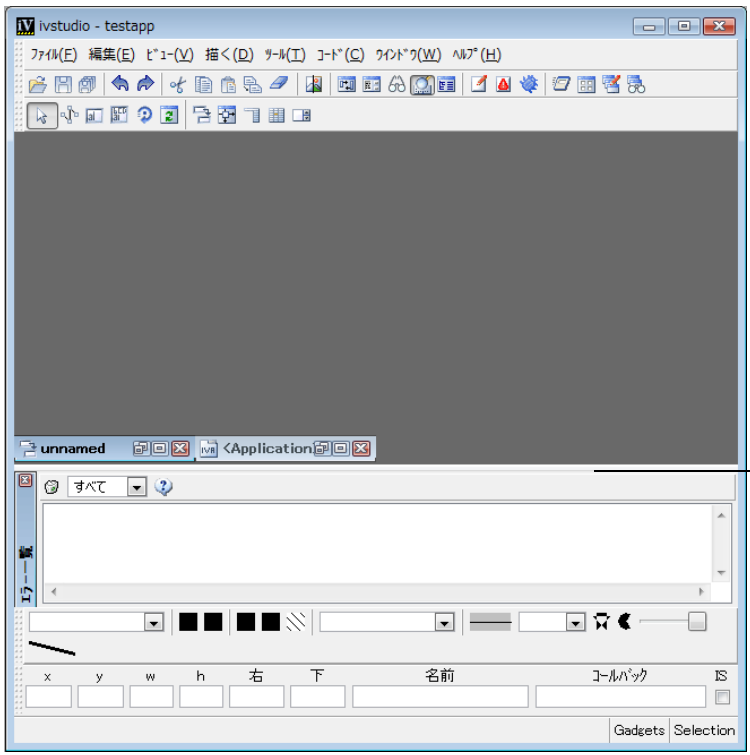


図15.3　スライダ・ペイン

自動スライダ作成の使用

デフォルトでは、ペイン・コンテナは自動的にスライダ・ペインをリサイズ可能ペインと伸縮ペインの間に作成します。

> ***メモ:*** *スライダ・ペインは実際には、メンバ関数* `IlvPanedContainer::updatePanes` *を呼び出した後にのみ作成されます。これはペインに元々割り当てられているインデックス番号に影響するため、とても重要です。たとえば、2 つのサイズ変更可能ペインを追加する空のコンテナを作成する場合、これらのインデックスはそれぞれ 0、1 となります。*`updatePanes` *を呼び出した後、リサイズ可能ペインのインデックスは、0 および2 となり、割り当てられているスライダ・ペインはインデックス番号 1 となります。339 ページのペインの取得を参照してください。*

ペイン・コンテナが自動的にスライダ・ペインを作成するとき、それは `IlvPanedContainer::createSliderPane` メンバ関数を呼び出します。カスタム・スライダ・ペインを作成するためにこれをオーバーライドすることができます。

スライダ・ペインを自動作成しない場合は、
`IlvPanedContainer::manageSliders` メンバ関数を、`false` を引数として、呼び出すことができます。この機能をオフにして、引き続きリサイズ可能および伸縮ペインをスライダを使用してリサイズする場合は、手動でスライダ・ペインを作成して、これらをペイン・コンテナに追加しなくてはなりません。

章

16

ペインおよびコンテナのドッキング

IBM® ILOG® Views Gadgets ライブラリは、ドッキング・ペインをサポートしています。

このセクションでは、ドッキング・ペインの概要とそれをグラフィカル・アプリケーションで使用する方法について説明します。このセクションを読む前に、まずペインについて理解しておいてください。ペインの詳細は、15 章*ペイン*を参照してください。

この章では、以下のトピックを取り上げます。

- *ドッキング・ペインおよびドッキング可能コンテナの概要*
- *ドッキング・ペインの作成*
- *ドッキング操作の制御*
- *ドッキング・バーの使用*
- *ドッキング・ペインがある標準的アプリケーションの作成*

サンプル

ViewFile アプリケーション・チュートリアルを参照してください。

ドッキング・ペインおよびドッキング可能コンテナの概要

ドッキング可能コンテナは、特殊なタイプのペイン・コンテナで、これにペインをドッキングさせたりドッキング解除させたりすることができます。ペインのドッキングとは、任意の場所で対話的にペインをドッキング可能コンテナに追加することです。ペインのドッキング解除とは、そのドッキング可能コンテナからペインを、対話的に特殊なトップ・ビュー内部に移動させることです。

ドッキング・ペインを実装する IBM® ILOG® Views Gadgets ライブラリのクラスは、`IlvDockable` および `IlvDockableContainer` です。

次の図では、グラフィカル・インターフェースを構成するすべてのペインがドッキングされています。

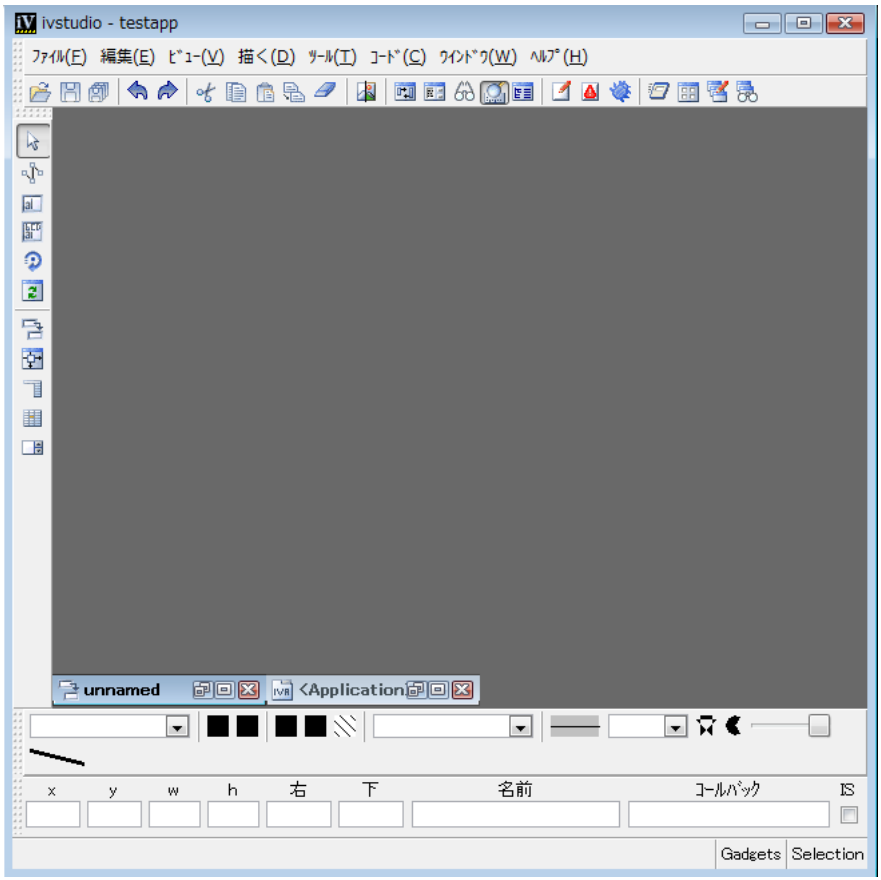


図16.1 *ドッキングされたペインのある GUI*

次の図では、メイン・メニュー・バーがドッキング解除されてトップ・ウィンドウ内で浮動しています。

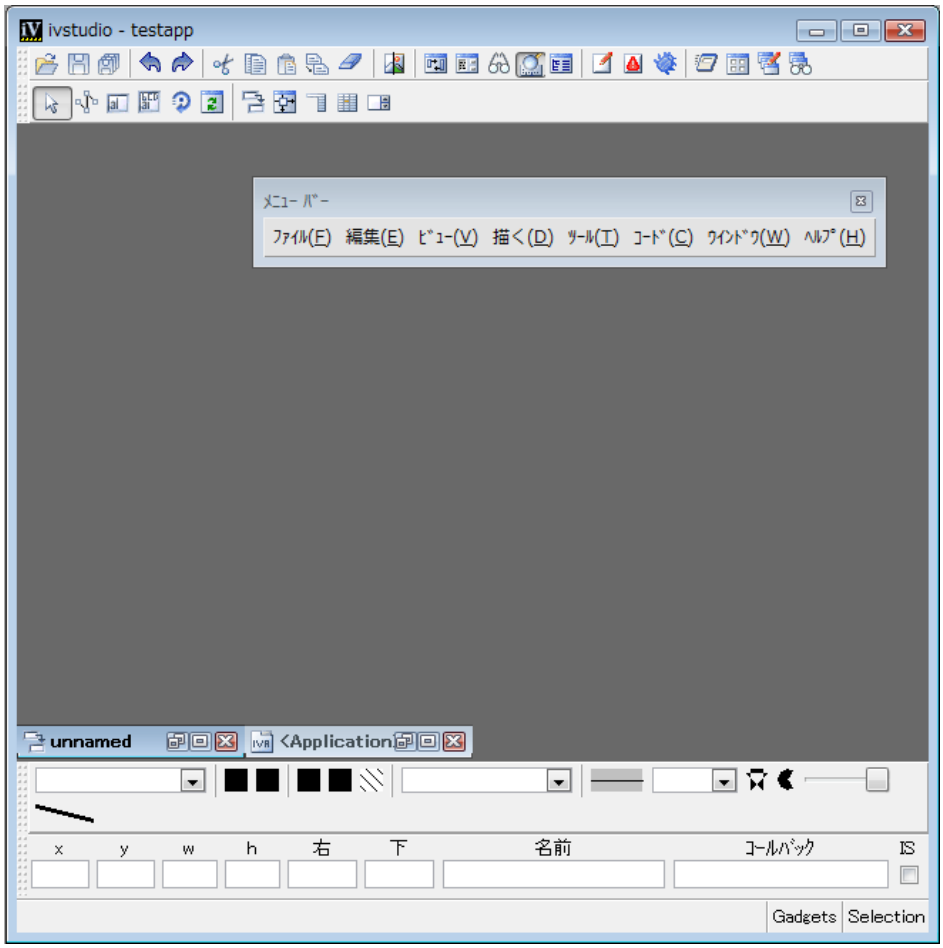


図16.2　ドッキング解除されたメイン・メニュー・バー

次の図では、メイン・メニュー・バーが再びGUIメイン・ウィンドウの右側にドッキングされています。

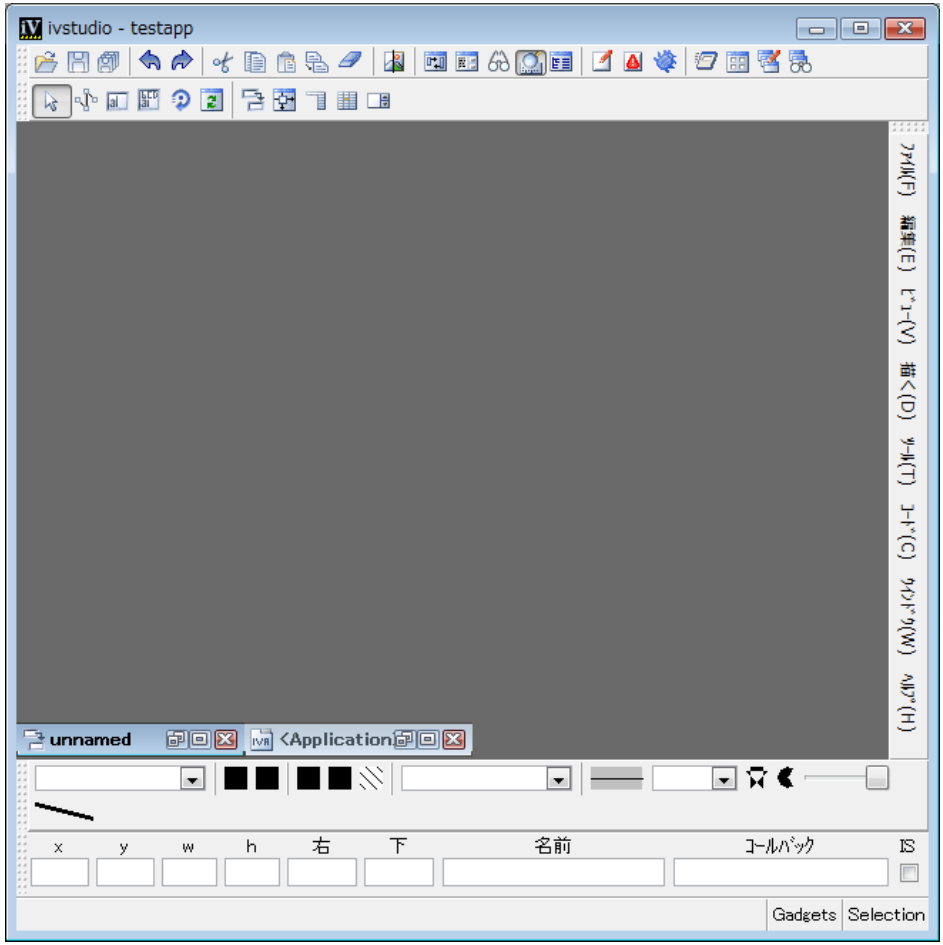


図16.3　再びドッキングされたメイン・メニュー・バー

ドッキング・ペインの作成

ドッキング・ペインの作成は、次の 2 つのコード・サンプルで示すとおり、通常のペインの作成とほとんど同じです。ペインについては、15 章*ペイン*を参照してください。

次のコード・サンプルでは、[ツリー] ペインがペイン・コンテナに addPane メンバ関数で追加されています。

```
IlvPanedContainer* container = new IlvPanedContainer(display,
                                                     "Paned Container",
                                                     "Paned Container",
                                                      IlvRect(0, 0, 500, 500),
                                                      IlvVertical);
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree", tree);
container->addPane(graphicPane);
```

次のコード・サンプルでは、同じ [ツリー] ペインがドッキング可能コンテナに、メンバ関数 IlvDockableContainer::addDockingPane で追加され、これをドッキング可能ペインにします。

```
IlvDockableContainer* container =
                            new IlvDockableContainer(display,
                                                     "Dockable Container",
                                                     "Dockable Container",
                                                     IlvRect(0, 0, 500, 500),
                                                     IlvVertical);
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree", tree);
container->addDockingPane(graphicPane);
```

2 つ目のコード・サンプルでは、ペイン・コンテナは、IlvPanedContainer のサブクラスである IlvDockableContainer です。そしてドッキング・ペインを作成するためにペインがこれにメンバ関数 IlvDockableContainer::addDockingPane で追加されています。

ドッキング・コンテナに addDockingPane メンバ関数で追加されたペインは、IlvDockable クラスのインスタンスに接続され、これがペインのドッキング操作を処理します。このクラスに関する詳細は、352 ページの *ドッキング操作の制御*を参照してください。

> *IlvDockable のサブクラスを使用する場合、addDockingPane を呼び出す前にこれを明示的に接続する必要がある点に留意してください。352 ページの IlvDockable クラスのインスタンスをペインに接続するを参照してください。*

ドッキング・ペインにはハンドルがあり、これをクリック、ドラッグすることでペインのドッキングを解除します。次の図を参照してください。

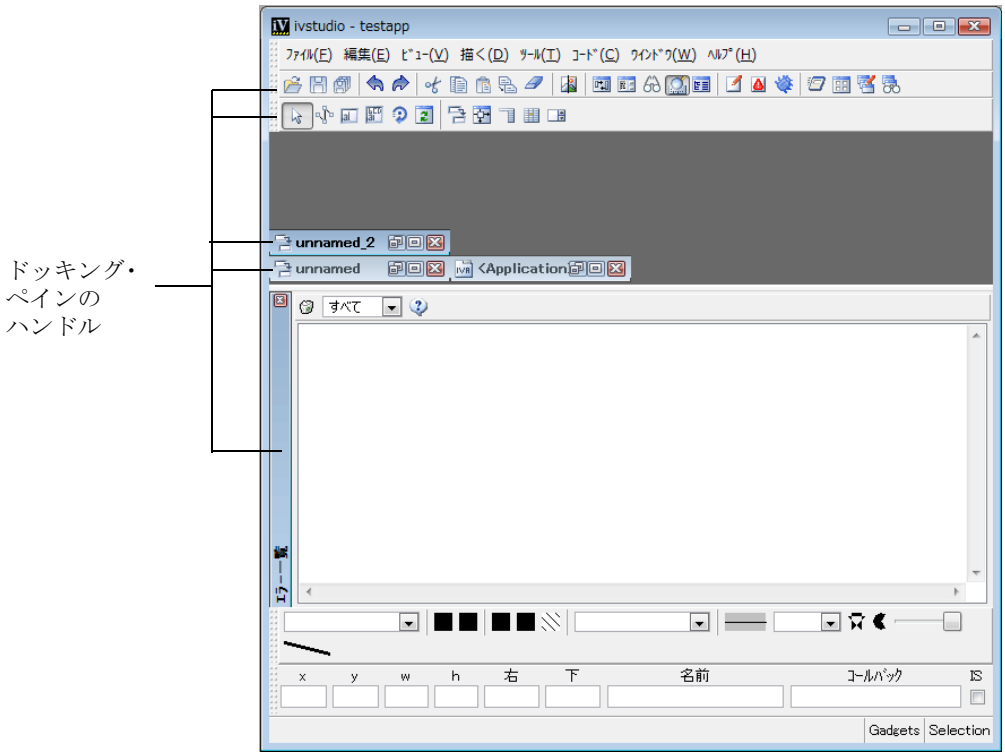


図16.4　ドッキング・ペインのハンドル

ペインをドラッグして移動させると、新しい位置に配置する助けとなるゴースト・イメージが表示されます。

> ***メモ:*** *ドッキング可能コンテナがハンドルをドッキング・ペインに追加するので、メンバ関数* `IlvDockableContainer::addDockingPane` *を呼び出すと、そのインデックスはもはや指定されたものではなくなります。ターゲット・ペイン・コンテナが水平の場合、ハンドルはペインの左側に追加され、コンテナが垂直の場合はペインの上側に追加されます。ペイン・インデックスに関しては、342 ページの自動スライダ作成の使用 および349 ページの直交ドッキング可能コンテナの作成 を参照してください。*

直交ドッキング可能コンテナの作成

直交ドッキング可能コンテナは、高度な機能です。非標準的な振る舞いをするドッキング・ペインを作成したい場合に、この機能を使用します。詳細は、360 ページの *IlvDockableMainWindow クラスの使用*を参照してください。この機能を実装している `IlvDockableMainWindow` クラスの説明があります。

IlvDockableContainer クラスは、メンバ関数 createOrthogonalDockableContainer を提供します。これは true に設定されている場合、次のように addDockingPane メンバ関数の振る舞いを変更します。

- メイン・ドッキング可能コンテナに直交する内部ドッキング可能コンテナを作成します。

  「直交ドッキング可能コンテナ作成」機能はこの内部コンテナには適用されません。
- 内部ドッキング可能コンテナをビュー・ペインにカプセル化します。
- ドッキング可能コンテナにビュー・ペインを追加します。
- ドッキング・ペインおよびそのハンドルを内部ドッキング可能コンテナに追加します。

「直交ドッキング可能コンテナの作成」機能がオフのときに、ペインを垂直ドッキング可能コンテナに追加すると、次のようになります。

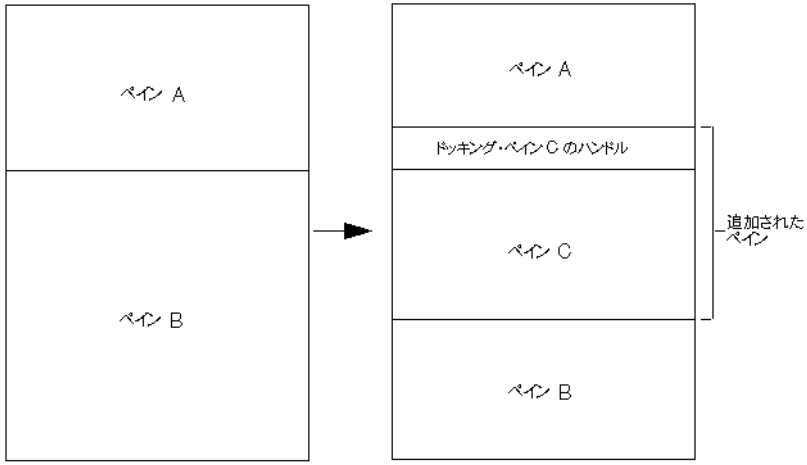


図 16.5 *直交ドッキング可能コンテナ作成機能がオフの場合*

「直交ドッキング可能コンテナの作成」機能がオンであれば、次のようになります。

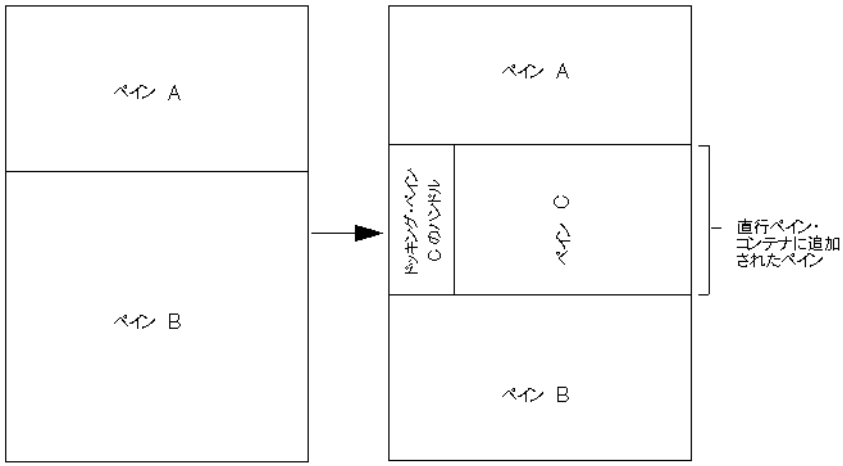


図16.6 *直交ドッキング可能コンテナ作成機能がオンの場合*

この機能は、下図のようなペイン構造を取得するため、他のペインをペイン C のドッキング可能コンテナにドッキングすることを可能にします。

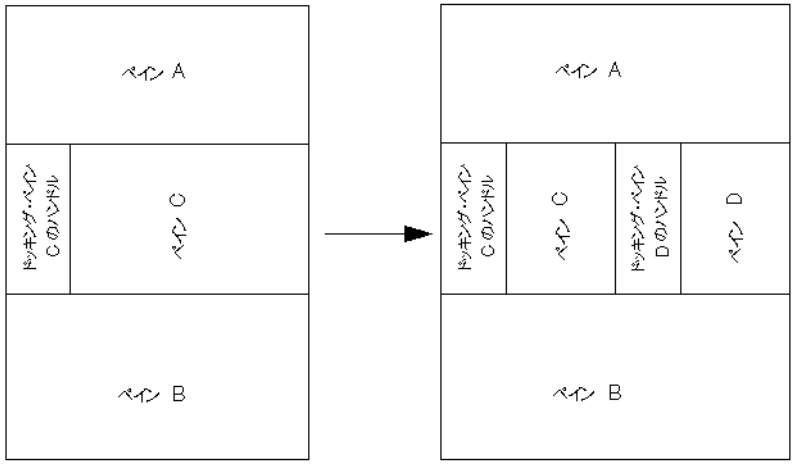


図16.7 *ペインD はペインC のドッキング可能コンテナにドッキングされている*

「直交ドッキング可能コンテナを作成」機能がオンのとき、ペインが実際に追加されるドッキング可能コンテナは、メンバ関数 `addDockingPane` を呼び出したものではありません。また、追加されたペインのインデックスは変更になっている可能性もあります。そのため、そのコンテナ内部のインデックスではなく、名前を使用してペインを取得することをお勧めします。

ドッキング操作の制御

特定のペインに関連するドッキング操作を、IlvDockable クラスで管理することができます。各ドッキング・ペインはそれに接続している IlvDockable クラスのインスタンスを持っています。このインスタンスは、メンバ関数 IlvDockableContainer::addDockingPane を使用してペインを追加するときに、自動作成することも、またはユーザが指定することもできます。

このセクションでは、以下のトピックを取り上げます。

- *IlvDockable クラスのインスタンスをペインに接続する*
- *ペインのドッキング/ ドッキング解除*
- *ドッキング操作のフィルタリング*

IlvDockable クラスのインスタンスをペインに接続する

IlvDockable クラスのインスタンスをペインに接続するには、まず次のように IlvDockable クラスのインスタンスあるいはサブクラスのインスタンスを作成しなくてはなりません。

```
IlvDockable* dockable = new IlvDockable();
```

次に、スタティック・メンバ関数 IlvDockable::SetDockable を使用して、これをペインに設定することができます。

ペインに接続されている IlvDockable インスタンスを取得するには、次を呼び出します。

```
IlvDockable* dockable = IlvDockable::GetDockable(pane);
```

このメンバ関数は、pane がドッキング・ペインでない場合は、0 を返します。

IlvDockable インスタンスに接続されているペインを取得するには、次を呼び出します。

```
IlvPane* pane = dockable->getPane();
```

ペインのドッキング / ドッキング解除

ペインがドッキングされているとき、メンバ関数 IlvDockable::unDock を使用して、ドッキング解除することができます。

ペインがドッキング解除されているとき、メンバ関数 IlvDockable::dock を使用して、ドッキングすることができます。このメンバ関数は、ペインをドッキングさせるために IlvDockableContainer::addDockingPane を呼び出します。

ペインがドッキングされているかどうかを調べるには、IlvDockable::isDocked メンバ関数を使用します。

ユーザ・インタラクションの制御

ドッキング・ペインのハンドルをダブルクリックして、これをドッキングさせたりドッキング解除させたりすることができます。

ペインをドッキングさせないようにするには、ドッキング可能コンテナにペインをドラッグしている間、Ctrl キーを押します。

ドッキング操作を取り消す場合は、Esc キーを押します。

ドッキング操作のフィルタリング

ドッキング・ペインはアプリケーション内のあらゆるドッキング可能コンテナに付加することが可能です。ただし、ドッキング操作を制御して、ドッキング・ペインを任意のコンテナに付加できないようにすることもできます。

ペインをドッキング可能コンテナにドラッグするとき、仮想メンバ関数 IlvDockable::acceptDocking が呼び出されます。これが true を返す場合、ペインはドッキングさせることができます。そうでない場合、ドッキング操作は許可されません。

acceptDocking で何を確認するかを、ここで簡単に説明します。

- ターゲット・コンテナが現在のペイン・コンテナと同じ場合、acceptDocking は IlTrue を返します。
- ターゲット・コンテナに、メンバ関数 IlvDockableContainer::acceptDocking でペインがドッキングできるかどうかを問い合わせます。ドッキング可能コンテナが IlFalse を返す場合、ドッキングは許可されず、acceptDocking は IlFalse を返します。デフォルトでは、メンバ関数 acceptDocking はコンテナのドッキング可能状態を返します。この状態をメンバ関数 IlvDockableContainer::setDockable で変更することができます。
- ドッキング方向の設定は、ターゲット・コンテナの方向と比較されます。両方の方向が合致しない場合、ドッキングは実行されず、acceptDocking は IlFalse を返します。ドッキング方向をメンバ関数 IlvDockable::setDockingDirection を使用して設定することができます。ペインを常に水平にドッキングさせたい場合など、この機能が役立ちます。デフォルトでは、ペインは垂直および水平コンテナ両方にドッキングさせることができます。

ドッキング・バーの使用

ほとんどの GUI アプリケーションにはドッキング・バーが含まれています。これらの振る舞いは、標準ドッキング・ペインの振る舞いとわずかに異なります。

このセクションでは、ドッキング・バーの概要を説明します。以下のトピックから構成されています。

- *IlvAbstractBarPane クラスの使用*
- *ドッキング・バーのカスタマイズ*

IlvAbstractBarPane クラスの使用

クラス `IlvAbstractBarPane` は、特にツールバーおよびメニュー・バーを処理するためのペインを定義します。このクラスは、`IlvAbstractBar` オブジェクトをカプセル化する `IlvGraphicPane` クラスのサブクラスです。これは、バーの向きを管理します。

ドッキング・バーがドッキングされたら、その方向を新しい位置に応じて変更しなくてはなりません。

下の図は、水平および垂直に配置された同じツールバーを表しています。

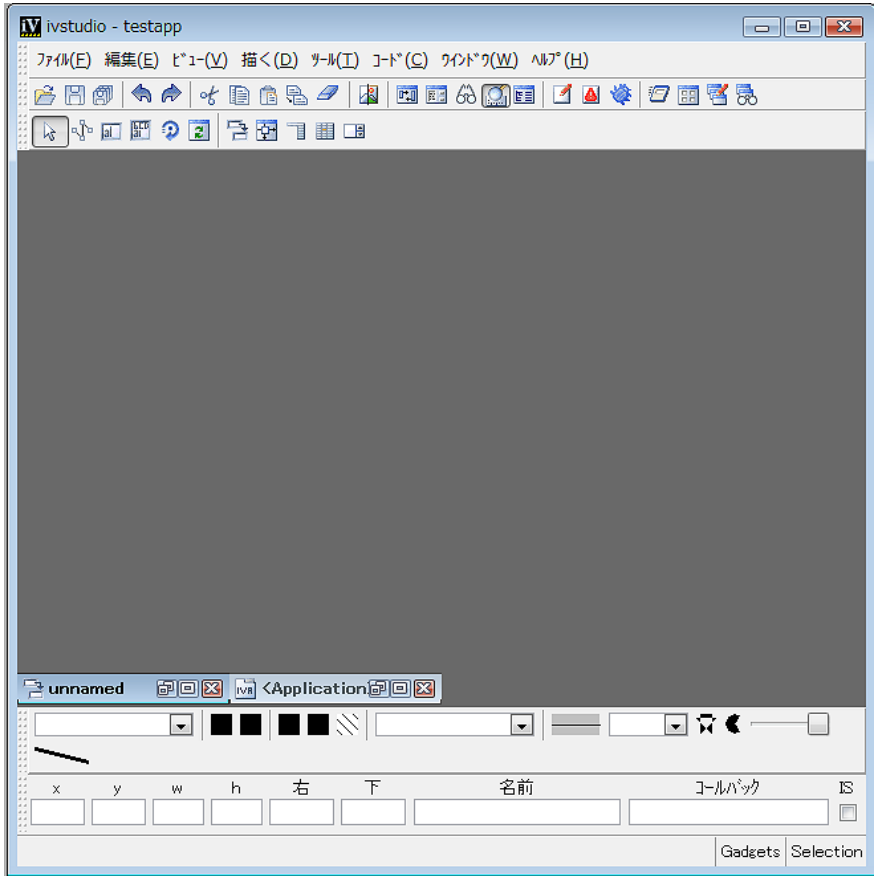


図16.8　水平ツールバー

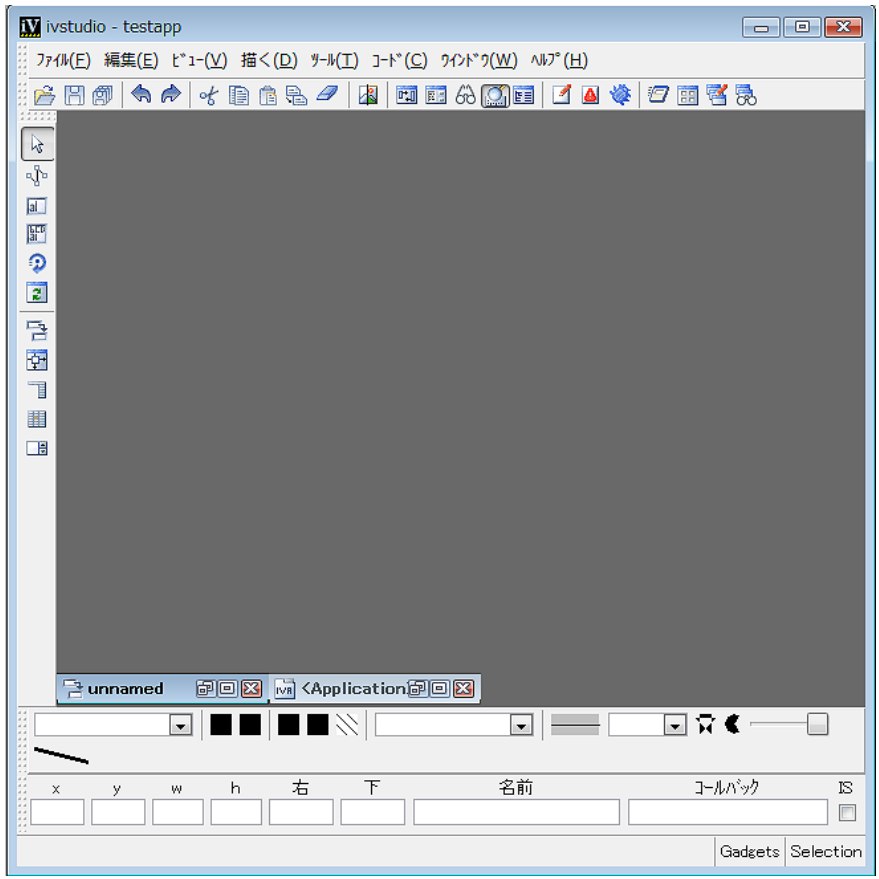


図16.9　垂直ツールバー

> ***メモ:*** *このクラスは、*`IlvDockable`*のそれ自体のサブクラスを管理しているため、変更できません。*

ドッキング・バーのカスタマイズ

`IlvAbstractBarPane`クラスには、独自のニーズに合わせて再定義できる仮想メンバ関数があります。

- `IlvAbstractBarPane::orientationChanged` - ペインによってカプセル化されたツールバーの向きが変更されるたびに呼び出されます。

- `IlvAbstractBarPane::geometryChanged` - ペインによってカプセル化されたツールバーの向きが変更されるたびに呼び出されます。311 ページの *ジオメトリの変更をバーに通知する*を参照してください。

次の例は、バー方向に応じてラベルの向きを変更する `IlvAbstractBarPane` クラスのサブクラスを表しています。

```
class MyMainMenuBarPane
: public IlvAbstractBarPane
{
public:
    MyMainMenuBarPane(const char* name, IlvAbstractBar* bar)
        : IlvAbstractBarPane(name, bar) {}
    virtual void setContainer(IlvPanedContainer* container)
    {
        IlvAbstractBarPane::setContainer(container);
        if (container)
            checkLabelOrientation();
    }
    virtual void orientationChanged()
    {
        checkLabelOrientation();
        IlvAbstractBarPane::orientationChanged();
    }
    void checkLabelOrientation()
    {
        IlvDockable* dockable = IlvDockable::GetDockable(this);
        getBar()->setLabelOrientation(dockable && dockable->isDocked()
                                      ? getBar()->getOrientation()
                                      : IlvHorizontal,
                                      IlFalse,
                                      IlFalse);
    }
};
```

`checkLabelOrientation` メンバ関数は、バーの向きが変更されるたびに呼び出されます。ペインがドッキングされている場合はバー・ラベルの向きをバーの向きに、バーがドッキングされていない場合は `IlvHorizontal` に設定します。

ドッキング・ペインがある標準的アプリケーションの作成

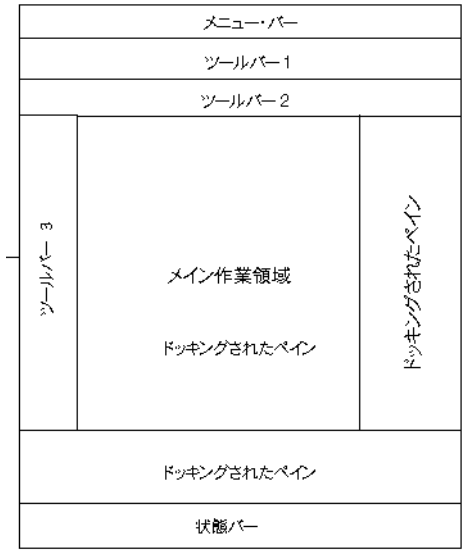
ドッキング・ペイン付きの GUI アプリケーションはすべて、多少なりとも外観が似ています。IBM® ILOG® Views Gadgets で、ドッキング・ペインがある標準的な GUI アプリケーションを容易に作成することができます。

このセクションでは、以下のトピックを取り上げます。

- *標準的レイアウトの定義*
- *IlvDockableMainWindow クラスの使用*

標準的レイアウトの定義

一般的に、標準的 GUI アプリケーションは、次のようなレイアウトになっています。



標準的レイアウトは上下左右をいくつかのペインで囲まれたメイン作業領域と呼ばれる中央のエリアから構成されていることが、図からわかります。

これは、ドッキング・ペインのある典型的な GUI アプリケーションの例です。

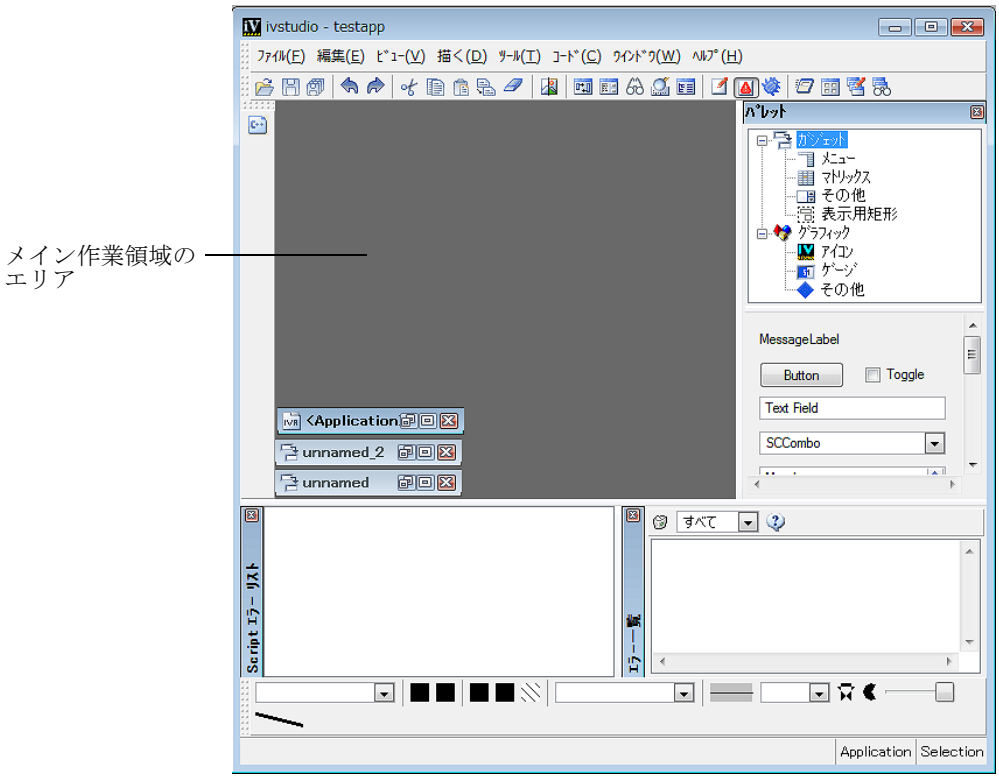
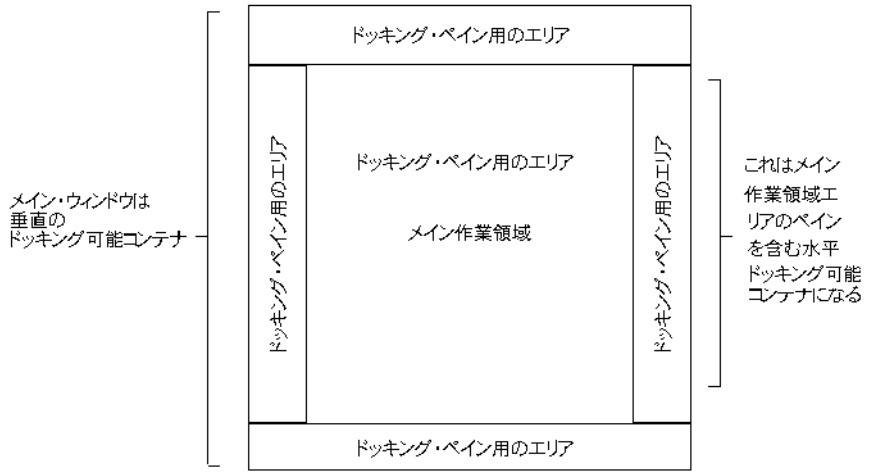


図 16.10 *ドッキング・ペインのある典型的な GUI アプリケーション*

ドッキング・ペインの機能を使用して、次のようなペイン構造を持つ標準的な GUI アプリケーションを作成することができます。



このレイアウトでは、359 ページの、図 16.10 に示すように、メイン作業領域の周囲のどこにでもペインを追加することが可能です。

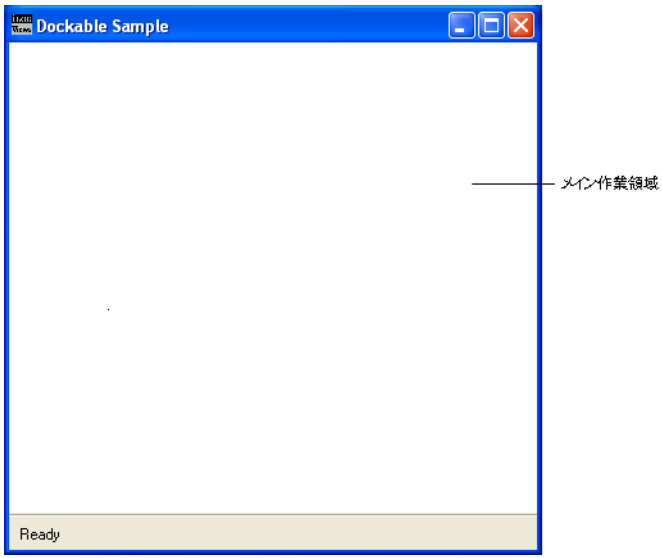
IlvDockableMainWindow クラスの使用

`IlvDockableMainWindow` クラスは、358 ページの *標準的レイアウトの定義* で説明されているレイアウトを実装します。このクラスを使用して、ペインがどのように構成されるかが正確にわからなくても、特定のペインに対してどこにペインを追加すればいいか指定することができます。メンバ関数 `IlvDockableMainWindow::addRelativeDockingPane` を使用して新しいペインを追加するのは、次のような文章で配置場所を指定するのと同じくらいに簡単です。「メニュー・バーをメイン作業領域のエリア上部に配置する。」

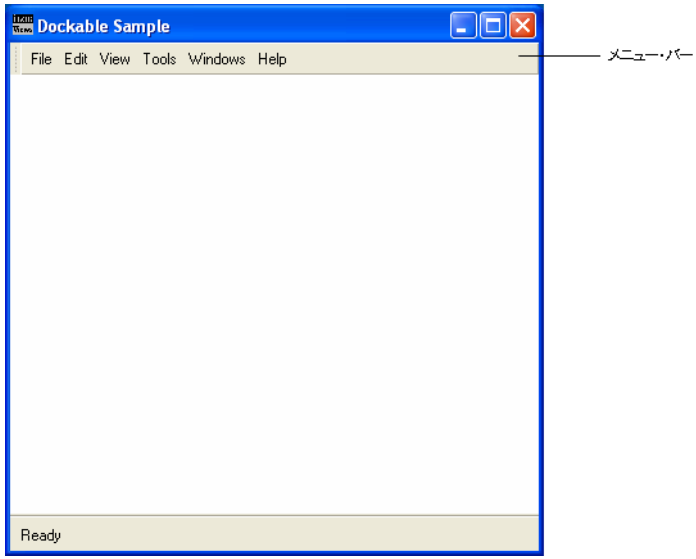
ペインのインデックスではなくて名前を提供するだけでよいため、アプリケーションのインターフェース全体を作成することがとても簡単になります。

次は、メンバ関数 `IlvDockableMainWindow::addRelativeDockingPane` を使用して取得する場合の例です。

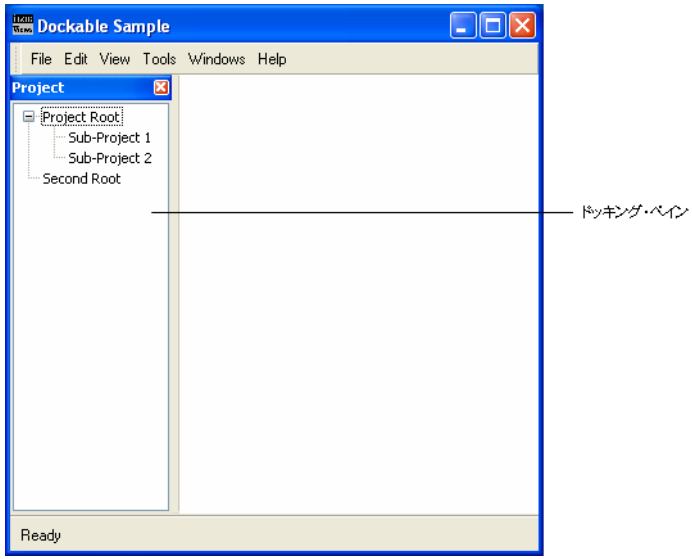
`IlvDockableMainWindow` のインスタンスを作成すると、次のペイン・レイアウトができます。



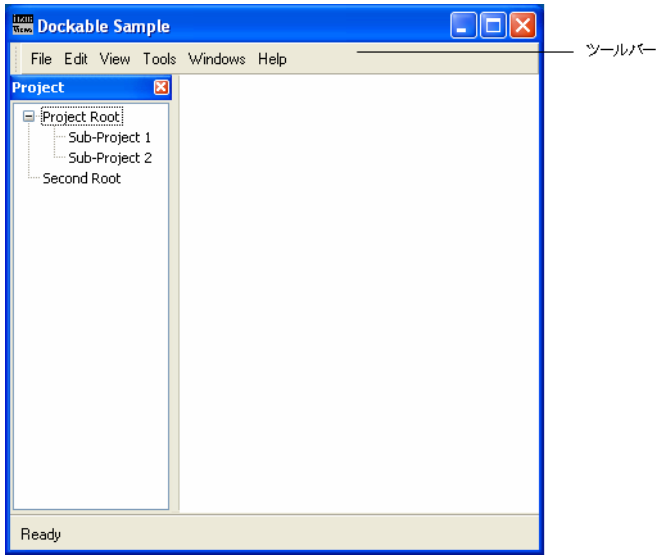
次に、下図のように、メニュー・バーがメイン作業領域のエリア上部に追加されます。



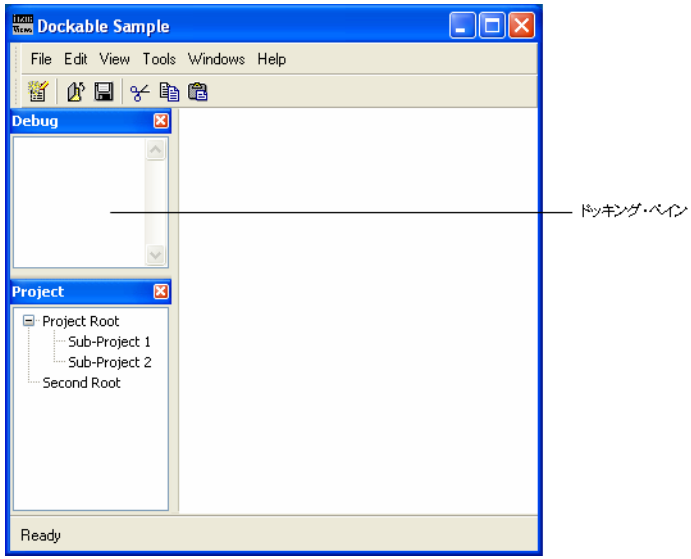
さらに、下に示すとおり、ドッキング・ペインがメイン作業領域エリアの左側に追加されます。



そしてツールバーが、下に示すとおり、メニュー・バーの下に追加されます。



最後に、下に示すとおり、2 つ目のドッキング・ペインが最初のドッキング・ペインの上に追加されます。


Dockable Sample
File Edit View Tools Windows Help
Debug
Project
Project Root
Sub-Project 1
Sub-Project 2
Second Root
Ready
ドッキング・ペイン

章

17

ビュー・フレーム

IBM® ILOG® Views Gadgets ライブラリは、ビュー・フレームをサポートしています。

この章では、ビュー・フレームの概念とグラフィカル・アプリケーション内での使用方法について説明します。以下のトピックから構成されています。

- *ビュー・フレームの概要*
- *ビュー・フレームのあるデスクトップの作成*
- *ビュー・フレームの管理*
- *ビュー・フレームの [最小化]、[最大化]、[元のサイズに戻す]*
- *ビュー・フレームを閉じる*
- *クライアント・ビュー・フレームの変更*

ビュー・フレームの概要

ビュー・フレームは、クライアント・ビューをカプセル化するタイトル・バーのある特殊なコンテナです。タイトル・バーは、アイコン、ラベル、そして複数のボタンから構成されています。ビュー・フレームは、デスクトップ・ビューとよばれる親ビュー内に表示されます。

ビュー・フレームは、クラス `IlvViewFrame` のインスタンスです。デスクトップ・ビューは常にクラス `IlvDesktopManager` のインスタンスにリンクされており、デスクトップ・ビュー内のすべてのフレームを管理します。

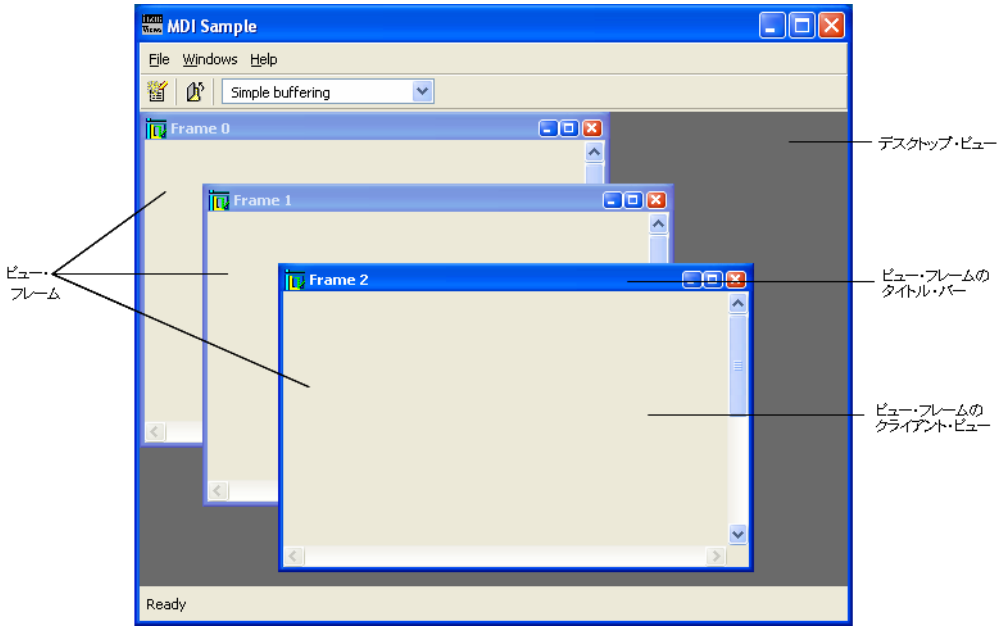


図17.1 *フレームで構成されるアプリケーション*

ビュー・フレームのあるデスクトップの作成

この章では、ビュー・フレームを含むデスクトップの作成方法を説明します。次のセクションに分けて説明します。

- *デスクトップの作成*
- *ビュー・フレームの作成*

デスクトップの作成

最初のステップは、デスクトップ・ビューおよびデスクトップ・マネージャの作成です。

```
IlvView* desktopView = new IlvView(...);
IlvDesktopManager* desktop = new IlvDesktopManager(desktopView);
```

メモ: デスクトップ・ビューは、IlvView のあらゆるサブクラスのインスタンスにすることができます。

スタティック・メソッド IlvDesktopManager::Get を使用して、ビューにリンクしている IlvDesktopManager インスタンスを取得することができます。

```
IlvDesktopManager* desktop = IlvDesktopManager:Get(view);
```

デスクトップ・マネージャに関連付けられたデスクトップ・ビューを調べるには、メソッド IlvDesktopManager::getView を使用します。

```
IlvView* view = desktop->getView();
```

デスクトップ・ビューが削除されると、デスクトップ・マネージャに通知されますが、デスクトップ・マネージャは削除はされません。

ビュー・フレームの作成

デスクトップ・マネージャが作成されると、デスクトップ・ビューの子ウィンドウとしてビュー・フレームを作成することができます。

```
IlvViewFrame* vframe = new IlvViewFrame(desktopView,
                                        "Frame 0",
                                        IlvRect(0, 0, 100, 100));
```

新しいビュー・フレームは自動的に、親ビュー (つまり、デスクトップ・ビュー) にリンクされている IlvDesktopManager のインスタンスで管理されます。デスクトップ・マネージャがビュー・フレームの親ビューにまったく付加されていない場合、デスクトップ・ビューとしてビュー・フレームの親ビューを使用してデフォルト・デスクトップ・マネージャが作成されます。このデフォルト・デスクトップ・マネージャは内部的に管理されるので、削除する必要はありません。

ビュー・フレームのデスクトップ・マネージャを調べるには、次を使用します。

```
IlvDesktopManager* desktop = vframe->getDesktopManager();
```

次を利用して、デスクトップ・マネージャによって管理されているフレームのリストを取得することもできます。

```
IlvUInt count;
IlvViewFrame* const* frames = desktop->getFrames(count);
```

ビュー・フレームの管理

このセクションでは、以下のトピックを取り上げます。

- *クライアント・ビューの作成*
- *タイトル・バーの変更*
- *ビュー・フレーム・メニューの変更*

クライアント・ビューの作成

ビュー・フレームの作成時にはクライアント・ビューはありません。ビュー・フレームにクライアント・ビューを追加するには、ビュー・フレーム内に子ウィンドウを作成する必要があります。

```
IlvGadgetContainer* clientView =
            new IlvGadgetContainer(vframe, IlvRect(0, 0, 200, 200));
```

ビュー・フレームはクライアント・ビューの形状に合わせてリサイズされます。

> ***メモ:*** *クライアント・ビューは、*`IlvView`* *のあらゆるサブクラスのインスタンスにすることができます。*

ビュー・フレームに関連付けられているクライアント・ビューを調べるには、次を使用します。

```
IlvView* clientView = vframe->getClient();
```

> ***メモ:*** *ビュー・フレームは1 つのクライアント・ビューのみを扱います。*

タイトル・バーの変更

タイトル・バーはアイコン、タイトル、そして 3 つのボタンから構成されています。

図 17.2 *ビュー・フレームのタイトル・バー*

タイトル・バーのアイコンを変更するには、メソッド `IlvViewFrame::setIcon` を使用します。

```
IlvBitmap* bitmap = ...
vframe->setIcon(bitmap);
```

タイトルを変更するには、メソッド `IlvViewFrame::setTitle` を使用します。

```
vframe->setTitle("Frame Title");
```

タイトル・バーの右にある 3 つのボタンは、フレームの 3 つの状態を切り替えるために使用します。詳細は次のとおりです。

ビュー・フレーム・メニューの変更

各ビュー・フレームは、タイトル・バーの左端にあるアイコンをクリックした時に表示されるポップアップ・メニューを有しています。

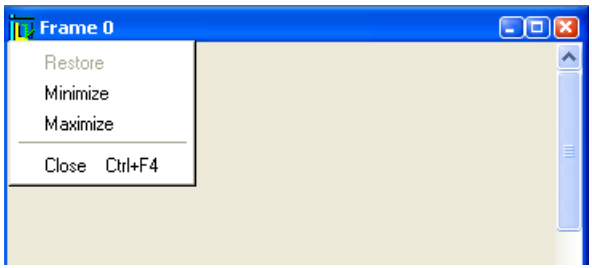


図 17.3 *ビュー・フレームのポップアップ・メニュー*

デフォルトでは、ビュー・フレームのポップアップ・メニューには次の選択が含まれます。[元のサイズに戻す]、[最小化]、[最大化]、そして [閉じる] です。新規項目を追加することもできます。

このメニューにアクセスするには、以下を使用します。

```
IlvPopupMenu* popup = vframe->getMenu();
```

ビュー・フレームの [最小化]、[最大化]、[元のサイズに戻す]

ビュー・フレームの状態は、標準、最小、最大のいずれかになります。

フレームの状態を取得するには、メソッド `IlvViewFrame::getCurrentState` を使用します。可能な戻り値は、`NormalState`、`MinimizedState`、および `MaximizedState` です。

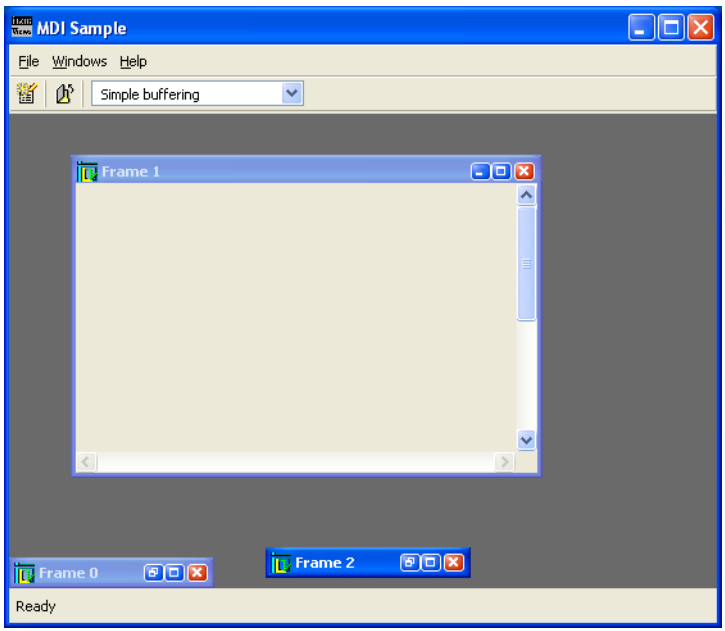


図17.4　標準および最小化されたビュー・フレーム

標準ビュー・フレーム

デフォルトでは、ビュー・フレームは標準サイズで表示されます。最小化あるいは最大化した後に、この状態にフレームを戻すには、メソッド `IlvViewFrame::restoreFrame` を使用します。

```
vframe->restoreFrame();
```

このメソッドは、フレームが既に標準の状態にある場合は何も行いません。

タイトル・バーにある [元のサイズに戻す] ボタンをクリックして、ビュー・フレームを初期状態に戻すこともできます。

最小化されたビュー・フレーム

ビュー・フレームが最小化されている時、タイトル・バーのみが表示状態となり、その位置はデスクトップ・マネージャによって管理されます。フレームを最小化するには、メソッド `IlvViewFrame::minimizeFrame` を使用します。

```
vframe->minimizeFrame();
```

最小化されたビュー・フレームのリストは、デスクトップ・マネージャに管理されており、メソッド `IlvDesktopManager::getMinimizedFrames` を使用して、アクセスすることができます。

タイトル・バーの [最小化] ボタンをクリックしてビュー・フレームを最小化することもできます。

ビュー・フレームの最大化

ビュー・フレームを最大化すると、クライアント・ビューがデスクトップのビュー全体に表示されます。

ビュー・フレームを最大化するには、メソッド IlvViewFrame::maximizeFrame を使用します。

```
vframe->maximizeFrame();
```

タイトル・バーの [最大化] ボタンをクリックしてビュー・フレームを最大化することもできます。

ビュー・フレームが最大化されると、そのタイトル・バーおよびそれに含まれるボタンは非表示となります。この場合、デスクトップ・マネージャは別の場所にこれら 3 つのボタンを表示させることができます。

次の行は、フレームが最大化されたときに、デスクトップ・マネージャに container にタイトル・バーのボタンを表示するように伝えます。

```
IlvContainer* container = ....
desktop->makeMaximizedStateButtons(container->getHolder());
```

次の行は、デスクトップ・マネージャに toolbar にタイトル・バーのボタンを表示するように伝えます。

```
IlvToolBar* toolbar = ...
desktop->makeMaximizedStateButtons(toolbar);
```

ビュー・フレームを閉じる

(たとえば、[閉じる] ボタンを使用して) ビュー・フレームを閉じようとすると、IlvViewFrame::closeFrame メソッドが呼び出されます。デフォルトでは、このメソッドは、ビュー・フレームに設定されている破壊コールバックを呼び出します。つまり、ビュー・フレームの破壊方法を制御したい場合は、破壊コールバックを設定しなくてはならないということです。

例 :

```
vframe->setDestroyCallback(DestroyFrame);
```

次のコールバックで呼び出します。

```
static void DestroyFrame(IlvView* view, IlvAny)
{
    IlvIQuestionDialog dlg(view->getDisplay(), "Are you sure ?");
    dlg.moveToMouse();
```

```
    if (dlg.get())
       delete view;
}
```

フレームを削除する前に、確認のダイアログ・ボックスが表示されます。

> ***メモ:*** *破壊コールバックが設定されていない場合、ビュー・フレームを閉じようとしても何も起きません。*

クライアント・ビュー・フレームの変更

デスクトップ・マネージャの現在のビュー・フレームは、キーボード・フォーカスのあるビュー・フレームです。別のビュー・フレームをクリックして、これを新しい現在のビュー・フレームにして、現在のビュー・フレームを変更することができます。

コーディングで現在のビュー・フレームを変更することもできます。

```
desktop->setCurrentFrame(vframe);
```

現在のビュー・フレームが変更されると、仮想メソッド `IlvDesktopManager::frameSelectionChanged` が呼び出されます。`IlvDesktopManager` の独自のサブクラス内のこのメソッドをオーバーライドして、現在のビュー・フレームが変更になったときに、特定のアクションを実行させることができます。

章

18

ルック・アンド・フィールのカスタマイズ

この章では、ルック・アンド・フィール機構で使用するクラスを紹介します。以下のトピックから構成されています。

- *アーキテクチャの理解*
- *ユーザ定義コンポーネントをルック・アンド・フィールに依存させる*
- *既存のコンポーネントのルック・アンド・フィールの変更*
- *新規ルック・アンド・フィール・ハンドラの作成*

アーキテクチャの理解

このセクションの目的は、ガジェットがどのようにルック・アンド・フィールに対応するかを説明します。以下のトピックに関する情報が記載されています。

- *IlvLookFeelHandler*
- *IlvObjectLFHandler*
- *クラスのダイアグラム*

IlvLookFeelHandler

IlvLookFeelHandler クラスは、すべてのルック・アンド・フィール・ハンドラのベース・クラスです。これは、オブジェクト・ルック・アンド・フィール・ハンドラの集合として振る舞い、特定のルックに共通するプロパティを集めます。ルック・アンド・フィールに依存させる必要がある各コンポーネントは、IlvLookFeelHandler クラスのインスタンスへアクセスしなくてはなりません。描画プロセス中、コンポーネントは自らを描画するためにこのハンドラを使用します。同様に、コンポーネントがイベントを受け取るとき、ハンドラが定義した方法でイベントを扱うためにこのハンドラを使用します。

> ***メモ:*** *ルック・アンド・フィール・ハンドラは共有オブジェクトです。標準演算子* new *を使用してこれらを作成したり、削除することは避けてください。*

IlvLookFeelHandler オブジェクトにポインタを取得する

ガジェットがポインタを IlvLookFeelHandler サブクラス・インスタンスに取得するには 3 つの方法があります。

- オブジェクト・レベル

  メソッド IlvGraphic::getLookFeelHandler() は、オブジェクトにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルトの定義では、オブジェクト・ホルダによって定義されたルック・アンド・フィール・ハンドラを使用します。

- ホルダ・レベル

  メソッド IlvGraphicHolder::getLookFeelHandler() は、ホルダにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルトの定義では、ホルダ表示インスタンスによって定義されたルック・アンド・フィール・ハンドラを使用します。

- 表示レベル

  メソッド IlvDisplay::getLookFeelHandler() は、表示インスタンスにルック・アンド・フィール・ハンドラを問い合わせるために使用されます。デフォルト値は、アプリケーションが作成されたプラットフォームによって定義されます。詳細については、229 ページの *デフォルト・ルック・アンド・フィールの使用* を参照してください。

IlvObjectLFHandler

ガジェットがそのルック・アンド・フィール・ハンドラを取得すると、その特定オブジェクト・ルック・アンド・フィール・ハンドラを要求しなくてはなりません。このオブジェクト・ルック・アンド・フィール・ハンドラは、IlvObjectLFHandler クラスで実装されます。ルック・アンド・フィールに依存さ

せる必要がある各コンポーネントは、IlvObjectLFHandler クラスのサブクラスを作成しなくてはなりません。

IlvObjectLFHandler オブジェクトにポインタを取得する

IlvLookFeelHandler クラスは IlvObjectLFHandler のハッシュ・テーブルを扱います。IlvObjectLFHandler クラスの各インスタンスは、そのクラス情報を使用して取得されます。たとえば、次のコードは IlvButton クラスのオブジェクト・ルック・アンド・フィール・ハンドラを取得します。

```
IlvLookFeelHandler lfh = display->getLookFeelHandler();
IlvButtonLFHandler* buttonLF = (IlvButtonLFHandler*)
      lfh->getObjectLFHandler(IlvButton::ClassInfo());
```

> ***メモ:*** *getObjectLFHandler によって返された値は、ボタン・オブジェクト・ルック・アンド・フィール・ハンドラ用のベース・クラスである* IlvButtonLFHandler *ポインタにキャストされます。*

ポインタがその特定オブジェクト・ルック・アンド・フィール・ハンドラに取得されると、ボタンは次のコードを使用して自らを描画します。

```
void
IlvButton::draw(IlvPort* dst,
                const IlvTransformer* t,
                const IlvRegion* clip) const
{
   IlvButtonLFHandler* lfhandler = (IlvButtonLFHandler*)
         getObjectLFHandler(IlvButton::ClassInfo());
    lfhandler->draw(this, dst, t, clip);
}
```

クラスのダイアグラム

次のダイアグラムは、ルック・アンド・フィール・プロセスで中心的な役割を果たす 3 つの項目の関係を示しています。この 3 つとは、オブジェクト、ルック・アンド・フィール・ハンドラ、そしてオブジェクト・ルック・アンド・フィール・ハンドラです。

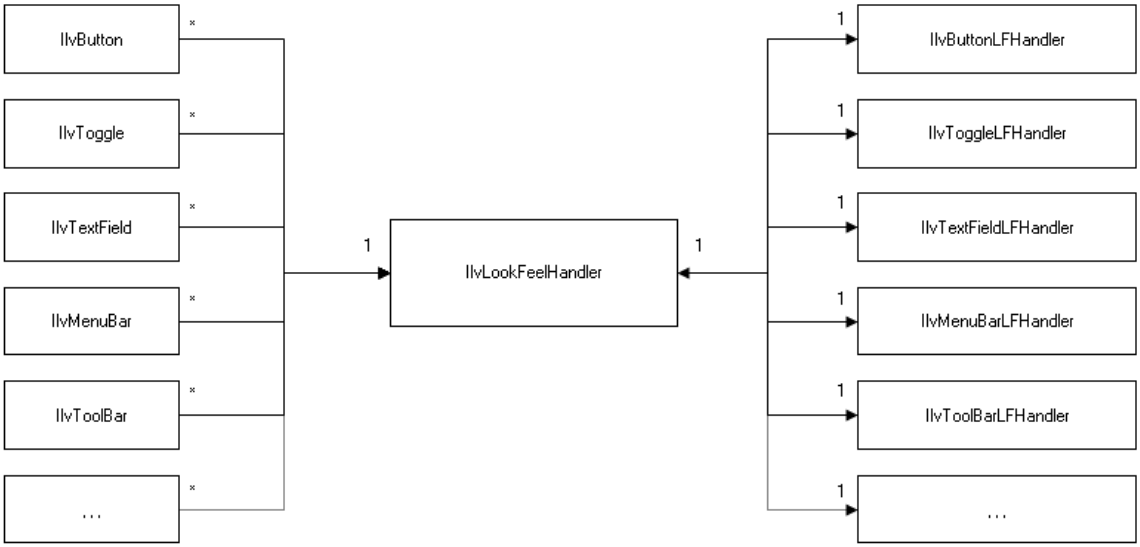


図 18.1 *ルック・アンド・フィール・プロセスに含まれる複数クラス間の関係*

次のダイアグラムは、`IlvButton` の描画中のイベントのトレースを Motif 風に描いたものです。

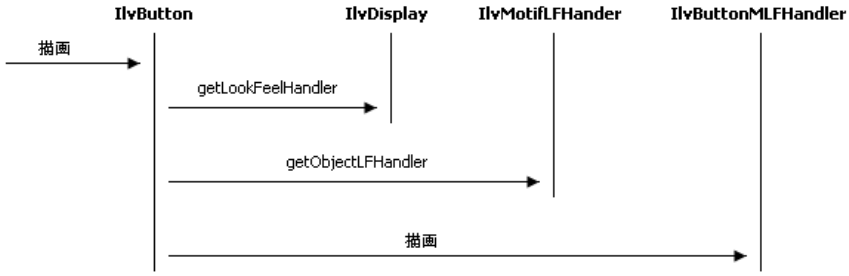


図 18.2 *イベント・トレース: ボタンの描画*

ユーザ定義コンポーネントをルック・アンド・フィールに依存させる

このセクションでは、ルック・アンド・フィールに依存するコンポーネントを新規に作成する方法について説明します。この章は、以下のトピックに分かれています。

- *新規コンポーネントの作成*
- *オブジェクト・ルック・アンド・フィール・ハンドラ API の定義*
- *オブジェクト・ルック・アンド・フィール・ハンドラのサブクラス化*
- *オブジェクト・ルック・アンド・フィール・ハンドラのインストール*

新規コンポーネントの作成

新規コンポーネントを適切に作成する方法に関する詳細情報は、Foundation ユーザ・マニュアル、「IlvGraphic: グラフィック・オブジェクト・クラス、新規グラフィック・オブジェクト・クラスの作成」にあります。

重要な点はクラス情報を適切に登録することです。これは、ルック・アンド・フィール依存コンポーネントを作成するのに欠かせません。

新規に作成されたコンポーネントが、`MyComponent`、`IlvGadget` のサブクラスであると仮定します。

オブジェクト・ルック・アンド・フィール・ハンドラ API の定義

オブジェクト・ルック・アンド・フィール・ハンドラ API は、デザインするコンポーネントによって異なります。一般的には、その外観と振る舞いをカスタマイズする方法を提供する必要があります。これは、次のメソッドをオブジェクト・クラスに追加することで行います。

```
class MyComponent : public IlvGadget
{
    ...
    virtual void draw(IlvPort* dst,
                      const IlvTransformer* t,
                      const IlvRegion* clip) const;
    virtual IlBoolean handleEvent(IlvEvent& event) const;
    ...

};
```

また、これらのメソッドをオブジェクト・ルック・アンド・フィール・ハンドラ・クラスに追加する必要があります。

```
class MyComponentLFHandler : public IlvObjectLFHandler
{
   MyComponentLFHandler(IlvLookFeelHandler* lfh) :
        IlvObjectLFHandler(MyComponent::ClassInfo(), lfh) {}
    ...
    virtual void draw(const MyComponent* object,
                      IlvPort* dst,
                      const IlvTransformer* t,
                      const IlvRegion* clip) const = 0;
    virtual IlBoolean handleEvent(MyComponent* object,
                                  IlvEvent& event) const = 0;
    ...
};
```

注意:

1. *オブジェクト・ルック・アンド・フィール・ハンドラは共有オブジェクトなので、オブジェクト・ルック・アンド・フィール・ハンドラ・クラスの各メソッドで* `MyComponent` *インスタンスへのアクセスを与える必要があります。メソッドの最初のパラメータを使用してこれを行うことができます。*
2. `MyComponentLFHandler` *のコンストラクタは、* `MyComponent::ClassInfo` *メソッドを使用してこのオブジェクト・ハンドラに* `MyComponent` *クラスをリンクします。このようにして、* `MyComponentLFHandler` *の各サブクラスは、* `MyComponent` *コンポーネント専用になります。*

`MyComponent` メソッドの実装は、以下のように行います。

```
void
MyComponent::draw(IlvPort* dst,
                  const IlvTransformer* t,
                  const IlvRegion* clip) const
{
    MyComponent* lfhandler = (MyComponentLFHandler*)
        getObjectLFHandler(MyComponent::ClassInfo());
    lfhandler->draw(this, dst, t, clip);
}
```

`handleEvent` メソッドも同様です。

```
IlBoolean
MyComponent::handleEvent(IlvEvent& event)
{
    MyComponentLFHandler* lfhandler = (MyComponentLFHandler*)
        getObjectLFHandler(ClassInfo());
    return lfhandler->handleEvent(this, event);
}
```

同じ方法で他の機能をコンポーネントに追加して、これらをルック・アンド・フィール依存にすることももちろん可能です。

オブジェクト・ルック・アンド・フィール・ハンドラのサブクラス化

オブジェクト・ルック・アンド・フィール・ハンドラの API が定義されると、多種のサブクラスを異なるルック・アンド・フィール・スタイルに応じて実装することができます。たとえば、ここで Motif ルック専用の `MyComponentLFHandler` のサブクラスである `MyComponentMLFHandler` クラスを作成します。

```
class MyComponentMLFHandler : public MyComponentLFHandler
{
    MyComponentMLFHandler(IlvLookFeelHandler* lfh) :
         MyComponentLFHandler(lfh) {}
    ...
    virtual void draw(const MyComponent* object,
                      IlvPort* dst,
                      const IlvTransformer* t,
                      const IlvRegion* clip) const;
    virtual IlBoolean handleEvent(MyComponent* object,
                                  IlvEvent& event) const;
    ...
};
```

オブジェクト・ルック・アンド・フィール・ハンドラをインストールして、対応するルック・アンド・フィールの設定時に使用されるようにする必要があります。

オブジェクト・ルック・アンド・フィール・ハンドラのインストール

オブジェクト・ルック・アンド・フィール・ハンドラを対応するルック・アンド・フィール・ハンドラにインストールするには、マクロ `IlvRegisterObjectLFHandler` を使用します。

```
IlvRegisterObjectLFHandler(IlvMotifLFHandler,
                           MyComponent,
                           MyComponentMLFHandler);
```

以前のコードは、Motif ルックを使用して表示される `MyComponentMLFHandler` クラス用オブジェクト・ルック・アンド・フィール・ハンドラとして `MyComponent` クラスを登録します。

オブジェクト・ルック・アンド・フィール・ハンドラのインスタンスを作成、あるいは削除する必要はありません。自動的に行われます。

既存のコンポーネントのルック・アンド・フィールの変更

このセクションでは、特化コンポーネントのルック・アンド・フィールを変更する方法について説明します。以下のトピックに関する情報が記載されています。

- *コンポーネント・オブジェクト・ルック・アンド・フィール・ハンドラのサブクラス化*
- *オブジェクト・ルック・アンド・フィール・ハンドラの置換*

コンポーネント・オブジェクト・ルック・アンド・フィール・ハンドラのサブクラス化

コンポーネントのルック・アンド・フィールを変更するには、まずそのオブジェクト・ルック・アンド・フィール・ハンドラ・ベース・クラスを識別する必要があります。通常、コンポーネント・クラスおよびそのオブジェクト・ルック・アンド・フィール・ハンドラは、同じヘッダー・ファイルで宣言されており、オブジェクト・ルック・アンド・フィール・ハンドラ・クラスの名前は、コンポーネント・クラスメイト文字列「LFHandler」が連結したものです。たとえば、IlvButton および IlvButtonLFHandler クラスの両方が <ilviews/gadgets/button.h> ヘッダー・ファイルに配置されています。

オブジェクト・ルック・アンド・フィール・ハンドラ・ベース・クラスが見つかったら、その API を注意深く観察し、どの仮想メンバ関数をオーバーライドするかを判断します。

次の例は、drawBackground メンバ関数が再定義された IlvButtonLFHandler のサブクラスです。

```
class MyButtonLFHandler : public IlvButtonLFHandler
{
    ...
    virtual void drawBackground(const IlvButton* button,
                                IlvPort* dst,
                                const IlvTransformer* t,
                                const IlvRegion* clip) const;
    ...
};
```

オブジェクト・ルック・アンド・フィール・ハンドラの置換

新規オブジェクト・ルック・アンド・フィール・ハンドラを定義したら、IlvLookFeelHandler インスタンスにこれをインストールしなくてはなりません。

オブジェクト・ルック・アンド・フィール・ハンドラをインストールする最も簡単な方法は、IlvLookFeelHandler::addObjectLFHandler メソッドをコンポーネントのルック・アンド・フィール・ハンドラで呼び出すことです。

```
IlvButton* button = ....
IlvLookFeelHandler* lfh = button->getLookFeelHandler();
MyButtonLFHandler* mylfh = new MyButtonLFHandler(lfh);
lfh->addObjectLFHandler(mylfh);
```

ルック・アンド・フィール・ハンドラをこの方法で変更することで、同じルック・アンド・フィール・ハンドラを参照する他のボタンにも影響を与えます。デフォル

トでは、ルック・アンド・フィール・ハンドラは 1 つのみで、IlvDisplay クラスに所有されています。特化コンポーネントのルック・アンド・フィールのみを変更し、デフォルトのルック・アンド・フィール・ハンドラを変更したくない場合は、次の手順に従います。

- 新規ルック・アンド・フィール・ハンドラ・インスタンスを、IlvLookFeelHandler::Create メソッドを使用して作成します。
- オブジェクト・ルック・アンド・フィール・ハンドラを、IlvLookFeelHandler::addObjectLFHandler を使用してインストールします。
- 新しいルック・アンド・フィール・ハンドラ・インスタンスを、コンポーネント上に、IlvGadget::setLookFeelHandler メソッドを使用してインストールします。

次のコードは新規ルック・アンド・フィール・ハンドラを Motif ルック用に作成し、これにオブジェクト・ルック・アンド・フィール・ハンドラをインストールします。最後に、新しいルック・アンド・フィール・ハンドラはコンポーネント上にインストールされます。

```
IlvLookFeelHandler* lfh = IlvLookFeelHandler::Create("motif");
MyButtonLFHandler* mylfh = new MyButtonLFHandler(lfh);
lfh->addObjectLFHandler(mylfh);
IlvButton* button = ....
button->setLookFeelHandler(lfh);
```

> ***メモ:*** 最初の 2 つの手順は、IlvLookFeelHandler のサブクラスを作成して単一アクションで実行させることができます。さらに詳しい情報は、380 ページの *新規ルック・アンド・フィール・ハンドラの作成* を参照してください。

新規ルック・アンド・フィール・ハンドラの作成

新規ルック・アンド・フィール・ハンドラを作成するには、次のいずれかを行います。

- 直接 IlvLookFeelHandler クラスをサブクラス化する。
- または、既存で定義済みのルック・アンド・フィール・ハンドラ・クラスの 1 つをサブクラス化する (IlvMotifLFHandler、IlvWindowsLFHandler、IlvWindows95LFHandler、IlvWindowsXPLFHandler)。

2 つ目のオプションは、すべての登録済みガジェットにオブジェクト・ルック・アンド・フィール・ハンドラを提供する必要がないので、より簡単です。ルック・アンド・フィールを変更させたいオブジェクトのみにオブジェクト・ルック・アンド・フィール・ハンドラを提供するだけです。

ILVHOME/samples/gadgets/lookfeel ディレクトリにあるサンプル lookfeel は、新規ルック・アンド・フィール・ハンドラを作成する方法を示しています。

新規ルック・アンド・フィール・ハンドラの登録

動的にルック・アンド・フィール・ハンドラを作成することができるように、これを適切に登録する必要があります。これを行うには、次のマクロをクラス宣言の中に追加する必要があります。

```
DeclareLookFeelTypeInfo();
```

その後、定義ファイルの中で次のマクロを使用します。

```
IlvPredefinedLookFeelMembers(MyLookFeelHandler, "MyLook");
IlvRegisterLookFeelClass(MyLookFeelHandler, BaseClass);
```

BaseClass が新規ルック・アンド・フィール・ハンドラのベース・クラスである場合。

これでクラスが適切に登録されました。次のコードを使用して、そのインスタンスを作成することもできます。

```
IlvLookFeelHandler* lfh = IlvLookFeelHandler::Create(IlGetSymbol("MyLook"));
```

新規ルック・アンド・フィール・ハンドラへのオブジェクト・ルック・アンド・フィール・ハンドラの登録

オブジェクト・ルック・アンド・フィール・ハンドラを新規ルック・アンド・フィール・ハンドラに登録するには、次のいずれかを行います。

- IlvLookFeelHandler::createObjectLFHandler メソッドをオーバーライドする。
- または、IlvRegisterObjectLFHandler マクロを使用する。

たとえば、IlvRegisterObjectLFHandler マクロを使用して、定義ファイルでコーディングすることができます。

```
IlvRegisterObjectLFHandler(MyLookFeelHandler, IlvButton, MyButtonLFHandler);
```

これによって、オブジェクト・ルック・アンド・フィール・ハンドラ・クラス MyButtonLFHandler が IlvButton クラス用にルック・アンド・フィール・ハンドラ MyLookFeelHandler へ登録されます。これは、ルック・アンド・フィール MyLookFeelHandler を持つ IlvButton オブジェクトが、そのオブジェクト・ルック・アンド・フィール・ハンドラを取得しようとすることを意味し、MyButtonLFHandler インスタンス上でポインタを取得します。

付録

A

状態の編集

状態によって、アプリケーションの異なるコンテキストや状態をあらかじめ定義できます。特定のコンテキストで、アプリケーションがパネルを開いたり閉じたり、グラフィック・オブジェクトを表示 / 非表示にしたり、そのセンシティビティ、色、その他プロパティを変更することができます。これらすべての設定を、*状態条件*と呼びます。状態は、状態条件をまとめたものです。原則的に、状態機構が処理するコンテキストに属する場合、プログラミングでこれらの設定を変更することは推奨しません。

IBM® ILOG® Views Studio を使うと、アプリケーションの状態およびその条件をインタラクティブに定義することができます。

この付録では、IBM ILOG Views Studio の状態機構の使用例を示します。説明は、次の 2 つの部分に分かれています。

- *簡単なアプリケーションの作成*
- *表示状態の編集*

簡単なアプリケーションの作成

以下の例では、2 つのパネルがある簡単なアプリケーションを作成します。アプリケーションの起動時は、一方のパネルのみが表示されます。目的は、最初のパネルのボタンをクリックして 2 つ目のパネルを開くことです。

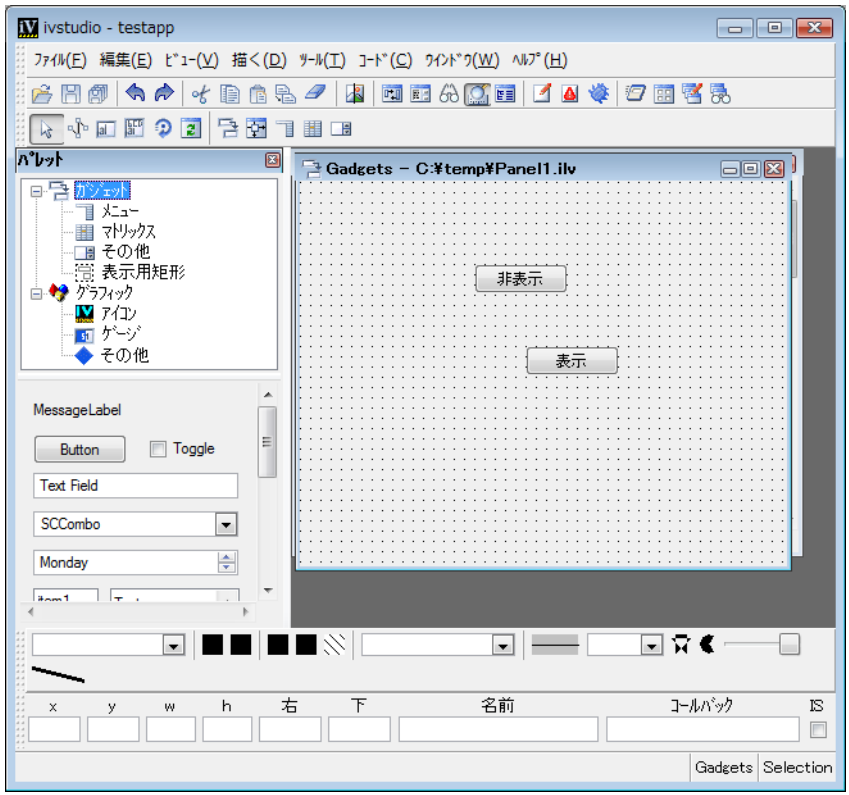
すでに IBM® ILOG® Views Studio で作業をしている場合は、開いているすべてのバッファを閉じるために、ウィンドウ・メニューから [すべてを閉じる] を選択します。次に、表示されるサブメニューの [ファイル] メニューおよびアプリケーションから [新規] を選択して、新しいアプリケーション・バッファ・ウィンドウを開きます。

最初のパネルの作成

最初のパネルを作成するには、次の手順に従います。

1. ファイル・メニューから [新規] を選択し、サブメニューの [ガジェット] を選択して新しい Gadgets ウィンドウ・バッファを開きます。
2. パレット・パネルの上側ペインでツリーの [ガジェット] をクリックします。
3. 2 つのボタンをパレット・パネルの下側ペインから Gadgets バッファ・ウィンドウへドラッグします。
4. ボタンをダブルクリックして、関連する詳細設定パネルを開きます。
5. [一般] ページの [名前] フィールドに、`ShowButton` および `HideButton` を入力します。
6. [詳細] ページの [ラベル] フィールドに「表示」および「非表示」を入力します。
7. パネルが適切な大きさになるようにリサイズします。
8. Gadgets バッファ・ウィンドウを任意のディレクトリに `panel1.ilv` として保存します。

   Panel1 は、次のように表示されます。



9. メイン・ウィンドウ・ツールバーの [パネル・クラス・エディタ] アイコンをクリックして、クラス・パレットを開きます。
10. クラス・パレットの [パネル・クラスの登録] アイコンをクリックして、Panel1 パネル・クラスを作成します。

    これを行うには、`panel1.ilv` Gadgets バッファ・ウィンドウがアクティブになっていることを確認してください。
11. アプリケーション・バッファ・ウィンドウをクリックして、これを前面に移動します。

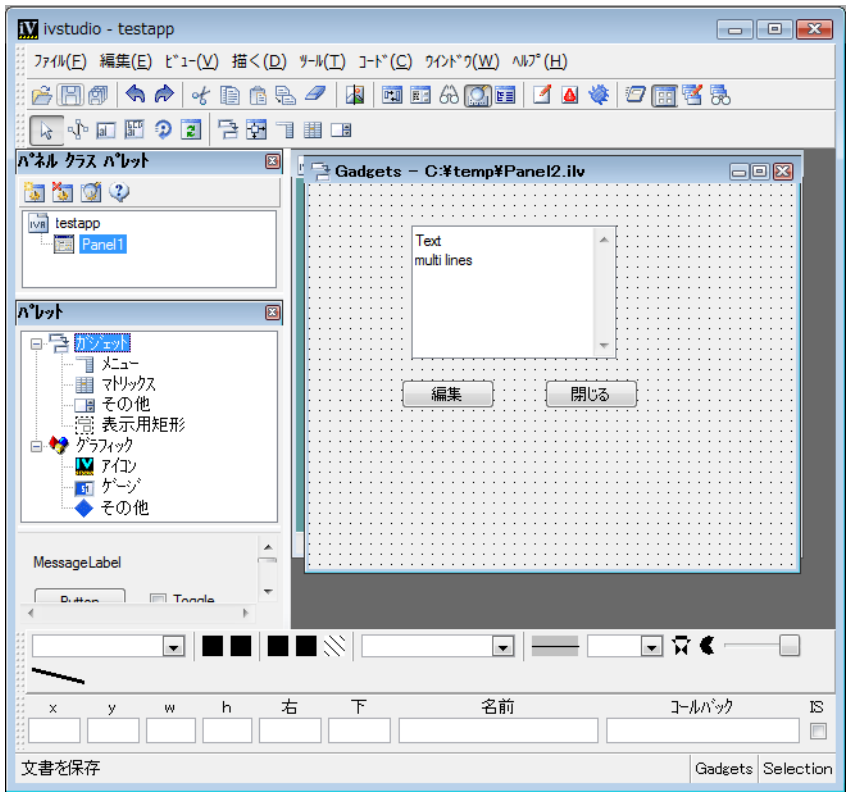
    [ウィンドウ] メニューから <Application> を選択することもできます。
12. [Panel1] アイコンをクラス・パレットからドラッグして、アプリケーション・ウィンドウ・バッファ内にドロップします。

2 番目のパネルの作成

次に 2 番目のパネルを作成します。

1. ファイル・メニューから [新規] を選択し、サブメニューの [ガジェット] を選択して新しい Gadgets バッファ・ウィンドウを開きます。
2. 必要に応じて、パレット・パネルの上側ペインで、ツリーの [ガジェット] をクリックします。
3. 複数行テキスト・ガジェット (`IlvText`) をカレント・バッファ・ウィンドウへドラッグします。
4. テキスト・ガジェットをダブルクリックして、その詳細設定パネルを開きます。
5. [一般] ページの [名前] フィールドに `Text` を入力します。
6. テキストの下に 2 つのボタンをドラッグします。
7. ボタンをダブルクリックして、関連する詳細設定パネルを開きます。
8. [一般] ページの [名前] フィールドに `EditButton` および `CloseButton` を入力します。
9. [詳細] ページの [ラベル] フィールドに「編集」および「閉じる」を入力します。
10. バッファを、同じディレクトリに `panel2.ilv` として保存します。

    Panel2 は、次のように表示されます。



11. クラス・パレットの [パネル・クラスの登録] アイコンをクリックして、Panel2 パネル・クラスを作成します。

    これを行うには、`panel2.ilv` Gadgets バッファ・ウィンドウがアクティブになっていることを確認してください。

12. アプリケーション・バッファ・ウィンドウをクリックして、これを前面に移動します。

    [ウィンドウ] メニューから <Application> を選択することもできます。

13. [Panel2] アイコンをクラス・パレットからドラッグして、アプリケーション・ウィンドウ・バッファ内にドロップします。

14. Panel2 のタイトル・バーをダブルクリックして、その詳細設定を開きます。

15. パネル・インスタンスの詳細設定にある [一般] ページで [表示] トグル・ボタンをオフにします。

16. メイン・ウィンドウ・ツールバーの [テスト] ボタンをクリックします。

アプリケーション起動時に Panel2 が非表示になっているのがわかります。

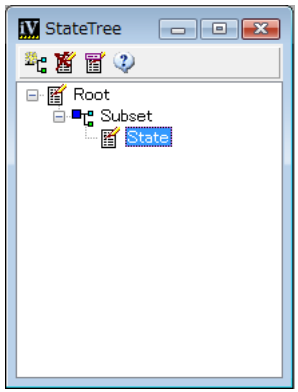
再度 [テスト] ボタンをクリックして、テスト・パネルを閉じます。

17. アプリケーションを、同じディレクトリに `myapp.iva` として保存します。

状態パネル

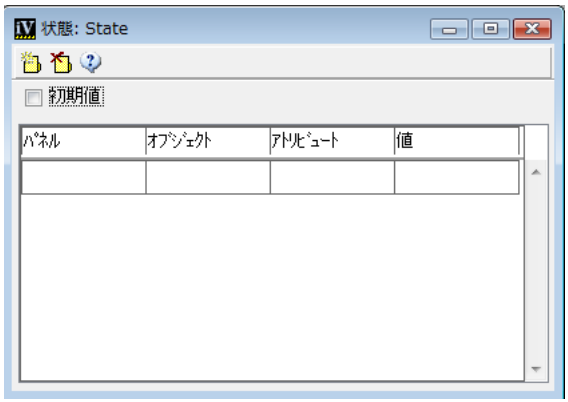
IBM® ILOG® Views Studio では、状態を編集する次の 2 つのパネルを提供します。

- アプリケーションの状態階層全体を管理するための状態ツリー。



図A.1 *状態ツリー・パネル*

- 状態ツリーで選択された状態のプロパティを詳細設定するための状態の詳細情報パネル



図A.2 *状態の詳細情報パネル*

IBM ILOG Views Studio のコマンド・パネルを使用してこれらのパネルを開くことができます。メイン・ウィンドウ・ツールバーのコマンド・アイコンをクリックします。次に、コマンド・パネルのコマンド・リストから [EditStates] を選択します。

アプリケーションに定義された状態がない場合、Edit State コマンドはルート状態、サブセット、および状態を作成します。状態サブセットについては、この章で後述します。

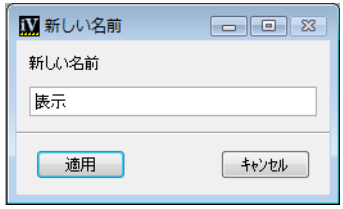
表示状態の編集

アプリケーションには、2 つの状態が必要です。最初の状態は、最初のパネルのみが表示状態にある場合の初期ルート状態です。2 番目の状態は、Panel2 が表示状態になっている状態です。

この 2 番目の表示状態に名前を付けるには

1. 状態ツリー・パネルで [State] を選択します。
2. 状態ツリー・ツールバーで [名前の変更] をクリックします。

選択した状態に新しい名前を入力するダイアログ・ボックスが開きます。



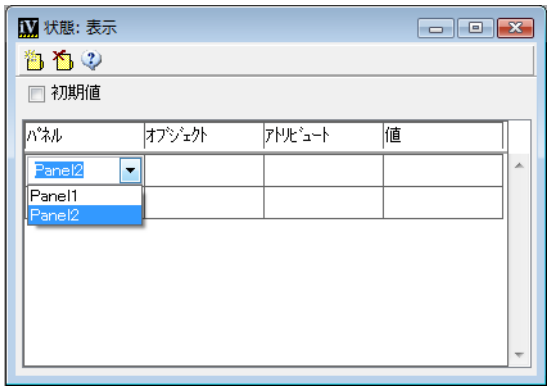
3. 「表示」と入力して [適用] をクリックします。

これで、アプリケーションに 2 つの状態、ルートおよび表示が設定されました。状態の詳細情報を使用して、各状態に条件を定義することができます。アプリケーションが表示状態にある時、Panel2 を表示するとします。

これを行うには、Panel2 の表示アトリビュートを true に設定します。

1. 必要に応じて、アプリケーション・バッファ・ウィンドウをアクティブにします。
2. 表示状態が、状態ツリー・パネルで選択されていることを確認してください。
3. ステータスの詳細情報内で、パネル列の最初の行をクリックします。

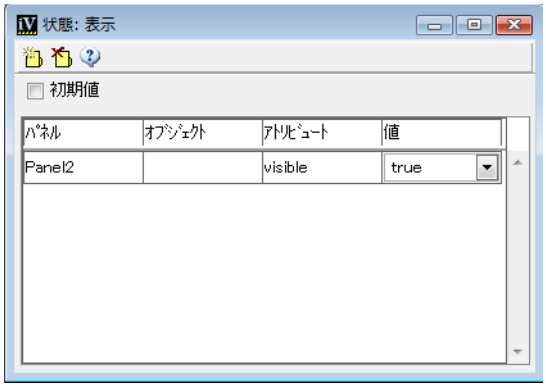
   アプリケーションからパネル・インスタンスのリストのあるコンボ・ボックスが表示されます。
4. リストの中から、[Panel2] を選択します。



5. [アトリビュート] をクリックします。

   パネルに関連する状態条件のリストが表示されます。
6. リストから [visible] を選択します。
7. [値] 列をクリックします。

   visible に関連する値のリストが表示されます。
8. リストから [true] を選択します。



この状態条件のターゲットはパネルです。それはパネル名で識別されます。状態条件のターゲットはオブジェクトではないので、オブジェクト列は空欄のままです。

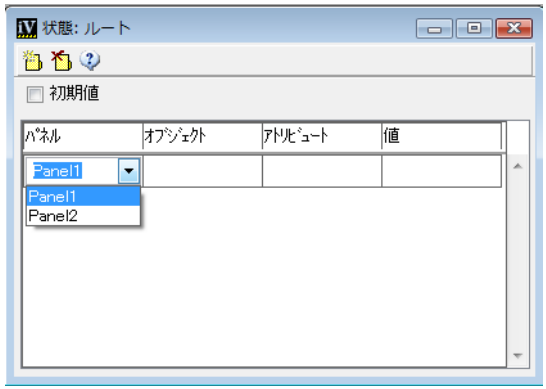
状態の連鎖

アプリケーションを起動すると、ルート状態が自動的に選択されます。[表示] ボタンをクリックして、表示状態に移動するとします。

表示状態を表示ボタンに付加するには、次の手順に従います。

1. 必要に応じて、アプリケーション・バッファ・ウィンドウをアクティブにします。
2. 状態ツリー・パネルでルート状態を選択します。
3. ステータスの詳細情報内で、パネル列の最初の行をクリックします。

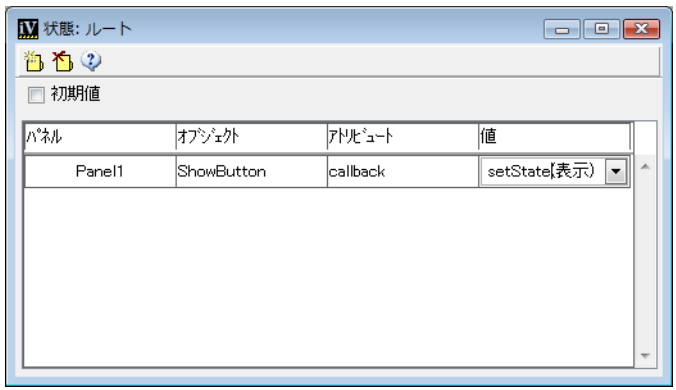
   アプリケーションからパネル・インスタンスのリストのあるコンボ・ボックスが表示されます。
4. リストから、[Panel1] を選択します。



5. オブジェクト列で、コンボ・ボックスのオブジェクト・リストから [ShowButton] を選択します。
6. [アトリビュート] 列をクリックします。

   `IlvButton` オブジェクトに関連する状態条件のリストが表示されます。
7. リストから [`callback`] を選択します。
8. シートの [値] 列をクリックします。

   関連するコールバックのリストが表示されます。
9. [`setState(ï\é¶)`] を選択します。



アプリケーションのテストを行う場合は、Panel1 のみが表示されます。しかし、ここでは [表示] ボタンをクリックすると Panel2 が表示され、表示状態になります。

[表示] ボタンのラベルおよびコールバックの変更

状態機構は、状態を設定したり、そのままにすることができる定義済みのコールバックを提供します。コールバックは、特定状態にあるオブジェトに付加された状態条件です。これは、異なる状態でオーバーライドすることができます。

次の練習問題では、アプリケーションを表示状態からルート状態に戻すための [表示] ボタンを設定します。また、表示状態が選択されたときにボタン・ラベルがルートを表示するように変更します。

1. 状態ツリー・パネルで表示状態を選択します。
2. 必要に応じて、新しい行を追加するためにステータス状態の詳細情報の左上にある [新しい条件] アイコンをクリックします。
3. ステータスの詳細情報内で、パネル列の空欄の行をクリックします。

   アプリケーションからパネル・インスタンスのリストのあるコンボ・ボックスが表示されます。
4. リストから、[Panel1] を選択します。
5. オブジェクト列で、コンボ・ボックスのオブジェクト・リストから [ShowButton] を選択します。
6. [アトリビュート] 列をクリックして、リストから `label` を選択します。

   `label` が表示されるまでリストをスクロールダウンします。
7. [値] フィールドに 「ルート」 を入力して、Enter を押します。

   新しい行は、最後の列に到達したときに Enter を押すと自動的に追加されます。
8. 新しい空欄列の [アトリビュート] 列をクリックします。

   パネルおよびオブジェクト名は自動的に 1 つ前の行からコピーされ、関連する状態条件名が表示されます。
9. `callback` を選択します。
10. [値] 列をクリックして、`leaveState( 表示 )` を選択します。

アプリケーションのテストを行い、([表示] ボタンをクリックして) 表示状態に移ると、[表示] ボタンのラベルは [ルート] に変わります。もう一度クリックすると、初期ルート状態に戻ります。[表示] ボタンのラベルおよびコールバックは回復され、Panel2 は非表示になります。つまり、状態をそのままにする時は、状態条件によって変更されたプロパティは回復されます。

サブ状態の作成：編集状態

同パネルが表示状態である場合、Panel2 のテキスト・フィールドを編集可能にしたくないとします。しかし、編集状態では、テキストを編集したいとします。

表示状態がアクティブの時に、Panel2 のテキスト・フィールドを編集不可にするには、次の手順に従います。

1. 状態ツリー・パネルで表示状態を選択します。
2. 必要に応じて、新しい行を作成するために状態の詳細情報の [新しい条件] アイコンをクリックします。
3. 新しい行のパネル列で、コンボ・ボックスのパネル・リストから Panel2 を選択します。
4. オブジェクト列で、コンボ・ボックスのオブジェクト・リストから [Text] を選択します。
5. [アトリビュート] 列をクリックして、リストから `editable` を選択します。

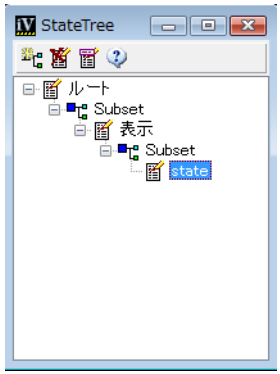
   条件はオブジェクト・タイプに依存しているため、関連する条件のリストはボタン用に選択したものとは違っていることがわかります。
6. [値] 列をクリックして、`false` を選択します。
7. アプリケーションをテストして、テキスト・フィールドが表示状態では編集不可であることをチェックします。

ここからは、表示条件から継承される表示のサブ状態、編集サブ状態を定義します。そのためには、次の手順に従います。

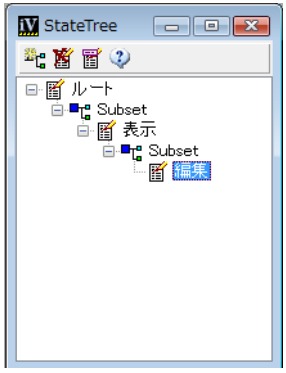
1. 状態ツリー・パネルで表示状態を選択します。
2. 状態ツリー・ツールバーで [新しいサブセット] アイコンをクリックします。

   新しいツリー・アイテムが作成、選択されます。ツールバーが少し変更になっていることがわかります。新しいサブセット・アイコンが新しい状態アイコンに置き換わっています。
3. ツールバーで [新しいステータス] アイコンをクリックします。

   新しい状態アイテムがツリー内に作成されました。

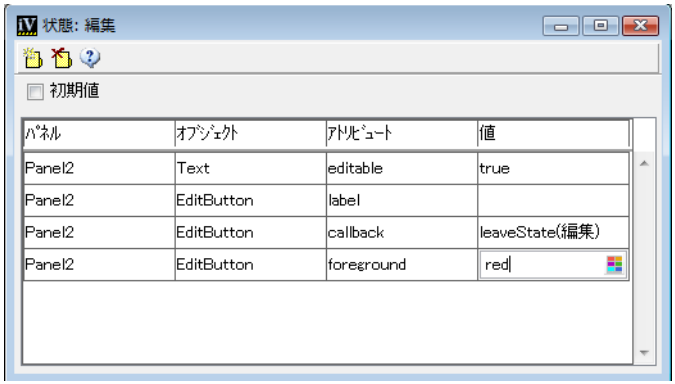


4. 状態の名前を [編集] に変更するために [名前の変更] をクリックします。



5. 状態ツリーで選択した編集状態をそのままにするか、必要に応じて選択します。
6. 必要に応じて、新しい行を作成するために状態の詳細情報の [新しい条件] アイコンをクリックします。
7. 空欄の行のパネル列で、コンボ・ボックスから Panel2 を選択します。
8. オブジェクト列で、オブジェクトのリストから [Text] を選択します。
9. [アトリビュート] 列で `editable` を選択します。
10. [値] 列で `true` を選択します。
11. ステータスの詳細情報ツールバーの新規条件をクリックして、ステータスの詳細情報パネルに新しい行を作成します。
12. パネル列の Panel2 およびオブジェクト列の [EditButton] を選択します。

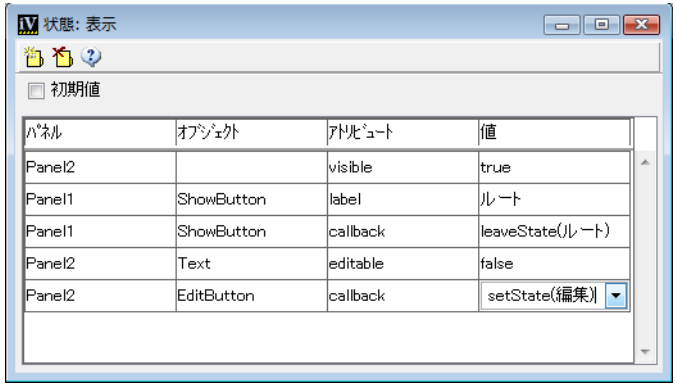
13. [アトリビュート] 列で label を選択します。
14. [値] 列に Apply を入力し、Enter を押します (あるいは、ツールバーの新規条件をクリックします)。
15. すぐ下の空欄の行で、[アトリビュート] 列をクリックして callback を選択します。
16. [値] 列で leaveState(ï"èW) を選択します。
17. このボタンに別の状態条件を作成し、そのアトリビュートとして foreground を選択します。
18. [値] 項目をクリックして、色のセレクタから色を選択します (例 : 赤)。



状態の詳細情報パネルの [値] 列をクリックしたとき、詳細設定の振る舞いは選択されたアトリビュートに依存することに注意してください。定義済みの値の文字列リストあるいは専用セレクタがアクティブになる場合があります。

表示状態に戻り、編集状態を Panel2 の [編集] ボタンに付加します。

1. 状態ツリー・パネルで表示状態を選択します。
2. 状態の詳細情報で、空欄の行を作成します。
3. パネル列のコンボ・ボックスから Panel2 を、そしてオブジェクト列のコンボ・ボックスから [EditButton] を選択します。
4. [アトリビュート] 列で callback を選択します。
5. [値] 列で setState(編集) を選択します。



6. ここで、アプリケーションのテストを行います。表示状態になっている場合は、Panel2 の [編集] ボタンをクリックして編集状態にします。

次のような外観になるように、状態条件を変更、追加して、アプリケーションの状態を拡張することもできます。

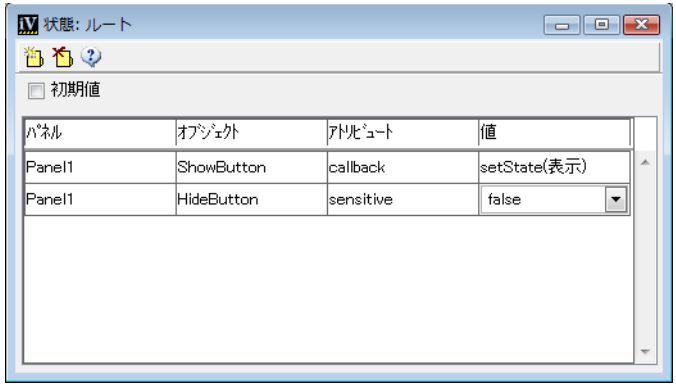
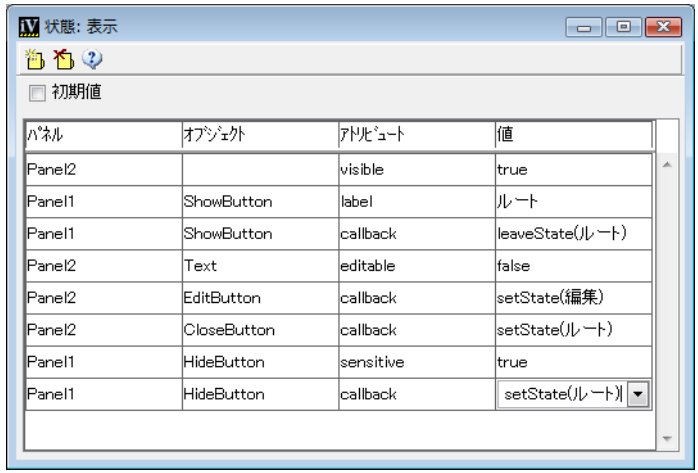
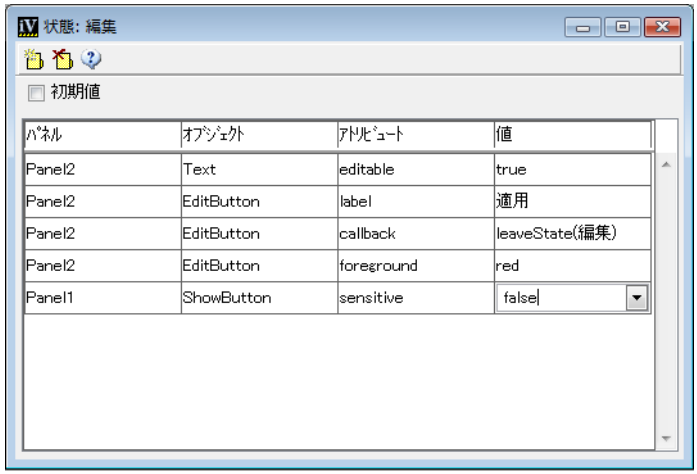


図 A.3 *ステータスの詳細情報のルート・シート*



図A.4 *ステータスの詳細情報の表示シート*



図A.5 *ステータスの詳細情報の編集シート*

状態ファイル

状態を定義したアプリケーションを保存するとき、状態定義は `.ivs` ファイルにアプリケーション・ファイルと同じ名前、同じディレクトリで保存されます。該当するアプリケーション・ファイルを読み込むときに、このファイルは自動的に読み込まれます。生成された C++ コードでは、定義された状態条件が存在する場合、状態ファイルが読み込まれます。状態ファイルは、`ILVPATH` 環境変数 (あるいは同等のリソース) によって指定されたディレクトリ内に配置される必要があります。

索引

数字

A

B

H

I

S

き

く

こ

さ

し

す

せ

の

は

ひ

め

も