IBM ILOG Views

Gadgets V5.3

チュートリアル

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

前書き　　このチュートリアルについて 3

前提事項 3

表記法 3

書体の規則 3

命名規則 4

チュートリアル 1　　ViewFile アプリケーション : シンプルなファイル・ブラウザの作成 3

ステップ 1: ブラウザ・ウィンドウの構築 4

アプリケーションに適切なガジェットを選択する 4

コンテナの選択 5

メイン・ウィンドウの実装 6

アプリケーションの作成 8

アプリケーションのプレビュー 10

ステップ 2: ファイルのブラウズ 10

ファイル・ブラウザの作成 11

ガジェット・アイテムの作成 18

ファイル・ビューアのインスタンス化 19

アプリケーションのプレビュー 20

ステップ 3: ドッキング・バーの追加 21

メイン・ウィンドウのドッキングを準備する 21

ドッキング・ツールバーの作成 23
ドッキング・メニュー・バーの作成 25
アプリケーションに変更を統合する 26
アプリケーションのプレビュー 27
ステップ 4: ビュー・フレームの追加 27
デスクトップ・ビューの選択 27
ビュー・フレームの作成 30
新しいメニューおよびアイテムをビュー・フレームへ追加する 31
アプリケーションのプレビュー 34
チュートリアル 2　ガジェットのカスタマイズ 37
ステップ 1: グラフィカルな外観変更によるガジェットの拡張 37
既存ガジェットのサブクラスを作成する 38
API を新規ガジェット・クラスへ追加する 40
新規ガジェットの描画方法を変更する 43
新規ガジェット・クラスのテスト 45
ステップ 2: 振る舞い変更によるガジェットの拡張 46
既存ガジェットの振る舞いを変更する適切な方法の選択 47
汎用インタラクタの作成 47
専用インタラクタの作成 49
新しいインタラクタのテスト 52
ステップ 3: 複合ガジェットの作成 52
他のガジェットから構成されるガジェットの作成 53
ガジェットでキーボード・フォーカスを処理する 56
ガジェットでのイベント処理 57
ガジェットへコールバックを追加する 59
複合ガジェットのテスト 59
索引 61

このチュートリアルについて

本チュートリアルでは、シンプルなファイル・ブラウザの作成方法およびガジェットのカスタマイズ方法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。

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;
  ```

章

1

ViewFile アプリケーション: シンプルなファイル・ブラウザの作成

このチュートリアルでは、IBM® ILOG® Views Gadgets API を使用して ViewFile アプリケーションを作成する方法を説明します。ViewFile アプリケーションは、ハードディスクに保存されているファイルのリストをブラウズするための小さなグラフィカル・ユーザ・インターフェースです。

> ***メモ:*** IBM ILOG Views *GUI エディタである* IBM ILOG Views *Studio を使用して同じ種類のアプリケーションを構築することもできます。*

このチュートリアルでは、ツリー・ガジェット、シート、ツールバーなどの多くの異なる IBM ILOG Views ガジェットを組み合わせてグラフィカルで高度に直観的なアプリケーションを作成する方法を学びます。最終アプリケーションには、ドッキング・ツールバーや複数フレームなどの洗練された機能が含まれます。

このチュートリアルには、4 つのステップがあります。

- *ステップ1: ブラウザ・ウィンドウの構築*
- *ステップ2: ファイルのブラウズ*
- *ステップ3: ドッキング・バーの追加*
- *ステップ4: ビュー・フレームの追加*

完成したアプリケーションは、次のようになります。

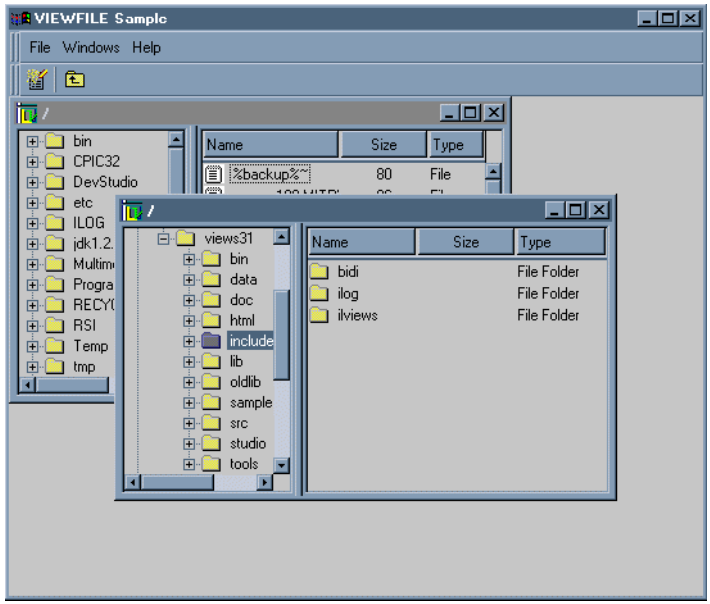


図 1.1 *最終 ViewFile アプリケーション*

ステップ 1: ブラウザ・ウィンドウの構築

この最初のステップでは、ファイル階層を表示するウィンドウの作成方法を説明します。

次のタスクを実行する方法を紹介します。

- *アプリケーションに適切なガジェットを選択する*
- *コンテナの選択*
- *アプリケーションの作成*
- *アプリケーションのプレビュー*

アプリケーションに適切なガジェットを選択する

アプリケーション・ウィンドウは、2 つの部分から構成されています。

- ウィンドウの左側は、ハードディスク内のフォルダをツリー構造で表示します。

- ウィンドウの右側は、左側で開かれたフォルダに含まれるファイルを表示します。

IBM® ILOG® Views では、アイテムの階層リストは、クラス `IlvTreeGadget` によって表現されます。このクラスは、ウィンドウの左側を構築するのに使用されます。ウィンドウの右側を表現するには、`IlvSheet` が使用されます。このクラスのインスタンスは各種情報を表示できるため、表示オブジェクトの種類を変更することなくデータ表示を変更することができます。

IBM ILOG Views は多くのガジェットを提供しているため、アプリケーションに最適なガジェットを見つけのるが困難な場合もあるかもしれません。
IBM ILOG Views Gadgets の主要プロパティの一覧は、IBM ILOG Views Gadgets の概要の章を参照してください。各ガジェットについては、*IBM ILOG Views Gadgets ユーザ・マニュアル*の関連するセクションに詳細があります。

ニーズにあったガジェットが見つからない場合、既存のガジェットをサブクラス化してそのルックあるいは振る舞い、またはその両方を変更することができます。

コンテナの選択

`IlvTreeGadget` および `IlvSheet` オブジェクトは、これらを表示するコンテナに追加します。複数の種類からコンテナを選択できますが、ViewFile アプリケーションに適しているのは 2 つのみ、`IlvGadgetContainer` および `IlvPanedContainer` です。

`IlvGadgetContainer` は、ガジェットを格納する `IlvContainer` のサブクラスです。これは、キーボード・フォーカスおよびアタッチメントを処理します。これら 2 つの機能に関しては、ガジェットの理解を参照してください。

`IlvPanedContainer` は `IlvGadgetContainer` のサブクラスです。ガジェット・コンテナとは異なり、ペイン・コンテナはグラフィック・オブジェクトを処理しません。代わりに、`IlvPane` クラスの特殊オブジェクトを処理します。ペイン・コンテナの水平または垂直方向に応じて、ペインは左から右、あるいは上から下へと配列されます。`IlvPane` は、2 つの定義済みサブクラスを持ち、そのうち 1 つはグラフィック・オブジェクトを格納する `IlvGraphicPane` クラスです。ペインの詳細については、ペインを参照してください。

`IlvPanedContainer` は、このアプリケーションにより適しています。これにより同じ高さの隣接する 2 つのペインから構成されるウィンドウを作成することができるからです。 1 つはツリー・ガジェットを保持し、もう 1 つはシートを保持するためのものです。また、`IlvGadgetContainer` によって提供される複雑なアタッチメント・モデルは必要ありません。しかし、どちらのタイプのコンテナも、同じアプリケーションに組み合わせることができます。

たとえば、ダイアログ・ボックスおよび複雑なグラフィック・パネルの構築にガジェットを使用でき、それによりこれらが有する精巧なアタッチメント・モデル

を利用することができます。さらに、これらをグローバル・アプリケーション・レベルでペインにカプセル化できます。

メイン・ウィンドウの実装

メイン・ウィンドウは、`IlvPanedContainer` のサブクラスである `FileViewerWindow` クラスで実装します。このクラスは、`viewerw.h` ファイルで宣言し、`viewerw.cpp` ファイルで定義します。

ペインの作成

`FileViewerWindow` クラスは、自動的に 2 つのグラフィック・ペインを作成します。1 つはツリーのカプセル化、もう 1 つはシートのカプセル化用です。これらのペインは、ここに示すように `FileViewerWindow::initLayout` メンバ関数で作成

されます。(このメンバ関数は、FileViewerWindow コンストラクタから呼び出されます)。

```
void
FileViewerWindow::initLayout(const IlvRect& size)
{
    // Create the tree gadget in which the folder hierarchy will
    // be displayed.
    IlvTreeGadget* tree =
        new IlvTreeGadget(getDisplay(),
                          IlvRect(0,
                          0,
                          IlvMax((IlvDim)100,
                                 (IlvDim)(size.w()/3)),
                          size.h()));
    // Encapsulate the tree gadget into a graphic pane and name it
    // DirectoryHierarchy.
    IlvGraphicPane* treePane = new IlvGraphicPane("DirectoryHierarchy", tree);
    // The pane is set to resizable. It is possible to
    // resize it using a splitter interactively.
    treePane->setResizeMode(IlvPane::Resizable);
    // Set also a minimum size on the horizontal direction.
    treePane->setMinimumSize(IlvHorizontal, 100);
    // Add the pane to the container.
    addPane(treePane);

    // Create the sheet.
    IlvSheet* sheet =
        new IlvSheet(getDisplay(),
                     IlvRect(0, 0, size.w(), size.h()),
                     1,
                     1,
                     size.w()/4,
                     25,
                     2,
                     IlvFalse,
                     IlvFalse);
    // Encapsulate it into a graphic pane giving FileList as name.
    IlvGraphicPane* listPane = new IlvGraphicPane("FileList", sheet);
    // The sheet is elastic: when the container will be resized, only the
    // sheet will be resized (because the tree is only resizable, not elastic).
    listPane->setResizeMode(IlvPane::Elastic);
    // Add the pane to the container.
    addPane(listPane);
    // Update the container.
    updatePanes();
}
```

initLayout メンバ関数は、まずツリー・ガジェットを作成します。次に、これをグラフィック・ペインにカプセル化して、ペイン・コンテナに追加します。同様

に、シートを作成して、グラフィック・ペインにカプセル化して、ペイン・コンテナに追加します。

> ***メモ:*** *シートには、行列が1 つしかありません。この段階ではブラウズするファイルが表示されたときにシートがどのように見えるのかはっきりしないためです。このシートは伸縮自在でツリーが残したスペースを占めるため、パラメータ (*`size.w(),4`*) で指定されたシートの幅は、無意味であることに注意してください。*

メンバ関数 `IlvPanedContainer::updatePanes` は、2 つのグラフィック・ペインの追加を有効にするために呼び出されます。詳細については、ペインの「ペイン・コンテナのレイアウトを変更する」のセクションを参照してください。

> ***メモ:*** *メンバ関数* `IlvPanedContainer::updatePanes` *を呼び出すと、スライダ・ペインをツリーとシート・ペインの間に挿入します。この2 つはリサイズ可能なため自動的に挿入されます。*

アプリケーションの作成

これでファイル階層表示用クラスが完成し、インスタンス化できます。これをインスタンス化するには、`IlvApplication` のサブクラスである `FileViewerApplication` を使用します。`IlvApplication` は、1 セットのパネルから構成される基本的なアプリケーションを表します。IBM® ILOG® Views Studio では、アプリケーションのコード生成にこれを使用します。

`FileViewerApplication` クラスは、`viewfile.h` ファイルで宣言し、`viewfile.cpp` ファイルで定義します。

メイン・ウィンドウの作成

`IlvApplication` クラスは、プログラムの最初にアプリケーション・パネルを作成するために呼び出される `makePanels` 仮想メンバ関数を含んでいます。

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();
    // Show it.
    getMainWindow()->show();
}
```

メンバ関数 FileViewerApplication::initMainWindow は、FileViewerWindow のインスタンスを作成します。

```
void
FileViewerApplication::initMainWindow()
{
    // Create the main window.
    IlvRect rect(0, 0, 500, 300);
    IlvContainer* mainWindow = createMainWindow(rect);
    // Name it to be able to retrieve it.
    mainWindow->setName(getName());
    // Quit the application when the user asks for termination.
    mainWindow->setDestroyCallback(IlvAppExit);
    // Add the panel to the application.
    addPanel(mainWindow);
}
```

メンバ関数 createMainWindow は、クラス FileViewerWindow のインスタンスを返します。

```
IlvContainer*
FileViewerApplication::createMainWindow(const IlvRect& rect) const
{
    return new
    FileViewerWindow(getDisplay(),getName(),getName(),rect,IlvFalse);
}
```

アプリケーションのインスタンス化

FileViewerApplication クラスは、主要エントリ・ポイントでインスタンス化されます。

```
int main(int argc, char* argv[])
{
    IlvSetLocale();
    FileViewerApplication* appli =
        new FileViewerApplication("VIEWFILE Sample", 0, argc, argv);
    if (!appli->getDisplay())
        return -1;
    appli->run();
    return 0;
}
```

> ***メモ:*** IlvSetLocale *への呼び出しは、単に* IBM ILOG Views *に現在の場所を使用するように伝えます。アプリケーションをローカライズしたい場合は、この関数を呼び出さなくてはなりません。詳細については、*IBM ILOG Views *の「国際化」にある「ローカライズされた環境で実行するプログラムの作成」を参照してください。*

アプリケーションのプレビュー

これでステップ 1 は完了です。アプリケーションは次のように見えるはずです。

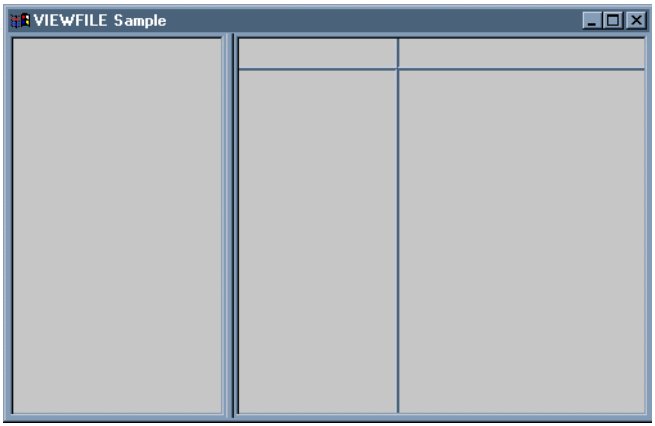


図 1.2 *ステップ 1 完了後の ViewFile アプリケーション*

ステップ 2: ファイルのブラウズ

4 ページの *ステップ 1: ブラウザ・ウィンドウの構築*で、ファイルの階層リストを表示するグラフィック・オブジェクトの作成方法を学びました。この 2 番目のステップでは、ファイル・ブラウザの定義方法およびインスタンス化の方法を説明します。つまり、グラフィック・オブジェクトにデータを埋めるのに使用されるオブジェクトです (ツリーおよびシート)。ハードディスクをブラウズするのに使用されるクラスは `FileViewer` です。これは、`viewer.h` ファイルで宣言し、`viewer.cpp` ファイルで定義します。

このステップでは、次のタスクを実行する方法を紹介します。

- *ファイル・ブラウザの作成*
- *ガジェット・アイテムの作成*
- *ファイル・ビューアのインスタンス化*
- *アプリケーションのプレビュー*

> ***メモ:*** *このステップ用のコードに見られるクラス* `IlPathName` *および* `IlString` *は、コードを読みやすくするために使用されます。*

ファイル・ブラウザの作成

FileViewer クラスは、長くて複雑です。このセクションではその主要部分を説明します。

- *コンストラクタ*
- *ツリーの初期化*
- *シートの初期化*
- *ディレクトリのスキャン*
- *シートの更新*

コンストラクタ

FileViewer コンストラクタは、IlvTreeGadget および IlvSheet をそのパラメータとして必要とします。両方のオブジェクトは initObjects メンバ関数によって初期化されます。

```
FileViewer::FileViewer(IlvTreeGadget* tree, IlvSheet* sheet)
: _tree(tree), _sheet(sheet)
{
    // Initialize the gadgets.
    initObjects();
}
```

ツリーの初期化

ツリー・オブジェクトは initTree メンバ関数で初期化されます。

```
void
FileViewer::initTree()
{
    // Set up the tree gadget.
    _tree->removeAllItems(IlvFalse);
    _tree->removeCallback(IlvTreeGadget::ExpandCallbackType(),
                          TreeItemExpanded);
    _tree->addCallback(IlvTreeGadget::ExpandCallbackType(),
                       TreeItemExpanded,
                       this);
    _tree->removeCallback(IlvTreeGadget::SelectCallbackType(),
                          TreeItemSelected);
    _tree->addCallback(IlvTreeGadget::SelectCallbackType(),
                       TreeItemSelected,
                       this);
}
```

このメンバ関数は、ツリーの全アイテムを削除して、ツリーで新しいフォルダが選択されたときにシートを更新する選択コールバックおよび、ツリー・アイテムが展開されるときにトリガされるコールバックの 2 つを設定します。この 2 つ目のコールバックは、FileViewer オブジェクトが、操作速度を遅らせる原因となる一

度でのファイル階層全体の読み込みを行わないようにします。フォルダは展開された時にのみ、読み込まれます。

> ***メモ:*** *各コールバックは、同じコールバックが2 度登録されるのを避けるために追加される前に削除されます*

詳細は、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。

シートの初期化

シートは、initSheet メンバ関数で初期化されます。

```
void
FileViewer::initSheet()
{
    // Set up the sheet.
    _sheet->scrollBarShowAsNeeded(IlvTrue, IlvFalse);
    _sheet->hideScrollBar(IlvHorizontal, IlvFalse);
    _sheet->adjustLast(IlvTrue);
    _sheet->reinitialize(3, 1);
    _sheet->setNbFixedColumn(0);
    _sheet->scrollToColumn(0);
    _sheet->setExclusive(IlvTrue);
    _sheet->allowEdit(IlvFalse);

    // Create the header.
    _sheet->set(0, 0, new IlvLabelMatrixItem("Name"));
    _sheet->setItemAlignment(0, 0, IlvLeft);
    _sheet->setItemRelief(0, 0, IlvTrue);
    _sheet->setItemSensitive(0, 0, IlvFalse);
    _sheet->setItemGrayed(0, 0, IlvFalse);
    _sheet->set(1, 0, new IlvLabelMatrixItem("Size"));
    _sheet->setItemRelief(1, 0, IlvTrue);
    _sheet->setItemSensitive(1, 0, IlvFalse);
    _sheet->setItemGrayed(1, 0, IlvFalse);
    _sheet->set(2, 0, new IlvLabelMatrixItem("Type"));
    _sheet->setItemAlignment(2, 0, IlvLeft);
    _sheet->setItemRelief(2, 0, IlvTrue);
    _sheet->setItemSensitive(2, 0, IlvFalse);
    _sheet->setItemGrayed(2, 0, IlvFalse);
}
```

このメンバ関数は、3 つの列と 1 つの行のみでシート・ヘッダーを表示するようにシートを変更します。下図に示すように、列 1 はファイル名、列 2 はファイル・サイズ、列 3 はファイル・タイプの情報を提供します。

メモ: ヘッダーを構成するアイテムは、非センシティブに設定されています。つまり、ユーザはこれらを選択することができません。

ディレクトリのスキャン

`fillTree` メンバ関数は、ツリーが初期化されるとツリーをデータで埋めます。任意のディレクトリの内容を読み込み、対応するアイテムをツリーに作成します。

```
void
FileViewer::fillTree(IlvPathName& path, IlvTreeGadgetItem* parent)
{
    // Read the 'path' directory and insert each sub-folder as a child of
    // the item 'parent'.
    if (path.openDir()) {
        IlvPathName file;
        while (path.readDir(file)) {
            if (file.isDirectory() &&
                file.getDirectory(IlvFalse) != IlvString(".") &&
                file.getDirectory(IlvFalse) != IlvString("..")) {
                IlvTreeGadgetItem* item =
                     (IlvTreeGadgetItem*)createFileItem(file, _tree);
                item->setUnknownChildCount(IlvTrue);
                // Insert the directory 'file' into parent.
                parent->insertChild(item);
                // Compute the absolute path name.
                IlvPathName* absPathName=new IlvPathName(path);
                absPathName->merge(file);
                item->setclientData(absPathName)
            }
        }
        path.closeDir();
        // Sort the added items.
        _tree->sort(parent, 1);
    }
}
```

`FileViewer::createFileItem` メンバ関数は、各ツリー・アイテムを作成します。これらは、親アイテム、つまり読み込まれているフォルダに追加されます。追加されたアイテムは、`IlvTreeGadgetItemHolder::sort` メンバ関数によってソートされます。

createFileItem の内容は、18 ページの*ガジェット・アイテムの作成*に記述されています。

メモ:

1. *メンバ関数* IlvTreeGadgetItem::setUnknownChildCount *が、展開コールバックが呼び出されることを確実にするため、追加された各アイテム用に呼び出されます。詳細については、共通ガジェットの使用の「アイテム特性の変更」を参照してください。*
2. *アイテムのクライアント・データは、アイテムがポイントするディレクトリを参照するパス・オブジェクトを保存するのに使用されます。このオブジェクトは、毎回アイテムの絶対パスを再計算することを避けるキャッシュです。*

fillTree メソッドは、ツリーの 1 つのノードのみを埋めます。ツリー全体は、実際には展開コールバックによって構築されます。

```
static void
TreeItemExpanded(IlvGraphic* g, IlvAny arg)
{
    IlvTreeGadgetItem* item = ((IlvTreeGadget*)g)->getCallbackItem();
    if (!item->hasChildren()) {
        FileViewer* viewer = (FileViewer*)arg;
        IlvPathName* path = (IlvPathName*)item->getClientData();
        viewer->updateTree(*path, item);
     }
}
```

このコールバックは、折りたたまれたフォルダを展開しようとするたびに呼び出されます。arg パラメータは、コールバックが設定されたときに指定されたとおり、FileViewer オブジェクトへのポインタです。アイテムのクライアント・データとして保存されているパスは、ツリー更新のために取得、使用されます。フォルダが一度も開かれていない場合のみツリーを更新するので、updateTree メソッドを呼び出す前に展開されたアイテムには子がないことを確認してください。

```
void
FileViewer::updateTree(IlvPathName& path, IlvTreeGadgetItem* item)
{
     _tree->initReDrawItems();
     // Reset the tree, if needed.
     if (!item)
         initTree();
     // Fill it using the 'path' directory.
     fillTree(path, item? item : _tree->getRoot());
     // Finally, redraw.
     _tree->reDrawItems();
}
```

このメンバ関数は、fillTree メンバ関数を呼び出し、ツリーのクリーニングやスマート再描画処理などの追加操作を実行します。ガジェット・アイテムの「ガジェット・アイテムの再描画」を参照してください。

シートの更新

IlvSheet オブジェクトによってウィンドウの右側に表示されているファイルのリストは、ツリーで選択されているアイテムが変更するたびに更新されなくてはなりません。更新は、IlvTreeGadget クラスの選択コールバックによって実行されます。

```
static void
TreeItemSelected(IlvGraphic* g, IlvAny arg)
{
    // Retrieve the item that has triggered the callback.
    IlvTreeGadgetItem* item = ((IlvTreeGadget*)g)->getCallbackItem();
    // In the case of a selection (this callback is also called when an item
    // is deselected) update the sheet.
    if (item->isSelected()) {
        // Retrieve the path of the item.
        IlvPathName* pathname = (IlvPathName*)item->getClientData();
        // Retrieve the file viewer instance.
        FileViewer* viewer = (FileViewer*)arg;
        // Update the sheet.
        viewer->updateSheet(*pathname);
    }
}
```

このコールバックは、新しいアイテムがツリーで選択されるたびに呼び出されます。コールバックをトリガしたアイテムがメンバ関数 IlvTreeGadget::getCallbackItem によって取得されると、アイテムが選択されているかどうかをチェックする必要があります (ツリー選択コールバックはアイテ

ムが選択解除されたときも呼び出されるため)。次に、アイテムのクライアント・データに保存されているパスが、updateSheet メンバ関数に渡されます。

```
void
FileViewer::updateSheet(IlvPathName& path)
{
    _sheet->initReDrawItems();
    // Reset the sheet.
    initSheet();
    // Read all the files contained in 'path'.
    if (path.openDir()) {
        IlvPathName file;
        while (path.readDir(file)) {
            if (!file.isDirectory() ||
                (file.getDirectory(IlvFalse) != IlvString(".") &&
                 file.getDirectory(IlvFalse) != IlvString("..."))) {
                if (!file.isDirectory())
                    file.setDirectory(path);
                // Add the file to the sheet.
                addFile(file);
             }
         }
         path.closeDir();
    }
    // Recompute the column sizes.
    _sheet->fitWidthToSize();
    // Invalidate the whole sheet.
    _sheet->getHolder()->invalidateRegion(_sheet);
    // Finally, redraw.
    _sheet->reDrawItems();
}
```

このメンバ関数は、ディレクトリ・パスに指定されている全ファイルを addFile メンバ関数を使用してシートに追加します。シートの再描画も管理します。

```
void
FileViewer::addFile(const IlvPathName& file)
{
    // Create the gadget item that will be inserted into the sheet.
    IlvGadgetItem* item = createFileItem(file, _sheet);
    // Encapsulate it with a matrix item.
    IlvAbstractMatrixItem* mitem = new IlvGadgetItemMatrixItem(item);
    // Add a new row in the matrix .
    _sheet->insertRow((IlvUShort) -1);
    IlvUShort row = (IlvUShort)(_sheet->rows() - 1);

    // Set the item in the first column.
    _sheet->set(0, row, mitem);
    _sheet->resizeRow((IlvUShort)(row + 1), item->getHeight() + 1);
    _sheet->setItemAlignment(0, row, _sheet->getItemAlignment(0, 0));
    // File Size in the second column.
    _sheet->setItemSensitive(1, row, IlvFalse);
    _sheet->setItemGrayed(1, row, IlvFalse);
    _sheet->setItemAlignment(1, row, _sheet->getItemAlignment(1, 0));
    ifstream ifile(file.getString(), IlvBinaryInputStreamMode);
    if (!(!ifile)) {
        ifile.seekg(0, ios::end);
        streampos length = ifile.tellg();
        mitem = new IlvIntMatrixItem(length);
        _sheet->set(1, row, mitem);
    }
    // File Type in the third column.
    mitem = new IlvLabelMatrixItem(file.isDirectory()
                                   ? "File Folder"
                                   : "File");
    _sheet->setItemSensitive(2, row, IlvFalse);
    _sheet->setItemGrayed(2, row, IlvFalse);
    _sheet->setItemAlignment(2, row, _sheet->getItemAlignment(2, 0));
    _sheet->set(2, row, mitem);
}
```

このメンバ関数は、その引数として提供されているファイル・パラメータに対応するアイテムを作成し、これをシートの最初の列に追加します。それから、ファイル・サイズを取得して、これを IlvIntMatrixItem オブジェクトとして 2 番目の列に配置します。最後に、ファイル・タイプを IlvLabelMatrixItem オブジェクトとして 3 番目の列に置きます。

2 番目と 3 番目の両方の列アイテムは、非センシティブに設定されています。つまりユーザはこれらを選択することができません。

> ***メモ:*** *シートの最初の列のファイル名は、ツリー構築に使用されたものと同じメソッドで作成されます。*

ガジェット・アイテムの作成

メンバ関数 FileViewer::createFileItem でアイテムを作成します。

```
IlvGadgetItem*
FileViewer::createFileItem(const IlvPathName& file,
                           const IlvGadgetItemHolder* holder) const
{
    // Compute the item label.
    IlvString filename = file.isDirectory()
        ? file.getString()
        : file.getBaseName();
    // If file is a directory, remove the trailing '/'.
    if (file.isDirectory())
        filename.remove(filename.getLength() - 1);
    return
        holder->createItem(filename,
                           0,
                           getBitmap(file, IlvGadgetItem::BitmapSymbol()),
                           getBitmap(file,
                                     IlvGadgetItem::SelectedBitmapSymbol()));
}
```

ガジェット・アイテムは、ピクチャおよびラベルの両方を含むことができます。ガジェット・アイテムの詳細については、ガジェット・アイテムを参照してください。アイテム・ラベルはファイル・パラメータから計算されます。アイテムは 2 つのビットマップに割り当てられます。 1 つは選択されたときのアイテムを表示し、もう 1 つは選択されていないときのアイテムを表します。

次に、アイテムをファクトリ・メンバ関数 IlvGadgetItemHolder::createItem を使用して作成します。これはそのパラメータとして提供されているホルダ・インスタンスで呼び出されます。ガジェット・アイテムの「ガジェット・アイテムの作成」を参照してください。

13 ページの*ディレクトリのスキャン*では、ツリーおよびシート両方のアイテムを同メソッドの呼び出しで作成します。クラス IlvTreeGadget および IlvSheet の両方がクラス IlvGadgetItemHolder から継承しているからです。createFileItem メンバ関数と IlvTreeGadget クラスのインスタンスをそのホルダ・パラメータとして呼び出すと、IlvTreeGadgetItem のインスタンスを返し、これを IlvSheet のインスタンスと呼び出すと、IlvGadgetItem クラスのインスタンスを返します。

メンバ関数 IlvGadgetItemHolder::createItem は、2 つのビットマップをその 3 番目および 4 番目のパラメータとして取ります。これは、定義済みビットマップ付きアイテムの作成を容易にします。

ただし、createItem への呼び出しは、ビットマップのガジェット・アイテムへの割り当て方法を示す次のコードで置き換えることができます。

```
IlvGadgetItem* item = holder->createItem(filename);
item->setBitmap(IlvGadgetItem::BitmapSymbol(),
                getBitmap(file, IlvGadgetItem::BitmapSymbol()));
item->setBitmap(IlvGadgetItem::SelectedBitmapSymbol(),
                getBitmap(file, IlvGadgetItem::SelectedBitmapSymbol()));
return item;
```

ここで使用される getBitmap メソッドはファイル・タイプ / アイテム・ステータスのペアに対応するビットマップを返します。コードは、次に示すようにとても単純です。

```
IlvBitmap*
FileViewer::getBitmap(const IlvPathName& file,
                      const IlvSymbol* state) const
{
     if (state == IlvGadgetItem::BitmapSymbol())
         return getBitmap(file.isDirectory()
                          ? folderBm
                          : fileBm);
     else
         return getBitmap(file.isDirectory()
                          ? sfolderBm
                          : sfileBm);
}
```

folderBm、fileBm、sfolderBm、および sfileBm は、次の条件で定義された記号です。

- folderBm は、フォルダに使用されるビットマップの名前です。
- fileBm は、非選択ファイルに使用されるビットマップの名前です。
- sfolderBm は、選択されたフォルダに使用されるビットマップの名前です。
- sfileBm は、選択されたファイルに使用されるビットマップの名前です。

ファイル・ビューアのインスタンス化

FileViewer クラスが、以上で完成しました。これをアプリケーションにインスタンス化するには、メンバ関数 FileViewerApplication::configureApplication

への呼び出しを、セクション 8 ページの *メイン・ウィンドウの作成*で定義された FileViewerApplication::makePanels に追加しなくてはなりません。

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();
    // Initialize the application.
    configureApplication();
    // Show it.
    getMainWindow()->show();
}
```

FileViewerApplication::configureApplication メンバ関数は、FileViewer クラスをインスタンス化し、初期化します。

```
void
FileViewerApplication::configureApplication()
{
    FileViewer* viewer = createFileViewer(getMainWindow());
    FileViewerApplication::SetFileViewer(getMainWindow(), viewer);
    viewer->init(IlvPathName("/"));
}
```

> ***メモ:*** FileViewerApplication::SetFileViewer *を呼び出すと、メイン・ウィンドウがビューアに接続されます。特定のウィンドウに接続されたファイル・ビューアを、*FileViewerApplication::GetFileViewer *で取得することができます。*

ファクトリ・メソッド createFileViewer は、FileViewer クラスの新しいインスタンスを返します。

```
FileViewer*
FileViewerApplication::createFileViewer(FileViewerWindow* window) const
{
    return new FileViewer(window->getDirectoryHierarchy(),
                          window->getFileList());
}
```

アプリケーションのプレビュー

これでステップ 2 は完了です。アプリケーションは次のように見えるはずです。

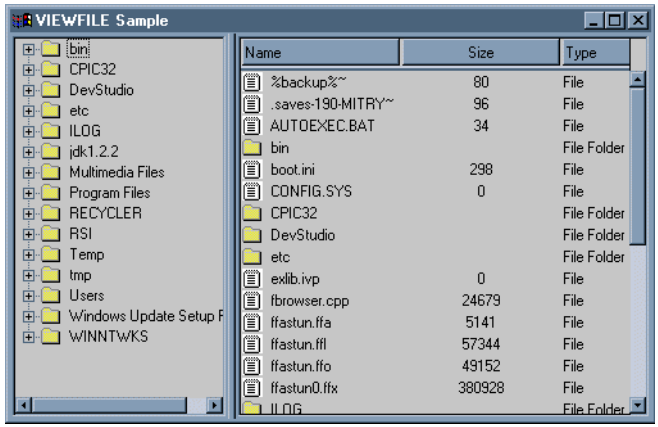


図1.4 *ステップ2 完了後の ViewFile アプリケーション*

ステップ 3: ドッキング・バーの追加

このステップでは、ドッキング・バー (メニュー・バーおよびツールバー) を ViewFile アプリケーションに追加する方法を説明します。ドッキング・ペインについては、ペインおよびコンテナのドッキングを参照してください。

このステップでは、次のタスクを実行する方法を紹介します。

- *メイン・ウィンドウのドッキングを準備する*
- *ドッキング・ツールバーの作成*
- *ドッキング・メニュー・バーの作成*
- *アプリケーションに変更を統合する*
- *アプリケーションのプレビュー*

このステップで加えられるすべての変更は、ファイル `viewfile.cpp` に反映されます。

メイン・ウィンドウのドッキングを準備する

ドッキング・ペインおよびドッキング・ツールバーとメニュー・バーは、ドッキング可能コンテナにドッキングする必要があります。つまり、アプリケーション・メイン・ウィンドウはドッキング操作を可能にするように変更されなくてはなりません。

メイン・ウィンドウの変更

IlvDockableContainer、IlvPanedContainer のサブクラスは、ドッキング・ペイン処理のベース・クラスです。このステップでは、メイン・ウィンドウはメンバ関数 FileViewerApplication::createMainWindow で IlvDockableMainWindow オブジェクトとして定義されています (8 ページの*メイン・ウィンドウの作成*を参照)。IlvDockableMainWindow は、ドッキング・ペインを組み込んでいるアプリケーション用標準レイアウトを実装する IlvDockableContainer のサブクラスです。ドッキング・ペインおよびコンテナの「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」を参照してください。

```
IlvContainer*
FileViewerApplication::createMainWindow(const IlvRect& rect) const
{
    return new IlvDockableMainWindow(getDisplay(),
                                     getName(),
                                     getName(),
                                     rect,
                                     0,
                                     IlvFalse);
}
```

これでドッキング・ペインをアプリケーションに追加することができます。

ペイン作成の準備

メンバ関数 FileViewerApplication::makePanels は、ペイン、およびペイン・コンテナ・レイアウトを更新する updatePanes を作成する initPanes への呼び出しを含むように変更しなくてはなりません。ペインの「ペイン・コンテナのレイアウトを変更する」のセクションを参照してください。

```
void
FileViewerApplication::makePanels(){
    // Initialize the main window.
    initMainWindow();
    // Initialize the panes.
    initPanes();
    // Initialize the application.
    configureApplication();
    // Update the main window layout.
    getMainWindow()->updatePanes(IlvTrue);
    // Show it.
    getMainWindow()->show();
}
```

initPanes メンバ関数は、メニュー・バーを作成する initMenuBar およびツールバーを作成する initToolBar を呼び出します。

```
void
FileViewerApplication::initPanes()
{
    // Initialize the menu bar.
    initMenuBar();
    // Initialize the toolbar.
    initToolBar();
}
```

ドッキング・ツールバーの作成

initToolBar メンバ関数で、ツールバーを作成します。

```
void
FileViewerApplication::initToolBar()
{
    IlvToolBar* toolbar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Item Up One Level.
    toolbar->insertBitmap(getBitmap("upBm"));
    toolbar->getItem(0)->setCallback(UpOneLevel);
    toolbar->getItem(0)->setClientData(this);
    toolbar->getItem(0)->setToolTip("Up One Level");
    // Encapsulate the toolbar into a graphic pane.
    IlvGraphicPane* toolbarPane = new IlvAbstractBarPane("Toolbar", toolbar);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(toolbarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
}
```

ツールバーには、「レベルを 1 つ上げる」アイテムのみが含まれています。選択されると、このアイテムは選択されたフォルダの親を探索しようとします (選択されたフォルダのアイテムは、ファイル・ブラウザの右側のシートに表示されます)。「レベルを 1 つ上げる」アイテムは、ツールチップに関連付けられています。ツールバーは、メイン・ウィンドウの作業領域の上に追加されるタイプ IlvAbstractBarPane の特殊グラフィック・ペインにカプセル化されます。

詳細については、メニュー、メニュー・バー、およびツールバーの「ツールバーでツールチップを使用する」およびドッキング・ペインおよびコンテナの「IlvAbstractBarPane クラスの使用」を参照してください。

> ***メモ:*** *アイテムのクライアント・データは* this *に設定され、アプリケーションへのポインタをそのコールバック関数から取得できるようにします(クライアント・データは、アイテムがアクティブにされるときに呼び出されるコールバックの 2 番目のパラメータです)。*

```
static void
UpOneLevel(IlvGraphic* g, IlvAny arg)
{
    // Retrieve a pointer on the application.
    FileViewerApplication* application = (FileViewerApplication*)arg;
    // Retrieve a pointer on the main window.
    FileViewerWindow* window = (FileViewerWindow*)
        application->getMainWindow()->getMainWorkspaceViewPane()->getView();
    // Retrieve a pointer on the file viewer.
    FileViewer* viewer = FileViewerApplication::GetFileViewer(window);
    if (viewer) {
        // Find the last selected item of the tree.
        IlvTreeGadgetItem* item =
            viewer->getTreeGadget()->getLastSelectedItem();
        // And select its parent.
        if (item && item->getParent() && item->getParent()->getParent())
            viewer->getTreeGadget()->selectItem(item->getParent(),
                                                IlvTrue,
                                                IlvTrue,
                                                IlvTrue);
        }
}
```

ドッキング・メニュー・バーの作成

initMenuBar メンバ関数で、メニュー・バーを作成します。

```
void
FileViewerApplication::initMenuBar()
{
    // The menu bar is in fact an IlvToolBar.
    IlvToolBar* menubar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Add two items
    menubar->addLabel("File");
    menubar->addLabel("Help");
    // Create the pane that will encapsulate the menu bar.
    IlvGraphicPane* menubarPane = new ApplicationMenuBarPane("Menu Bar",
                                                             menubar);
    // Change the mode of the menu bar to make it show items on several
    // rows, if needed.
    menubar->setConstraintMode(IlvTrue);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(menubarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
    // Now fill the menus with popup menus.
    IlvPopupMenu* menu;
    // Menu File: New / Separator / Exit.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("Exit");
    menu->getItem(0)->setCallback(ExitApplication);
    menu->getItem(0)->setClientData(this);
    menubar->getItem(0)->setMenu(menu, IlvFalse);
    // Menu Help: About.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("About");
    menubar->getItem(1)->setMenu(menu, IlvFalse);
    menu->getItem(0)->setCallback(ShowAboutPanel);
    menu->getItem(0)->setClientData(this);
}
```

メニュー・バーが作成され、2 つのアイテム、ファイルおよびヘルプが初期化されます。次に、バーの向き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である ApplicationMenuBarPane クラスのペインにカプセル化されます。

ドッキング・ペインおよびコンテナのセクション「ドッキング・バーのカスタマイズ」を参照してください。

メニュー・バーのヘルプ・アイテムには、バージョン情報アイテムを含むサブメニューがあります。このアイテムは、アプリケーション名を示すダイアログ・ボックスを表示するコールバックに付加されます。

```
void
FileViewerApplication::showVersion()
{
    IlvIInformationDialog dialog(getDisplay(), "VIEWFILE Tutorial");
    dialog.moveToMouse();
    dialog.showModal();
}
```

詳細については、ダイアログ を参照してください。

アプリケーションに変更を統合する

最後のステップは、ファイル・ビューアのビューをアプリケーション・メイン・ウィンドウのメイン作業領域へ含めるためにメンバ関数 `FileViewerApplication::configureApplication` を変更することです。この作業領域は、デフォルトで作成されたビュー・ペインによって表され、`IlvDockableMainWindow::getMainWorkspaceViewPane` を使用して取得することができます。

ドッキング・ペインおよびコンテナの「IlvDockableMainWindow クラスの使用」を参照してください。

```
void
FileViewerApplication::configureApplication()
{
    // Create the file viewer window.
    FileViewerWindow* window = createFileViewerWindow(getMainWindow(),
                                                       IlvRect(0, 0, 400, 200));
    // Replace the view of the main workspace pane with the file viewer window.
    getMainWindow()->getMainWorkspaceViewPane()->setView(window);
    // Create the file viewer using the file viewer window.
    FileViewer* viewer = createFileViewer(window);
    // Link the file viewer with its window.
    // This is used in the UpOneLevel callback.
    SetFileViewer(window, viewer);
    // Initialize the file viewer.
    viewer->init(IlvPathName("/"));
}
```

ファクトリ・メンバ関数 `createFileViewerWindow` は、`FileViewerWindow` オブジェクトを提供されたビューのサブ・ビューとして作成します。作成されたビューは、メイン作業領域を表すビュー・ペインとして設定されています。最後に、`FileViewer` オブジェクトが以前のステップと同じメソッドを使用して作成され、初期化されます。

アプリケーションのプレビュー

これでステップ 3 は完了です。アプリケーションは次のように見えるはずです。

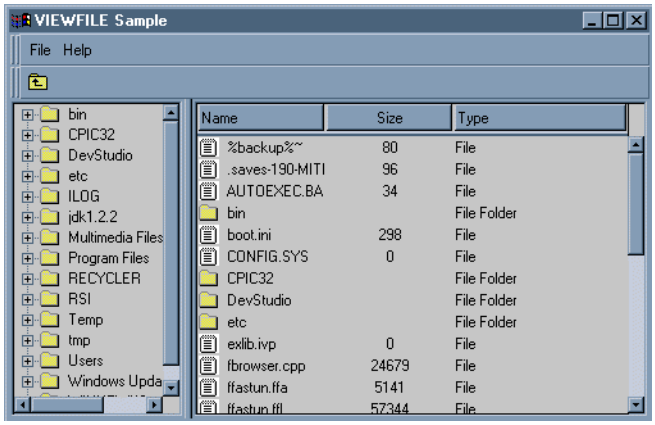


図 1.5 *ステップ3 完了後の ViewFile アプリケーション*

ステップ 4: ビュー・フレームの追加

この最後のステップでは、ビュー・フレーム・サポートを ViewFile アプリケーションに追加する方法を説明します。ビュー・フレームをアプリケーションに含めると、ファイルの複数の階層リストを同時に表示してブラウズすることができるようになります。詳細については、ビュー・フレームを参照してください。

このステップで加えられるすべての変更は、ファイル `viewfile.cpp` に反映されます。

このステップでは、次のタスクを実行する方法を紹介します。

- *デスクトップ・ビューの選択*
- *ビュー・フレームの作成*
- *新しいメニューおよびアイテムをビュー・フレームへ追加する*
- *アプリケーションのプレビュー*

デスクトップ・ビューの選択

ビュー・フレームは、デスクトップ・ビュー内に表示されます。最初の操作は、ビュー・フレームが表示される場所でデスクトップ・ビューを選択することです。

デスクトップ・ビューを配置する最適の場所は、固有のファイル・ビューアが 21 ページの*ステップ3: ドッキング・バーの追加*に表示されるメイン作業領域のビュー・ペインです。すべてのビュー・フレームを管理するデスクトップ・マネージャは、このビューから作成することができます。

デスクトップ・マネージャの作成

`FileViewerApplication::makePanels` メンバ関数は、メイン作業領域のビュー・ペインを使用してデスクトップ・マネージャを作成する `initDesktopManager` への呼び出しを含むように変更されなくてはなりません。

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();

    // Initialize the desktop manager.
    initDesktopManager();

    // Initialize the panes.
    initPanes();

    // Initialize the application.
    configureApplication();

    // Update the main window layout.
    getMainWindow()->updatePanes(IlvTrue);

    // Show it.
    getMainWindow()->show();
}
```

次は、`initDesktopManager` の詳細です。

```
void
FileViewerApplication::initDesktopManager()
{
    createDesktopManager(getMainWindow()->
    getMainWorkspaceViewPane()->getView());
}
```

ファクトリ・メンバ関数 `FileViewerApplication::createDesktopManager` は、`IlvDesktopManager` クラスのインスタンスを返します。

```
IlvDesktopManager*
FileViewerApplication::createDesktopManager(IlvView* view) const
{
    return new IlvDesktopManager(view);
}
```

デスクトップ・マネージャの初期化

デスクトップ・マネージャを初期化するために FileViewerApplication::configureApplication も変更しなくてはなりません。

```
void
FileViewerApplication::configureApplication()
{
    // The desktop manager is maximized by default.
    getDesktopManager()->
       makeMaximizedStateButtons((IlvToolBar*)
                                 ((IlvGraphicPane*)getMainWindow()->
                                   getPane("Menu Bar", IlvTrue))->getObject());
    getDesktopManager()->maximize(0);

    // Create a frame initialized at "/".
    getDesktopManager()->setCurrentFrame(createNewFrame(IlvRect(0,
                                         0,
                                         400,
                                         200), "/"));
}
```

IlvDesktopManager::makeMaximizedStateButtons は、ビュー・フレームのボタンが最大化されたときに、どこに表示されるべきかを指定します。

ビュー・フレームの「ビュー・フレームの最大化」を参照してください。

ボタンは、次に示すように、メニュー・バー内部に表示されます。

次に、デスクトップ・マネージャが最大化され、デフォルト・フレームが FileViewerApplication::createNewFrame によって作成されます。

ビュー・フレームの作成

メンバ関数 FileViewerApplication::createNewFrame は、ファイル・ビューアおよびその関連するウィンドウをカプセル化する新しいフレームを作成します。

```
IlvViewFrame*
FileViewerApplication::createNewFrame(const IlvRect& rect,
const char* path) const
{
    // Create a view frame in the desktop manager view.
    IlvViewFrame* vframe = new IlvViewFrame(getDesktopManager()->getView(),
                                                                  path,
                                                                  rect,
                                                                  IlvFalse);
     vframe->setDestroyCallback(DestroyFrame);
     // Create a file viewer window inside the view frame.
     FileViewerWindow* viewerWindow = createFileViewerWindow(vframe, rect);
     // Create the file viewer in the file viewer window.
     FileViewer* viewer = createFileViewer(viewerWindow);
     // Associate the viewer window with the viewer.
     SetFileViewer(viewerWindow, viewer);
     // Initialize the file viewer.
     viewer->init(IlvPathName(path));
     return vframe;
}
```

メンバ関数 IlvView::setDestroyCallback は、作成したフレームの削除を処理する破壊コールバックを設定します。ビュー・フレームの「ビュー・フレームを閉じる」を参照してください。

次に、ファイル・ビューア・ウィンドウをビュー・フレームのサブビューとして作成します。ビュー・フレームの「クライアント・ビューの作成」を参照してください。最後に、ファイル・ビューアはビュー・フレームに接続され、初期化されます。

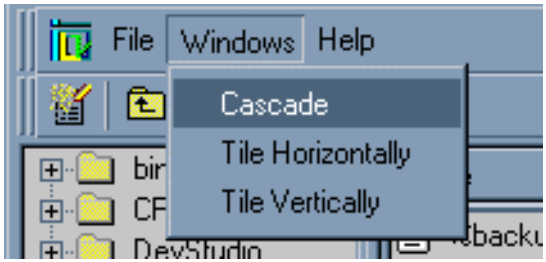
新しいメニューおよびアイテムをビュー・フレームへ追加する

新しいメニュー・アイテムをメニュー・バーに追加するには、メンバ関数 `FileViewerApplication::initMenuBar` を次のように変更しなくてはなりませ

ん。

```
void
FileViewerApplication::initMenuBar()
{
    // The menu bar is in fact an IlvToolBar.
    IlvToolBar* menubar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Add three items.
    menubar->addLabel("File");
    menubar->addLabel("Windows");
    menubar->addLabel("Help");
    // Create the pane that will encapsulate the menu bar.
    IlvGraphicPane* menubarPane = new ApplicationMenuBarPane("Menu Bar",
                                                             menubar);
    // Change the mode of the menu bar to make it show items on several
    // rows, if needed.
    menubar->setConstraintMode(IlvTrue);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(menubarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
    // Now fill the menus with popup menus.
    IlvPopupMenu* menu;
    // Menu File: New / Separator / Exit.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("New (Ctrl+N)");
    menu->getItem(0)->setBitmap(getBitmap("newBm"));
    menu->getItem(0)->setCallback(AddNewFrame);
    menu->getItem(0)->setClientData(this);
    menu->getItem(0)->setAcceleratorText("Ctrl+N");
    menu->getItem(0)->setAcceleratorKey(IlvCtrlChar('N'));
    menu->addItem(IlvMenuItem());
    menu->addLabel("Exit");
    menu->getItem(2)->setCallback(ExitApplication);
    menu->getItem(2)->setClientData(this);
    menubar->getItem(0)->setMenu(menu, IlvFalse);
    // Menu Windows: Cascade / Tile Horizontally / Tile Vertically.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("Cascade");
    menu->getItem(0)->setCallback(CascadeFrames);
    menu->getItem(0)->setClientData(this);
    menu->addLabel("Tile Horizontally");
    menu->getItem(1)->setCallback(TileHorizontallyFrames);
    menu->getItem(1)->setClientData(this);
    menu->addLabel("Tile Vertically");
    menu->getItem(2)->setCallback(TileVerticallyFrames);
    menu->getItem(2)->setClientData(this);
    menubar->getItem(1)->setMenu(menu, IlvFalse);
    // Menu Help: About.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("About");
    menubar->getItem(2)->setMenu(menu, IlvFalse);
    menu->getItem(0)->setCallback(ShowAboutPanel);
    menu->getItem(0)->setClientData(this);
}
```

メニュー・バーには、次の図に示すように、サブアイテムを含む Windows メニューがあります。



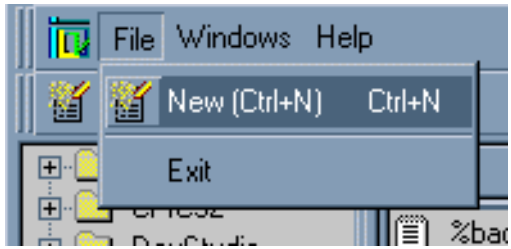
Windows メニューによって使用されるコールバックは、対応するメンバ関数をデスクトップ・マネージャで呼び出します。

```
static void
CascadeFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->cascadeFrames();
}

static void
TileHorizontallyFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->tileFrames(IlvHorizontal);
}

static void
TileVerticallyFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->tileFrames(IlvVertical);
}
```

メンバ関数 `FileViewerApplication::initToolBar` も、選択されたときに新しいフレームを作成する [新規作成] アイテムを含むように変更されています。



```
void
FileViewerApplication::initToolBar()
{
    IlvToolBar* toolbar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Item New.
    toolbar->insertBitmap(getBitmap("newBm"));
    toolbar->getItem(0)->setCallback(AddNewFrame);
    toolbar->getItem(0)->setClientData(this);
    toolbar->getItem(0)->setToolTip("New");
    // Separator.
    toolbar->addItem(IlvMenuItem());
    // Item Up One Level.
    toolbar->insertBitmap(getBitmap("upBm"));
    toolbar->getItem(2)->setCallback(UpOneLevel);
    toolbar->getItem(2)->setClientData(this);
    toolbar->getItem(2)->setToolTip("Up One Level");
    // Encapsulate the toolbar into a graphic pane.
    IlvGraphicPane* toolbarPane = new IlvAbstractBarPane("Toolbar", toolbar);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(toolbarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
}
```

新しいアイテムによってトリガされたコールバックは、メンバ関数 `FileViewerApplication::createNewFrame` を呼び出し、新しいフレームをアクティブにします。

```
static void
AddNewFrame(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    IlvViewFrame* vframe =
       application->createNewFrame(IlvRect(0, 0, 400, 200), "/");
    application->getDesktopManager()->setCurrentFrame(vframe);
}
```

> ***メモ:*** *[新規作成] アイテムは、アクセラレータ・キーに関連付けられており、Ctrl+N キーを押すと新しいフレームを作成できます。メニュー、メニュー・バーとツールバーの「アクセラレータをメニュー・アイテムに関連付ける」を参照してください。*

アプリケーションのプレビュー

これで、チュートリアルが完了しました。アプリケーション・ウィンドウは、次のように見えるはずです。

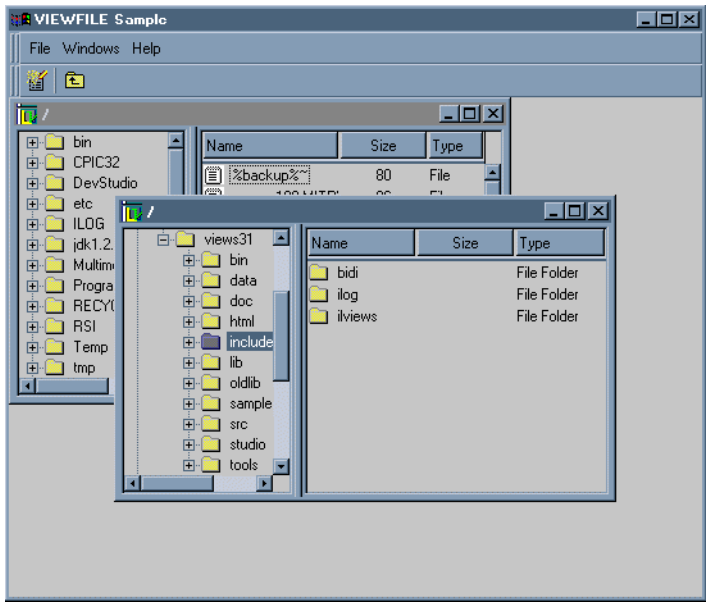


図 1.6 *最終 ViewFile アプリケーション*

章

2

ガジェットのカスタマイズ

このチュートリアルでは新しいガジェットの作成方法を説明します。新しいガジェット・クラスの作成は、新しいグラフィック・クラス作成 (つまり、`IlvGraphic` のサブクラス) に類似していますが、より多くの作業が要求されます。たとえば、ガジェットはそれ自体の振る舞いを定義し (`IlvGadget::handleEvent` メソッドを通じて)、キーボード・フォーカスの処理方法を定義しなくてはなりません。

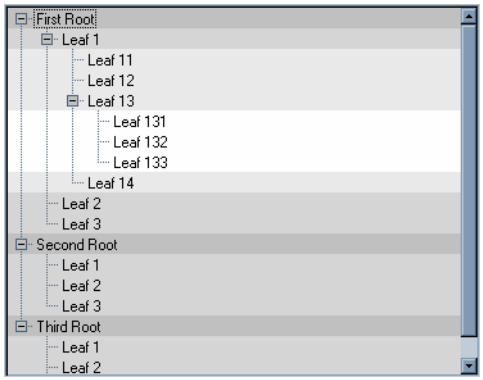
このチュートリアルには、3 つのステップがあります。

- *ステップ 1: グラフィカルな外観変更によるガジェットの拡張*
- *ステップ 2: 振る舞い変更によるガジェットの拡張*
- *ステップ 3: 複合ガジェットの作成*

ステップ 1: グラフィカルな外観変更によるガジェットの拡張

最初のステップでは、既存のガジェットのグラフィカルな外観を変更するために、そのサブクラスを作成する方法を説明します。

このステップでは、`IlvTreeGadget` クラスが拡張されます。その描画を変更して、次の図に示すように、アイテムの子の色を変更できるようにします。



このステップでは、次のタスクを実行する方法を紹介します。

- *既存ガジェットのサブクラスを作成する*
- *API を新規ガジェット・クラスへ追加する*
- *新規ガジェットの描画方法を変更する*
- *新規ガジェット・クラスのテスト*

既存ガジェットのサブクラスを作成する

既存ガジェットの新しいサブクラスの作成は、既存の `IlvGraphic` クラスのサブクラスの作成に類似しています。`IlvGraphic` オブジェクトのサブクラスを適切に作成する主要ステップをここで紹介します。

サブクラスの宣言

選択したグラフィックのサブクラスとして、サブクラスを宣言します。このチュートリアルでは、`IlvTreeGadget` は、色付きのアイテムを表示できる新しいオブジェクトを作成するためにサブクラス化されています。このサブクラスは、`ColoredTreeGadget` と呼ばれます。これは `ColoredTreeGadget` クラスの宣言です。

```
class ColoredTreeGadget
: public IlvTreeGadget {
public:
    ColoredTreeGadget(IlvDisplay*        display,
                      const IlvRect&     rect,
                      IlvUShort          thickness = IlvDefaultGadgetThickness,
                      IlvPalette*        palette   = 0);
    virtual ~ColoredTreeGadget();
};
```

この宣言は、`ColoredTreeGadget` クラスのコンストラクタおよびデストラクタから構成されています。

動的なタイピングおよび永続性を可能にするために ColoredTreeGadget の登録を追加します。

```
DeclareTypeInfo();
DeclareIOConstructors(ColoredTreeGadget);
DeclareGraphicAccessors();
```

DeclareGraphicAccessors マクロは、queryValue および applyValue メソッドを宣言して、新しいオブジェクトをスクリプト可能にします。

メソッドの実装

宣言された ColoredTreeGadget メソッドの実装を次に示します。

まず、コンストラクタが次のように実装されます。

```
ColoredTreeGadget::ColoredTreeGadget(IlvDisplay*        display,
                                     const IlvRect&     rect,
                                     IlvUShort          thickness,
                                     IlvPalette*        palette )
: IlvTreeGadget(display, rect, thickness, palette)
{
}
```

コンストラクタのコピー、DeclareIOContructors マクロによって宣言されています。

```
ColoredTreeGadget::ColoredTreeGadget(const ColoredTreeGadget& source)
: IlvTreeGadget(source)
{
}
```

IO コンストラクタ、これも DeclareIOConstructors マクロによって宣言されています。

```
ColoredTreeGadget::ColoredTreeGadget(IlvInputFile& is, IlvPalette* palette)
: IlvTreeGadget(is, palette)
{
}
```

write メソッド、DeclareTypeInfo マクロによって宣言されています。

```
void
ColoredTreeGadget::write(IlvOutputFile& os) const
{
    IlvTreeGadget::write(os);
}
```

デストラクタ

```
ColoredTreeGadget::~ColoredTreeGadget()
{
}
```

アクセサ関連メソッド、DeclareGraphicAccessors マクロによって宣言されています。

```
IlvValue&
ColoredTreeGadget::queryValue(IlvValue& value) const
{
    return IlvTreeGadget::queryValue(value);
}

IlvBoolean
ColoredTreeGadget::applyValue(const IlvValue& value)
{
    return IlvTreeGadget::applyValue(value);
}

void
ColoredTreeGadget::GetAccessors(const IlvSymbol* const** a,
                                const IlvValueTypeClass* const** t,
                                IlvUInt& c)
{
}
```

それから、copy および read メソッドの実装が IlvPredefinedIOMembers マクロによって与えられます。

```
IlvPredefinedIOMembers(ColoredTreeGadget)
```

最後に、クラスは IlvTreeGadget クラスのサブクラスとして登録されます。

```
IlvRegisterClass(ColoredTreeGadget, IlvTreeGadget);
```

これらのアイテムは、ColoredTreeGadget クラスを適切に登録するために必要です。これらのメソッドの多くは空ですが、次の手順でデータを埋めます。

API を新規ガジェット・クラスへ追加する

ColoredTreeGadget クラスの宣言は、coltree.h ファイルにあり、その実装は coltree.cpp ファイルにあります。

ColoredTreeGadget クラスは、以下を必要とします。

- ツリー・アイテムを特定の色に関連付ける方法。この色は、このアイテムの子の背景を描くのに使用されます。
- ツリー・アイテムの色塗りを有効 / 無効にする方法。
- アクセサをこれに追加する。

アイテムを色に関連付ける

ツリー・アイテムを色に関連付けるために、setChildrenBackground メソッドが追加されました。これは、色を保存するためにツリー・アイテムでプロパティ・セットを使用します。

```
void
ColoredTreeGadget::setChildrenBackground(IlvTreeGadgetItem* item,
                                         IlvColor* color,
                                         IlvBoolean redraw)
{
    // Retrieve the old color.
    IlvPalette* oldPalette =
                (IlvPalette*)item->getProperty(GetChildrenBackgroundSymbol());
    // Compute the new color.
    IlvPalette* palette = color
                          ? getDisplay()->getPalette(0, color)
                          : 0;
    // Lock it.
    if (palette)
        palette->lock();
    // Unlock the old one.
    if (oldPalette)
        oldPalette->unLock();
    // Set the property to the item.
    item->setProperty(GetChildrenBackgroundSymbol(), (IlvAny)palette);
    // Redraw if asked.
    if (redraw)
        reDraw();
}
```

GetChildrenBackgroundSymbol スタティック・メソッドはアイテム上のプロパティ・セットを識別する記号を返します。その定義は簡単です。

```
static IlvSymbol*
GetChildrenBackgroundSymbol()
{
    // This symbol is used to connect a tree gadget item to the color of its
    // children.
    static IlvSymbol* symbol = IlvGetSymbol("ChildrenBackground");
    return symbol;
}
```

getChildrenBackground メソッドは、任意のツリー・アイテムに関連付けられている色を返します。

```
IlvColor*
ColoredTreeGadget::getChildrenBackground(const IlvTreeGadgetItem* item) const
{
    // Returns the color stored in the property list of the specified item.
    IlvPalette* palette =
        (IlvPalette*)item->getProperty(GetChildrenBackgroundSymbol());
    return palette
           ? palette->getForeground()
           : 0;
}
```

色塗りを有効 / 無効にする

ツリー・アイテムの色塗りを有効 / 無効にできるようにするコードを実装します。これを行うために、保護されたメンバ変数を ColoredTreeGadget クラスに追加します。

```
protected:
    IlvBoolean _drawChildrenBg;
```

デフォルトで、このブール型値は IlvTrue へのコンストラクタによって初期化されます。これらは _drawChildrenBg メンバ変数へのアクセスを付加するメンバ関数です。

```
IlvBoolean isDrawingChildrenBackground() const;
void drawChildrenBackground(IlvBoolean value,
                            IlvBoolean redraw = IlvTrue);
```

_drawChildrenBg メンバ変数は保存する必要があるため、IO コンストラクタおよび write メソッドの両方を変更しなくてはなりません。

```
ColoredTreeGadget::ColoredTreeGadget(IlvInputFile& is, IlvPalette* palette)
: IlvTreeGadget(is, palette),
  _drawChildrenBg(IlvTrue)
{
    // Read the _drawChildrenBg flag.
    int drawChildrenBg;
    is.getStream() >> drawChildrenBg;
    _drawChildrenBg = (IlvBoolean)drawChildrenBg;
}

void
ColoredTreeGadget::write(IlvOutputFile& os) const
{
    IlvTreeGadget::write(os);
    // Write the _drawChildrenBg flag.
    os.getStream() << IlvSpc() << (int)_drawChildrenBg << IlvSpc();
}
```

アクセサの追加

オブジェクトにアクセサを追加すると、これらをスクリプトに利用できるようになります。以下の記述は、_drawChildrenBg メンバ変数へのアクセサの例を表しています。

まず、この変数にアクセスするために使用される記号を定義します。これは、スタティック関数 GetDrawChildrenBackgroundSymbol によって行われます。

```
static IlvSymbol*
GetDrawChildrenBackgroundSymbol()
{
    // This symbol is used to access to drawChildrenBackground accessor of
    // the colored tree gadget.
    static IlvSymbol* symbol = IlvGetSymbol("drawChildrenBackground");
    return symbol;
}
```

次に、新しいアクセサが GetAccessors に登録されます。

```
void
ColoredTreeGadget::GetAccessors(const IlvSymbol* const** a,
                                const IlvValueTypeClass* const** t,
                                IlvUInt& c)
{
    DeclareAccessor(GetDrawChildrenBackgroundSymbol(),
                    IlvValueBooleanType,
                    a,
                    t,
                    c);
}
```

queryValue メソッドは、アクセサの値に問い合わせが行われたときに呼び出されます。

```
IlvValue&
ColoredTreeGadget::queryValue(IlvValue& value) const
{
    if (value.getName() == GetDrawChildrenBackgroundSymbol())
        return value = isDrawingChildrenBackground();
    else
        return IlvTreeGadget::queryValue(value);
}
```

applyValue メソッドは、アクセサの値が変更されたときに呼び出されます。

```
IlvBoolean
ColoredTreeGadget::applyValue(const IlvValue& value)
{
    if (value.getName() == GetDrawChildrenBackgroundSymbol()) {
        drawChildrenBackground((IlvBoolean)value, IlvFalse);
        return IlvTrue;
    } else
        return IlvTreeGadget::applyValue(value);
}
```

新規ガジェットの描画方法を変更する

次のステップは、ColoredTreeGadget の描画方法を変更することです。グラフィカル・オブジェクトの描画を変更する標準的な方法は、IlvGraphic レベルで定義されているその draw メソッドをオーバーライドすることです。

```
virtual void draw(IlvPort* dst,
                  const IlvTransformer* t = 0,
                  const IlvRegion*      clip = 0) const = 0;
```

ただし、IlvTreeGadget は複雑なオブジェクトで、その描画のカスタマイズに役立つその他のメソッドが複数あります。各アイテムが描画される前に背景を描画す

るとします。IlvGadgetItemHolder::drawGadgetItem メソッドを使用することができます。これは、各アイテムを描画するためにツリーが呼び出します。

```
void
ColoredTreeGadget::drawGadgetItem(const IlvGadgetItem* item,
                                  IlvPort* port,
                                  const IlvRect&     rect,
                                  const IlvTransformer* t,
                                  const IlvRegion* clip) const
{
    if (isDrawingChildrenBackground()) {
         // Check if the item being drawn has a special palette.
         IlvPalette* palette = getBackgroundPalette((IlvTreeGadgetItem*)item);
         if (palette) {
             // Compute the visible bounding box.
             IlvRect bbox;
             visibleBBox(bbox, t);
             // Move and resize it to match the item bounding box.
             bbox.y(rect.y());
             bbox.h(rect.h());
             if (clip)
                 palette->setClip(clip);
             port->fillRectangle(palette, bbox);
             if (clip)
                 palette->setClip();
        }
    }
    // Draw the item.
    IlvTreeGadget::drawGadgetItem(item, port, rect, t, clip);
}
```

getBackgroundPalette メソッドは、ツリー・アイテムの背景を描画するために使用されたパレットの取得に使用されます。

```
IlvPalette*
ColoredTreeGadget::getBackgroundPalette(const IlvTreeGadgetItem* item) const
{
    // Returns the palette that will be used to draw the background of ?item?
    // This information is stored in its parent
    if (item->getParent()) {
        IlvPalette* palette = (IlvPalette*)
            item->getParent()->getProperty(GetChildrenBackgroundSymbol());
        if (!palette)
            palette = getBackgroundPalette(item->getParent());
        return palette;
    } else
        return 0;
}
```

getBackgroundPalette メソッドは、背景パレットとして設定されている「アイテム」の最初の親を見つけようとします。これは、ツリー・アイテムはその色を親から継承することを意味します。アイテムの背景色を変更すると、その子の背景色を変えるだけでなく、その孫などの背景色も変えることになります。

新規ガジェット・クラスのテスト

ファイル main.cpp は、ColoredTreeGadget クラスをテストするコードを含んでいます。これは、ColoredTreeGadget インスタンスの記述を含むファイル coltree.ilv を読み込み、レベルに応じて全アイテムの色を変更します。

```
    // Read the file that contains the colored tree.
    container->readFile("../doc/gadgets/tutorials/custgad/data/coltree.ilv");

    // Retrieve the tree.
    ColoredTreeGadget* tree = (ColoredTreeGadget*)container->getObject("Tree");

    // Set the background of the tree to gray.
    // This color will be used as the reference color to compute the children
    // colors.
    tree->setBackground(display->getColor("gray"));

    // Now change the color of each level of items.
    tree->applyToItems(ChangeColor, (IlvAny)tree);
```

applyToItems メソッドは、ツリーの全アイテムに ChangeColor 関数を適用します。これは、ColoredTreeGadget::setChildrenBackground メソッドを呼び出すこの ChangeColor 関数の記述です。

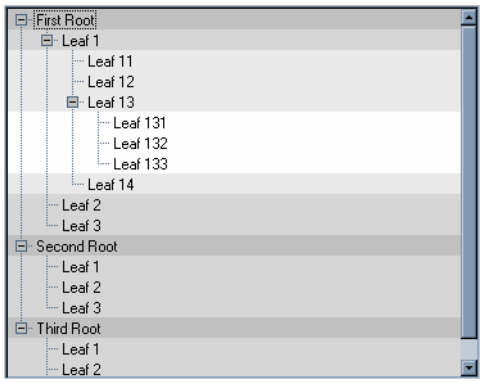
```
static IlvBoolean
ChangeColor(IlvGadgetItem* item, IlvAny arg)
{
    // The argument is a pointer to the ColoredTreeGadget instance.
    ColoredTreeGadget* tree = (ColoredTreeGadget*)arg;
    // Change the background of the children of 'item'.
    tree->setChildrenBackground((IlvTreeGadgetItem*)item,
                                GetChildrenColor((IlvTreeGadgetItem*)item,
                                                 tree->getBackground()),
                                                 IlvFalse);
    // Continue.
    return IlvTrue;
}
```

別のスタティック関数を使って、ツリー・アイテムおよび基準色を使用して色を計算します。

```
static IlvColor*
GetChildrenColor(IlvTreeGadgetItem* item, IlvColor* color)
{
    // Get the item level to choose the right color.
    IlvUInt level = item->getLevel();
    // Compute the HSV components of the reference color.
    IlvFloat h, s, v;
    color->getHSV(h, s, v);
    // Increase the V component.
    v = (IlvFloat)IlvMin((IlvFloat)1., (IlvFloat)(v + level*.08));
    // Return the new color.
    return color->getDisplay()->getColor(h, s, v);
}
```

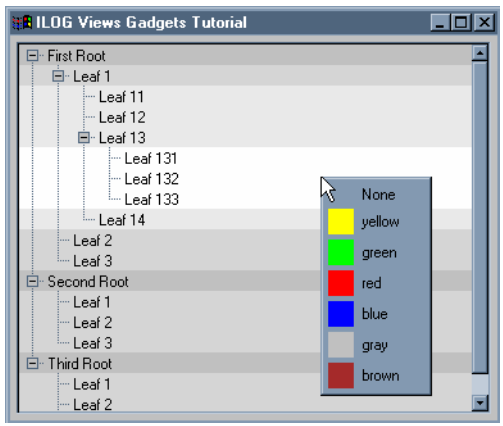
この関数は、アイテムのレベルおよび指定された基準色を使用して、新しい色を計算します。この新しい色は HSV モデルの基準色の V コンポーネントを変更して計算されます。

これでステップ 1 は完了です。テスト・プログラムは、次のように見えるはずです。



ステップ 2: 振る舞い変更によるガジェットの拡張

このステップでは、既存のガジェットの振る舞いの変更方法を説明します。このチュートリアルのステップ 1 で作成した `ColoredTreeGadget` の拡張法を学びます。ユーザが右マウス・ボタンでツリーをクリックしたときに文脈依存型メニューが表示されるように、ツリーの振る舞いを変更します。この文脈依存型メニューで、次の図で示すように、クリックした場所のアイテムの色を変更することができます。



このステップでは、次のタスクを実行する方法を紹介します。

- *既存ガジェットの振る舞いを変更する適切な方法の選択*
- *汎用インタラクタの作成*
- *専用インタラクタの作成*
- *新しいインタラクタのテスト*

既存ガジェットの振る舞いを変更する適切な方法の選択

既存ガジェットの振る舞いを変更するには 2 つの方法があります。

1. ガジェットをサブクラス化してその `handleEvent` メソッドをオーバーライドする。

   これは振る舞いの変更が、それが適用する種類のガジェットと強くリンクされている場合に、より論理的な方法です。

2. ガジェットに設定される新しいインタラクタを作成する。

   新しいインタラクタを作成するのは、オブジェクトの振る舞いを一時的に変更したい場合、あるいは新しいインタラクタが汎用であるとき、つまり異なる複数のガジェット上にこれを設定することができる場合に適した方法です。

汎用インタラクタの作成

このチュートリアルでは、2 番目の方法 (新しいインタラクタを作成する) を選択します。この例を実装する振る舞い (右クリックで文脈依存型メニューを表示する) は汎用性があるため、どのガジェット・クラスにも使用できます。

`ContextualMenuInteractor` クラス用の完全なコードは、`ctxminter.h` および `ctxminter.cpp` にあります。次に示すのは、クラスの記述です。

目的

`ContextualMenuInteractor` クラスの目的は、文脈依存型メニューを処理し、接続されているガジェットの振る舞いを維持する汎用インタラクタを提供することです。この理由により、`ContextualMenuInteractor` クラスは

IlvGadgetInteractor クラスのサブクラスでなくてはなりません。つまり、すべてのガジェットに設定されている定義済みインタラクタ・クラスです。

```
class ContextualMenuInteractor
: public IlvGadgetInteractor
{
public:
   virtual IlBoolean handleEvent(IlvGraphic* obj,
                                 IlvEvent&,
                                 const IlvTransformer* t);
   virtual IlBoolean shouldShowMenu(IlvGraphic*,
                                    IlvEvent&,
                                    const IlvTransformer*) const;
    virtual IlvPopupMenu* getMenu(IlvGraphic*,
                                  IlvEvent&,
                                  const IlvTransformer*) const = 0;
};
```

次に、これらの各メソッドの詳細を検討します。

s houldShowMenu メソッド

shouldShowMenu メソッドは、指定されたイベントに対してポップ・アップ・メニューを表示する必要がある場合、IlvTrue を返します。デフォルトの振る舞いは、マウスの右ボタンがリリースされると IlvTrue を返します。このメソッドは、他の全イベントに対してポップ・アップ・メニューを表示するようにサブクラスで再定義することもできます。

```
IlBoolean
ContextualMenuInteractor::shouldShowMenu(IlvGraphic*,
                                         IlvEvent& event,
                                         const IlvTransformer*) const
{
    // The contextual menu is displayed by default when the right button
    // is released.
    return event.type() == IlvButtonUp && event.button() == IlvRightButton;
}
```

getMenu メソッド

getMenu メソッドは、純粋であるため、表示するポップ・アップ・メニューを返すためにサブクラスで再定義される必要があります。

```
  virtual IlvPopupMenu* getMenu(IlvGraphic*,
                                IlvEvent&,
                                const IlvTransformer*) const = 0;
```

このメソッドは、shouldShowMenu メソッドが IlvTrue を返したときに handleEvent メソッドにより呼び出されます。

handleEvent メソッド

以下に handleEvent メソッドの実装を示します。

```
IlBoolean
ContextualMenuInteractor::handleEvent(IlvGraphic* obj,
                                      IlvEvent& event,
                                      const IlvTransformer* t)
{
    // Check that the object is a gadget.
    IlvGadget* gadget = accept(obj) ? IL_CAST(IlvGadget*, obj) : 0;
    if (gadget && gadget->isActive()) {
        // Is it time to display the contextual menu?
        if (shouldShowMenu(obj, event, t)) {
            // Get the menu.
            IlvPopupMenu* menu = getMenu(obj, event, t);
            // Show the menu.
            if (menu) {
                menu->get(IlvPoint(event.gx(), event.gy()),
                          /* transient */ 0);
                return IlvTrue;
            }
        }
    }
    // Default behavior of the gadget.
    return IlvGadgetInteractor::handleEvent(obj, event, t);
}
```

専用インタラクタの作成

ColoredTreeGadget クラスに専用の ContextualMenuInteractor クラスのサブクラスが、作成されます。ColoredTreeGadgetInteractor と呼ばれるこのサブクラスは、getMenu メソッドをオーバーライドし、次の図で示すように、いくつかの共通色が利用できるメニューを作成します。

色を選択すると、ポップ・アップ・メニューが表示される前にマウスが置かれていたアイテムの色が変更されます。

メニューの作成

以下に getMenu メソッドの実装を示します。

```
IlvPopupMenu*
ColoredTreeGadgetInteractor::getMenu(IlvGraphic* graphic,
```

```
                                    IlvEvent& event,
                                    const IlvTransformer* t) const
{
    ColoredTreeGadget* tree = (ColoredTreeGadget*)graphic;
    // Find the item located at the event location.
    IlvTreeGadgetItem* item = tree->pointToItemLine(IlvPoint(event.x(),
                                                             event.y()),
                                                    (IlvTransformer*)t);
    if (!item)
        return 0;

    // Create the menu if needed.
    static IlvPopupMenu* menu = 0;
    if (!menu) {
        // Create the pop-up menu.
        menu = new IlvPopupMenu(tree->getDisplay());
        // Fill it with predefined colors.
        AddColor(menu, 0);
        AddColor(menu, "yellow");
        AddColor(menu, "green");
        AddColor(menu, "red");
        AddColor(menu, "blue");
        AddColor(menu, "gray");
        AddColor(menu, "brown");
    }
    // Set the item that asked for the contextual menu so that the menu
    // will be able to use it later.
    SetSelectedItem(menu, tree, item);
    // Return the menu.
    return menu;
}
```

マウスの下に位置するアイテムを `IlvTreeGadget::pointToItemLine` メソッドを使用して取得した後、スタティック関数 `AddColor` を呼び出すことでメニューが定義済みの色で作成されます。

メモ: *メニューは 1 回のみ作成されます。*

それからメニューがツリーおよびツリー・アイテムに、スタティック・メンバ関数 `SetSelectedItem` を呼び出して接続されます。このスタティック・メンバ関数は、51 ページの*メニュー・コールバック*に説明されているように、メニュー・コールバックも設定します。

最後に、メニューが返されます。

スタティック関数 `AddColor` の内容をより詳細に示します。

```
static void
AddColor(IlvPopupMenu* menu, const char* color)
{
    // This static function is used to fill the contextual menu.
    IlvDisplay* display = menu->getDisplay();
    // A menu item is created with the specified color name.
    IlvMenuItem* item = new IlvMenuItem(color? color : "None");
    if (color) {
```

```
        // An IlvFilledRectangle object is created with the specfied color.
         IlvPalette* palette = display->getPalette(0, display->getColor(color));
        IlvFilledRectangle* rectangle =
            new IlvFilledRectangle(display,
                                   IlvRect(0, 0, 20, 20),
                                   palette);
        // The IlvFilledRectangle object is set as the picture of the item.
        item->setGraphic(rectangle);
    }
    // The item is inserted into the pop-up menu.
    menu->insertItem(item);
}
```

関数の 2 番目のパラメータとして与えられた名前を使用してメニュー・アイテムを作成した後、IlvFilledRectangle が右パレットで作成され、メニュー・アイテムのピクチャとして設定されます。次に、メニュー・アイテムがメニューに挿入されます。

メニュー・コールバック

メニュー・アイテムが選択されたときにアイテムの色がどのように変更されるかの詳細を次に示します。メニュー・コールバックのコード詳細を示します。

```
static void
ChangeColor(IlvGraphic* g, IlvAny arg)
{
    IlvPopupMenu* menu = (IlvPopupMenu*)g;
    // Retrieve the selected item of the pop-up menu.
    IlvShort selected = menu->whichSelected();
    if (selected != -1) {
        // Retrieve its graphical representation.
        IlvSimpleGraphic* graphic =
            (IlvSimpleGraphic*)menu->getItem(selected)->getGraphic();
        // Use the second argument of the callback: A pointer to the colored
        // tree gadget object.
        ColoredTreeGadget* tree = (ColoredTreeGadget*)arg;
        // Retrieve the item whose color is going to be changed.
        IlvTreeGadgetItem* item =
            ColoredTreeGadgetInteractor::GetSelectedItem(menu);
        // Change the color of all the children of the parent item.
        if (item && item->getParent())
            tree->setChildrenBackground(item->getParent(),
                                        graphic
                                        ? graphic->getForeground()
                                        : (IlvColor*)0);
    }
}
```

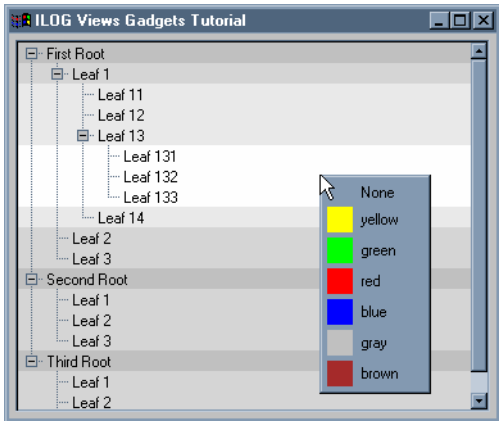
選択されたメニュー・アイテムのピクチャから色が取られ、setChildrenBackground メソッドを呼び出して ColoredTreeGadget に設定されます。

新しいインタラクタのテスト

ファイル `main.cpp` は、`ColoredTreeGadgetInteractor` クラスをテストするコードを含んでいます。ステップ 1 の `main.cpp` ファイルと同じですが、`ColoredTreeGadget` インスタンスにインタラクタが設定される点が異なります。

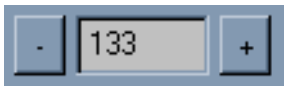
```
tree->setInteractor(new ColoredTreeGadgetInteractor());
```

これでステップ 2 は完了です。テスト・プログラムは、次のように見えるはずです。



ステップ 3: 複合ガジェットの作成

このステップでは、新しいガジェットの作成方法、つまり `IlvGadget` クラスの直接サブクラスの作成方法を説明します。作成するガジェットは、3 つのガジェット、つまり、1 個のテキスト・フィールドおよび 2 つのボタンから構成される複合ガジェットです。`UpDownField` ガジェットは、次のように表示されます。



このステップでは、次のタスクを実行する方法を紹介します。

- *他のガジェットから構成されるガジェットの作成*
- *ガジェットでキーボード・フォーカスを処理する*
- *ガジェットでのイベント処理*
- *ガジェットへコールバックを追加する*

◆ *複合ガジェットのテスト*

他のガジェットから構成されるガジェットの作成

複合ガジェットの作成とは、1 つ以上の既存のガジェットから構成される新しいガジェットを作成するということです。スクロール操作を可能にする内部 `IlvScrollBar` オブジェクトを有する `IlvTreeGadget` がこれに相当します。これはまた、このステップで作成される `UpDownField` クラスにもあてはまります。

複合ガジェットの宣言

`UpDownField` が構成されているオブジェクトへのポインタを維持するために、保護されたメンバ変数を使用します。

```
class UpDownField
: public IlvGadget
{
     ....
protected:
    IlvTextField* _textField;
    IlvButton*    _rightButton;
    IlvButton*    _leftButton;
   };
```

コンポーネントの作成

最初に、複合ガジェットを構成するオブジェクトを作成しなくてはなりません。`UpDownField` クラスは、クラスの各コンストラクタに呼び出される保護されたメンバ関数 `init` を持っています。

```
void
UpDownField::init(const char* label)
{
    // Compute the bounding boxes of each element.
    IlvRect r1, r2, r3;
    computeRects(r1, r2, r3);
    // Text field.
    _textField   =
        new IlvTextField(getDisplay(),
                         label,
                         r2,
                         getThickness(),
                         getPalette());
    _focusGadget = _textField;
    // Left Button.
    _leftButton  = new IlvButton(getDisplay(), "-", r1, getThickness(),
                                 getPalette());
    _leftButton->setCallback(_internal_Down, this);
    // Right Button.
    _rightButton = new IlvButton(getDisplay(), "+", r3, getThickness(),
                                 getPalette());
    _rightButton->setCallback(_internal_Up, this);
}
```

init メソッドはまず、全オブジェクトのバウンディング・ボックスを computeRects メンバ関数を呼び出して計算します。それから、オブジェクトを作成して初期化します。

> ***メモ:*** *2 つのボタンにはコールバックが与えられます。左ボタンのコールバックは、*UpDownField::decrement *メソッドを呼び出し、右ボタンコールバックは* UpDownField::increment *メソッドを呼び出します。*

レイアウト

以下に、UpDownField クラスのレイアウトを担当する computeRects メソッドの記述です。

```
void
UpDownField::computeRects(IlvRect& r1,
                          IlvRect& r2,
                          IlvRect& r3,
                          const IlvTransformer* t) const
{
    IlvRect rect = _drawrect;
    if (t)
        t->apply(rect);
    r1.moveResize(rect.x(), rect.y(), (IlvDim)ButtonWidth, rect.h());
    IlvDim width = rect.w() - (2*(ButtonWidth + Margin));
    r2.moveResize(rect.x() + (IlvPos)(ButtonWidth + Margin),
                  rect.y(),
                  (IlvDim)IlvMax(width, (IlvDim)0),
                  rect.h());
    IlvPos deltaX = (IlvPos)(rect.w() - ButtonWidth);
    r3.moveResize(rect.x() + (IlvPos)IlvMax(deltaX, (IlvPos)0),
                  rect.y(),
                  ButtonWidth,
                  rect.h());
    r1.intersection(rect);
    r2.intersection(rect);
    r3.intersection(rect);
}
```

- r1 は、左ボタンのバウンディング・ボックスです。
- r2 は、テキスト・フィールドのバウンディング・ボックスです。
- r3 は、右ボタンのバウンディング・ボックスです。

このメソッドは、毎回複合ガジェットを移動、リサイズするたびに呼び出されます。すべてのグラフィック・オブジェクトと同様に、applyTransform メソッドはどちらのケースでも呼び出されます。

```
void
UpDownField::applyTransform(const IlvTransformer* t)
{
    IlvGadget::applyTransform(t);
    IlvRect r1, r2, r3;
    computeRects(r1, r2, r3);
```

```
    _rightButton->moveResize(r3);
    _textField->moveResize(r2);
    _leftButton->moveResize(r1);
}
```

これはまた、コンポーネント・バウンディング・ボックスについて複合ガジェットに問い合わせを行うのにも使用されます。

コンポーネントの描画

以上で、複合ガジェットのコンポーネントが配置されました。複合ガジェットは、これらを各コンポーネントの draw メンバ関数を呼び出して描画します。

```
void
UpDownField::draw(IlvPort* dst,
                  const IlvTransformer* t,
                  const IlvRegion* clip) const
{
    _textField->draw(dst, t, clip);
    _rightButton->draw(dst, t, clip);
    _leftButton->draw(dst, t, clip);
}
```

IlvGraphic メンバ関数の再定義

IlvGraphic クラスのいくつかのメンバ関数は、複合ガジェットのコンポーネントへの委譲を可能にするために再定義されています。setPalette メソッドを例として挙げます。

```
void
UpDownField::setPalette(IlvPalette* palette)
{
    IlvGadget::setPalette(palette);
    _textField->setPalette(palette);
    _rightButton->setPalette(palette);
    _leftButton->setPalette(palette);
}
```

UpDownField のパレットが変更されると、UpDownField の各コンポーネントに同じパレットが設定されます。

最後に、setHolder メソッドを次のようにオーバーライドしなくてはなりません。

```
void
UpDownField::setHolder(IlvGraphicHolder* holder)
{
   IlvGadget::setHolder(holder);
   _textField->setHolder(holder);
   _rightButton->setHolder(holder);
   _leftButton->setHolder(holder);
}
```

このメソッドは、UpDownField がホルダに追加、あるいは削除されるたびに呼び出されます。これによって UpDownField のコンポーネントすべてが同じホルダを

持つようになります。ガジェットが適切に振る舞うためにはホルダを必要とするため、これは必須です。

ガジェットでキーボード・フォーカスを処理する

グラフィックおよびガジェット・クラスでのイベント処理の主な違いは、ガジェットはキーボード・フォーカスを持てるという点です。ガジェットがキーボード・フォーカスを持てば、キーボード・イベントはこのガジェットに送られるようになります。詳細は、ガジェットの理解にある「フォーカス管理」を参照してください。

キーボード・フォーカス処理には、次のような意味があります。

- フォーカス・イベントに反応すること。ガジェットにフォーカスが与えられようとするとき、IlvKeyboardFocusIn イベントを受け取ります。同様に、ガジェットがフォーカスを失いかけるとき、IlvKeyboardFocusOut イベントを受け取ります。
- フォーカス描画の処理。フォーカスの描画機構には 2 つのメソッドが含まれています。IlvGraphic::computeFocusRegion および IlvGraphic::drawFocus です。最初のメソッドは、フォーカス描画の位置に関する情報を与えます。2 番目のメソッドはフォーカスを描画します。

UpDownField の各コンポーネントにフォーカスを与えることができなくてはなりません。フォーカスが UpDownField に与えられると、それを送信するために選んだコンポーネントに転送されます。保護されたメンバ変数が、UpDownField の現在のフォーカス・コンポーネントにポインタを維持するために追加されます。

```
protected:
   IlvGadget*    _focusGadget;
```

setFocus という名前のメソッドも、UpDownField 内の現在フォーカスされているオブジェクトを変更するために追加されます。

```
void
UpDownField::setFocus(IlvGadget* gadget)
{
    IlvRegion region;
    // Send a focus_out event to the gadget that loses the focus.
    if (_focusGadget) {
        IlvEvent fo;
        fo._type = IlvKeyboardFocusOut;
        _focusGadget->computeFocusRegion(region, getTransformer());
        _focusGadget->handleEvent(fo);
        _focusGadget = 0;
    }
    _focusGadget = gadget;
    // Send a focus_in event to the gadget that receives the focus.
    if (_focusGadget) {
        IlvEvent fi;
        fi._type = IlvKeyboardFocusIn;
        _focusGadget->handleEvent(fi);
```

```
        _focusGadget->computeFocusRegion(region, getTransformer());
    }
    if (getHolder())
        getHolder()->reDraw(&region);
}
```

メソッドは最初に IlvKeyboardFocusOut イベントをキーボード・フォーカスを失うコンポーネントに送ります。それから、IlvKeyboardFocusIn イベントを新しくフォーカスされたコンポーネントに送ります。最後に、変更された領域が IlvGraphicHolder API を使用して再描画されます。

以下はフォーカスの描画方法を表しています。computeFocusRegion および drawFocus メソッドがこのプロセスに含まれています。

```
void
UpDownField::drawFocus(IlvPort* dst,
                       const IlvPalette* palette,
                       const IlvTransformer* t,
                       const IlvRegion* clip) const
{
    _focusGadget->drawFocus(dst, palette, t, clip);
}

void
UpDownField::computeFocusRegion(IlvRegion& region,
                                const IlvTransformer* t) const
{
    _focusGadget->computeFocusRegion(region, t);
}
```

UpDownField は、現在フォーカスされているオブジェクトに委譲します。次のセクションでは、UpDownField のキーボード・フォーカスをどのように変更するかを説明します。

ガジェットでのイベント処理

ガジェット・クラスのイベント用エントリ・ポイントは、IlvGadget::handleEvent メソッドです。このメソッドは、ガジェット・ホルダがガジェットによって処理するイベントを受け取ったときにガジェット・ホルダによって呼び出されます。フォーカスされたオブジェクトへイベントのほとんどを委譲するため、UpDownField::handleEvent は非常に単純です (_focusGadget メンバ変数によってポイントされています)。

```
IlvBoolean
UpDownField::handleEvent(IlvEvent& event)
{
    IlvBoolean result = IlvFalse;
    switch (event.type()) {
    case IlvButtonDown:
        {
            // Changing focus on click
            IlvRect r1, r2, r3;
            IlvPoint evp(event.x(), event.y());
```

```
            computeRects(r1, r2, r3,  getTransformer());
            if (r2.contains(evp) && _focusGadget != _textField)
                setFocus(_textField);
            else
            if (r3.contains(evp) && _focusGadget != _rightButton)
                setFocus(_rightButton);
            else
            if (r1.contains(evp) && _focusGadget != _leftButton)
                setFocus(_leftButton);
            result = _focusGadget->handleEvent(event);
            break;
        }
    case IlvKeyDown:
        {
            // Moving focus with the Tab key.
            if (event.data() == IlvTab &&
                (!(event.modifiers() & IlvShiftModifier)) &&
                (!(event.modifiers() & IlvCtrlModifier))) {
                if (_focusGadget == _textField)
                    setFocus(_rightButton);
                else
                if (_focusGadget == _rightButton)
                    setFocus(_leftButton);
                else
                if (_focusGadget == _leftButton)
                    setFocus(_textField);
                return IlvTrue;
            }
            // Moving focus with the Shift Tab key.
            if (event.data() == IlvTab &&
                ((event.modifiers() & IlvShiftModifier)) &&
                (!(event.modifiers() & IlvCtrlModifier))) {
                if (_focusGadget == _leftButton)
                    setFocus(_rightButton);
                else
                if (_focusGadget == _rightButton)
                    setFocus(_textField);
                else
                if (_focusGadget == _textField)
                    setFocus(_leftButton);
                return IlvTrue;
            }
        }
    default:
        result = _focusGadget->handleEvent(event);
    }
    return result;
}
```

handleEvent メソッドの主要部分は、キーボード・フォーカス管理を処理します。IlvButtonDown のケースは、複合ガジェットがボタン・ダウン・イベントを受け取ったときに、現在フォーカスされているオブジェクトの変更を行います。IlvKeyDown ケースは、複合ガジェットが TAB キーあるいは Shift-TAB キー・イベントを受け取ったときに、現在フォーカスされているオブジェクトの変更を行います。デフォルト・ケースは、単に現在フォーカスされているオブジェクトに委譲します。

ガジェットへコールバックを追加する

このステップの最初に示したように、UpDownField の左および右ボタンには、押されたときに通知を可能にするコールバックが割り当てられています。2 つのコールバックが定義されています。左ボタンが押されたときに呼び出される Down コールバック、および右ボタンが押されたときに呼び出される Up コールバックです。

これらのコールバックは、getCallbackType メソッドで宣言されています。

```
IlvUInt
UpDownField::getCallbackTypes(const char* const**      names,
                              const IlvSymbol* const** types) const
{
    IlvUInt count = IlvGadget::getCallbackTypes(names, types);
    AddToCallbackTypeList(count, names, types,
                          "Down", downCallbackType());
    AddToCallbackTypeList(count, names, types,
                          "Up", upCallbackType());
    return count;
}
```

> ***メモ:*** *このメソッドを再定義することは必須ではありませんが、*UpDownField *によって処理されているコールバックのリストをエディタ (*IBM ILOG Views *Studio など) が問い合わせることができるようになるため、エディタでの* UpDownField *の統合が容易になります。*

次に、UpDownField::increment メソッドが Up コールバックを呼び出すために実装され、UpDownField::decrement メソッドが Down コールバックを呼び出すために実装されます。

```
void
UpDownField::increment()
{
    callCallbacks(upCallbackType());
}

void
UpDownField::decrement()
{
    callCallbacks(downCallbackType());
}
```

複合ガジェットのテスト

ファイル main.cpp は、UpDownField クラスをテストするコードを含んでいます。テキスト・フィールドの値を変更するために UpDownField クラスの Up および Down コールバックを実装する方法を示します。Down コールバックは、テキスト・フィールドの値を減少させ、Up コールバックはこれを増加させます。

まず、コンテナが作成され、UpDownField クラスのインスタンスがこれに追加されます。

```
IlvDialog* dialog = new IlvDialog(display,
                                  title,
                                  title,
                                  IlvRect(0, 0, 100, 100));
UpDownField* but = new UpDownField(display, IlvRect(5, 5, 100, 23) , "0");
dialog->addObject(but);
```

次に、UpDownField のコールバックが設定されます。

```
but->setUpCallback(Increment);
but->setDownCallback(Decrement);
```

以下にコールバックの実装を示します。

```
static void Increment(IlvGraphic* g, IlvAny)
{
    char buffer[1000];
    UpDownField * obj = (UpDownField*)g;
    const char* label = obj->getLabel();
    if (label && *label) {
        IlvInt value = ((IlvInt)atof(label))+1;
        sprintf(buffer, "%ld", value);
        obj->setLabel(buffer, IlvTrue);
    } else
        obj->setLabel("0", IlvTrue);
}

static void Decrement(IlvGraphic* g, IlvAny)
{
    char buffer[1000];
    UpDownField * obj = (UpDownField*)g;
    const char* label = obj->getLabel();
    if (label && *label) {
        IlvInt value = ((IlvInt)atof(label))-1;
        sprintf(buffer, "%ld", value);
        obj->setLabel(buffer, IlvTrue);
    } else
        obj->setLabel("0", IlvTrue);
}
```

これらのコールバックは、UpDownField インスタンスのテキスト・フィールドの値を UpDownField::getLabel を呼び出して取得し、値 1 を追加 / 削除して変更し、UpDownField::setLabel を呼び出してテキスト・フィールドのラベルとしてこれを設定します。

これで、ステップ 3 が完了しました。

索引

C

C++
　前提条件 **3**

V

Viewfile チュートリアル **3**

か

ガジェット・カスタマイズのチュートリアル **37**

ち

チュートリアル
　viewfile **3**
　ガジェットのカスタマイズ **37**

ひ

表記法 **3**

ま

マニュアル
　表記法 **3**
　命名規則 **4**

め

命名規則 **4**