IBM ILOG Views

Prototypes 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

本書では、IBM® ILOG® Views の Prototype パッケージ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

このマニュアルは、以下の章で構成されています。

- **1 章** では、プロトタイプの概念について説明します。
- **2 章** Prototype 拡張機能がある IBM® ILOG® Views Studio ウィンドウ、パネル、およびコマンドについて説明します。

- **3 章**では、グラフィック・オブジェクトと、これらに振る舞いを割り当ててプロトタイプを作成するための IBM ILOG Views Studio の使用法について説明します。
- **4 章**では、プロトタイプの操作に使用するクラスおよびメソッドについて、および BGO をフルに活用するためのアプリケーションの構成方法について説明します。IBM ILOG Views Studio で作成したプロトタイプを C++ アプリケーションで使用する方法を説明します。
- **5 章**は、プロトタイプで定義済みの振る舞いの一覧です。

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。
- 初出の*斜体*の用語には、ユーザ・マニュアルの用語集で解説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。

- IBM® ILOG® Views 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

Prototype パッケージの概要

Prototype パッケージでは、*Business Graphic Objects (BGO)* と呼ばれるカスタム・ドメイン特有のグラフィック・オブジェクトを作成できます。これらのオブジェクトは、IBM ILOG Views Studio の Prototype 拡張機能を使用して、C++ を記述することなく対話的に作成されます。

このセクションでは、BGO の概念を説明し、IBM ILOG Views Studio で作成したプロトタイプ操作に使用されるクラスおよびメソッドについて説明します。

Prototype パッケージの概要

このセクションでは、BGO を作成するために IBM® ILOG® Views の Prototype パッケージの使用方法を説明します。IBM ILOG Views BGO は、プロトタイプ・デザイン・パターンに基づいているので、しばしばプロトタイプと呼ばれています。

Prototype パッケージの概要として、このセクションでは次の項目を扱います。

- *ビジネス・グラフィック・オブジェクト*
- *IBM ILOG Views Studio の Prototypes 拡張機能を使用して BGO を作成する*
- *アプリケーションでプロトタイプを使用する*
- *プロトタイプを使用する理由*

- *プロトタイプ・デザイン・パターン*
- *アクセサを使用してグラフィックおよび対話的な振る舞いを指定する*

ビジネス・グラフィック・オブジェクト

アプリケーション開発者は、ユーザが対話的に使用できるようなドメイン固有アプリケーション・オブジェクトを表現するために、グラフィック・オブジェクトをカスタマイズ化する必要に迫られることがよくあります。IBM® ILOG® Views Prototypes パッケージは、そのようなビジネス・グラフィック・オブジェクト (BGO) を構築するための簡単で効率的なソリューションを提供します。BGO は、IBM ILOG Views Studio の Prototype 拡張機能を使用して作成します。BGO の作成にはコーディングは必要ありません。3 つの基本手順を行うことで作成できます。

1. BGO のアプリケーション・インターフェースを、オブジェクトのドメインを表現するタイプ化されたアトリビュートのセットとして定義します。たとえば発電所のボイラを表すボイラ・オブジェクトの場合、温度、容量、レベル、インプット・バルブ、アウトプット・バルブなどのアトリビュートを設定できます。

2. 線、テキスト、画像などの基本的な IBM ILOG Views グラフィック・オブジェクトを使って、オブジェクトの外観を定義します。他の BGO を含めてオブジェクトを構造化することもできます。たとえば、ボイラ・オブジェクトは矩形で、矩形内のゲージで温度およびレベルを、そしてインプットおよびアウトプット・バルブを矩形内のトグル・ボタンで表すことができます。

3. 振る舞いをグラフィック・オブジェクトに付加して、これらをアプリケーション・オブジェクトのステータスをどのように表すか、さらにユーザ・イベントにどのように反応するかを定義します。動的に形状のアトリビュートを変更したり、オブジェクトをアニメーション化したり、BGO を互いに接続してユーザ・インターフェースでのオブジェクトのステータスを反映させることができます。たとえば、塗りつぶしの振る舞いをレベル・アトリビュートに付加すると、ボイラのレベルをそのグラフィック表示と同期できます。

次に、BGO のインスタンスを作成し、基本 IBM ILOG Views グラフィック・オブジェクトで行う場合と同様、それらのインスタンスをマネージャやコンテナ内で使用します。アプリケーション・オブジェクトは対応する BGO にリンクできます。表示、同期化、およびユーザ・インタラクションは、Prototypes パッケージによって処理されます。BGO はいつでも編集・変更できます。プロトタイプのインスタンスは自動的に更新されます。

ドメイン固有オブジェクト用にパワフルで直接操作できるインターフェースを作成するのは、フォーム・ベースのインターフェースを作成するのと同じ位簡単ですが、直接操作のインターフェースの方がユーザにとってより魅力的でわかりやすいものとなります。

図 1.1 は、プロトタイプで構築したアプリケーション・パネルの例を表しています。

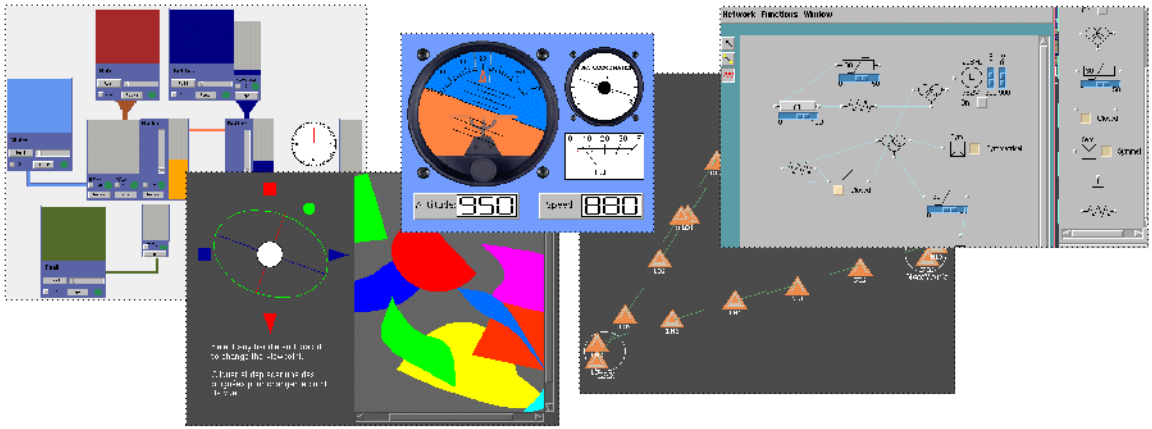


図 1.1 *プロトタイプ・アプリケーションの例*

IBM ILOG Views Studio の Prototypes 拡張機能を使用して BGO を作成する

IBM ILOG Views Studio を使用して BGO を作成すると、以下の処理が行えます。

- 基本的なグラフィック・オブジェクトを組み合わせて、独自の BGO 外観を設計する。
- 定義済み振る舞いを付加する、あるいはスクリプトを記述することで、グラフィックの振る舞いおよび BGO の対話的な側面を定義する。
- パネルで再利用したり、変更、インスタンス化できる*プロトタイプ*・オブジェクトとして BGO をライブラリに保存する。BGO はたいていの場合プロトタイプとして使用されるため、用語「プロトタイプ」と「BGO」は相互交換的に用いられます。
- 独自のプロトタイプのインスタンスをマネージャ、またはコンテナに追加する。
- プロトタイプおよびプロトタイプを保持するパネルの振る舞いをテストする。

これらの操作はすべて、WYSIWYG (what you see is what you get) モードで、C++ コーディングなしに行われます。

アプリケーションでプロトタイプを使用する

基本的 IBM ILOG Views グラフィック・オブジェクトを含むファイルのロードと同じ方法で、プロトタイプのインスタンスを含む IBM® ILOG® Views ファイルを、

マネージャあるいはコンテナにロードすることができます。また、プロトタイプのインスタンスを作成し、それらをアプリケーション・オブジェクトに付加して、マネージャあるいはコンテナに配置することもできます。

プロトタイプは、IlvGraphic のサブクラスではありません。プロトタイプは、IlvGroup クラスのオブジェクトに含まれるグラフィック・オブジェクトのグループです。プロトタイプの定義はファイルに保存されているので、プロトタイプを変更する場合にアプリケーションを再コンパイルする必要がありません。

プロトタイプ・インスタンスを IlvManager あるいは IlvContainer に配置するには、これらを IlvProtoGraphic と呼ばれる IlvGraphic の特殊サブクラスに埋め込まなくてはなりません。IBM ILOG Views Studio を使用してプロトタイプをマネージャあるいはコンテナに配置する場合、IBM ILOG Views Studio はカプセル化する IlvProtoGraphic を作成します。IlvGraphic と同じ方法で IlvProtoGraphic を操作することができます。IlvGroupHolder クラスを使用して、任意のビュー (コンテナあるいはマネージャ) のプロトタイプ・インスタンスを取得し、表示させたいアプリケーション値に応じてプロトタイプ・インスタンスのプロパティを変更することができます。

プロトタイプを使用する理由

BGO を定義するには、プロトタイプを使用するか、IBM ILOG Views メソッドの直接呼出しを使用して、IlvGraphic のサブクラス用に C++ コードを記述してオブジェクトを描画します。プロトタイプは直接コーディングの代替として使用します。

プロトタイプを使ったアプローチには、次の利点があります。

- 短期間の開発で対話的な GUI デザイン・プロセスが可能。
- アプリケーションの実装とユーザ・インターフェースの実装が明らかに分離されているので、メンテナンスとデバッグが容易に行える。
- ILOG Views Studio への完全統合。ユーザ・インターフェース・デザイナはプログラムするのではなく描画します。
- C++ プログラミングのスキルをほとんど必要としない。

その結果、オブジェクトのグラフィカルな外観をデザインする作業は、プログラマ以外に割り当てることができます。このような作業には、たとえば、グラフィック・デザイナの方がより適していることもあります。IBM ILOG Views Studio には、グラフィック・デザイナが使い慣れているツールに相当するような描画プログラムもあります。

プロトタイプはその効率性を強く強調してデザインされ、実装されています。プロトタイプは、常に直接 C++ コーディングを行うのと同じくらい効率的ではありませんが (派生ではなく複合に基づいているため)、アプリケーションは性能上の

問題を引き起こすことなく、何千ものプロトタイプ・インスタンスを作成できます。

プロトタイプ・デザイン・パターン

BGO 作成のプロセスは、プロトタイプ・デザイン・パターンに基づいています。ユーザは基本オブジェクトをグループ化し、グループをそこから複製 (またはインスタンス) を作成できるモデル (またはプロトタイプ) として使用できます。プロトタイプを変更すると、そのインスタンスもすべて自動的に更新されます。

プロトタイプ・デザイン・パターンを使用すると、WYSIWYG エディタで複雑なグラフィック・オブジェクトを作成し、そのオブジェクトをアプリケーション・パネルの構築に直ちに使用できます。

アクセサを使用してグラフィックおよび対話的な振る舞いを指定する

BGO はパブリック・アトリビュートのセットを定義します。これらのアトリビュートは、BGO のインターフェースをプログラムするアプリケーションに対応します。アトリビュートを任意の値に設定して、BGO の外観を変更することができます。いつでもこれらのアトリビュートを問い合わせることができます。

これらアトリビュートのそれぞれにいくつかの振る舞いを付加できます。振る舞いは、アトリビュートが変更された、あるいは問い合わされたときに実行される二次作用を定義します。たとえば、Condition 振る舞いを Temperature アトリビュートに付加できます。温度が変更すると条件が評価され、オブジェクトのグラフィック外観が変化します。Condition の振る舞いは、温度が定義済みの閾値を超えたときにオブジェクトの色を赤に設定することができます。また対話的な振る舞いを BGO に付加することもできます。たとえば、ユーザが温度をクリックしたときに温度が調整されるように指定できます。

アトリビュートおよび振る舞いは、アクセサ (クラス `IlvAccessor` のオブジェクト) で実装されます。アクセサはグラフィック・オブジェクトに付加し、次の処理が行えます。

- アトリビュート値を保存
- 二次作用の実行
- ユーザ・イベントの追跡

アクセサ機構により、複雑な振る舞いを定義できます。アプリケーション全体のロジックを再作成するためにアクセサを組み合わせることができます。しかしアクセサ機構は、オブジェクトおよびオブジェクトの対話的な振る舞いの定義にのみ使用することをお勧めします。アクセサ機構をアプリケーション・ドメインの機能の実装に使用しないでください。これにより、プログラムの健全なモジュラーの側面を維持します。

全体としてみると、BGO のアクセサは、*データ・フロー・グラフィック・プログラム*を定義します。データ・フロー・プログラミングは、C++ や Java などのプログラミング言語で使用される古典的な*コントロール・フロー・モデル*同様にパワフルです。ただし、データ・フロー・プログラミングは、小さなグラフィック志向のプログラムの定義により適しています。

複雑なグラフィックの振る舞いの定義を簡単にするために、スクリプト・アクセサでグラフィックおよ対話的な振る舞いを IBM ILOG Script プログラムとして定義することができます。これにより、より複雑な計算を実行し、IBM ILOG Script 機能全体へアクセスできます。

章

2

ユーザ・インターフェースおよびコマンド

この章では、ドメイン特有のアプリケーション・オブジェクトの完全に対話的なグラフィカル・ユーザ・インターフェースの開発を容易にするための拡張機能、IBM® ILOG® Views Studio の Prototype 拡張機能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。

- *概要*
- *メイン・ウィンドウ*
- *パレット・パネル*
- *グループの詳細情報パネル*
- *Prototype 拡張機能コマンド*

> ***メモ:** IBM ILOG Views Studio の Prototype 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。*

概要

IBM® ILOG® Views Studio の Prototype 拡張機能で、プロトタイプと呼ばれる複雑なグラフィック・オブジェクトを、IBM® ILOG® Views Views グラフィック・オブジェクトを対話的に組み合わせて定義することができます。これらのプロトタイプに振る舞いを付加し、アプリケーションのグラフィカルで対話的な部分全体を定義することができます。

プロトタイプはインスタンス化して、アプリケーション・ウィンドウ、詳細設定、あるいは直接操作インターフェースに基本的な構築ブロックとして使用することができます。これは、各アプリケーション・オブジェクトが直接、対話的でグラフィカルな表現に結び付けられている場合です。

Prototype 拡張機能付きの IBM ILOG Views Studio は、新しいワークフローを定義して高度に対話的なインターフェースを構築します。グラフィカル・エディタでアプリケーションの対話的な部分を開発し、これをライブラリに保存してから、C++ で書かれた中心的機能にリンクさせます。

Prototype 拡張機能付き IBM ILOG Views Studio の起動

2D Graphics Pro パッケージを既にインストールしてある場合は、IBM ILOG Views Studio が起動されるときに、Prototype 拡張機能は自動的に読み込まれます。拡張機能は、設定ファイルで `smproto` と呼ばれています。現在は使用されていない機能を有効にする互換性拡張機能も使用できます。`smoldpro`。このマニュアルでは、`smproto` 拡張機能についてのみ説明します。これらの拡張機能をインストールあるいはアンインストールするには、*IBM ILOG Views Studio の概要* のプラグインの読み込みのセクションを参照してください。

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。

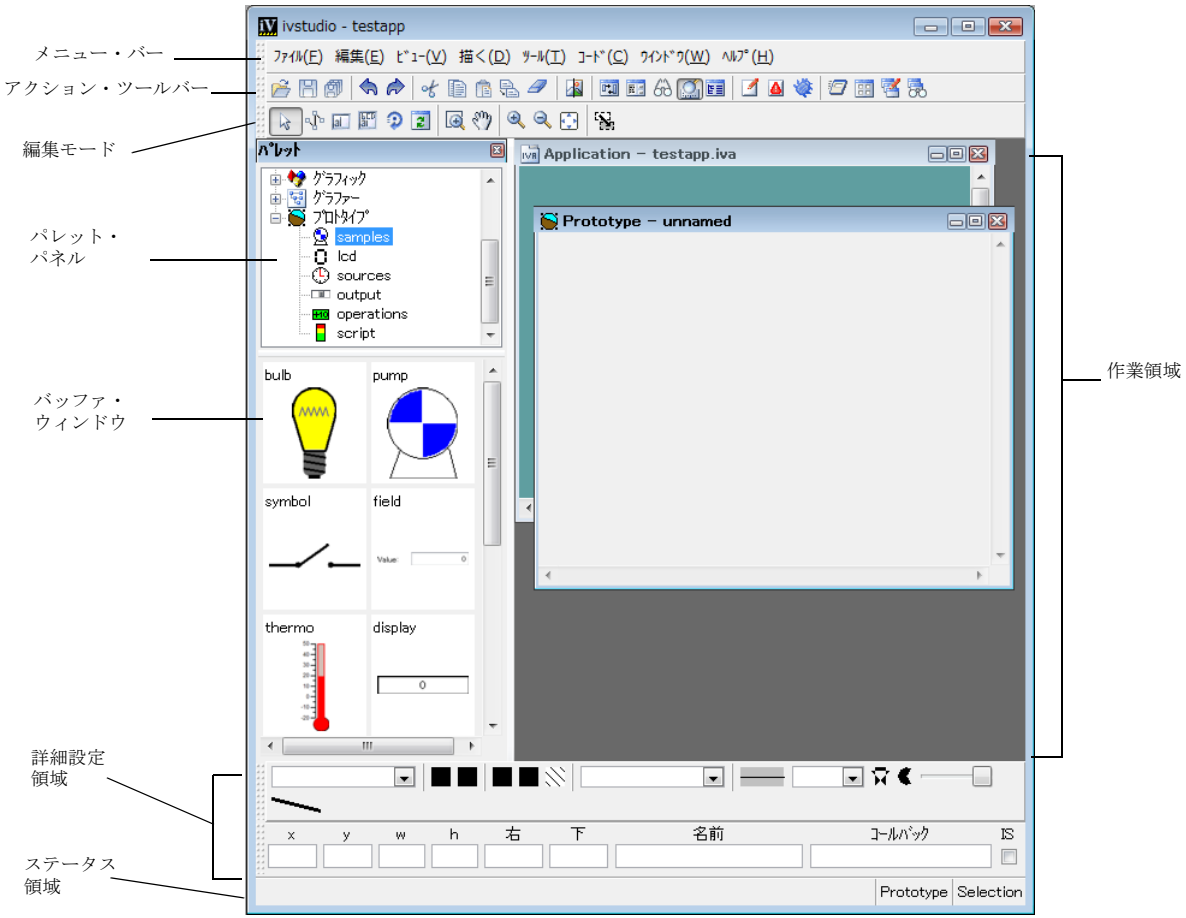


図2.1　Prototype 拡張機能のある IBM ILOG Views Studio メイン・ウィンドウ（起動時）

IBM ILOG Views Studio Foundation パッケージのみがインストールされているときと同じ数のメイン・ウィンドウが表示されます。ただし、Protoypes 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。次に各項目を簡単に紹介します。

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 Prototype 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。

- 2D Graphics
- Grapher
- Prototype

IBM ILOG Views Studio を起動すると、空の 2D グラフィック・バッファがデフォルトで表示されます。

現在メイン・ウィンドウに読み込まれているバッファの間で切り替えると、各バッファ・タイプには独自の編集モードのセットがあることがわかります。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。

2D Graphics バッファ・ウィンドウ

2D グラフィック・バッファは、Foundation パッケージのデフォルトです。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。

Grapher バッファ・ウィンドウ

Grapher バッファ・ウィンドウで、`IlvGrapher` の内容を編集できます。`IlvGrapher` を使用して、ノードとリンクの読み込み、編集、保存ができます。

新規の Grapher バッファ・ウィンドウを作成する方法は次の通りです。

1. [ファイル] メニューから [新規] を選択します。
2. 次に、表示されるサブメニューで [グラファー] 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 `NewGrapherBuffer` コマンドを実行することもできます。

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

Prototype バッファ・ウィンドウ

Prototype バッファ・ウィンドウは、プロトタイプを作成して操作するために使用されます。グラフィック・オブジェクトおよびプロトタイプは、パレット・パネルからアクティブの Prototype バッファ・ウィンドウにドラッグすることができます。

新しい Prototype 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. 次に、表示されるサブメニューで [プロトタイプ] を選択します。

あるいは、プロトタイプ・パレットで [プロトタイプ] をダブルクリックすると、Prototype バッファ・ウィンドウが自動的に開き、これを変更したり、そのアトリビュートや振る舞いの詳細設定を行うことができます。

メニュー・バー

Prototypes 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るようになります。

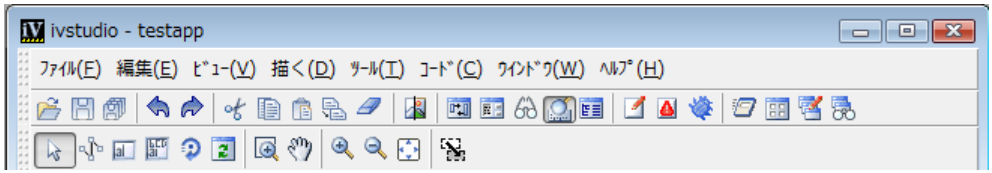


図 2.2　IBM ILOG Views Studio Prototype 拡張機能メニュー・バー

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 *Prototype 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 > プロトタイプ	NewPrototypeEditionBuffer
新規 > プロトタイプ・ライブラリ ...	NewProtoLibrary
新規 > プロトタイプ・グラファー	NewPrototypeGrapherBuffer *メモ: このコマンドは現在使用されていませんが、以前のバージョンとの互換性のために供給されています。*
プロトタイプ・ライブラリを保存 ...	SaveProtoLibraryAs
プロトタイプ・ライブラリを閉じる	CloseProtoLibrary

[描く] メニュー・コマンド

メニュー項目	コマンド
グループ化	GroupIntoGroup
プロトタイプの編集	EditPrototype
プロトタイプの削除	DeletePrototype

[ツール] メニュー・コマンド

メニュー項目	説明
グループの詳細情報	これは、現在選択されているプロトタイプ・インスタンスあるいは `IlvGroup` オブジェクトのグループ・インスタンスを開きます。

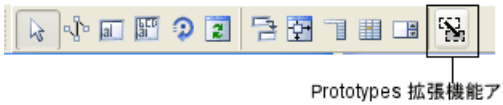
[ビュー] メニュー・コマンド

メニュー項目	コマンド
トグル・アニメーション・タイマー	ToggleTimers

アクション・ツールバー

アクション・ツールバーは Foundation パッケージと同じです。

編集モード・ツールバー



IBM ILOG Views Studio の Prototype 拡張機能には、通常の IBM ILOG Views Studio 編集モードとは別の編集モードが含まれています。

グループ接続モード

プロトタイプ・インスタンスの値を接続するためにグループ接続モードを使用します。接続モードは、プロトタイプ・インスタンス間の接続を定義するために使用されます。*プロトタイプ・インスタンスの接続* を参照してください。

パレット・パネル

プロトタイプ・パレットは、図 2.3 に示すとおり、パレット・パネルに含まれています。これは定義した、あるいは読み込んだ各種プロトタイプ・ライブラリを表示し、またプロトタイプのアイコンを Prototype バッファにドラッグしてインスタンス化することができます。プロトタイプは他のグラフィック・オブジェクト同様に操作されます。各ライブラリは、パレット・パネルでそれ独自のパネルを定義します。

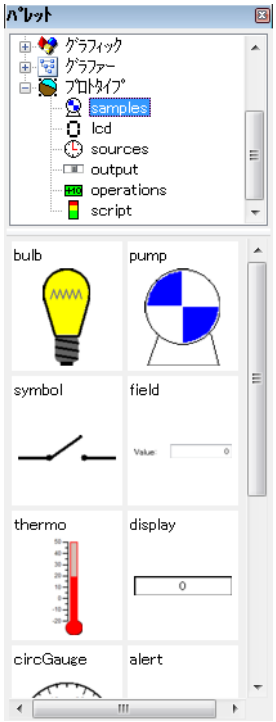


図2.3 samples プロトタイプ・ライブラリを表示するパレット・パネル

Prototype 拡張機能がインストールされているとき、IBM ILOG Views Studio は次のライブラリを起動時に読み込みます。

ライブラリ	説明
samples	起動時に読み込まれるサンプル・ライブラリ
sources	値ソースを含むプロトタイプ
output	ガジェットおよび定義する出力値を含むプロトタイプ
lcd	LCD ディスプレイ (1 桁および 4 桁)
operations	プロトタイプの接続およびこれらの値への操作の実行に使用されるプロトタイプ
script	スクリプト・アクセサを使用するプロトタイプ

これらのプロトタイプ・ライブラリの 1 つを開くには、パレット・パネルの上側ペインでプロトタイプ・パレット内の名前をクリックします。

プロトタイプ・アイコンをダブルクリックしてプロトタイプ定義を表示できます。これで Prototype バッファ・ウィンドウにプロトタイプが読み込まれ、グループの詳細情報パネルが開きます。

> ***メモ:*** *プロトタイプ・インスタンスを含むパネル・ファイルを読み込むとき、必要なプロトタイプ・ライブラリが自動的にプロトタイプ・パレットに読み込まれます。*

`<ILVHOME>/samples/protos` ディレクトリに、プロトタイプの使用法に関する他のサンプルがあります。

グループの詳細情報パネル

Prototypes 拡張機能は追加パネルを提供しており、これによりインターフェースおよびプロトタイプのグラフィックや振る舞いを、図 2.4 に示すように、定義することができます。グループおよびプロトタイプ・インスタンスをカスタマイズするためにも使用されます。

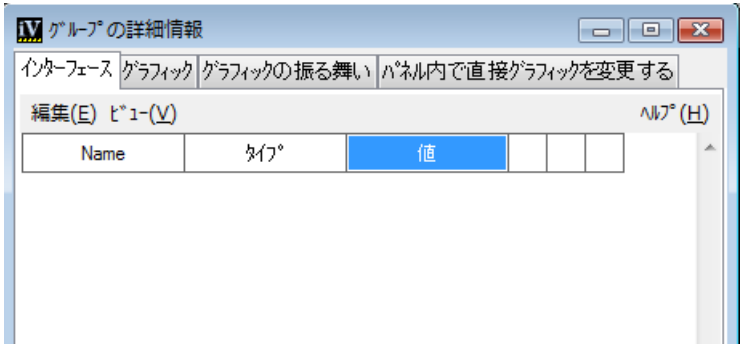


図 2.4 *グループの詳細情報パネル*

パネルへのアクセス

パネルには、次のいずれかの方法でアクセスできます。

- [ツール] メニューから [グループの詳細情報] を選択します。
- 新規 Prototype バッファ・ウィンドウを作成します。
- Prototype バッファ・ウィンドウでプロトタイプをダブルクリックします。
- ツール・メニューからコマンドを選択し、リストから `ShowGroupInspector` コマンドを選択し、[適用] をクリックします。

グループの詳細情報を構成する要素

グループの詳細情報パネルには、4 つのタブがあります。

- [インターフェース] タブは、プロトタイプのアトリビュートを定義し、プロトタイプ・インスタンスをカスタマイズするために使用されます。
- [グラフィック] タブはプロトタイプを構成するグラフィック・オブジェクトを表示して編集するために使用されます。
- [グラフィックの振る舞い] タブは、プロトタイプのグラフィック振る舞いを定義するために使用されます。
- [パネル内で直接グラフィックを変更する] タブは、プロトタイプの対話的な振る舞いを定義するために使用されます。

完全に文脈依存型のハイパーテキスト・ヘルプが、詳細設定の [ヘルプ] をクリックすると利用できます。このヘルプは、[ヘルプを閉じる] ボタンをクリックして非表示にすることができます。

グループの詳細情報パネルの機能の詳細は、*プロトタイプの作成と使用* を参照してください。

Prototype 拡張機能コマンド

このセクションでは、IBM® ILOG® Views Studio の Prototype 拡張機能で使用可能な追加の定義済みコマンドの一覧表を、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・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、コマンドが属するカテゴリ、およびコマンドの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。

CloseProtoLibrary

ラベル	プロトタイプ・ライブラリを閉じる
パス	メイン・ウィンドウ :[ファイル] メニュー
カテゴリ	プロトタイプ
アクション	パレット・パネルに現在表示されているプロトタイプ・ライブラリを閉じます。

ConvertProtoManager

ラベル	ProtoManager の変換
パス	ProtoManager の変換 :[編集] メニュー
カテゴリ	プロトタイプ
アクション	現在アクティブのプロトタイプ・インスタンス・バッファの内容をコピーする通常の Studio バッファを作成します。このコマンドは `IlvPrConvertProtoManager` 関数を使用して変換を行います。Views 3.1 プロトタイプから最新の API への切り替えを助けるものです。

DeletePrototype

ラベル	プロトタイプの削除
パス	メイン・ウィンドウ :[編集] メニュー
カテゴリ	プロトタイプ
アクション	プロトタイプをそのライブラリから削除します。

EditPrototype

ラベル	プロトタイプの編集
パス	メイン・ウィンドウ :[編集] メニュー
カテゴリ	プロトタイプ
アクション	新規 Prototype バッファ・ウィンドウで選択したプロトタイプを編集し、プロトタイプ・インスタンス用にグループの詳細情報パネルを開きます。

GroupIntoGroup

ラベル	IlvGroup
パス	メイン・ウィンドウ :[描く] メニュー > グループ化

カテゴリ	プロトタイプ
アクション	選択されたオブジェクトを `IlvGroup` にグループ化します。

NewProtoLibrary

ラベル	プロトタイプ・ライブラリ ...
パス	メイン・ウィンドウ :[ファイル] メニュー > 新規
カテゴリ	プロトタイプ
アクション	新規プロトタイプ・ライブラリを作成します。このライブラリは、パレット・パネルで表示されます。ファイル・セレクタ・ダイアログ・ボックスが開き、ライブラリ・ファイル (`.ipl`) を選択します。

NewPrototype

ラベル	新しいプロトタイプ
パス	メイン・ウィンドウ :[ファイル] メニュー > 新規
カテゴリ	プロトタイプ
アクション	プロトタイプを描画し、編集するために使用されるバッファ・ウィンドウを作成します。

メモ: *このコマンドは現在使用されていませんが、以前のバージョンとの互換性のために供給されています。*

NewPrototypeEditionBuffer

ラベル	プロトタイプ
パス	メイン・ウィンドウ :[ファイル] メニュー > 新規
カテゴリ	プロトタイプ
アクション	プロトタイプを描画し、編集するために使用されるバッファ・ウィンドウを作成します。

NewPrototypeGrapherBuffer

ラベル	プロトタイプ・グラファー
パス	メイン・ウィンドウ :[ファイル] メニュー > 新規
カテゴリ	プロトタイプ
アクション	`IlvProtoGrapher` クラスのインスタンスを作成できるようにします。

メモ: *このコマンドは現在使用されていませんが、以前のバージョンとの互換性のために供給されています。*

OpenProtoLibrary

ラベル	プロトタイプ・ライブラリを開く ...
パス	メイン・ウィンドウ :[ファイル] メニュー > 開く ...
カテゴリ	プロトタイプ
アクション	プロトタイプ・ライブラリ・ファイルが開きます。選択ダイアログ・ボックスが開き、ファイル (`.ipl`) を選択します。

SaveProtoLibraryAs

ラベル	プロトタイプ・ライブラリに名前を付けて保存 ...
パス	メイン・ウィンドウ :[ファイル] メニュー
カテゴリ	プロトタイプ
アクション	現在選択されているプロトタイプ・ライブラリのコピーを別のファイルに保存します。

SelectGroupConnectionMode

ラベル	グループ接続
パス	[編集モード] ツールバー
カテゴリ	プロトタイプ
アクション	グループ接続モードを選択します。

SelectGroupSelectionMode

ラベル	グループ選択
パス	[編集モード] ツールバー
カテゴリ	プロトタイプ
アクション	グループ選択モードを選択します。

SelectNodeSelectionMode

ラベル	ノード選択
パス	[編集モード] ツールバー
カテゴリ	プロトタイプ
アクション	プロトタイプ・バッファでノード選択モードを選択します。このモードでは、プロトタイプを編集中にノードを選択して詳細設定を行うことができます。

ShowGroupInspector

ラベル	グループの詳細情報
パス	[ツール] メニュー

カテゴリ	プロトタイプ
アクション	グループの詳細情報パネルを表示 / 非表示にします。

ToggleTimers

ラベル	トグル・アニメーション・タイマー
パス	[表示] メニュー
カテゴリ	プロトタイプ
アクション	プロトタイプのアニメーション・アクセサのアニメーション・タイマーをオンまたはオフにしてプロトタイプを編集し、その振る舞いをテストします。

UngroupIlvGroups

ラベル	グループ解除
パス	メイン・ウィンドウ :[描く] メニュー
カテゴリ	プロトタイプ
アクション	このコマンドは一般のグループ解除コマンドを置き換えて、`IlvGroup` オブジェクトを考慮します。

章

3

IBM ILOG Views Studio を使った BGO の作成

この章では、Prototype 拡張機能でどのように複合グラフィック・オブジェクトを作成し、これを対話的なポイント・アンド・クリック編集でアプリケーション・インターフェース、グラフィックの振る舞い、および対話的な振る舞いに割り当てるかを説明します。これらのグラフィック・オブジェクトは、アプリケーションイン・インターフェースに従ってドメイン特有のオブジェクトにリンクさせ、ドメイン・オブジェクトの直接操作編集である完全な WYSIWYG を可能にします。

以下のトピックに関する情報が記載されています。

- *プロトタイプの作成と使用*
- *プロトタイプ・ライブラリの読み込みおよび保存*
- *パネルでプロトタイプ・インスタンスを作成 / 編集する*
- *プロトタイプ・インスタンスの接続*

> ***メモ :*** *IBM ILOG Views Studio の Prototypes 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。*

プロトタイプの作成と使用

プロトタイプの作成および使用に関して、次のセクションに分けて説明します。

- *プロトタイプ・ライブラリの作成*
- *プロトタイプの作成*
- *アトリビュートの定義*
- *プロトタイプの描画*
- *グラフィックの振る舞いを定義する*
- *対話的な振る舞いの定義*
- *プロトタイプのテスト*
- *プロトタイプの保存*

プロトタイプ・ライブラリの作成

BGO をライブラリに作成して、それをまとめて取得したり操作できるようにします。

新規のプロトタイプ・ライブラリを作成するには、次の手順に従います。

1. メイン・ウィンドウの [ファイル] メニューから、[新規] > [プロトタイプ・ライブラリ] を選択します。

   ファイル・セレクタが表示されます。

2. ディレクトリを選択し、それに許可を書き込み、新規ライブラリの名前を入力します (拡張子は、`.ipl`)。[保存] をクリックします。

   新規ページが、作成したライブラリに対応してパレット・パネルに表示されます。

プロトタイプの作成

次は、プロトタイプの作成に関するタスクです。

- グループの詳細情報パネルの [インターフェース] タブでプロトタイプのアトリビュートを定義。
- Prototype バッファ・ウィンドウでプロトタイプを構成するグラフィック要素を描画。
- グループの詳細情報・パネルを使用してプロトタイプのグラフィックの振る舞いを定義。

- グループの詳細情報パネルを使用してプロトタイプの対話的な振る舞いを定義。

これらのタスクはインターリーブすることができます。つまり、いつでもプロトタイプのアトリビュート、グラフィック要素、振る舞いを追加、編集あるいは削除することができます。

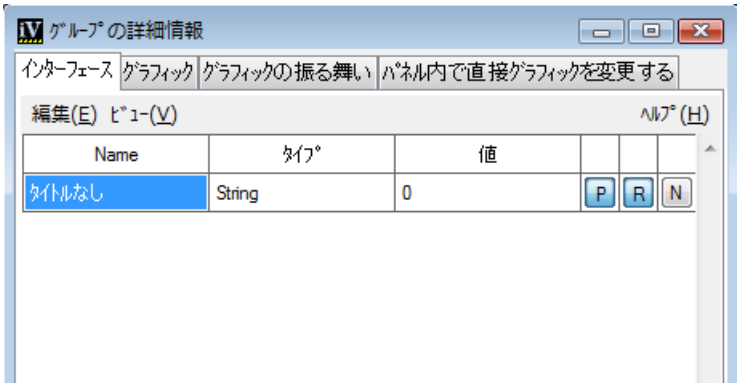
プロトタイプは、Prototype バッファ・ウィンドウで作成されます。これらのタスクを開始する前に、次の処理を行います。

1. [ファイル] メニューから [新規] > [プロトタイプ] を選択して Prototype バッファ・ウィンドウを開きます。
2. [ツール] メニューから [グループの詳細情報] を選択して、グループの詳細情報パネルを表示します。

アトリビュートの定義

次の手順で、プロトタイプあるいはグループの外部アトリビュート (あるいは「プロパティ」) を定義して編集します。(プロパティは、プロトタイプあるいはグループに、アプリケーションあるいはその他のオブジェクトからアクセスする方法を決定します。)

1. グループの詳細情報パネルの [インターフェース] タブを開きます。このタブでアトリビュートのセットを定義し、それぞれにタイプおよびデフォルト値を与えます。



2. [編集] > [新規アトリビュート] を選択するか、Ctrl+N でアトリビュートを追加します。

   表に新規行「タイトルなし」が表示されます。

3. アトリビュートの名前を指定するには、[タイトルなし] ボックスをクリックします。このアトリビュートに名前を入力します。この名前は、一意にする必要があります。このアトリビュートの振る舞いにアクセスするのに使用します。

4. アトリビュートのタイプを指定するには、隣の [タイプ] コンボ・ボックスを 2 回クリックします (あるいは、キーボードのみを使用している場合は F2 キーを押します)。タイプを指定するプルダウン・メニューを選択します。

   すべてのアトリビュートはタイプに分けられています。各タイプは、アトリビュートに割り当てることのできる値の種類を表しており、その意味を決定する助けとなります。利用できるタイプは以下のとおりです。

   - **値** - アトリビュートは、直接設定したり問い合わせることのできる値を保持します (文字列、色、整数など)。アトリビュートを任意の値タイプに設定すると、コンボ・ボックスがこのタイプを直接表示します。
   - **リファレンス** - グループの他の内部アトリビュートへの参照。たとえば、「温度」という名前のアトリビュートを作成し、それに「スライダ」グラフィック・オブジェクトの「値」アトリビュートを参照することで、「温度」のより適切な名前の下でスライダの「値」内部アトリビュートにアクセスできます。グループが温度計を表現する場合に便利です。これはプログラム言語のポインタ、あるいはエイリアスと同じです。リファレンスへのタイプを設定してあれば、プレフィックス「 ^ 」のある参照アトリビュートが、アトリビュート・タイプを記述するコンボ・ボックスに表示されます。
   - **アトリビュートのグループ化** - アトリビュートという名前のグループにあるすべてのサブアトリビュートは、集合的にアドレス指定され、同じ値を割り当てられます。たとえば、「前景」という名前のアトリビュートを作成し、これにタイプ「グループ」を与えることで、グループに含まれる全オブジェクトの前景を同じ値に設定するアトリビュートを作成します。
   - **スクリプト** - スクリプトが実行されます。このスクリプトは、アトリビュートを定義する値を返します。[グラフィックの振る舞い] タブを使用して、値を定義する関数の名前を変更します。関数の名前は、アトリビュートのこのタイプを選択したときに「()」が付いて表示されます。
   - **タイプなし** - 純粋に機能的ないくつかのアトリビュートは、タイプに分かれていないものもあります。

   > ***メモ:*** *タイプも振る舞いもないアトリビュートは、システムで保持できません。アトリビュートを作成した後にそのタイプを「なし」に設定することは、アトリビュートを削除することと同じです。*

5. アトリビュートの値列に、アトリビュートのデフォルト値を入力します。

6. アトリビュートにその他のパラメータを設定するには、ページの右側のボタンを使用します。ボタンを放したときに、プロパティが設定されます。

- **パブリック** (詳細設定の P ボタン) - アトリビュートは外部オブジェクトによって表示可能です。アトリビュートはデフォルトではパブリックですが、これらのアトリビュートを非表示にして内部処理を実行するためだけに使用することもできます。
- **永続性** (詳細設定の R ボタン) - アトリビュート値は、グループが保存されるときに保存され、ユーザが最後に設定した値のセットを維持します。デフォルトでは、アトリビュートには永続性があります。読み書きを最適化したり、ファイルが読み込まれるときにプロトタイプの元のステータスにアトリビュートを常に復元させるには、非永続として設定します。
- **通知** (詳細設定の N ボタン) - これが設定されていると、アトリビュートは他のアトリビュートに値が変更されたことを通知します。これによって、他のアトリビュートがそれ自体を更新できるようになります。*Notify* を参照してください。

7. 追加するアトリビュートに手順 2 から 6 までを繰り返し、プロトタイプのインターフェース詳細を作成します。

[インターフェース] タブで [編集] メニューを使用する

プロトタイプのインターフェースを指定すると、次の処理を行うことができます。

◆ 他のプロトタイプのインターフェースをインポートして、そのプロトタイプから定義済みのアトリビュートおよび振る舞いを追加します。

[編集] > [継承] を選択して、利用できるリストから継承したいアトリビュートのあるプロトタイプを選択します。

表に灰色で表示された継承したアトリビュートを持つ新規アトリビュートが作成されます。これらの継承アトリビュートを直接編集することはできませんが、これらを他のアトリビュートを介して参照できます。

いくつかの継承されたアトリビュートは、既に他のアトリビュートあるいはグラフィック・ノードを参照している場合があります。そのため、すべてのプロトタイプを他のプロトタイプにインポートできるわけではありません。

◆ アトリビュートの整列

アトリビュートを選択して、[編集] > [上に移動] あるいは [編集] > [下に移動] を選択します。

◆ アトリビュートの削除

アトリビュートを選択して、[編集] > [削除] を選択します。

◆ 切り取り / コピー / 貼り付け : アトリビュート・ツリーの一行目を選択して、[編集] > [コピー] または [編集] > [切り取り] を選択すると、アトリビュート全体とその振る舞いをコピーしたり、切り取ることができます。まずアトリビュートを挿入する行を選択して、[編集] > [貼り付け] を選択すると、アトリビュート・クリップボードの内容を貼り付けることができます。

[インターフェース] タブで [ビュー] メニューを使用する

[インターフェース] タブのこのメニューは、グループあるいはプロトタイプのアトリビュートの他のビューを表しています。これにより、任意のグループあるいはプロトタイプについて、編集するアトリビュートのタイプを選択できます。

- インターフェース - グループあるいはプロトタイプ用に定義されている全アトリビュートへのアクセス、編集ができます。これはデフォルト表示です。
- パブリック・アトリビュート - プロトタイプのパブリック・アトリビュートのみを表示します。これらは他のオブジェクトおよびアプリケーションによってのみ見ることができます。
- 変更値 - プロトタイプと異なるプロトタイプ・インスタンスの値を一覧表示します。これらの値は、プロトタイプ・インスタンスとともに保存されます。
- すべてのアトリビュート - すべてのプロトタイプ値およびサブ値を一覧表示します。これらの値は変更可能ですが、新規設定を他の振る舞いがオーバーライドする場合、これらの変更がプロトタイプとともに保存される、ということではありません。

プロトタイプの描画

Prototype バッファ・ウィンドウを使用してプロトタイプのグラフィック表示を定義します。

- グラフィック・オブジェクトをバッファへドラッグ・アンド・ドロップして、編集モードで線、多角形などを作成することができます。Prototype バッファ・ウィンドウには、IBM ILOG Views Studio 2D Graphics バッファ・ウィンドウのすべてのプロパティがあります。
- プロトタイプを描画するとき、メイン・ウィンドウに追加して表示されるグループの詳細情報パネルのグラフィック・ページでその構造を見ることができます。図 3.1 はこの例を表しています。グラフィック・ノードのリストが上から下へまとめて表示されます。グラフィック・オブジェクトをプロトタイプに追加すると、ツリー構造が更新されます。グラフィック・ノードは、(IBM ILOG Views Studio で行うように) Prototype バッファ・ウィンドウで直接選択することもできますし、グループの詳細情報パネルに表れるツリー内で選択することもできます。

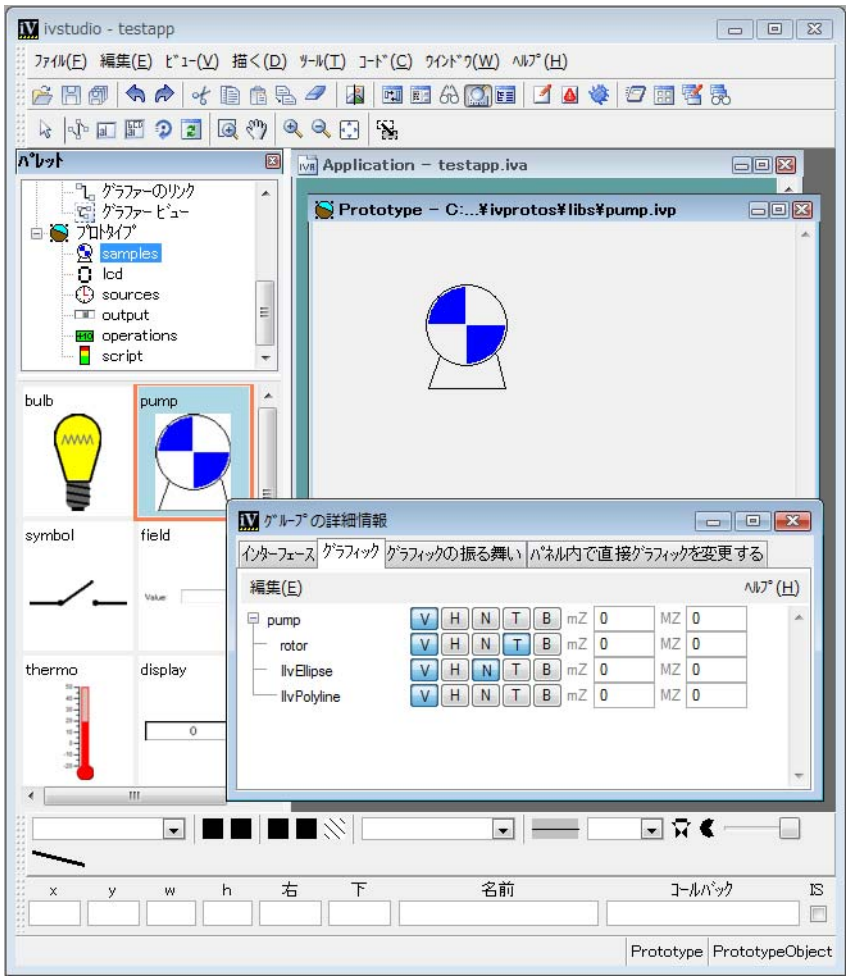


図3.1 *グループの詳細情報パネルのグラフィック・ページ*

プロトタイプ・ノードの編集

グループの詳細情報のグラフィック・ページ上のフィールドを使用して (図 3.2 を参照してください)、プロパティの要素 (あるいはノード) に関連付けられているプロパティの数を変更することができます。

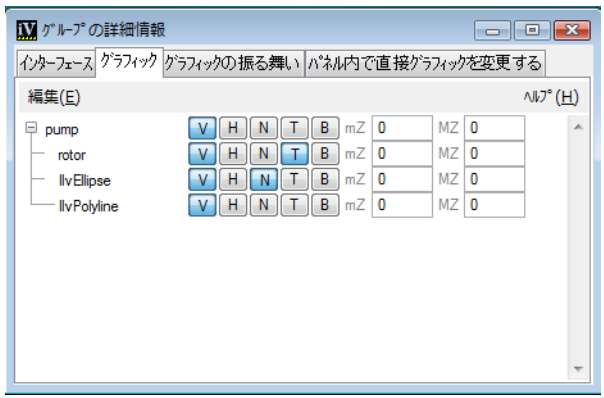


図3.2　グループの詳細情報パネルのグラフィック・ページ

- 選択したノードがグラフィック・ノードである場合、プロパティはこの特殊ノードにのみ適用されます。
- 選択したノードがグループ・ノードの場合、つまり、プロトタイプのルート・ノード、プロトタイプのサブグループ、あるいはプロトタイプ・インスタンスである場合、このプロパティは選択したグループのすべての子グラフィック・ノードに適用されます。

次の表は、グラフィック・ページにあるフィールドを表しています。

表3.1　グループの詳細情報パネルのグラフィック・ページのフィールド

フィールド	説明
ノード名	このテキスト・フィールドは、ノードの名前を変更するために使用されます。IBM ILOG Views Studio メイン・ウィンドウのオブジェクト詳細情報の名前フィールドも使用できます。 *メモ: ノードには、英数文字 (A ～z、a ～z、0 ～9) のみを使用します。*
(V) 表示	このトグルは、プロトタイプのグラフィック・オブジェクトの可視性を制御します。
(H) アプリケーション内で非表示	このボタンが設定されている場合、選択されたグラフィック・オブジェクトは、IBM ILOG Views Studio でプロトタイプあるいはプロトタイプのインスタンスを編集中にのみ表示されます。オブジェクトは最終アプリケーションでは非表示になります。このプロパティを使用して、「操作」プロパティ・ライブラリのインスタンスのような中間「計算」プロトタイプ・インスタンスを作成することができます。

表 3.1 *グループの詳細情報パネルのグラフィック・ページのフィールド（続き）*

フィールド	説明
(N) グラファー・ノード	このボタンが設定されている場合、グラフィック・オブジェクトがグラファー・ノードとしてプロトタイプがグラファーでインスタンス化されたときに追加されますこれにより、グラファー・ノードとしてプロトタイプ・インスタンスを使用することができます。(このボタンは現在使用されていませんが、互換性の理由から保持されています)。
(T) 変換	このボタンは、グラフィック・オブジェクトがひずみなく任意に確実に変換されるようにするため、トランスフォーマがグラフィック・ノードに関連付けられているかどうかを制御します。ローカル・トランスフォーマがないと、いくつかの IBM ILOG Views オブジェクトはリサイズされたときに元のジオメトリを失います。ローカル・トランスフォーマを使用することで、オブジェクトのジオメトリがジオメトリ変換によって変更されないようにします。一方で、ローカル・トランスフォーマの使用によってメモリの消費量は高まります。 このボタンを選択すると、標準選択モードを使ってノードのグラフィック・オブジェクトを詳細設定する必要があります。グループ選択モードを使用する場合、選択したオブジェクトは `IlvTransformedGraphic` のサブクラスのインスタンスとなり、詳細設定できません。
(B) サイズを拘束	これを設定すると、このフラグがオブジェクトのズーム可能性を制限します。このフラグを設定して mZ および MZ を 0 のままにすると、両方を 1 に設定するのと同じです。ただし、効率は高まります。このオプションを設定し、mZ および MZ が 0 ではなく、インスタンスのビューのズーム倍率が MZ より大きい、あるいは mZ より小さい場合、このフラグはオブジェクトを非表示にします。
(mZ) 最小ズーム	ゼロでない場合、このアトリビュートはオブジェクトの最小サイズを制限します。オブジェクトを保持するビューのスケーリング係数がこの値を下回る場合、オブジェクトはこれ以上大きくなりません。最小ズームおよび最大ズームが同じ値に設定されている場合、オブジェクトのサイズは変わりません。1 に設定されている場合、作成された時のサイズのままになります。
(MZ) 最大ズーム	ゼロでない場合、このアトリビュートはオブジェクトの最大サイズを制限します。オブジェクトを保持するビューのスケーリング係数がこの値を上回る場合、オブジェクトはこれ以上大きくなりません。

プロトタイプ・ノードの構造化

プロトタイプを構造化するには、グラフィック・ノードをサブグループにグループ化します。たとえば、オブジェクトのグループを回転させたり、その色を変更したいときなど、プロトタイプ・アクセサを定義するときにこれは便利です。サブグループを作成するには、次の処理を行います。

1. Prototype バッファ・ウィンドウで、グループ化したいグラフィック・オブジェクトを選択します。
2. [描く] メニューから [グループ化] を選択します。

   オブジェクトが `IlvGroup` クラスのインスタンスにグループ化され、プロトタイプにサブグループ・ノードが作成されます。ノード・ツリーに構造の変化が表示されます。

グループ選択モードを使用して、サブグループを選択して全体を移動させることができます。このモードは、選択したグループをグループの周囲に破線を描いて表示します。標準選択モードで、サブグループ内のグラフィック・オブジェクトを個別に選択することができます。

編集しているプロトタイプに他のプロトタイプのインスタンスを含めるには、次の処理を行います。

1. まだ開かれていない場合は [ツール] メニューから [パレット] を選択して、プロトタイプ・パレットをアクティブにします。
2. プロトタイプ・ライブラリを選択します。
3. プロトタイプを Prototype バッファ・ウィンドウにドラッグ・アンド・ドロップします。プロトタイプの詳細設定の [ノード] ページは、サブグループ・ノードの場合と同様にプロトタイプ・インスタンスに新規ノードを表示します。

オブジェクトを追加して、[インターフェース] タブに戻って内部ノードを参照する新規アトリビュートを定義したり、[グラフィックの振る舞い] タブでプロトタイプに動的振る舞いを定義することができます。

グラフィックの振る舞いを定義する

プロトタイプのグラフィックの振る舞いを、グループの詳細情報の [グラフィックの振る舞い] ページを使用して定義します (図 3.3 を参照してください)。グラフィックの振る舞いは、アトリビュートの変更がどのようにプロトタイプの視覚的側面に影響するかを決定します。各アトリビュートに振る舞いを追加することができます。以下は、値が変更されるたびに行われる指示です。

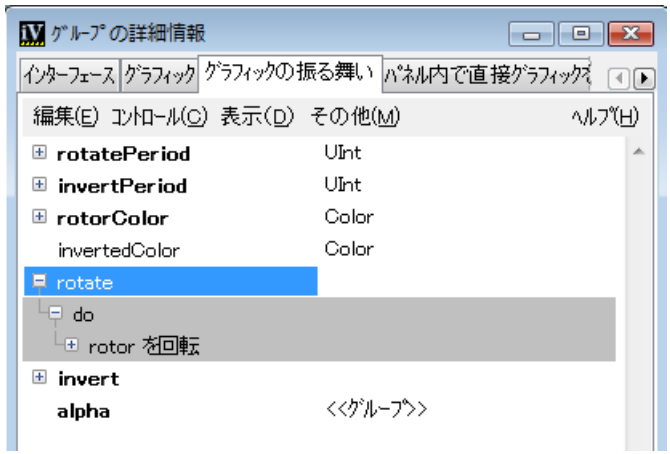


図3.3 *グループの詳細情報の [グラフィックの振る舞い] タブ*

振る舞いを追加するには、次の処理を行います。

1. リストのアトリビュートを選択します。
2. [コントロール]、[表示]、[その他] メニューから、追加する振る舞いを選択します。
   - 振る舞いのコントロールにより、一定の条件下であるアトリビュートが他のアトリビュートの変更をトリガできるようになります。たとえば、温度が任意の閾値を上回る場合に温度計を赤で表示させたい場合、ゲージの前景値に赤を割り当てる温度アトリビュート上に条件アクセサを追加します。
   - 振る舞いの表示により、回転、ズーム、可視性あるいはオブジェクトのアニメーションなどのオブジェクトのグラフィック・プロパティを変化させることができます。

さらに、グループあるいはプロトタイプのグラフィック外観が、1 つのアトリビュートの値が変更したときに完全に調整されるように、アトリビュートが他のアトリビュートにその変更を通知させるようにすることができます。[コントロール] メニューの通知の振る舞いで、それを監視している他のアトリビュートに振る舞いを実行するように伝え、[その他] メニューの監視の振る舞いで、あるアトリビュートが他のアトリビュートを監視していることを示すことができます。

すべての定義済み振る舞いの効果については、*定義済みアクセサ*を参照してください。

また、オンライン・ヘルプからこのページにアクセスすることもできます。メニュー・バーから [ヘルプ] を選択します。[コントロール]、[表示]、[その他] メニューから振る舞いを選択すると、振る舞いの効果を説明するヘルプがパネルの左ペインに表示されます。

[グラフィックの振る舞い] タブで [編集] メニューを使用する

[グラフィックの振る舞い] タブでは、次の処理が行えます。

- 計算の中間ステータスで使用される中間アトリビュートあるいは非表示アトリビュートを追加します。

1. [編集] > [新規アトリビュート] を選択し、アトリビュートを追加します。

   新しい名前の付いていないアトリビュートが表示されます。

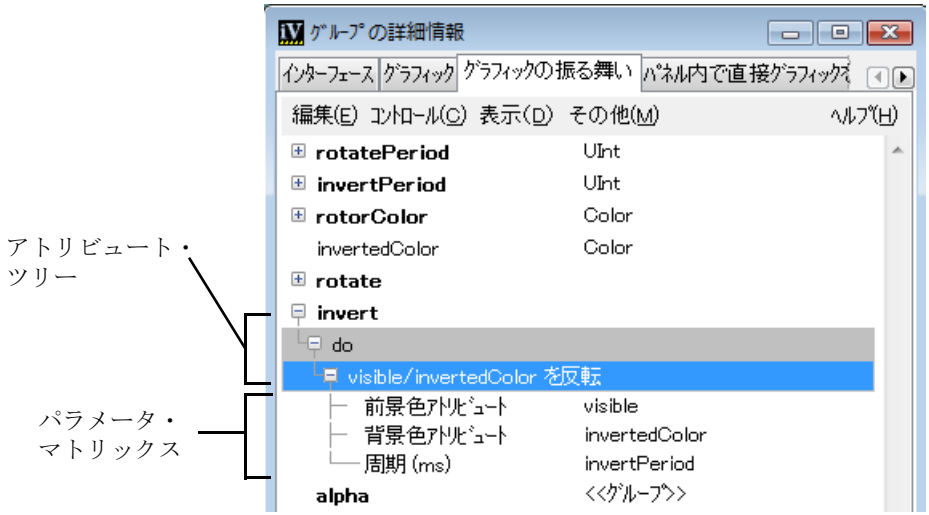
2. その名前およびタイプを、[インターフェース] タブで説明したように設定します。

3. アトリビュートが選択されているとき、1 つ以上の振る舞いをこれに追加することができます。

- 切り取り / コピー / 貼り付けの振る舞い。アトリビュート・ツリーの最初の行を選択してから [編集] > [コピー] あるいは [編集] > [切り取り] を選択して、アトリビュートおよびその振る舞い全体をコピーしたり、切り取ることができます。振る舞いの行を選択して、[編集] > [コピー]、[編集] > [切り取り] を選択して単一の振る舞いをコピーしたり、切り取ることもできます。まずアトリビュートを挿入する行を選択して、[編集] > [貼り付け] を選択すると、アトリビュート・クリップボードの内容を貼り付けることができます。
- 選択したアトリビュート、振る舞い、パラメータを削除する場合は、[編集] > [削除] を選択します。
- 振る舞いを上下に移動します。振る舞いは、上から下への順番でトリガされます。トリガされる順番を決定することができます。

アクセサ・パラメータの設定

クラスによっては、振る舞いを完全に定義するために追加パラメータが必要となります。振る舞いパラメータは、アトリビュート・ツリーの右のマトリックスで編集します。



グループの詳細情報は、コンボ・ボックスあるいはダイアログ・ボックスのパラメータ値を簡単に選択するだけでプロトタイプに複雑な振る舞いを定義できるようになっています。

各マトリックスの行はパラメータに対応しています。

- 左の列はパラメータ値を含んでいます。
- 右の列はパラメータ・レベルを含んでいます。

振る舞いが追加されると、パラメータ・マトリックスはデフォルトあるいは適切な値を入力する必要がある空白の値で初期化されます。

パラメータ値を編集するには、マトリックスの対応するアイテムを 2 回クリックします。これで値アイテムの上に、コンボ・ボックスあるいはテキスト・フィールドのいずれかの編集フィールドが作成されます (図 3.4 を参照)。コンボ・ボックスは、このパラメータに関連する値で初期化されます。

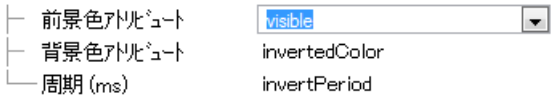


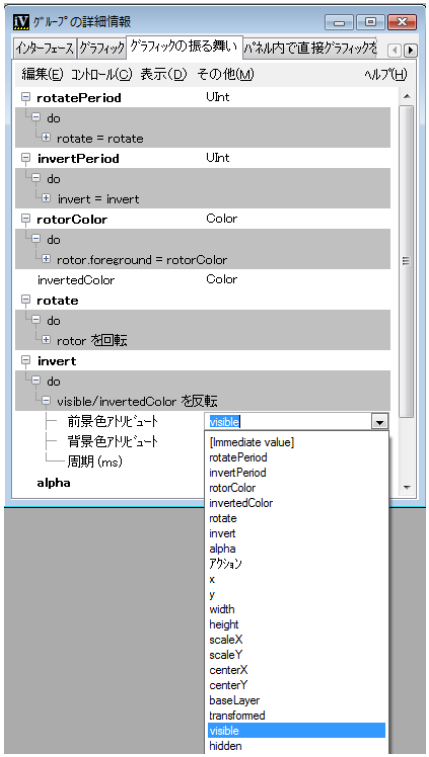
図 3.4 *コンボ・ボックスとデフォルト値の例*

パラメータには、以下の表で示すように 4 つのタイプがあります。

表 3.2 *振る舞いパラメータ*

パラメータ・タイプ	説明	記号
リテラル / 明示	値は明示的に指定されなくてはならない文字列あるいは列挙型タイプです。	(e)
入力	値はアクセサが評価されるときに問い合わせが行われます。これらの値は、定数 (文字列あるいは数値)、他のアトリビュートへの参照、あるいは定数および参照の組み合わせの式となります。	
出力	値は振る舞いが評価されるときに変更されます。(`changeValue` メソッドへの呼び出しが行われます。) したがって、値は、プロトタイプの既存のアトリビュートを参照する名前になります。	
オブジェクト / ノード	パラメータの値は、既存ノードの名前である必要があります。いくつかのアクセサは、パラメータとして特定の種類のオブジェクトのみを受け取ります。たとえば、表示の振る舞いはグラフィック・ノードの上でのみ機能します。	

入力パラメータの場合、編集フィールドはアクセサのツリーならびに 2 つの特殊アイテムをツリーの始めに含むコンボ・ボックスです。



- `[Immediate value]` - このアイテムが選択されていると、編集フィールドはパラメータの即値を編集するために設定されます。値タイプを決定できる場合、値セレクタ (コンボ・ボックスまたはリソース・セレクタ) が作成されます。即値の直接入力もできます。値が数値またはブール型値でない場合、値は二重引用符で囲むことができます (たとえば、色を `"red"` として入力します)。値は式にすることもできます。
- `[All types / Matching types]` - このアイテムは 2 つの値の間をトグルします。`[All types]` は、そのタイプが予想タイプと異なるものも含め、すべてのアクセサを一覧表示します。`[Matching types]` は、その値について、予想タイプと一致するアクセサのみを一覧表示します。通常は、編集フィールドは前のフィールドから取得できる情報を使用して初期化されることが多いため、パラメータ値の編集は上から下へと行う方が効率的です。

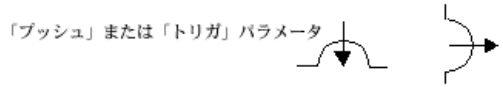
入力パラメータ式には、次を含めることができます。

- 定数：数値あるいは文字列 (引用符の間に置く)
- 変数：プロトタイプ値あるいはノード・アトリビュート
- 算術演算子および括弧：(+, -, *, **, /, %, ==, !=, >, >=, <, <=, &&, ||)

◆ 定義済み関数：abs、acos、asin、atan、ceil、cos、exp、floor、log、rand、rint、round、sin、sqrt、tan。(関数の意味については、標準 C/C++ ライブラリ・ユーザ・マニュアルを参照してください。)

> ***メモ:*** *C/C++ の場合とは異なり、rand は整数引数を取ります。非 0 引数は、乱数を発生させるときにシードとして乱数ジェネレータが使用します。そうでない場合、rand(0) は乱数ジェネレータが最後に初期化されたときに始まる乱数シーケンスの次の整数を返します。*

振る舞いの通知には、単にアトリビュートの値を設定する代わりに、それらの監視アトリビュートに値を伝達するという二次作用があります。この場合、監視アトリビュートの振る舞いは遂次評価されます。そのような振る舞い (トリガ・アクセサ) は出力パラメータの上に外向き矢印を表示し、接続された値は内向き矢印を表示します。



いくつかの振る舞いは、パラメータの可変数を持つことができます。これらのアクセサは、パラメータ列の最後の行、「<< アイテムを追加するためにクリックしてください >>」で識別されます。グループの詳細情報の編集マトリックスでこれらの振舞い用に新しい行を作成するには、最後のパラメータの値フィールドで Enter キーを押すか、または示されているフィールドをクリックします。

対話的な振る舞いの定義

ユーザの振る舞いをグループの詳細情報の [パネル内で直接グラフィックを変更する] タブで追加し、ユーザ・アクションがどのようにプロトタイプのアトリビュートに影響するかを決定します。

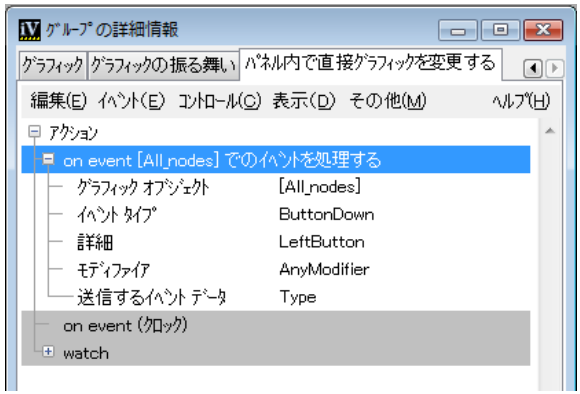


図3.5　グループの詳細情報の [パネル内で直接グラフィックを変更する] タブ

このページは、[グラフィックの振る舞い] タブと同じように機能し、振る舞いを一覧表示します。このページは始めは空白で、ユーザ・アクションによってトリガされる振る舞いがこのページに表示されます。

各アトリビュートに、値が変更される度に実行される指示となる振る舞いを追加することができます。

1. [イベント] メニューのアイテムを選択して、対話的な振る舞いを追加します。
   - ユーザ・アクションがコールバックで呼び出す場合、[イベント] > [コールバック] を選択していくつかのオブジェクト・インタラクタおよびコールバックでトリガされる振る舞いを追加します。
   - それ以外の場合、ボタン・クリックのようなユーザ・イベントを直接処理するには、[イベント] > [イベント] を選択して単純なユーザ・イベントによってトリガされる振る舞いを追加します。
2. 追加された振る舞いそれぞれにパラメータを入力します。
3. トリガするアクセサを追加したら、グラフィック振る舞いに対して行ったのと同じように、[コントロール] からアイテム を追加、あるいは [表示] からアイテムの振る舞いを追加することで、ユーザ・アクションでトリガされる振る舞いを追加することができます。

一般的には、インタラクション・アクセサによってプロトタイプのパブリック・アトリビュートのみを変更し、その変更によって表示される振る舞いのすべてを更新するようにします。

プロトタイプの編集

プロトタイプを作成して保存したら、パレットでプロトタイプを選択し、[描く] > [グループの編集] (Ctrl+E) の順にクリックして選択したプロトタイプを

編集することができます。変更を保存するとき、すべての変更が作成したインスタンスに伝播されます。

たとえば、sample パレットで bulb を選択し、アイコン上でダブルクリックすると、プロトタイプの編集バッファが開き、[グループの詳細情報] ページでインターフェースとグラフィックの振る舞いを編集することができます。

プロトタイプのテスト

プロトタイプに振る舞いを定義したら、プロトタイプ・アトリビュートを変更してこれらをテストすることができます。

1. グループの詳細情報の [インターフェース] タブを選択します。
2. アトリビュート値を設定するには、マトリックス・アイテムをクリックします。
3. 値のタイプに応じて、コンボ・ボックス、リソース・セレクタ、あるいは単純なテキスト・フィールドが作成されます。右マウス・ボタンでシングル・クリックすると、リストあるいはセレクタ (該当する場合) が表示されます。
4. プロトタイプ値は、フィールドの編集後に Enter キーを押したとき、あるいは値の編集後に他のフィールドに移動したときに `changeValue` メソッドの呼び出しを通じて変更されます。

これにより、プロトタイプのアトリビュートを異なる値に設定したとき、プロトタイプのグラフィック表示がどのように変更するかをテストすることができます。

プロトタイプの対話的な振る舞いをテストするには、次の処理を行います。

1. パネルを、アクティブ・モードに切り替えます。
2. プロトタイプのさまざまなアイテムをクリック・アンド・ドラッグして、グループの詳細情報パネルで定義したインタラクションがどのようにアトリビュートに影響しているかを見ることができます。

プロトタイプの保存

プロトタイプを既存のライブラリに保存するには、次の処理を行います。

1. [ファイル] メニューから [名前を付けて保存] を選択します。
2. プロトタイプがライブラリの一部でない場合、プロトタイプをプロパティ・ライブラリに追加するかどうかのメッセージが表示されます。[はい] をクリックします。
3. 表示されているダイアログ・ボックスから、プロトタイプを追加したいライブラリを選択します。
4. 続くダイアログ・ボックスで、プロトタイプの名前を入力します。

プロトタイプ・ライブラリ以外の場所にプロトタイプを保存するには、次の処理を行います。

1. [ファイル] メニューから [名前を付けて保存] コマンドを選択します。
2. 「プロトタイプをライブラリに保存しますか ?」という質問には [いいえ] と答えます。
3. ファイル・セレクタ・ダイアログ・ボックスが表示され、ファイル名を指定するよう指示されます。`.ivp` 拡張子を付けて名前を指定します。

> ***メモ:*** *プロトタイプ・ライブラリに含まれていないプロトタイプはプロトタイプ・パレットには表示されないため、IBM ILOG Views Studio 内部からこのプロトタイプのインスタンスをパネルに作成することはできません。各プロトタイプをそれ独自のファイルに保存するのではなく、プロトタイプ・ライブラリを使用することをお勧めします。*

プロトタイプに更に変更を加えた場合、[ファイル] メニューの [保存] コマンドを使用して、これを再び同じプロトタイプ・ライブラリかファイルに保存することができます。

[名前を付けて保存] コマンドを使うと、プロトタイプを異なるライブラリに保存する、異なる名前を付ける、あるいはライブラリから削除することができます。

プロトタイプが保存されると、そのプロトタイプのインスタンスを含むすべてのパネルは新しいプロトタイプ定義で更新されます。

プロトタイプ・ライブラリの読み込みおよび保存

プロトタイプ・ライブラリを読み込むには、次の処理を行います。

1. [ファイル] メニューから [開く] を選択します。
2. ファイル・タイプのボックスで [プロトタイプ・ライブラリ (`*.ipl`)] を選択します。
3. 読み込むライブラリ・ファイルの名前を参照します。

   ライブラリが読み込まれると、パレット・パネルのプロトタイプ・パレットに新しいパレットとして追加されます。

プロトタイプを作成あるいは編集するたびにプロトタイプ・ライブラリを保存する必要はありません。プロトタイプ・ライブラリは、プロトタイプの編集中に必要に応じて自動的に保存されます。

現在のプロトタイプ・ライブラリの名前 (つまり、プロトタイプ・パレットの表示ページに表示されるライブラリ名) を変更するには、次の処理を行います。

1. [ファイル] メニューから [プロトタイプ・ライブラリを保存] コマンドを使用します。
2. 新しいディレクトリ、およびプロトタイプ・ライブラリ・ファイルに (`.ipl` 拡張子を付けて) 名前を選択します。

   プロトタイプ・ライブラリ名がそれに従って変更され、ライブラリのすべてのプロトタイプが新しいディレクトリに保存されます。

[ファイル] メニューから [プロトタイプ・ライブラリを閉じる] を選択して、不要なプロトタイプ・ライブラリを閉じます。このコマンドは、プロトタイプ・パレットに現在表示されているライブラリを削除します。

> ***メモ:*** *ライブラリに含まれているプロトタイプは実際にはメモリから削除されていません。したがって、パネルあるいは他のプロトタイプから参照することができます。*

パネルでプロトタイプ・インスタンスを作成 / 編集する

このセクションでは、パネルを作成するために定義した、あるいは読み込んだプロトタイプをインスタンス化する方法について説明します。

バッファ・タイプの選択

プロトタイプ・スタジオには、パネルにできる 2 種類のバッファ・ウィンドウがあります。2D Graphics および Grapher です。Gadgets 拡張機能がインストールされている場合は、Gadgets バッファ・ウィンドウも利用できます。

- グラフィックを多用するアプリケーションには 2D Graphics バッファ・ウィンドウを使用します。これは、プロトタイプが線、矩形およびスプラインなどの 2D グラフィック・オブジェクトを含んでいる場合です。
- グラフィック・オブジェクトをグラファー・リンクを使用してプロトタイプ・インスタンスに接続するため、グラファー機能が必要な場合は、Grapher バッファ・ウィンドウを使用します。
- IBM ILOG Views Control パッケージがインストールされており、プロトタイプにガジェットが含まれている場合は、Gadgets バッファ・ウィンドウを使用します。

プロトタイプ・インスタンスを使用するパネルを作成するには、適切なバッファ・タイプを [ファイル] > [新規] から選択します。

プロトタイプ・インスタンスの作成

プロトタイプ・インスタンスを作成するには、次の処理を行います。

1. プロトタイプ・ライブラリをパレット・パネルの上側ペインから選択します。
2. 選択するプロトタイプのアイコンをバッファ・ウィンドウまでドラッグします。

あるいは、

1. プロトタイプを表すアイコンをクリックし、選択します。
2. バッファ・ウィンドウの矩形をクリックしてドラッグします。描画したばかりの矩形で定義されるバウンディング・ボックスのあるプロトタイプのインスタンスが作成されます。

プロトタイプ・インスタンスをカプセル化する `IlvProtoGraphic` オブジェクトの形でプロトタイプがインスタンス化されます。

プロトタイプ・インスタンスの編集

プロトタイプ・インスタンスは、グループの詳細情報を使用して編集します。グループの詳細情報を表示するには、[ツール] メニューから [グループの詳細情報] を選択します、あるいは、プロトタイプ・インスタンスをダブルクリックします。インスタンスを選択しているとき、そのアトリビュートはグループの詳細情報の [インターフェース] タブに表示されます。

> ***メモ:*** *[グラフィックの振る舞い] および [パネル内で直接グラフィックを変更] は、プロトタイプ・インスタンスの場合はオフになっています。これらはプロトタイプを編集しているときのみ使用できます。グループの詳細情報を使用してアクセサ値を編集する方法についての説明は、グラフィックの振る舞いを定義するを参照してください。*

パネルの読み込みと保存

プロトタイプ・インスタンスを含むパネルは、[ファイル] メニューから [開く]、[名前を付けて保存] コマンドを使用して、標準 `.ilv` ファイルとして読み込まれ、保存されます。

プロトタイプ・インスタンスの接続

プロトタイプ は、他のプロトタイプのインスタンスのアトリビュートに接続することのできる通知アトリビュートを定義することができます。通知アトリビュー

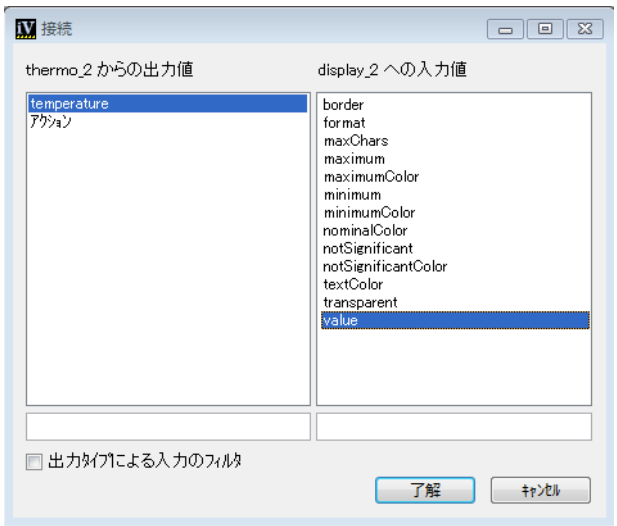
トが変更されると、その値は接続しているオブジェクトのアトリビュートに割り当てられます。

プロトタイプ・インスタンスのアトリビュートを接続するには、次の処理を行います。

1. [編集モード] ツールバーから [グループ接続] モードを選択します。

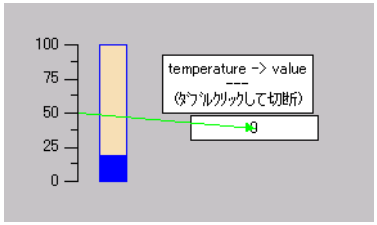
2. 通知アトリビュート (送信する値) を定義するプロトタイプ・インスタンスをクリックします。
3. 接続ラインをアトリビュートを接続させたいインスタンス (アトリビュート変更の通知を受け取るインスタンス) までドラッグします。

   接続の 2 つの値ダイアログ・ボックスが表示されます。



4. 左側ペインで、最初のインスタンスから通知アトリビュートを選択します。
5. 右側ペインで、2 番目のインスタンスから入力アトリビュートを選択します。

同じ 2 つのオブジェクトの間に複数の接続を行うことができます。グループ接続モードがアクティブのとき、既存の接続は緑色の線で表示されます。緑色の線をクリックすると、接続の詳細 (出入力値の名前) が表示されます。



接続を削除するには、次の処理を行います。

1. 接続ラインをダブルクリックします。

   [接続の削除] ダイアログ・ボックスが表示されます。

2. 削除したい接続を選択し、[適用] をクリックします。

次の章では、C++ で定義されているプロトタイプをアプリケーション・オブジェクトへリンクする方法について説明します。

章

4

C++ アプリケーションでプロトタイプを使用する

この章では、C++ アプリケーションでのプロトタイプの使用方法について説明します。このセクションでは、次のトピックを扱います。

- *アーキテクチャ*
- *プロトタイプを使用した C++ アプリケーションのプログラミング*
- *アプリケーション・オブジェクトへプロトタイプをリンクする*
- *プロトタイプの高度な使用*

アーキテクチャ

Prototypes パッケージは、IBM® ILOG® Views Foundation パッケージの上で定義されており、次のタスクを実行できます。

- 基本的グラフィック・オブジェクトをグループとして組み立てます (クラス `IlvGroup`)。
- 定義済みのアクセサ・オブジェクトあるいはスクリプトを使用してグループの振る舞いを指定します。
- プロトタイプを定義し、マネージャでプロトタイプ・インスタンスを作成します。プロトタイプは、`IlvPrototype` と呼ばれる `IlvGroup` のサブクラスです。

- プロパティ・インスタンス間のプロパティを接続します。
- アプリケーション・オブジェクトとプロトタイプ・インスタンスをリンクします。

Prototypes パッケージのアーキテクチャを図 4.1 に示します。

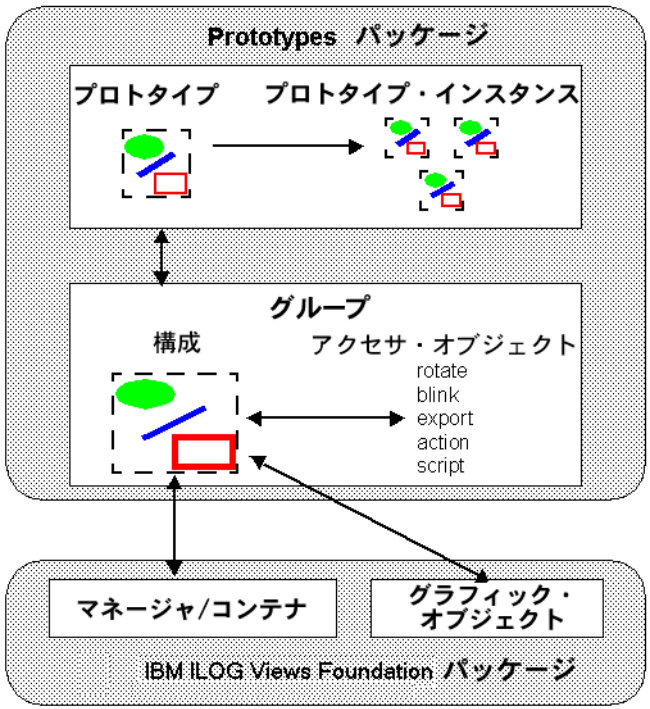


図 4.1 Prototypes パッケージのアーキテクチャ

グループ

グループは Prototypes パッケージの基本コンポーネントです。

Prototypes パッケージで BGO を作成するには、まずグループを作成する基本 IBM ILOG Views グラフィック・オブジェクトを組み合わせる必要があります。グループ内の任意の IBM ILOG Views グラフィック・オブジェクトを使用できます。サブグループを作成して、構造化オブジェクトを構築することもできます。

グループは C++ のクラス `IlvGroup` で表されます。`IlvGroup` オブジェクトは、`IlvGroupNode` クラスの次のサブクラスで表示されるノードの階層を含んでいます。

- IlvGraphicNode は、グラフィック・オブジェクト (IlvGraphic のサブクラスのインスタンス) を保持するノードです。グループは、そのグラフィック要素それぞれに 1 つのグラフィック・ノードを含んでいます。
- IlvSubGroupNode は、サブグループ、つまり、他のグループに含まれるグループを保持します。このクラスは、オブジェクト階層を作成するために使用されます。

> ***注意:*** IlvGroup *オブジェクトと* IlvGraphicSet *オブジェクトは異なります。* IlvGroup *は、マネージャに含まれるグラフィック・オブジェクトの論理階層です。* IlvGraphicSet *とは異なり、* IlvGroup *は* IlvGraphic *のサブクラスではありません。* IlvProtoGraphic *は、* IlvGroup *をカプセル化してマネージャあるいはコンテナに配置するための* IlvGraphic *のサブクラスです。*
>
> IlvValueSourceNode *と呼ばれる* IlvGroupNode *の 3 番目のクラスはパッケージには含まれていますが、現在は使用されていません。*

アトリビュートおよびアクセサ・オブジェクト

Prototypes パッケージを使用すると、オブジェクトのグラフィックの外観だけでなくその振る舞いも定義できます。グループの振る舞いは、そのアトリビュート (プロパティとも呼ばれる) によって制御されます。これらのアトリビュートは、個別名を有しグループの外部インターフェースを表します。つまり、その外観がどのようにアプリケーションから制御されるかを表します。

グループのアトリビュートおよびそれらの振る舞いは、アクセサ・オブジェクトで定義されています。各アクセサ・オブジェクトには名前とタイプがあり、設定の効果を実装したり、グループに値を取得します。いくつかのアクセサ・オブジェクトは、同じ名前を持つことができます。つまりそれらは同じアトリビュートに属するということです。これは、アトリビュート値を設定すると複数の二次作用が生じる可能性があるということです。

アクセサは、オブジェクトの他のアトリビュートあるいはアプリケーション・データにリンクさせることができます。これらは、ユーザ・イベントあるいはアプリケーションの指示に反応してステータスあるいは外観の変更を定義し、拡張機能を使ってオブジェクトのグラフィックおよび対話的な振る舞いを指定します。アクセサ・オブジェクトは、IlvAccessor のサブクラスのインスタンスです。

言い換えれば、アクセサ・オブジェクトと値の関係は次のようになります。

- アトリビュートを通じてグループと対話します。
- グループには、それに付加されているアクセサ・オブジェクトのセットがあります。各アクセサ・オブジェクトは、BGO のアトリビュート (あるいはファセット) を定義する名前に関連付けられています。

- `IlvGroup::changeValue` メソッドは、与えられた名前のアクセサ・オブジェクトすべての `changeValue` メソッドを呼び出し、それによってアトリビュート値を*設定*します。
- `IlvGroup::queryValue` メソッドは、与えられた名前のアクセサ・オブジェクトすべての `queryValue` メソッドを呼び出し、それによってアトリビュート値を*取得*します。
- 各振る舞いクラスの効果は、`changeValue` および `queryValue` メソッドの実装によって定義されています。
- いくつかのアクセサは、ユーザ・インタラクションを通じてあるいはアプリケーションによって設定することができ、それにより他の振る舞いを連鎖的にトリガします。

グループのアクセサ・オブジェクトとアトリビュート / 振る舞いの関係を、図 4.2 に示します。

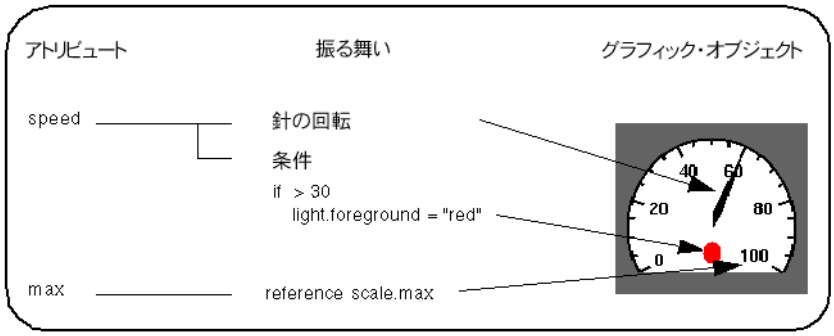


図 4.2 *アクセサ・オブジェクト、アトリビュート、および振る舞いの関係*

この例は、計器を表すグループを示しています。計器には 2 つのアトリビュート、`speed` と `max` があります。

- `speed` アトリビュートは、それぞれグラフィック振る舞いを持つ 2 つのアクセサ・オブジェクトによって実装されます。
  - 回転アクセサ・オブジェクト - `speed` アトリビュートが変更されたとき、このアクセサ・オブジェクトは計器の針を回転させます。
  - 条件アクセサ・オブジェクト - `speed` アトリビュートが変更されたとき、このアクセサ・オブジェクトは、値が 30 を上回る場合に円の色を変更します。
- `max` アトリビュートは、グループ・レベルで基本的グラフィック・オブジェクトのプロパティを参照するリファレンス・アクセサ・オブジェクトによって実装されています。`max` アトリビュートが変更されたとき、このアクセサ・オブジェクトはスケール・グラフィック・オブジェクトの最高速度を変更します。

次のタイプのアクセサを、アトリビュートに付加できます。

- **データ・アクセサ** - データ・アイテムをどのように保存するか (ローカルまたはノード内)、そしてそのタイプは何かを示します。これらは、通常のプログラム言語における変数宣言に相当します。これらのアクセサの 1 つのみが各アトリビュートに示されます。
- **コントロール・アクセサ** - 他のアトリビュートに基づいて条件指示、評価、および代入を行います。これらは、入力パラメータを取り、他のパラメータに対する出力効果があります。典型的な例としては、条件代入の条件アクセサやトグル・アクセサがあります。
- **Notify アクセサ** - 評価サイクルの入力ポイントを定義します。アプリケーション (`pushValue` を行うとき) あるいはユーザ (コールバックがコールバック・アクセサをトリガするとき) のいずれかが値をプッシュし、アクセサにこれらを強制的に処理させます。アトリビュート間の接続は、評価を他の値に伝達するために Watch および Notify アクセサによって行われます。
- **表示アクセサ** - グループのノードの視覚的表示へのアトリビュートの二次作用を定義します。これらは、プログラム言語の描画ライブラリへの呼び出しに相当します。これらが設定されると、ノードの回転あるいはグラフィック・コンポーネントの可視性の変更など、グラフィック・プロパティを変更します。
- **アニメーション・アクセサ** - グラフィック・アトリビュートを定期的に変更する特殊なケースの表示アクセサです。
- **その他のアクセサ** - 前のカテゴリに当てはまらない 2 つのアクセサから構成されます。デバッグ・アクセサおよびプロトタイプに継承アクセサです。

定義済みアクセサ・クラスすべての完全な説明については、*定義済みアクセサ*を参照してください。

アクセサ・パラメータ

アクセサは、任意のアトリビュートが設定されたときに、他のオブジェクトあるいはアトリビュートに対して実行される二次作用を定義します。つまり、プログラム言語の関数と同様に、アクセサはその効果をカスタマイズするためにパラ

メータを取らなくてはなりません。アクセサに設定できる 4 種類のパラメータの説明は、ï 4.1 にあります。

表 4.1 *アクセサ・パラメータ*

パラメータ・タイプ	説明
直接パラメータ	値は明示的に指定されなくてはならない文字列あるいは列挙型タイプです。
入力パラメータ	値はアクセサが評価されるときに問い合わせが行われます。これらの値は、定数 (文字列あるいは数値)、他の値へのリファレンス (ノードのアトリビュートあるいはプロトタイプ値)、あるいは定数およびリファレンスの組み合わせである式になります。
出力パラメータ	値は、アクセサが評価されるときに変更されます (`changeValue` メソッドへの呼び出しが行われます)。したがって、値は、ノードあるいはプロトタイプ値の既存のアトリビュートを参照する名前である必要があります。
オブジェクト / ノード・パラメータ	パラメータの値は、既存ノードの名前である必要があります。いくつかのアクセサは、パラメータとして特定の種類のオブジェクトのみを受け取ります。たとえば、表示アクセサはグラフィック・ノードの上でのみ機能します。

入力パラメータ式には、次を含めることができます。

- **定数**：数値あるいは文字列
- **変数**：プロトタイプ値あるいはノード・アトリビュート
- **算術演算子および括弧**：(+, -, *, **, /, %, ==, !=, >, >=, <, <=, &&, ||)
- **定義済み関数**：abs、acos、asin、atan、ceil、cos、exp、floor、log、rand、rint、round、sin、sqrt、tan。

> ***メモ :*** *C/C++ の場合とは異なり、rand() は整数引数を取ります。この引数が非ゼロの場合、ランダム数が生成される前にランダム生成のシードとして使用されます。そうでない場合、rand(0) は乱数ジェネレータが最後に初期化されたときに始まる乱数シーケンスの次の整数を返します。*

プロトタイプとインスタンス

グラフィックの内容とグループの振る舞いを定義すると、これをプロトタイプとして保存できます。プロトタイプは BGO のモデルです。プロトタイプを直接コーディングで作成することもできますが、通常は、IBM ILOG Views Studio でプロトタイプを作成します。*コーディングによるプロトタイプの作成*を参照してください。

プロトタイプは、クラス `IlvProtoLibrary` によって表されるプロトタイプ・ライブラリを使用して、保管、読み込み、保存することができます。プロトタイプからプロトタイプ・インスタンスを作成できます。プロトタイプ・インスタンスはプロトタイプの完全コピーです。

プロトタイプは `IlvPrototype` クラスによって表され、プロトタイプ・インスタンスは `IlvProtoInstance` クラスで表されます。これらのクラスはどちらも `IlvGroup` のサブクラスです。

プロトタイプ・インスタンスをファイルに保存すると、マネージャはそのインスタンスに対して変更されたプロパティの値のみを書き込みます。プロトタイプ・インスタンスを構成しているグラフィック・オブジェクトは、ファイルには保存されません。したがって、プロトタイプの定義を完全に変更したり、グラフィック・オブジェクトの追加、削除などができます。変更したプロトタイプのインスタンスは、新規の定義で自動的に更新されます。

マネージャとコンテナでグループおよびインスタンスを表示する

グループ、プロトタイプ、およびインスタンスをアプリケーション (`IlvManager` あるいは `IlvContainer`) のパネルに表示するには、これらを `IlvProtoGraphic` オブジェクトに配置し、オブジェクトをマネージャあるいはコンテナに追加する必要があります。`IlvProtoGraphic` は、グループのすべてのグラフィック・オブジェクトをカプセル化するように設計されている `IlvGraphic` のサブクラスです。コンテナあるいはマネージャのプロパティを拡張するクラスであり、便利な関数でグループを直接追加したり取得できるようにする `IlvGroupHolder` を使用して、グループをマネージャに追加することもできます。このクラスは、`IlvGroup` の周囲でそれをラップするために `IlvProtoGraphic` を作成します。

> ***メモ:*** *特別マネージャおよびコンテナ・クラス* `IlvProtoManager`、`IlvProtoContainer`、*および* `IlvProtoGrapher` *は、コンテナあるいはマネージャ内の* `IlvGroup` *オブジェクトの直接処理を可能にするために追加されています。これらのクラスは、互換性の理由から保持されています。これらは、古いクラスですので使用しないでください。*

アトリビュートの接続

グループには読み書き可能アトリビュートがあり、それに付加されたアクセサ・オブジェクトによって定義されています。グループまたは要素の 1 つから生成されるイベントに似た Norify アトリビュートがあることもあります。

Notify アトリビュートは、他のグループのアトリビュートへ接続することができます。アトリビュートが変更されると、変更はこれに接続されているグループに伝達されます。これは、接続アトリビュートへプッシュされる Notify アトリビュートの値として参照されます。

アプリケーション・オブジェクトをプロトタイプへリンクする

プロトタイプおよびパネルを定義したら、これらを C++ で定義されている実際のアプリケーション・データおよびプロセスに接続します。

作成するインターフェースのタイプに応じて、プロトタイプをアプリケーション・オブジェクトにリンクするために使用できるメソッドが 3 つあります。

- 表示にグラフィックが多く、アプリケーション・オブジェクトや値がわずかしかない場合は、直接値を与えて任意のパネルのプロトタイプ・インスタンスにアプリケーション・オブジェクトをリンクします。

  これは一般的には、定義済みのグラフィック・コンポーネントのみで構成される静的な要約表示で使用されます。アプリケーション値をプロトタイプ・インスタンス経由で変更するときに、アプリケーションでユーザ入力を期待しない場合は、値の直接入力が便利です。`<ILVHOME>/samples/protos` の `base_feed` サンプルは、シンプルなコントロール・パネルにこのアプローチを使う方法を示しています。

- WYSIWYG、直接操作アプリケーション・オブジェクト・エディタを構築するために、`IlvGroupMediator` を使用する場合があります。このクラスで、アプリケーション・オブジェクトをパネルの任意の `IlvGroup` (あるいはプロトタイプ・インスタンス) にリンクさせて、このアトリビュートのインタラクティブな編集を可能にすることができます。グループ・メディエータを使うと、アプリケーション・オブジェクトをオブジェクトのエディタとして機能する任意のプロトタイプに動的に結合したり、結合を解除させることができます。

  このタイプの一般的なアプリケーションは、Prototypes Studio のガイドの詳細設定などの WYSIWYG 詳細設定です。`<ILVHOME>/samples/protos` の `inspector` サンプルは、この種のエディタの例です。これは、ビューのビュー・ポイントをインタラクティブに制御する 2D 変換マトリックス・エディタの構築方法を示しています。

- 任意のアプリケーション・オブジェクトにリンクされている各インスタンスで、動的にプロトタイプのインスタンスを多数作成するには、`IlvProtoMediator` を使用します。このクラスはプロトタイプをインスタンス

化して、これを作成されたときに任意のクラスのアプリケーション・オブジェクトにリンクします。これにより、インターフェース・デザインとアプリケーション・デザインが明白に分離されます。これで各々を共通のアプリケーション・インターフェースとは別に開発できます。

このタイプの典型的なアプリケーションでは、パネルを表示させ、多数のクラスの多数のオブジェクトを表してそれを同時に編集できます。各アプリケーション・クラスはプロトタイプにリンクされており、クラスの各インスタンスはプロトタイプのインスタンスにリンクされています。

地図表示およびすべてのグラフ表示は、`IlvProtoMediator` を使用するプロトタイプから利益を受けるアプリケーションの例です。`<ILVHOME>/samples/protos/interact_synoptic` は、このタイプのアプリケーションの例で、フライトと空港の各々がプロトタイプ・インスタンスで表示されているとてもシンプルな航空シミュレータを表しています。シミュレータはフライトのアトリビュートの変更のみを扱い、プロトタイプは最善の表示を行うために描画エディタでインクリメンタルに改善できます。

プロトタイプを使用した C++ アプリケーションのプログラミング

一般的には、IBM® ILOG® Views Studio でプロトタイプを作成し、これらをアプリケーションで使用します。以下のセクションでは、アプリケーションにプロトタイプを追加するために使用する C++ API と、これらのプロトタイプの操作方法を説明します。

> ***メモ:*** *一般的ではありませんが、プロトタイプを直接コーディングで作成することもできます。この場合の詳細については、コーディングによるプロトタイプの作成 を参照してください。*

このセクションでは、次の項目を説明します。

- *ヘッダ・ファイル*
- *プロトタイプ・インスタンスを含むパネルの読み込み*
- *プロトタイプの読み込み*
- *プロトタイプ・インスタンスの作成*
- *プロトタイプ・インスタンスの削除*
- *グループおよびプロトタイプ・インスタンスの取得*
- *アトリビュートの取得と設定*
- *ユーザ定義および定義済みのアトリビュート*

ヘッダ・ファイル

アプリケーションが必要なライブラリ・パッケージにリンクされていることを確認するために、まずグラフィック・オブジェクト、アクセサ (IlvUserAccessor のサブクラス)、および読み込もうとするプロトタイプで使用されるインタラクタに対応するヘッダー・ファイルを含めなくてはなりません。

すべての定義済みアクセサ・クラスを含めるには、ヘッダー・ファイル <ILVHOME>/include/ilviews/protos/allaccs.h を使用します。

次は、あらゆるタイプのグラフィック・オブジェクトを含むプロトタイプを読み込むことのできるアプリケーションを構築するために含めるヘッダー・ファイルの典型的なセットです。

```
#include <ilviews/protos/protogr.h> // for IlvProtoGraphic.
#include <ilviews/protos/allaccs.h>  // for all accessors.
#include <ilviews/graphics/all.h> // for all graphic objects.
#include <ilviews/gadgets/gadgets.h> // if you use gadgets in your prototypes.
#include <ilviews/graphics/inter.h> // for all object interactors.
```

以下のヘッダ・ファイルを追加することもできます。

```
#include <ilviews/protos/groupholder.h> // to get the groups attached
                                        // to a given container or manager.

#include <ilviews/protos/proto.h> // to manipulate prototypes and
                                  // their libraries.

#include <ilviews/protos/grouplin.h> // to attach prototypes to
                                     // application objects.
```

使用するプロトタイプが前もってわかっている場合、allaccs.h、graphics.h、および gadgets.h の代わりに必要なヘッダー・ファイルのみを含めることで、アプリケーションのサイズを小さくすることができます。

プロトタイプ・パッケージを使用するアプリケーションをコンパイルするには、これらをライブラリ ilvproto でコンパイルする必要があります。このライブラリも、以下のライブラリを必要とします。ilvgrapher、ilvmgr、および使用プラットフォーム別の標準 IBM ILOG Views ライブラリ。ilvgdpro ライブラリは、プロトタイプの古い機能を使用するアプリケーションで必要となる場合があります。

プロトタイプ・インスタンスを含むパネルの読み込み

プロトタイプ・インスタンスを含む .ilv ファイルを読み込むには、read、または IlvManager、あるいは IlvContainer の readFile メソッドを使用します。

```
Container->readFile("protoSample.ilv");
```

ファイルで使用されるすべてのファイルは、それらのプロトタイプ・ライブラリから自動的に読み込まれます。プロトタイプ・ライブラリは、表示パスを使用し

てファイル・システム内で検索されます。たとえば、パネルが /usr/home/yourdir/protolibs/mylib.ipl にある mylib と呼ばれるプロトタイプ・ライブラリからのプロトタイプを含む場合、/usr/home/yourdir/protolibs/ を ILVPATH 環境変数に含める必要があります。

コンテナあるいはマネージャ内でグループの処理を可能にするために、IlvGroupHolder クラスがすべての必要なインターフェースを提供します。IlvGroupHolder クラスのインスタンスは、自動的にコンテナあるいはプロトタイプ・インスタンスを含むマネージャに付加されます。このクラスは、グループ (その結果としてプロトタイプ・インスタンスも) の追加、削除、取得のメソッドを提供します。コンテナ、マネージャあるいはグラフィック・ホルダを、次の汎用メソッドで取得することができます。

- `IlvGroupHolder* groupHolder = IlvGroupHolder::Get(manager);`
- `IlvGroupHolder* groupHolder2 = IlvGroupHolder::Get(manager ->getHolder());`
- `IlvGroupHolder* groupHolder3 = IlvGroupHolder::Get(container);`

プロトタイプの読み込み

コーディングによってプロトタイプのインスタンスを作成する場合があります。プロトタイプのインスタンスを作成するには、まずこれらを読む込みます。プロトタイプ・ライブラリ全体を読む込んだ後、それが含む 1 つ以上のプロトタイプを読み込みます。これを行うには、IlvProtoLibrary クラスのインスタンスを作成し、その load メソッドを呼び出します。

```
IlvProtoLibrary* lib = new IlvProtoLibrary(display, "mylib");
if(!lib->load())
    IlvFatalError("Could not load prototype library");
```

表示パスにないプロトタイプ・ライブラリを読み込みたい場合は、コンストラクタを呼び出してライブラリが位置するディレクトリを指定することができます。

```
IlvProtoLibrary* lib = new IlvProtoLibrary(display, "mylib",
                                           "/usr/somewhere/protos");
if(!lib->load())
    IlvFatalError("Could not load prototype library");
```

プロトタイプ・ライブラリを読み込むと、そのすべてのプロトタイプあるいは特殊プロトタイプを次のメソッドで取得することができます。

```
IlUInt count;
IlvPrototype** protos = lib->getPrototypes(count);
```

または

```
IlvPrototype* proto = lib->getPrototype("myproto");
```

getPrototypes メソッドで返された配列は、new[] 演算子で割り当て、必要がなくなったときは delete[] 演算子で削除します。

代わりに、各プロトタイプを個別にグローバル関数 IlvLoadPrototype で読み込むこともできます。

```
IlvPrototype* proto = IlvLoadPrototype("mylib.myproto", display);
```

最初の引数は、プロトタイプ・ライブラリの名前およびプロトタイプの名前 (ピリオドで分けられている) を指定します。2 番目の引数は、アプリケーションが作成した IlvDisplay インスタンスです。プロトタイプ・ライブラリ・ファイルおよびプロトタイプ・ファイルは、表示パスを使用してファイル・システムの内で検索されます。

プロトタイプ・インスタンスの作成

プロトタイプのインスタンスを作成するには、メソッド clone を使用します。

```
IlvPrototypeInstance* instance = proto->clone("myinstance");
```

clone メソッドの引数は、新しいインスタンスの名前です。0 を渡すことができます。これで、名前が自動的に生成されます。

IlvGroupHolder クラスのインスタンスは、自動的にコンテナあるいはプロトタイプ・インスタンスを含むマネージャに付加されます。このクラスは、グループ (その結果としてプロトタイプ・インスタンスも) の追加、削除、取得のメソッドを提供します。

新しいプロトタイプ・インスタンスをマネージャあるいはコンテナに追加するには、マネージャ / コンテナ・クラスに付加されている IlvGroupHolder の addGroup メソッドを使用することができます。

```
IlvGroupHolder* groupHolder = IlvGroupHolder::Get(manager);
groupHolder->addGroup(instance);
```

代わりに IlvProtoGraphic オブジェクトを作成し、直接これをマネージャに配置できます。

```
IlvPrototype* proto;
// Create an instance of the prototype proto and places it
IlvProtoGraphic* protoGraphic1 = new IlvProtoGraphic(proto);
// Create an instance of a prototype
IlvProtoInstance* protoInstance = proto->clone(?instance2?);
IlvProtoGraphic* protoGraphic2 = new IlvProtoGraphic(protoInstance);
manager->addObject(protoGraphic1);
manager->addObject(protoGraphic2);
```

プロトタイプ・インスタンスをマネージャあるいはコンテナに追加したときに、プロトタイプ・インスタンスの位置を設定することがよくあります。この場合は、次の手順に従います。

- IlvProtoGraphic を移動します。

  ```
  manager->moveObject(protoGraphic1, 100, 100).
  ```

- プロトタイプ・インスタンスの x および y アトリビュートを設定します。1 回の呼び出しで複数の値を設定する方法については、*アトリビュートの取得と設定*を参照してください。

プロトタイプ・インスタンスの削除

プロトタイプ・インスタンスをそのコンテナあるいはマネージャから削除するには、IlvGroupHolder クラスの removeGroup メソッドを使用します。

```
groupHolder->removeGroup(instance);
```

埋め込まれた IlvProtoGraphic を、コンテナあるいはマネージャから削除することもできます。

```
manager->removeObject(protoGraphic);
```

プロトタイプ・インスタンスを完全に削除するには、delete 演算子を呼び出します。カプセル化する protoGraphic を削除することもできます。

グループおよびプロトタイプ・インスタンスの取得

マネージャあるいはコンテナに含まれるすべてのグループを取得するには、付加されているグループ・ホルダの getGroups メソッドを使用します。

```
IlUInt count;
IlvGroup** instances = groupHolder->getGroups(count);
```

> ***メモ:*** getGroups *メソッドで返されたポインタの配列は、*new[] *演算子で割り当て、必要がなくなったときは* delete[] *演算子で削除する必要があります。*

グループをその名前で取得するには、getGroup メソッドを使用します。

```
IlvProtoInstance* pump = (IlvProtoInstance*)groupHolder->getGroup("pump");
```

このメソッドは指定したグループが存在しない場合、0 を返します。

アトリビュートの取得と設定

プロトタイプ・インスタンスは、API に基づいて名付けられた同一のアトリビュート (プロパティあるいはアクセサとも呼ばれる) を通じて操作されます。この API は、クラス IlvGraphic から提供されるものと同じで、基本的に

IlvGraphic::changeValue および IlvGraphic::queryValue メソッドから構成されています。

名前の付いているアトリビュートは、IlvValue クラスのインスタンスによって表示され、下記によって定義されています。

- アトリビュート名。たとえば、ボタンのラベルにアクセスする「label」。
- 値、これは異なるタイプのものでも良い (たとえば、文字列、整数あるいはポインタ)。
- データのタイプに対応するタイプ。

値のタイプは、IlvValue クラスによって自動的に設定されます。定義済みタイプの値を初期化するためにコンストラクタを使用します (IlInt、const char*、IlvColor* など)。代入演算子 = を使用するか、定義済みタイプへ IlvValue をキャストさせて、値を変更することもできます。IlvValue クラスは、すべての変換を自動的に処理します。詳細は、IlvValue クラスを参照してください。

プロトタイプ・インスタンスに値を設定するには、IlvValue を作成し、IlvGraphic::changeValue メソッドを呼び出します。

```
IlvValue xval("x", (IlInt)100);
instance->changeValue(xval);
```

> ***メモ:*** *値* 100 *をタイプ* IlInt *に明示的にキャストしなければならないのは、整数とブール型のタイプにあいまいさが存在するためです。キャストしないと、(あるプラットフォーム上で) コンパイラが、タイプ* IlBoolean *の* IlvValue *を作成するコンストラクタを呼び出すことがあります。* IlvValue *の初期化に定数を使用するときは、常に明示的キャストを使用することをお勧めします。*

値を変更するときに毎回新しい IlvValue を作成する必要はありません。既存の IlvValue を使用して、このデータを代入演算子で変更することができます。

```
xval = (IlInt)200;
instance->changeValue(xval);
```

1 回の呼び出しで複数の値を設定することができます。これを行うには、IlvValue オブジェクトの配列を作成して初期化し、changeValues メソッドを呼び出します。次の例は、1 回の呼び出しでオブジェクトの位置を設定する方法を表しています。

```
IlvValue vals[] = {
                IlvValue("x", (IlInt)100),
                IlvValue("y", (IlInt)200)
};
instance->changeValues(vals, 2);
```

値を取得するには、queryValue メソッドを使用します。

```
IlvValue xval("x");
IlInt x = instance->queryValue(xval);
```

queryValue メソッドは、パラメータとして IlvValue リファレンスを取ります。IlvValue は取得する値の名前で初期化する必要があります。queryValue メソッドは取得した値をその引数に保存し、そのリファレンスを返します。例では、queryValue の結果を整数変数 x に代入することで、IlInt キャスト演算子への IlvValue を呼び出します。

1 回の呼び出しで複数の値を取得するには、IlvValue オブジェクトの配列を作成し、queryValues メソッドを呼び出します。

```
IlvValue vals[] = { "x", "y", "width", "height" };
instance->queryValues(vals, 4);
IlInt x = vals[0];
IlInt y = vals[1];
IlUInt width = vals[2];
IlUInt height = vals[3];
```

IlvValue クラスは、必要に応じて値を自動的に変換します。つまり、設定、あるいは取得する値のタイプを正確に知っている必要はありません。たとえば、オブジェクトの位置を、下記のように文字列値を使用して設定することができます。

```
IlvValue xval("x", "100");
instance->changeValue(xval);
```

逆に言えば、値を取得するとき、次のようにして必要なタイプに変換することができます。

```
IlvValue xval("x");
instance->queryValue(xval);
float x = xval;
```

ユーザ定義および定義済みのアトリビュート

プロトタイプおよびそのインスタンスには、3 種類のアトリビュート、ユーザ定義アトリビュート、定義済みアトリビュート、およびサブ・アトリビュートがあります。

ユーザ定義アトリビュート

ユーザ定義アトリビュートは、IBM ILOG Views Studio でデザインしたときにプロトタイプに付加したアクセサによって定義されるアトリビュートです。これらはプロトタイプによって異なります。ユーザ定義アトリビュートの設定あるいは取得の効果は、これを構成するアクセサ・オブジェクトによって決定されます。

たとえば、温度計を表すプロトタイプを作成したとします。temperature アトリビュートを、温度を計器の value アトリビュートにマッピングするリファレンス・アクセサを追加して定義します。プロトタイプのインスタンスによって表示されている温度を変更するには、次のように changeValue メソッドを使用します。

```
IlvValue tempVal("temperature");
tempVal = 22.5;
instance->changeValue(tempVal);
```

定義済みアトリビュート

グループの定義済みアトリビュートにより、すべてのプロパティが持つ位置、サイズ、可視性などの共通プロパティを変更したり取得することができます。

ほとんどの定義済みアトリビュートは、グループがマネージャあるいはコンテナに追加されたときにのみ有効になりますが、それよりも前に設定しておくことができます。これらはグラフィック・ノードに保存されますが、グループが追加されたときにのみ有効になります。

定義済みアトリビュートを、ï 4.2 に一覧表示します。

表 4.2 *プロトタイプおよびプロトタイプ・インスタンスの定義済みアトリビュート*

Attribute	タイプ	説明
layer	IlInt	グループのすべてのノードを任意のレイヤに移動させるためにこのアトリビュートを設定します。このアトリビュートを取得すると、グループのノードが含まれるレイヤを返します。すべてのノードが同じレイヤにある場合、結果は定義されません。
visible	IlBoolean	このアトリビュートを設定し、グループの表示 / 非表示を行います。このアトリビュートの取得は、グループのすべてのグラフィック・ノードが表示されている場合、IlTrue を返し、すべてが非表示の場合は IlFalse を返します。一部のノードが表示され、その他のノードが非表示である場合、結果は定義されません。
x	IlInt	このアトリビュートはグループ・バウンディング・ボックスの左上隅 (マネージャ座標内) の水平座標で、どのようなビュー・トランスフォーマも適用しません。
y	IlInt	このアトリビュートはグループ・バウンディング・ボックスの左上隅 (マネージャ座標内) の垂直座標で、いかなるビュー・トランスフォーマも適用しません。
width	IlUInt	このアトリビュートはグループ・バウンディング・ボックス (マネージャ座標内) の幅で、いかなるビュー・トランスフォーマも適用しません。

表 4.2 プロトタイプおよびプロトタイプ・インスタンスの定義済みアトリビュート (続

Attribute	タイプ	説明
height	IlUInt	このアトリビュートはグループ・バウンディング・ボックス (マネージャ座標内) の高さで、いかなるビュー・トランスフォーマも適用しません。
centerX	IlInt	このアトリビュートはグループ・バウンディング・ボックス (マネージャ座標内) の中心の水平座標で、いかなるビュー・トランスフォーマも適用しません。
centerY	IlInt	このアトリビュートはグループ・バウンディング・ボックス (マネージャ座標内) の中心の垂直座標で、いかなるビュー・トランスフォーマも適用しません。
interactor	const char*	インタラクタをグループのすべてのグラフィック・ノードに関連させるために、このアトリビュートを設定します。アトリビュートの値は、インタラクタ名です (たとえば、"Button")。このアトリビュートを取得すると、グループのグラフィック・ノードに関連しているインタラクタの名前を返します。すべてのノードが同じインタラクタを持つ場合、結果は定義されていません。

サブ・アトリビュート

プロトタイプのサブ・アトリビュートにより、プロトタイプに含まれているオブジェクトのアトリビュートに直接アクセスできます。サブ・アトリビュートの名前は、オブジェクトのパスとアトリビュート名を連結して作成します。サブ・アトリビュート名のコンポーネントは、ピリオドで分けられています。たとえば、プロトタイプが `title` と名付けられた `IlvLabel` を含む場合、そのラベルをアトリビュート名 `title.label` で設定あるいは取得できます。

ï 4.2 に一覧表示されている定義済みのプロパティすべては、特殊グラフィック・ノード用にアクセスすることができます。

アプリケーション・オブジェクトへプロトタイプをリンクする

このセクションでは、プロパティをアプリケーション・オブジェクトにリンクさせるのに使用する 3 つのメソッドについて説明します。

- *値の直接設定*: 単にアプリケーションからビューに値を渡す場合は、これがもっとも簡単な方法です。

- *グループ・メディエータの使用*: これにより、アプリケーションはインターフェースの駆動およびユーザによる値変更の通知を受け取ることができます。
- *プロト・メディエータの使用*: これにより、アプリケーション・クラスをプロトタイプにリンクさせるファクトリ・オブジェクトを構築でき、その結果、動的アプリケーションのインターフェースを自動的に作成します。

値の直接設定

サンプル base_feed (<ILVHOME>/samples/protos ディレクトリに含まれている) は、アプリケーションからインターフェースの駆動方法を示しています。プロトタイプのインスタンスを含むパネルをダウンロードする、あるいはマネージャあるいはコンテナでインスタンスを作成して、編集するインスタンスを取得します。

```
IlvGroupHolder* groupHolder= IlvGroupHolder::Get(manager);
IlvGroup* myThermometer= groupHolder->getGroup("thermometer");
```

次に、その値を IlvGroup::changeValue メソッドで変更します。

```
if (myThermometer)
   myThermometer->changeValue(IlvValue("temperature",(IlUInt) 20)));
```

グループ・メディエータの使用

グループ・メディエータ (クラス IlvGroupMediator) は、アプリケーションのオブジェクトをプロトタイプに接続するために使用され、オブジェクト (詳細設定とも呼ばれる) のインタラクティブ・グラフィック・エディタとして機能します。サンプル inspector および synoptic (<ILVHOME>/samples/protos ディレクトリに含まれる) は、グループ・メディエータを実装し、ベースラインとして使用されます。

次のコード・サンプルは、ユーザ・インターフェースとアプリケーション・コードを明確に分離するアプリケーションの開発方法を示しています。Machine ベース・クラスおよび Boiler 特殊クラスを含むアプリケーションを想定します。

```
class Machine { // The base class of most application objects.
protected:
     list<MachineObserver* > observers;
};
class MachineObserver { // A notification mechanism serving as a
                        // generic communication means between objects.
public:
     void observe(Machine* m) { m->observers.append(this); }
     virtual void notify (Machine*);
};
class Boiler :public Machine { // The class for which you want
                               // to create an object inspector.
public:
// Temperature is an attribute you want the user to have control of.
     void set_temperature(float) { ...
             for each observer in observers
                     observer->notify(this);
```

```
     }
     float get_temperature();
};
```

これらのクラスは、シミュレーション、プロセス・コントロールあるいはコンピュータ上の操作を、あらゆる種類のインタラクティブまたはグラフィックの振る舞いから独立して行います。グループ・メディエータにより、かなり複雑であると推定されるアプリケーション・クラス依存を導入することなく、グラフィカル・ユーザ・インターフェースを Boiler に実装することができます。

このためには、グラフィック表示を扱う IlvGroupMediator のサブクラスおよび Boiler クラスの機械のユーザ・インタラクションを作成します。

```
class BoilerUI :public IlvGroupMediator,  public MachineObserver {
public:
     BoilerUI(IlvGroup* ui, Boiler* b) :IlvGroupMediator(ui, b) {
          MachineObserver::observe(b);
          if (!temperatureSymbol)
            temperatureSymbol=IlvGetSymbol("temperature");
     }
     Boiler* boiler() { return (Boiler*) getObject(); }
     void queryValues(IlvValue* vals, IlUInt) const {
            if (vals[0].getName() == temperatureSymbol))
                 vals[0] = boiler()->get_temperature();
     }
     void changeValues(const IlvValue* vals, IlUInt) {
            if (vals[0].getName() == temperatureSymbol))
                  boiler->set_temperature(vals[0]);
     }
     void notify(Machine*) { update(); }
static IlvSymbol* temperatureSymbol;
};
```

このクラスは、プロトタイプ・インスタンスとアプリケーション・オブジェクト間のブリッジとして機能します。4 つのメソッドを定義します。

- コンストラクタはリンクを確立し、observe(b) ステートメントはアプリケーションに、ボイラに生じた内部変更を通知するように宣言します。
- オブジェクトのアトリビュートにユーザが変更を行ったときに呼び出される changeValue() メソッド。これは、オブジェクトに温度値を更新するように通知します。その他のアトリビュートも処理します。
- プロトタイプが値を更新する必要があるときに呼び出される queryValue() メソッド。オブジェクトの内部値を問い合わせ、ユーザ・インターフェースに伝達します。
- オブジェクトの内部アトリビュートが変更されたときに、この変更をユーザ・インターフェースに反映させるためにアプリケーション内部から明示的に呼び出す必要がある notify() メソッド。Boiler::set_temperature() への呼び出しは、自動的にすべてのオブザーバに通知されます。つまり、notify() メ

ソッドは、明示的に呼び出される必要はありません。このように、Observer/Observable デザイン・パターンを実装しない他のアプリケーションでは、内部コードの他の部分から notify() を呼び出す場合があります。

メディエータ・クラスが定義されると、アプリケーションのオブジェクトをボイラの詳細設定として使用されるプロトタイプ・インスタンスへ動的にリンクさせることができます。

```
IlvGroup* myBoilerInspector = groupHolder->getGroup("BoilerInspector");
BoilerUI* myBoilerUI = new BoilerUI(myBoilerInspector, myBoiler);
```

プロトタイプで詳細設定されているアプリケーション・オブジェクトはいつでも変更できます。

```
myBoilerUI->setObject(myOtherBoiler);
```

この機構は、いくつかのアプリケーション特有のコーディングを必要としますが、それは非常に一般的なものです。どのようなアプリケーション・データ構造もこれを使用するように適合できます。メディエータ・クラスのデザインが終わると、ユーザ・インターフェースおよびアプリケーションは完全に独立したエンティティになります。各々を、別個に開発、保守できます。ユーザ・インターフェースは、IBM ILOG Views Studio およびあらゆるアプリケーション開発環境を使用するアプリケーションを使用して開発されます。

グループ・メディエータは、ユーザ・インターフェースの不必要な更新を防ぐために使用するロック機構も備えています。上の例では、ボイラ set_temperature メソッドは、BoilerUI の notify() メソッドを呼び出してユーザ・インターフェースを更新します。値の変更は、UI から生じるため、この最後の更新は不必要な場合もあります。ロックしたフラグがそのような更新を防ぐかどうかをテストするには、次のようにします。

```
void BoilerUI::changeValues(const IlvValue* vals, IlUInt) {
    if (locked()) return;
    if (vals[0].getName() == temperatureSymbol))
          boiler->set_temperature(vals[0]);
}
```

プロト・メディエータの使用

プロト・メディエータ (クラス IlvProtoMediator) は、IlvGroupMediator のサブクラスで任意のクラスのプロトタイプ・インスタンスを動的に作成し、これらをマネージャあるいはコンテナに配置するために使用されます。目的は、各メイン・アプリケーション・クラス用に特定プロトタイプをデザインすることです。オブジェクトがアプリケーションで作成されると対応するプロトタイプがインスタンス化され、マネージャに配置されます。これにより、アプリケーションの機能的コアからユーザ・インターフェース・デザインを分離させて、アプリケーション全体用のグラフィカル・ユーザ・インターフェースを作成することができます。<ILVHOME>/samples/protos ディレクトリからの次のサンプルは、このデザイン・パターンを実装しています。(航空管制シミュレータを構築するための

interact_synoptic および製造プラント・シミュレータを構築するための synoptic)

たとえば、同じベース・アプリケーションを想定して (Machines および Boilers)、ユーザが同時に表示させ編集する各 Boiler インスタンスを設定するとしましょう。IlvProtoMediator のサブクラスを作成します。

```
class BoilerUI :public IlvProtoMediator, public MachineObserver {
public:
    BoilerUI(IlvManager*m,Boiler*b)
            :IlvProtoMediator(m,"BoilerPrototype",b)
        {
          observe(b);
          IlvSymbol* vals[2] = {
              IlvGetSymbol( "x"), IlvGetSymbol("y") };
          update(vals); // Sets the position of the current instance.
          // The application must have a way of specifying where to place
          // the object.Alternatively, you can handle the placement by
          // explicitly setting the x and y values of the BGO.
          install(m); // Place the prototype in the manager
        }
// Other methods are the same as the BoilerUI using the GroupMediator.
};
```

これで、アプリケーションは、その内部オブジェクトを作成するとすぐにプロトタイプ・インスタンス生成を行うグローバルな「ユーザ・インターフェース・ファクトリ」を持つことができます。このファクトリのコードは、次の擬似コードのようになります。

```
class myApplication {
      list<Boiler*> boilers;
      void initUI (IlvManager* m) {
           for each machine in boilers
               new BoilerUI(m, machine);
      }
      void add_boiler(Boiler* b) {
          boilers.append(b);
          new BoilerUI(getManager(), b);
      }
};
```

プロトタイプの高度な使用

このセクションでは、プロトタイプの使用に関する高度なトピックを扱います。

- *新規アクセサ・クラスの記述*
- *コーディングによるプロトタイプの作成*
- *Prototypes 拡張機能付き IBM ILOG Views Studio のカスタマイズ*

新規アクセサ・クラスの記述

Prototypes パッケージには、プロトタイプで複雑な振る舞いの定義を可能にする多くの定義済みアクセサ・クラスが含まれています。しかし、特殊なニーズのために特殊な振る舞いを実装する場合もあります。このセクションでは、プロトタイプ構築のためにアクセサ・クラスのセットを拡張する方法を説明します。独自の新しいアクセサ・クラスがどのように IBM® ILOG® Views Studio に統合されるかについても説明します。

アクセサのクラスを追加するには、2 つのクラスを記述する必要があります。

- 新規アクセサの効果を定義する `IlvUserAccessor` のサブクラス。
- アクセサが IBM ILOG Views Studio で編集される方法を定義する `IlvAccessorDescriptor` のサブクラス。

IBM ILOG Views 配布ディレクトリの `<ILVHOME>/samples` には、新規アクセサ・クラスの例があります (`gpacc.h` および `gpacc.cpp` ファイル)。詳細については、ディレクトリの `README` ファイルを参照してください。

IlvUserAccessor のサブクラス化

新規アクセサ・クラスを定義するには、`IlvUserAccessor` の直接サブクラスを記述するか、拡張したい機能を実装している既存のサブクラスから派生させます。このクラスを永続化させることもできます。

サブクラスの定義

`IlvUserAccessor` のサブクラスの宣言は、通常、次のようになります。

```
class MyAccessor:public IlvUserAccessor {
public:
    MyAccessor(const char* name,
               const IlvValueTypeClass* type,
               const char* param1,
               const char* param2);
    DeclareUserAccessorInfo();
    DeclareUserAccessorIOConstructors(MyAccessor);
protected:
    IlvSymbol* _param1;
    IlvSymbol* _param1;
    virtual IlBoolean changeValue(IlvAccessorHolder* object,
                                  const IlvValue& val);
    virtual IlvValue& queryValue(const IlvAccessorHolder* object,
                                 IlvValue& val) const;
}
```

次のメソッドは、新規アクセサ・クラスを作成するために再定義します。

- `MyAccessor`

```
MyAccessor(const char* name,
           const IlvValueTypeClass* type,
           const char* param1,
           const char* param2);
```

このコンストラクタは、コードでアクセサのインスタンスを作成するために使用されます。IBM ILOG Views Studio では、入力コンストラクタのみが使用されます。name パラメータは、アクセサによって処理されるアトリビュートの名前を定義し、type パラメータは、アトリビュートのタイプを定義します。コンストラクタには、param1 などの追加パラメータがあります。これらのパラメータは、IBM ILOG Views Studio で入力することができ、実行時に評価されるパラメータに対応する文字列になることがよくあります。

- changeValue

```
virtual IlBoolean changeValue(IlvAccessorHolder* object,
                              const IlvValue& val);
```

アクセサによって扱われるアトリビュートがプロトタイプ上、またはそのインスタンスの 1 つで、changeValue への呼び出しを使って変更されたときに changeValue メソッドが呼び出されます。このメソッドを使用して、アクセサの値変更の効果を定義します。アクセサがパラメータを使用する場合、これらのパラメータを評価しなくてはなりません。これは、直接値あるいは他のアクセサの名前のいずれかを含む文字列を評価する getValue メソッドを使用して行うことができます。

object パラメータは、アクセサが付加されるプロパティあるいはプロパティ・インスタンスです。val パラメータは新しい値を含んでいます。changeValue メソッドは、値の変更に成功した場合 IlTrue を返し、エラーが生じたときは IlFalse を返します (たとえば、パラメータの 1 つが評価できなかった場合)。

- queryValue

```
virtual IlvValue& queryValue(const IlvAccessorHolder* object,
                             IlvValue& val) const;
```

アクセサによって扱われるアトリビュートがプロトタイプ上、またはそのインスタンスの 1 つで、queryValue への呼び出しを使って取得されたときに queryValue メソッドが呼び出されます。このメソッドは、アクセサの ¢ 現在の £ 値をその val パラメータ内に保存する必要があります (それが適切な場合)。アクセサの中には現在の値を保存するものもあれば、保存しないもの (たとえば、条件アクセサはその値を保存しません) もあります。現在の値は、IlvValue の代入演算子を使用して val パラメータに保存されます。メソッドはその val パラメータを返さなくてはなりません。

- initialize

```
virtual void initialize(const IlvAccessorHolder* object);
```

initialize メソッドは、アクセサ・オブジェクトがプロトタイプあるいはプロトタイプ・インスタンスに関連付けられるときに呼び出されます。このメソッドを再定義して、あらゆる種類の初期化を実行することができます。

IlvUserAccessor サブクラスの永続化

グラフィック・オブジェクトのように、アクセサ・オブジェクトは永続的である必要があります。つまり、プロトタイプ定義ファイルに保存され、プロトタイプがロードされるときに読み込まれなくてはなりません。アクセサ・オブジェクトの永続化機構は、グラフィック・オブジェクトに使用する機構ととてもよく似ています。

まず、アクセサ・クラスの .h ファイルで、クラス宣言の public セクションの次のマクロを呼び出します。

```
DeclareUserAccessorInfo();
DeclareUserAccessorIOConstructors(MyAccessor);
```

これは自動的に IBM ILOG Views 実行時タイプ情報をサブクラス用に作成し、永続性を宣言してメソッドをコピーします。

.cpp ファイルで、次のメソッドを記述する必要があります。

- `MyAccessor(IlvDisplay* display, IlvGroupInputFile& f)`
- `MyAccessor::MyAccessor(const MyAccessor& source)`
- `MyAccessor::write(IlvGroupOututFile& f) const`

このコンストラクタは、アクセサ・オブジェクトの記述を入力ストリームから読み込みます。IlvGroupInputFile クラスは、IlvInputFile に類似しています。一般的には、その getStream メソッドのみを使用します。これは、istream オブジェクトへのリファレンスを返し、そこからアクセサ・オブジェクトの記述を読み込むことができます。ただし、便利なメソッド readValue を使用することもできます。writeValue メソッドは、スペースを含む文字列の周りに引用符を付け、readValue メソッドは、これらの引用符をチェックして文字列を正しく読み込みます。これらのメソッドを組み合わせて使用することで、入力 / 出力エラーを回避します。たとえば、メソッドの実装は、下記のようになります。

```
MyAccessor::MyAccessor(IlvDisplay* display, IlvGroupInputFile& f)
: IlvUserAccessor(display, f)
{
        _param1 = f.readValue();
        _param2 = f.readValue();
}
```

次に、プロトタイプがコピーされたとき、あるいはプロトタイプのインスタンスが作成されたときに呼び出されるコピー・コンストラクタを記述します。

```
MyAccessor::MyAccessor(const MyAccessor& source)
: IlvUserAccessor(source)
{
        _param1 = source._param1;
        _param2 = source._param2;
}
```

write メソッドは、アクセサの記述を保存するために再定義します。パラメータの保存に使用するフォーマットは、入力コンストラクタによって定義されているフォーマットと一致させます。

```
MyAccessor::write(IlvGroupOututFile& f) const
{
        IlvUserAccessor::write(f);
        f.writeValue(_param1); f << IlvSpc();
        f.writeValue(_param2); f << endl;
}
```

最後に、次のマクロを .cpp ファイルで呼び出します。

```
IlvPredefinedUserAccessorIOMembers(MyAccessor)
IlvRegisterUserAccessorClass(MyAccessor,IlvUserAccessor);
```

IlvAccessorDescriptor のサブクラス化

IlvUserAccessor のサブクラスを記述後、別のクラス、IlvAccessorDescriptor のサブクラスを記述します。このクラスは、IBM ILOG Views Studio のグループの詳細情報がアクセサ・クラスのパラメータを編集するために必要な情報を提供します。

IlvAccessorDescriptor サブクラスの名前は、IlvUserAccessor のサブクラスの名前と一致させます。たとえば、アクセサ・クラスが MyAccessor である場合、記述子クラスは MyAccessorDescriptorClass と呼ぶ必要があります。

ここでは、アクセサ記述子クラスだけを宣言すればいいようになっています。インスタンスは自動的に作成され、IlvRegisterUserAccessorClass マクロによってユーザ・アクセサ・サブクラスに関連付けられます。

次は、記述子クラスの一般的な例です。

```
class MyAccessorDescriptorClass :
public IlvAccessorDescriptor {
public:
    MyAccessorDescriptorClass()
        : IlvAccessorDescriptor("MyAccessor:an example",
                Miscellaneous,
                "example %s %s...",
                IlFalse,
                &IlvValueIntType,
                0,
                2,
                "Parameter #1", &IlvValueParameterTypeString,
                "Parameter #2", &IlvNodeNameParameterType) {}
};
```

アクセサ記述子クラスは、引数のないコンストラクタを要求するだけです。IlvAccessorDescriptor コンストラクタを呼び出さなくてはなりません。このコンストラクタのパラメータの詳細は、IlvAccessorDescriptor クラスの説明を参照してください。

コーディングによるプロトタイプの作成

プロトタイプは、IBM® ILOG® Views Studio を使用してグラフィカルにデザインするものです。しかし、場合によっては、C++ プログラムからプロトタイプを作成したり、既存のプロトタイプを変更する必要があります。このセクションでは、IBM ILOG Views Studio でデザインするのではなく、C++ のコーディングによるプロトタイプの作成方法について説明します。

新規プロトタイプの作成

プロトタイプは、IlvPrototype クラスのインスタンスで表現されています。新規プロトタイプを作成するには、次のコンストラクタを使用します。

```
IlvPrototype* proto = new IlvPrototype("myPrototype");
```

グラフィック・ノードの追加

最初のステップは、プロトタイプのグラフィック外観を定義することです。これは、グラフィック・オブジェクトを含むノードを追加して行います。これには、IlvGraphicNode クラスのインスタンスを作成し、これらを addNode メソッドでプロトタイプに追加します。

```
IlvLabel* label = new IlvLabel(display, 100, 100, "Hello");
IlvGraphicNode* node = new IlvGraphicNode(label, "label", IlTrue);
proto->addNode(node);
```

IlvGraphicNode コンストラクタには 3 つのパラメータがあります。

- IlvGraphic: プロトタイプに含めるグラフィック・オブジェクト。
- 文字列。ノード名。
- ブール型。ローカル・トランスフォーマをグラフィック・ノードに関連付けるべきかどうかを指定します (詳細は、IlvGraphicNode クラスを参照してください)。

アクセサ・パラメータでプロトタイプのグラフィック・ノードを参照する必要がある場合、それらに異なる名前を付けます。

サブグループの追加

プロトタイプにサブグループを追加して階層オブジェクトを作成することができます。これを行うには、IlvSubGroupNode クラスのインスタンスであるノードを追加します。このサブグループは、グラフィック・ノードを追加して独自に構築した IlvGroup にすることができます。あるいは、他のプロトタイプのインスタンスにすることもできます。

```
// Add a sub-group:
IlvGroup* subgroup = new IlvGroup("subgroup");
IlvLine* line1 = new IlvLine(display, IlvPoint(100, 100),
                                      IlvPoint(200, 200));
subgroup->addNode(new IlvGraphicNode(line1, "line1"));
IlvLine* line2 = new IlvLine(display, IlvPoint(100, 200),
```

```
                                        IlvPoint(200, 100));
subgroup->addNode(new IlvGraphicNode(line2, "line2"));
proto->addNode(new IlvSubGroupNode(subgroup));
// Add a prototype instance as a sub-group:
IlvPrototype* proto = IlvLoadPrototype("samples.pump", display);
IlvProtoInstance* instance = proto->clone();
proto->addNode(new IlvSubGroupNode(instance));
```

アクセサ・オブジェクトの追加

グラフィック・オブジェクトをプロトタイプに追加してプロトタイプを¢描く£と、そのプロパティを定義してプロパティの変更の効果を指定することができます。これを行うには、プロトタイプにアクセサ・オブジェクトを追加します。アクセサ・オブジェクトは、IlvUserAccessor のサブクラスのインスタンスです。

アクセサ・オブジェクトをプロトタイプに追加するには、IlvUserAccessor の適切なサブクラスのインスタンスを作成して、addAccessor メソッドを呼び出します。たとえば次のコードは、プロトタイプに 2 つのアクセサ・オブジェクトを追加します。値を保存する IlvValueAccessor と条件をテストし、その結果に応じてアトリビュートを変更する IlvConditionAccessor。

```
proto->addAccessor(new IlvValueAccessor("v", IlvValueFloatType));
proto->addAccessor(new IlvConditionAccessor("v", IlvValueFloatType,
                       display,
                       IlvConditionAccessor::IlvCondGreaterThan,
                       "100",
                       "label.label",
                       "Greater than 100",
                       "Smaller than 100"));
```

各アクセサ・クラスの完全な説明は、*定義済みアクセサ*および *IBM ILOG Views Prototypes リファレンス・マニュアル*を参照してください。

ライブラリにプロトタイプを追加する

保存して後で再読み込みができるように、プロトタイプは、プロトタイプ・ライブラリに保存しなくてはなりません。

新規のプロトタイプ・ライブラリを作成するには、IlvProtoLibrary クラスを使用します。

```
IlvProtoLibrary* protoLib = new IlvProtoLibrary(display,
                                    "myLib",
                                    "/usr/home/myhome/protos");
```

プロトタイプ・ライブラリは、プロトタイプをファイル・システム・ディレクトリに保存します (前の例の "/usr/home/myhome/protos")。このディレクトリを後で、setPath メソッドを使用して変更することができます。

プロトタイプを新規ライブラリに追加するには、addPrototype メソッドを呼び出します。

```
protoLib->addPrototype(proto);
```

プロトタイプの保存

プロトタイプを保存するには、IlvAbstractProtoLibrary::save メソッドを呼び出します。

```
myLib->save(0, IlTrue);
```

最初のパラメータはオプションの出力ストリームで、ここにライブラリ記述ファイルが保存されます。記述ファイルがそのデフォルトの位置に保存されるように、これを 0 に設定します (前の例の "/usr/home/myhome/protos/myLib.ipl")。2 番目のパラメータは IlTrue に設定して、すべてのプロトタイプが保存されるように指定します。

Prototypes 拡張機能付き IBM ILOG Views Studio のカスタマイズ

このセクションでは、Prototypes 拡張機能付き IBM® ILOG® Views Studio を拡張するために派生させるもっとも重要なクラスについて説明します。

拡張クラス

IBM ILOG Views Studio 拡張機能は IlvStPrototypeExtension クラスで表示され、<ILVHOME>/studio/ivstudio/protos/stproto.h で宣言されています。

```
class ILVSTPRCLASS IlvStPrototypeExtension
  : public IlvStExtension {
public:
    IlvStPrototypeExtension(IlvStudio* editor);
    static IlvStPrototypeExtension* Get(IlvStudio* editor);
};
```

このクラス (あるいはサブクラス) のインスタンスは、IlvStudio オブジェクトが作成された後および initialize メソッドが呼び出される前に作成されなくてはなりません。静的な Get メソッドは、IlvStPrototypeExtension の (固有の) インスタンスを返します。

バッファ・クラス

IBM ILOG Views Studio は、IlvStBuffer の 4 つのサブクラスを定義します。これらのクラスは、<ILVHOME>/studio/ivstudio/protos/stproto.h でも宣言されています。

IlvStPrototypeManagerBuffer

IlvStPrototypeManagerBuffer クラスは、「プロトタイプ・インスタンス (2D)」タイプのバッファを表しています。NewPrototypeManagerBuffer コマンドは、このクラスのインスタンスを作成します。IlvStPrototypeManagerBuffer によって制御されるマネージャは、IlvManager のインスタンスです。

```
class ILVSTPRCLASS IlvStPrototypeManagerBuffer
  : public IlvStBuffer
{
public:
```

```
    IlvStPrototypeManagerBuffer(IlvStudio*,
                                                const char* name,
                                IlvManager* = 0);
};
```

IlvStPrototypeEditionBuffer

IlvStPrototypeEditionBuffer クラスは、「プロトタイプ」タイプのバッファを表しています。つまり、プロトタイプの編集に使用されるバッファです。NewPrototypeEditionBuffer コマンドは、このクラスのインスタンスを作成します。IlvStPrototypeEditionBuffer によって制御されるマネージャは、IlvGadgetManager のインスタンスです。

```
class ILVSTPRCLASS IlvStPrototypeEditionBuffer
  : public IlvStPrototypeManagerBuffer
{
public:
    IlvStPrototypeEditionBuffer(IlvStudio*,
                                                const char* name,
                                IlvManager* = 0);
    void editPrototype(IlvPrototype* prototype,
                       IlBoolean fromLib = IlTrue,
    const char* filename = 0);
    IlvPrototype* getPrototype();
    IlvPrototype* getEditedPrototype();
   };
```

editPrototype メソッドは、prototype によって指定されているプロトタイプを編集できるようにバッファを初期化します。プロトタイプのコピーが作成され、関連するマネージャに保存されます。fromLib 引数は、編集されたプロトタイプがプロトタイプ・パレットに含まれているプロトタイプ・ライブラリに保存されているか、あるいは .ivp ファイルから読み込まれた「スタンドアロン」プロトタイプであるかを指定します。2 番目のケースでは、オプションの filename 引数は、.ivp ファイルのフル・パス名を含むことができます。

getPrototype() メソッドは、バッファに含まれているプロトタイプを返します。getEditedPrototype() メソッドは、バッファが現在ライブラリからのプロトタイプを編集している場合は、「元の」プロトタイプを返します。そうでない場合は、0 を返します。

章

5

定義済みアクセサ

アクセサは、値および BGO (`IlvGroup` あるいは `IlvPrototype`) の振る舞いを定義する基本的な構築ブロックです。アトリビュートは通常、データ・アクセサおよび 1 つ以上のアトリビュートが設定されるときにその二次作用を定義するコントロール・アクセサから構成されます。このセクションでは、プロトタイプ・ライブラリで定義済みのアクセサ・クラスを一覧表示します。次のトピックから構成されています。

- *概要*
- *データ・アクセサ*
- *コントロール・アクセサ*
- *表示アクセサ*
- *アニメーション・アクセサ*
- *トリガ・アクセサ*
- *その他のアクセサ*

概要

各アクセサ・クラスは、1 つ以上のサンプル・プロトタイプで表示されます。ほとんどのサンプルは、IBM® ILOG® Views Views 配布ディレクトリにあるプロトタイプ・ライブラリのいずれかに含まれています。

- `<ILVHOME>/data/ivprotos/libs`
- `<ILVHOME>/samples/protos/*/data/*.ipl` サブディレクトリ

サンプル・プロトタイプを見るには、次の処理を行います。

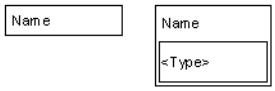
1. Prototype 拡張機能付き IBM ILOG Views Studio を起動します。
2. 対応するプロトタイプ・ライブラリを含む `.ipl` ファイルを開きます。
3. パレットのプロトタイプをダブルクリックします。

プロトタイプの振る舞いをグラフィックに表示する

各振る舞いクラスを表すサンプルでは、プロトタイプのアクセサに定義されるデータ・フローは次のグラフィック用語で表されています。

- 矩形はアクセサを表します (振る舞いの基本的要素)。
- アトリビュートは任意の名前の付いたアクセサのスタックで表されています。スタックでは、アトリビュートの値が変更されたり問い合わせが行われたときに、アクセサが上から下へと評価されます。
- 評価の順番は、そのスタックでのアクセサの相対的位置で表されます。
- はめ込まれた矩形は、任意のアトリビュートのタイプを表すために使用されます。

これらのアイテムを表すグラフィックを次に示します。

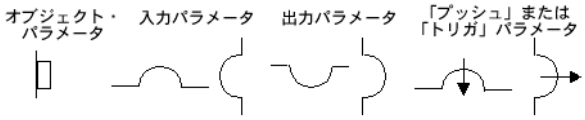


また、

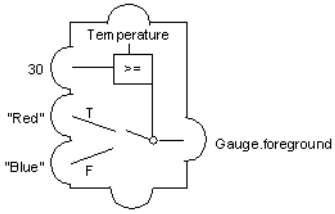
- アクセサの横のスロットは、アクセサのパラメータを表します。
- 丸いスロットは値のパラメータを表します。
- 四角いスロットはオブジェクトのパラメータを表します。
- 上部のスロットは値への入力アクセスを表します。

- 下部のスロットはその出力を表します。
- 左側のスロットはアクセサの入力パラメータを表します (アクセサは評価されるときにその値を問い合わせます)。
- 右側のスロットは出力パラメータを表します (アクセサは値を変更します)。
- 矢印のあるスロットは値が単に設定されるのではなく、プッシュされることを表します。矢印はトリガ・アクセサを表すのに使用されます。

これらのアイテムを表すグラフィックを次に示します。



モデルを完成させるために、リンクあるいは直接値を使って、アクセサ出力を他の入力アトリビュートに接続します。次の図は、条件アクセサとこれらの条件を表しています。温度が 30 度以上になると、計器オブジェクトの前景が赤になります。それ以外の場合は、青になるようにします。



データ・アクセサ

データ・アクセサは、値あるいは値へのポインタを保持します。これらは任意のアトリビュートのタイプを定義します。これは C++ などのプログラミング言語での変数宣言に似ています。すべてのアトリビュートは、これらのアクセサのうちの 1 つのみを含みます。

> ***メモ:*** *一部のアクセサでは、値 (インスタンスの回転) を保持します、つまり、これらを保持する値はその他のデータ・アクセサを必要としません。*

異なるデータ・アクセサについては、次を参照してください。

- *値*
- *リファレンス*

- *グループ化*
- *スクリプト*

値

値アクセサ (クラス `IlvValueAccessor`) により、値を保持するアトリビュートをプロトタイプに付加できます。値が変更されたときは保存されます。値の問い合わせが行なわれると、最後に格納された値が返されます。

パラメータ

- 値のタイプは導出できないため、パラメータではなく値のタイプが指定されなくてはなりません。

例:

`samples` プロトタイプ・ライブラリにある `pump` プロトタイプの `invertedColor` アトリビュートは、色名を一時的変数として保存します。

リファレンス

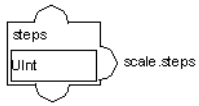
リファレンス・アクセサ (クラス `IlvNodeAccessor`) は、プロトタイプ・レベルでプロトタイプ・ノード (サブ・アトリビュートとも呼ばれます) の 1 つのアトリビュートを参照するために使用されます。対応するアトリビュートが変更されると、新しい値が指定されたサブ・アクセサに転送されます。逆に言えば、アトリビュートの問い合わせが行われるとき、まずノードから問い合わせ、その後でプロトタイプに転送されます。リファレンス・アクセサはプログラム言語の参照 (ポインタ、あるいはエイリアス) と似ています。

パラメータ

- **アクセサ** : ノード・アトリビュートあるいは値を保持するプロトタイプ 値のタイプは、アクセサが何をポイントするかによって決まります。

例

`samples` ライブラリの `thermo` プロトタイプにある `steps` アトリビュートは、直接 `scale` オブジェクトの `steps` アトリビュートをポイントします。アトリビュート `steps` が設定されていると、それは `scale.step` アトリビュートに代入されます。`scale.step` アトリビュートがプログラムによって変更されると、アトリビュートのすべての問い合わせで新しい値を返します。



グループ化

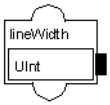
グループ・アクセサ (クラス `IlvGroupUserAccessor`) は、すべてのグループ・ノードで同じ名前の全サブ・アトリビュートを集合的に参照するアトリビュートを定義します。たとえば、`foreground` という名前で `Color` というタイプのこのアクセサを使用して、すべてのプロトタイプ要素の前景色を 1 回の代入で変更することができます。

パラメータ

- パラメータなし。アトリビュートの名前は、このアクセサによって参照されるサブ・アトリビュートを決定するときに使用されます。アクセサのタイプは暗黙的に決定されます。

例

`samples` プロトタイプ・ライブラリの `pump` プロトタイプで、`lineWidth` アトリビュートを追加することができます。アトリビュートはタイプ `UInt` でなくてはなりません。このアトリビュートをグループの詳細情報から変更すると ([インターフェース] タブを使用して)、定義された `lineWidth` を持つすべてのグラフィック・オブジェクトの線の幅を変更します。



スクリプト

スクリプト・アクセサ (`IlvJavaScriptAccessor`) を使うと、IBM ILOG Views Studio に含まれているスクリプト言語インタープリタを使用してプロトタイプの振る舞いをプログラムできます。

スクリプト・アクセサは、スクリプト関数の名前である 2 つのパラメータを持っています。

- `set` 関数は、アクセサの値が変更されるときに呼び出されます。形式は次のようになります。

```
function SetX(obj, newval)
{
   ...
}
```

obj 引数は、アクセサに関連付けられているプロトタイプです。newval 引数は、アトリビュートに割り当てられた新しい値です。

- get 関数は、アクセサの値の問い合わせを行うときに呼び出されます。形式は次のようになります。

```
function GetX(obj)
{
   ...
   return(val);
}
```

obj 引数は、アクセサに関連付けられているプロトタイプです。関数は、アトリビュートの新しい値となる値を返します。

スクリプト・アクセサと関連付けられている関数では、あらゆるプロトタイプ・アトリビュートあるいはプロトタイプ・ノードにアクセスして変更することができます。スクリプト・アクセサの 2 つの関数名のいずれかは、何の関数も呼び出さない none とすることができます。

スクリプト・アクセサに関連付けられている関数は、IBM ILOG Views Script Editor を使用して編集することができます。これらは、同じディレクトリに .ijs サフィックスを持つファイルとして、プロトタイプと同じファイル名で保存されます。そうでない場合は、プロトタイプ・ファイルあるいはそのライブラリ・ファイルに保存されます。

> ***メモ:*** *同じパネルで同じ関数名を持つ複数のプロトタイプ・インスタンスを読み込もうとするとき、命名の競合が起こる場合があります。そのため、すべてのプロトタイプ・スクリプト関数の名前のプレフィックスを、これらが属するプロトタイプ名にするようにします。たとえば、*samples.thermo *プロトタイプで、*Temp *値がスクリプト・アクセサを持つ場合、その関数を、*SamplesThermoTempGet() *および* SamplesThermoTempSet() *と呼びます。*

パラメータ

- **スクリプト関数 (set)**: アトリビュートが変更されたときに実行するスクリプト関数の名前。
- **スクリプト関数 (get)**: アトリビュートの問い合わせが行われたときに実行するスクリプト関数の名前。
- タイプは、set 関数が返す値によって決定され、あるいは get 関数にパラメータとして取得されます。つまり、動的に変更します。

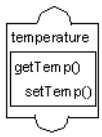
例

次の関数は、条件アクセサに似たアクションを実行するために使用されます。

```
function SetTemperature(obj, temperature)
{
```

```
    if(temperature > obj. threshold) {
         obj.gauge.foreground = "red";
    } else {
         obj.gauge.foreground = "blue";
    }
}

function GetTemperature(obj)
{
    return obj.gauge.foreground;
}
```



コントロール・アクセサ

コントロール・アクセサは、他のアトリビュートが評価されるときにこれらに二次作用を及ぼします。これらはコントロールの構造およびプログラム言語の指示を表しています。グループの詳細情報で、[グラフィックの振る舞い] および [パネル内で直接グラフィックを変更する] タブの各アトリビュートに付加されている「実行」節の下で、これらを見ることができます。

> ***メモ:*** *これらのアクセサは書き込み専用です。最後にテストされた値は記録しません。値にコントロール・アクセサのみを定義する場合、この値を読み戻すことはできません。アクセサに関連付けられている値を保存するには、値アクセサを同じ名前で定義する必要があります。*

異なるコントロール・アクセサについては、次を参照してください。

- *代入*
- *条件*
- *フォーマット*
- *インクリメント*
- *最小値 / 最大値*
- *複数*
- *Notify*
- *スクリプト*

- *スイッチ*
- *トグル*

代入

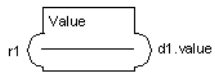
代入アクセサ (クラス `IlvTriggerAccessor`) は、値を他のアトリビュートあるいはサブ・アトリビュートに代入するのに使用されます。アトリビュートが設定されると、`target` パラメータに指定されているターゲット・アトリビュートに指定した値を代入します。

パラメータ

- **アトリビュート** : このアクセサが評価されるときに変更されるアトリビュート。
- **送る** : アトリビュートあるいはアトリビュートに代入される式。
- アクセサのタイプは確定されていないため、ここでは関係ありません。

例

`lcd` ライブラリの `lcd2` プロトタイプは、代入アクセサを使用します。



条件

条件アクセサ (クラス `IlvConditionAccessor`) は、アトリビュートが変更されたときに他のアトリビュートの条件による代入を実行するために使用されます。

最初のパラメータは、アトリビュートの新しい値に適用される条件演算子を定義します。たとえば、アトリビュートの値が `10` に変更された場合、演算子パラメータは `>`、そしてオペランドは 5、テストされた条件は `10 > 5` です。演算子が `[Operand_value]` の場合、テストされた条件は、オペランド・パラメータの値のみです (つまり、`changeValue` にパスされた新しい値は無視されます)。

テストの結果に応じて、アトリビュート・パラメータによって指定されるアトリビュートは、次の 2 つの値のうち 1 つに設定されます。Value if True あるいは Value if False です。オペランド・パラメータ、Value if True あるいは Value if False は、即値 (`1` あるいは `"red"` など)、使用される値の取得のために問い合わせを行う他のアトリビュート、あるいはこれらの即値あるいはアトリビュート名を含む式のいずれかです。

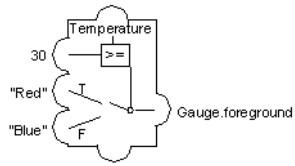
パラメータ

- **演算子** : 条件のテストに使用される演算子。次のいずれかのパレットを使用します。`==`、`!=`、`>=`、`<`、`<=`、または `[Operand_value]`。

- **オペランド**: オペランド値。
- **アトリビュート**: 条件に応じて、プロトタイプ値あるいは true あるいは false に設定されるノード・アトリビュートです。
- **Value if True**: 条件が true の場合、出力が設定される値 (あるいは非 0)。
- **Value if False**: 条件が false の場合、出力が設定される値 (あるいは非 0)。
- アクセサのタイプは確定されていないため、ここでは関係ありません。しかし、オペランド・タイプと互換性がなければいけません。

例

次の例では、thermo プロトタイプ・ライブラリの samples プロトタイプを示しています。temperature アトリビュートが 30 を上回る場合、計器は赤で描画されます。そうでない場合、青で描画されます。



フォーマット

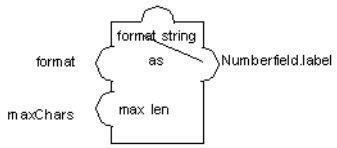
フォーマット・アクセサ (クラス IlvFormatAccessor) は、ユーザ指定のフォーマットを使用して倍精度の数値を文字列に変更するために使用することができます。フォーマットされた文字列は、その後、他のアクセサにコピーされます。値のフォーマットは、C ライブラリ関数 printf に定義されているフォーマット・パラメータによって指定されます。数値は、変換関数に IlDouble として渡されるので、フォーマットは %g 指定子を含んでいる必要があります。

パラメータ

- **フォーマット (printf-style)**: printf C ライブラリ関数に定義されているフォーマット文字列は、String 値である必要があります。このアクセサは倍精度の値を変換するだけなので、この文字列に少なくとも 1 つの %g が含まれている必要があります。
- **Max # of chars**: 変換後の文字列の最大長。この長さを超過する場合、値は * 文字で置き換えられます。整数値でなくてはなりません。
- **アトリビュート**: フォーマット値が代入されるアトリビュート。

例

samples プロトタイプ・ライブラリの display プロトタイプで、フォーマット・アクセサで値が表示される方法を NumberField.label で変更することができます。



インクリメント

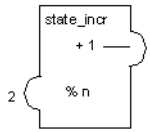
インクリメント・アクセサ (クラス IlvCounterAccessor) は、他のアトリビュートを増加させるために使用されます。このアトリビュートを含むアトリビュートが設定されるたびに、カウンタと呼ばれる別のアトリビュートが指定された最大値に達するまで 1 つずつ増加します。この値に達したとき、カウンタはゼロにリセットされます。

パラメータ

- **最大値** : 最大値 : 増加させる値が最大値と同じになると、0 にリセットされます。
- **アトリビュート** : 増加させるアトリビュート。
- アクセサのタイプは確定されていないため、ここでは関係ありません。

例

3 つのステータスのボタンは、MultiRep アクセサにリンクされているカウンタ・アクセサを使用して実装することができます。次のアクセサは、samples プロトタイプ・ライブラリの symbol プロトタイプに追加されます。グループの詳細情報の [インターフェース] タブで state_incr 値を変更すると、ステータスが増加し、表示が切り替わります。



最小値 / 最大値

最大 / 最小アクセサ (クラス IlvMinMaxAccessor) は条件アクセサに似ていますが、アトリビュートを最小および最大閾値に対してテストする場合の一般ケースを処理します。アトリビュートが変更されると、他のアトリビュートが設定されま

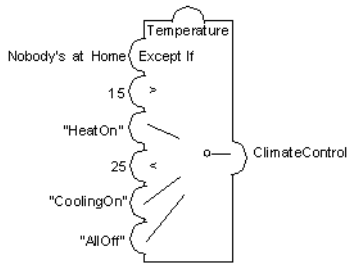
す。代入される値は、現在のアトリビュートが最小値より小さい、最小値と最大値の間、あるいは最大値より大きいかによって異なります。さらに、例外条件を指定することができます。例外条件が true の場合、値は変更されません。

パラメータ

- **最小値** : 最小値を定義します。
- **最大値** : 最大値を定義します。
- **の場合を除く** ：この値が true の場合、値は無視され、出力値あるいはアトリビュートは設定されません。式の結果はブール型値にならなくてはなりません。
- **アトリビュート** : 次の 3 つの値のうち 1 つに設定されるアトリビュート。
- **val < 最小値の時** : 値が最小値より小さい場合、アトリビュートが設定される値が設定されます。
- **最小値 < val < 最大値の時** : 値が最小値と最大値の間の場合、アトリビュートが設定される値が設定されます。
- **最大値 > val の時** : 値が最大値より大きい場合、アトリビュートが設定される値が設定されます。

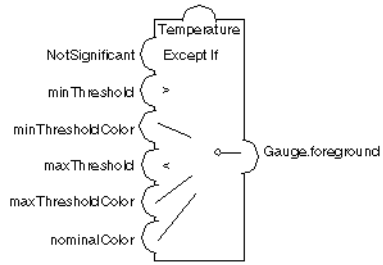
例 1

このアクセサは `Temperature` アトリビュートに付加されます。`Temperature` が設定されると、`Nobody?s at Home` が true である場合、何も行われません。`Temperature` が 15 を下回る場合、`HeatOn` が `ClimateControl` に代入されます。`Temperature` が 25 を上回る場合、`CoolingOn` が `ClimateControl` に代入されます。気温が 15 と 25 の間にある場合、`AllOff` が `ClimateControl` に代入されます。



例 2

この例では、`sample` ライブラリの `vertGauge` プロトタイプを示します。



複数

複数アクセサ (クラス `IlvCompositeAccessor`) は、アトリビュート値を複数の他のアトリビュートあるいはサブ・アトリビュートに割り当てます。たとえば、プロトタイプの単一パブリック値を使用して 2 つのグラフィック・ノードの色を変更するときに使用します。

パラメータ

- このアクセサには、パラメータの変数値があります。これらの各パラメータは、値が代入されるアトリビュートあるいはサブ・アトリビュートです。
- すべてのパラメータには、互換性のあるタイプがあります。

例

この例では、`thermo` プロトタイプの色アクセサを示します。

Notify

Notify アクセサ (クラス `IlvOutputAccessor`) は、付加されているアトリビュートへの `changeValue` 呼び出しを `pushValue` 呼び出しへ返します。特定のアトリビュートを監視する値は、すべての振る舞いを実行します。

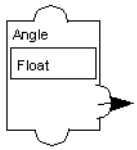
このアクセサは、通知する値に依存する他のアトリビュートの振る舞いをトリガします。たとえば、温度アトリビュートの再評価も行う閾値アトリビュートに変更を加えることができます。これは、Notify アクセサを閾値アトリビュートに、および監視 (閾値) の振る舞いを温度アトリビュートに付加することで行われます。

パラメータ

- パラメータなし。

例

次の例では、samples ライブラリの transformer プロトタイプの X_Scale アトリビュートを示します。



スクリプト

このアクセサについては、*スクリプト* のデータ・アクセサで説明されています。

スイッチ

スイッチ・アクセサ (クラス IlvSwitchAccessor) は、Switch ステートメントを実装しています。

パラメータ

- **Switch**: 整数を返す式。その結果に応じて、アトリビュート 0...N にパラメータの値が代入されます。
- **case 0**: プロトタイプまたは値 "" のアトリビュート。Switch が 0 に評価された場合、このパラメータで指定されたアトリビュートの振る舞いが実行されます。
- **case 1**: Switch が 1 に評価された場合、このパラメータで指定されたアトリビュートの振る舞いが実行されます。
- **...**
- **case N**: Switch が N 以上の値に評価された場合、このパラメータで指定されたアトリビュートの振る舞いが実行されます。

例

信号機を、設定に応じて以下のように実装できます。

```
Value    Integer
do
  Switch Value
  case 0 doRed
  case 1 doOrange
  case 2 doGreen
  case 3 Anomaly
```

```
doRed
do
  greenEllipse.visible=False
  orangeellipse.visible=False
  redEllipse.visible=True
  doBlink=False

doOrange
do
  greenEllipse.visible=False
  orangeellipse.visible=True
  redEllipse.visible=False
  doBlink=False

doGreen
do
  greenEllipse.visible=True
  orangeellipse.visible=False
  redEllipse.visible=False
  doBlink=False

Anomaly
do
  greenEllipse.visible=False
  orangeellipse.visible=True
  redEllipse.visible=False
  doBlink=true

doBlink Boolean
do
  blink orangeEllipse.visible 150
```

トグル

トグル・アクセサ (クラス `IlvToggleAccessor`) は、アトリビュートが設定されるたびに他のアトリビュートを true と false 間で切り替えます。トグル振る舞いを含むアトリビュートに代入される値は無視されます。

パラメータ

- **論理アトリビュート** : この振る舞いが評価されるときに切り替えられるアトリビュート。ブール型でなくてはなりません (たとえば、オブジェクトの可視性アトリビュート)。

例

次の例では、値 `toggle` を持つ `sources` プロトタイプ・ライブラリの `random` プロトタイプを示しています。

表示アクセサ

表示アクセサは、ノードのグラフィック表示を変更します。すべてのアクセサ・ネットワークは、最終的にオブジェクトの外観を変更します。したがって、何らかの表示アクセサを使用します。回転、スケール、変換などの一般表示アクセサは、グラフィック・ノードのサイズおよび位置を変更します。アクセサの 1 つである MultiRep はノードの可視性を制御し、塗りつぶしなどのアクセサはオブジェクト特有のプロパティを制御します。

異なる表示アクセサについては、次を参照してください。

- *塗りつぶし*
- *MultiRep*
- *回転*
- *ScaleX*
- *ScaleY*
- *TranslateX*
- *TranslateY*

塗りつぶし

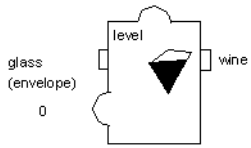
塗りつぶしアクセサは、プロトタイプに含まれている 2 つの多角形オブジェクト、塗りつぶす多角形および塗りつぶし多角形に作用します。アトリビュートの値は塗りつぶしレベルを表します。アトリビュートが変更されると、塗りつぶす多角形のポイントが変更され、指定されたレベルまで多角形を塗りつぶします。角度で、多角形を塗りつぶす方向を指定できます。

パラメータ

- **塗りつぶすグラフィック・ノード**: `IlvPolygon` グラフィック・ノードでなくてはなりません。
- **塗りつぶしグラフィック・ノード**: `IlvPolygon` グラフィック・ノードでなくてはなりません。
- **角度**: 塗りつぶしの角度を示す浮動小数点。

例

以下は、`bottle` プロトタイプで、2 つの多角形、グラスおよびワインです。塗りつぶしアクセサは、`level` プロパティを定義します。塗りつぶす多角形はグラスで、塗りつぶし多角形はワインです。



MultiRep

MultiRep アクセサ (クラス `IlvMultiRepAccessor`) は、プロトタイプの一部の異なる表示間を整数値で切り替えるために使用されます。パラメータは、異なる表示を定義するノードのリストを指定します。値が `n` に変更されると、アクセサはリストの `n` 番目のノードを表示し、他のすべてのノードを非表示にします。

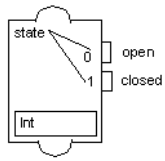
このアクセサは、パラメータの変数を受け入れます。パラメータ編集マトリックスの行で定義したように、多くの表示ステータスがあります。新しい行が最後のパラメータの値を確認すると、自動的に作成されます。

パラメータ

- **グラフィック・ノード**: 値が `0` のときに表示されるノードを定義します。グラフィック・ノードでなくてはなりません。
- **グラフィック・ノード**: 値が `1` のときに表示されるノードを定義します。グラフィック・ノードでなくてはなりません。
- この値のタイプは `Int` (整数) です。

例

`samples` プロトタイプ・ライブラリの `symbol` プロトタイプは、2 つの線で開くと閉じるの 2 ステータスの切り替えを表示するために使用します。



回転

回転アクセサ (クラス `IlvRotationAccessor`) で、オブジェクトの回転角度を任意の値に設定できます。このアクセサで定義される値は、回転が設定される角度 (度数で表示) です。角度は設定されるたびに保存されるため、値をリセットするとオブジェクトを古い角度と新しい角度の間のデルタ値に対応してオブジェクトが回転します。

最小角度、角度範囲、最小値、および値範囲パラメータは、入力値に与えられる新しい回転角度を計算するために使用されます。新しい回転は、次の式を使用して回転アクセサに代入された値から計算されます。

```
angle = minAngle + (value - minimum)* Anglerange / range
```

回転角度の初期値は、最小角度の値とみなされます。そのため回転オブジェクトの初期位置はこの値に対応している必要があります。

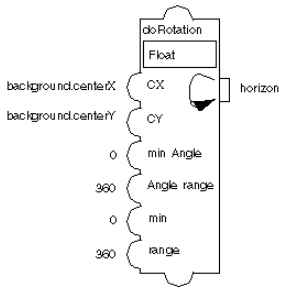
> ***メモ:*** *すべてのグラフィック・オブジェクトが回転に対してセンシティブであるわけではありません。矩形、楕円、およびテキスト・オブジェクトは回転しません。多角形およびスプラインを代わりに使用することをお勧めします。*

パラメータ

- **グラフィック・ノード** : 回転させるノードの名前。グラフィック・ノードでなくてはなりません。
- **中心 X**: 回転中心の X 座標。CenterX アクセサをこのパラメータ (浮動小数点あるいは整数) に使用することができます。
- **中心 Y** : 回転中心の Y 座標。CenterY アクセサをこのパラメータ (浮動小数点あるいは整数) に使用することができます。
- **角度の最小値** : 回転の計算に使用される最小角度 (浮動小数点あるいは整数)。
- **角度の範囲** : 回転の計算に使用される角度の範囲 (浮動小数点あるいは整数)。
- **最小値** : 回転の計算に使用される最小値 (浮動小数点あるいは整数)。
- **範囲** : 回転の計算に使用される値範囲 (浮動小数点あるいは整数)。
- **インタラクタを使用** : ノードをクリックしてアクセサを回転させるときに、イベント・アクセサのように振る舞うべきかどうかを指定するブール型。これが true に設定されている場合、ユーザはノードを回転させることができ、アクセサ値がそれに従って更新されます。
- この値のタイプは浮動小数点 (回転の角度) です。

例

次の例は、`samples` ライブラリの `transformer` プロトタイプに付加された回転アクセサを示しています。



ScaleX

ScaleX アクセサ (クラス `IlvZoomXAccessor`) を使うと、オブジェクトの水平スケーリング係数を設定できます。このアクセサの値が変更されると、オブジェクトは新しい値に基づいて縮小、拡大されます。スケーリング係数は、毎回設定されるたびに保存されるので、スケールを異なる値にリセットすると新旧のスケーリング係数のデルタでオブジェクトを縮小、拡大します。

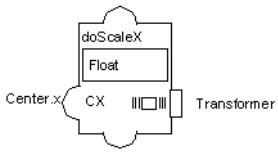
> ***メモ:*** *すべてのグラフィック・オブジェクトがスケーリング係数に対してセンシティブであるわけではありません。たとえば、テキスト・オブジェクトは縮小、拡大できません。*

パラメータ

- **グラフィック・ノード**: 縮小、拡大するグラフィック・ノードの名前。グラフィック・ノードでなくてはなりません。
- **中心 X**: スケールの中心の X 座標 (浮動小数点あるいは整数)。
- この値のタイプは浮動小数点です。

例

この例では、トランスフォーマ・オブジェクトに付加されたスケール・アクセサを示しています。このアクセサを使用しているフル・プロトタイプは、`samples` ライブラリの `transformer` プロトタイプです。



ScaleY

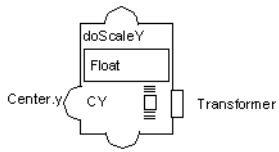
ScaleY アクセサ (クラス `IlvZoomYAccessor`) を使うと、オブジェクトの垂直スケーリング係数を設定できます。この値が変更されると、オブジェクトは新しい値に基づいて縮小、拡大されます。スケールは、毎回設定されるたびに保存されるので、スケーリング係数を異なる値にリセットすると新旧のスケーリング係数のデルタでオブジェクトがリサイズされます。

パラメータ

- **グラフィック・ノード**: 縮小、拡大するグラフィック・ノードの名前。グラフィック・ノードでなくてはなりません。
- **中心 Y**: スケールの中心の Y 座標。
- この値のタイプは浮動小数点です。

例

この例では、トランスフォーマ・オブジェクトに付加されたスケール・アクセサを示しています。このアクセサを使用しているフル・プロトタイプは、`samples` ライブラリの `transformer` プロトタイプです。



TranslateX

TranslateX アクセサ (クラス `IlvSlideXAccessor`) は、ノードを水平に、最小位置、位置範囲、最小値、および値範囲で決定される位置に移動させます。新しい位置は、次の式を使用して TranslateX に代入された値から計算されます。

```
x = xmin + (v - minimum) * xrange / range
```

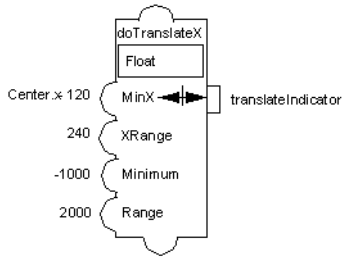
パラメータ

- **グラフィック・ノード**: 移動させるノードの名前。グラフィック・ノードでなくてはなりません。
- **最小値 X**: 最小位置の名前 (浮動小数点あるいは整数)。
- **X の範囲**: 位置範囲の名前 (浮動小数点あるいは整数)。
- **最小値**: 最小値の名前 (浮動小数点あるいは整数)。
- **範囲**: 値範囲の名前 (浮動小数点あるいは整数)。

- **インタラクタを使用** : ノードをクリックしてアクセサを回転させるときに、イベント・アクセサのように振る舞うべきかどうかを指定するブール型。これが true に設定されている場合、ユーザはノードを回転させることができ、アクセサ値がそれに従って更新されます。
- この値のタイプは浮動小数点です。

例

Translate アクセサの使用はスケール・アクセサと似ていますが、前者はオブジェクトのサイズではなく位置を変更します。`samples` ライブラリの `transformer` プロトタイプを参照してください。



TranslateY

TranslateY アクセサ (クラス `IlvSlideYAccessor`) は、ノードを垂直に、最小位置、位置範囲、最小値、および値範囲で決定される位置に移動させます。新しい位置は、次の式を使用して TranslateY に代入された値から計算されます。

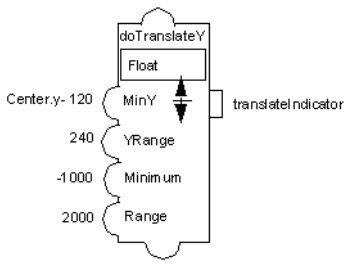
```
y = ymin + (v - minimum) * yrange / range
```

パラメータ

- **グラフィック・ノード** : 移動させるノードの名前。グラフィック・ノードでなくてはなりません。
- **Y の最小値** : 最小位置 (浮動小数点あるいは整数)。
- **Y の範囲** : 位置範囲 (浮動小数点あるいは整数)。
- **最小値** : 最小値 (浮動小数点あるいは整数)。
- **範囲** : 値範囲 (浮動小数点あるいは整数)。
- **インタラクタを使用** : ノードをクリックしてアクセサを回転させるときに、イベント・アクセサのように振る舞うべきかどうかを指定するブール型。これが true に設定されている場合、ユーザはノードを回転させることができ、アクセサ値がそれに従って更新されます。
- この値のタイプは浮動小数点 (移動の距離) です。

例

TranslateX および TranslateY の使用は、scaleX および scaleY と似ています。



アニメーション・アクセサ

アニメーション・アクセサ (クラス `IlvAnimationAccessor`) は、オブジェクトの外観を定期的に変更する表示アクセサのカテゴリです。アニメーション・アクセサは、アニメーションがオンであるかどうかを示すブール型の値を保持します。

効率上の理由から、アニメーション・アクセサではタイマのカウントごとにアトリビュートの再評価は行いません。そのため、アクセサのアトリビュートの 1 つを変更する場合、値をそれ自体に再度代入し、代入アクセサなどを使用してパラメータを強制的に更新させなくてはなりません。`samples` ライブラリの `pump` プロトタイプを参照してください。

異なるアニメーション・アクセサについては、次を参照してください。

- *点滅*
- *反転*
- *回転*

点滅

点滅アクセサ (クラス `IlvBlinkAccessor`) は、プロトタイプのオブジェクトを点滅させます。つまり、定期的な間隔でオブジェクトの表示と非表示を切り替えます。アトリビュートが `IlTrue` に設定されているとき、オブジェクトが点滅を始めます。アトリビュートが `IlFalse` に設定されると、点滅は止まります。

パラメータ

- **論理アトリビュート** : オブジェクトの可視性を制御するオブジェクト・アトリビュート。

- **周期 (ms)**: 2 回*点滅*する間の間隔をミリ秒で表したもの (浮動小数点あるいは整数)。
- この値のタイプはブール型です。

例

次の例では、点滅値付きの sources ライブラリの file プロトタイプを示しています。

反転

反転アクセサ (クラス IlvInvertAccessor) は、プロトタイプの要素の色を定期的に反転します。プロパティが IlTrue に設定されると、色反転が開始されます。アトリビュートが IlFalse に設定されると、色反転が止まります。

色が前景色および背景色として指定されている間、プロトタイプに定義されているすべての色、あるいはそのノードの色を使用することができます。

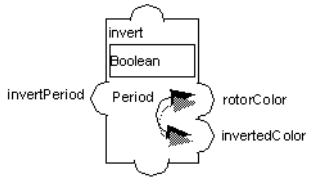
パラメータ

- **前景色アトリビュート** : ノード・アトリビュートあるいは値を保持するプロトタイプ。
- **背景色アトリビュート** : 背景色を保持するノード・アトリビュートあるいはプロトタイプ値。
- **周期 (ms)**: オブジェクト色の 2 つの反転間の間隔をミリ秒で表したもの (浮動小数点あるいは整数)。
- Type : ブール型 (アクセサが値を交換しているかどうか)。

例

この例は、samples プロトタイプ・ライブラリの pump プロトタイプを示しています。invert が true に設定されているとき、rotorColor および invertedColor が定期的に交換されます。この周期は、invert アトリビュートによって定義されます。

> ***メモ:*** invertPeriod *値には、代入の振る舞いがあります。*invert = invert *周期が変更されたときにアクセサで強制的に再評価されるようにし、内部タイマでその周期を更新します。*



回転

回転アクセサ (クラス `IlvRotateAccessor`) は、オブジェクトを定期的に回転させる振る舞いが `IlTrue` に設定されているとき、この振る舞いを定義します。

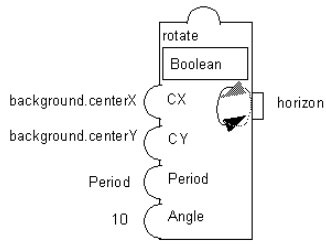
角度パラメータは度数を指定し、この数値でタイマが動くたびにオブジェクトを回転させます。中心 X および中心 Y パラメータは、回転の中心を定義します。これらのパラメータについては、回転ノードの中心は使用できません。オブジェクト回転中に発生する切り上げ問題により、わずかにこれが移動する場合があるからです。代わりに、プロトタイプの他の固定オブジェクトの中心を使用してください。この参照オブジェクトは、必要に応じて非表示にすることができます。

パラメータ

- **グラフィック・ノード**: 回転させるノードの名前。グラフィック・ノードあるいはサブグループ・ノードになります。
- **角度**: 各ステップでオブジェクトが回転する角度を度数で表したもの (浮動小数点あるいは整数)。
- **中心 X**: 回転中心の X 座標。CenterX アクセサをこのパラメータ (浮動小数点あるいは整数) に使用することができます。
- **中心 Y** : 回転中心の Y 座標。CenterY アクセサをこのパラメータ (浮動小数点あるいは整数) に使用することができます。
- **周期 (ms)**: オブジェクトが回転する間隔をミリ秒で表示したもの。整数値になります。

例

この例は、`samples` プロトタイプ・ライブラリの `pump` プロトタイプを示しています。回転アクセサが true に設定されていると、ノードが 10ms ごとに 20 度回転します。



トリガ・アクセサ

トリガ・アクセサは、アクセサのグラフで評価シーケンスの入力ポイントを定義します。トリガは、ユーザ・イベント (コールバックおよびイベント)、アプリケーション (pushValue メソッドを使用して) によるノードの変更、あるいは他のいくつかのノードの変更 (トリガおよび接続の組み合わせ) に反応できるアクセサです。

異なるトリガ・アクセサについては、次を参照してください。

- *コールバック*
- *Clock*
- *Watch*
- *イベント*

コールバック

このアクセサ (クラス IlvCallbackAccessor) は、所定のコールバックが指定したグラフィック・ノード上でのユーザ・アクションによって呼び出されるときに設定されるトリガを付加します。呼び出されるコールバックのノードは IlvGadget あるいは IlvGraphic でなくてはならず、これにインタラクタが付加されます。

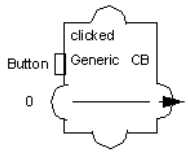
パラメータ

- **グラフィック・ノード** : そのコールバックがトリガされるグラフィック・ノードの名前。
- **コールバック名** : コールバックの名前。
- **入力** : コールバックがトリガされるときに送信される値。

例

次の例では、値 clicked を持つ sources ライブラリの random プロトタイプを示しています。clicked 値は、ユーザがボタンを押したときに 0 を、その出力にプッ

シュします。この出力は、toggle 値に接続されており、これで running 値を切り替えます。



Clock

Clock アクセサ (クラス IlvAnimationAccessor) は、アトリビュートを定期的にトリガし、付加されている振る舞いを実行します。0 に設定されていると、アクセサには何の振る舞いもありません。他の値に設定されていると、この値は振る舞いを定期的にトリガする内部タイマの周期として使用されます。

パラメータ

- この値のタイプは UInt です。非ゼロの場合、アトリビュートは何の効果もありません。それ以外の場合は、その値をタイマ周期として解釈します。

Watch

Watch アクセサ (クラス IlvLoopbackAccessor) は、付加されているアトリビュートに他の Notify アトリビュートを監視させます。

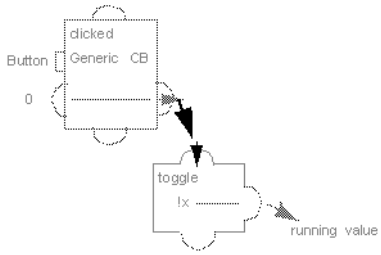
このアクセサ・クラスは、コールバックがトリガされるときにコールバック・アクセサとともに、プロトタイプの値を変更するためによく使用されます。Watch アクセサは、コールバックを含んだトリガするアトリビュートを変更の必要がある監視アトリビュートに接続します。

パラメータ

- **Notify アトリビュート** : 監視されるアトリビュート。このアトリビュートは、Notify あるいはコールバック・アクセサを持つアトリビュートの 1 つでなくてはなりません。

例

クリックされた Watch アクセサは、ユーザが clicked 値に付加されているボタンを押したときに、running アトリビュートを切り替える toggle 値に clicked 値をリンクします。



イベント

イベント・アクセサ (クラス `IlvEventAccessor`) は、ユーザあるいはアプリケーション・イベントに反応して振る舞いをトリガさせるために使用されます。マウス・ポインタがプロトタイプのノード上にある間に所定タイプのイベントが生じたとき、アクセサが付加されているアトリビュートが評価されます、つまり、そのすべての振る舞いが設定されます。

コールバックがその値をプロトタイプの他のアトリビュートにプッシュする間、イベントは、その付加されているアトリビュートに通知します。このイベントは、コールバックを付加した後で、監視アクセサをそれ自体を付加させることに似ています。

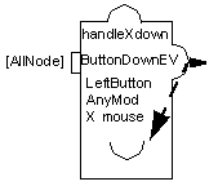
パラメータ

- **グラフィック・ノード** : グラフィック・ノードの名前。入力デバイスから受け取ったイベントは、振る舞いをトリガするためにこのグラフィックを経由してアクセサに送信されます。特別な値 `[All Nodes]` は、この値が、所定タイプのあらゆるイベントがプロトタイプのグラフィック・ノードのいずれかに達したときにトリガされることを示しています。
- **イベント・タイプ** : アクセサをトリガするイベント・タイプ。タイプには、標準 IBM ILOG Views イベント・タイプ、`AnyEvent`、`KeyUp`、`KeyDown`、`ButtonDown`、`ButtonUp`、`EnterWindow`, `LeaveWindow`, `PointerMoved`、`ButtonDragged`、`Repaint`、`ModifyWindow`、`Visibility`、`MapWindow`, `UnMapWindow`, `Reparent`、`KeyboardFocusIn`、`KeyboardFocusOut`、`DestroyWindow`、`ClientMessage`、および `DoubleClick` のいずれかを使用できます。
- **詳細** : イベントの詳細。このパラメータは、イベントの追加フィルタリングを示しており、イベント・タイプによって異なります。たとえば、`ButtonDown` イベントの場合、詳細は次のようになります。`AnyButton`、`LeftButton`、`RightButton`、`MiddleButton`、`Button4`、あるいは `Button5` です。`KeyDown` イベントの場合、詳細パラメータは、キー、あるいはアクセサをトリガする `AnyKey` を示しています。有効なキーのリストについては、`IlvEvent` クラスを参照してください。

- **モディファイア** : どのモディファイアを押すかを示します。次の値を設定できます。`AnyModifier`、`NoModifier`、`Shift`、`Ctrl`、`Meta`、`Alt`、`Num`、`Lock`、`Alt+G` あるいは `Shift+Crtl`、`Ctrl+Shift+Alt` などの以前のモディファイアの組み合わせ。
- **送信するイベント・データ** : 現在の値にプッシュされるイベント・アトリビュート。タイプ、詳細、X (ウインドウに対するマウスの水平位置)、Y (ウインドウに対するマウスの垂直位置)、GlobalX、GlobalY (画面に対するマウスの位置) などになります。

例

次の例では、`samples` ライブラリの `EventScaleY` 値を持つ `transformer` プロトタイプを示します。



その他のアクセサ

これらのアクセサは、現在の既存のカテゴリには適しません。

その他のアクセサについては、次のセクションに説明があります。

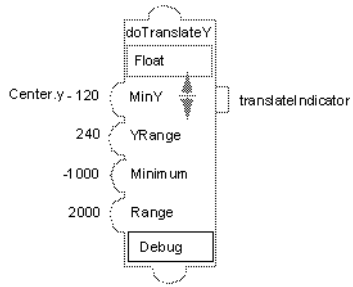
- *デバッグ*
- *プロトタイプ*

デバッグ

デバッグ・アクセサ (クラス `IlvDebugAccessor`) は、プロトタイプのデバッグを行うために使用されます。対応する値が変更されたり、問い合わせが行われた時に、コンソールや出力ウインドウへメッセージを印刷します。

例

次の doTranslateY アクセサの問い合わせが行われた、あるいは変更されたとき、メッセージが出力コンソールに印刷され、現在の値を表示します。



プロトタイプ

プロトタイプ・アクセサ (クラス `IlvPrototypeAccessor`) により、既存のプロトタイプのすべてのアクセサから新しいプロトタイプを継承します。新しいプロトタイプは、既存のプロトタイプのすべてのアクセサがこれに追加されたかのように振る舞います。これは、複雑な振る舞いのライブラリを構築するとき、およびこれらを他のプロトタイプで再利用するときに便利です。プロトタイプを含むプロトタイプ・ライブラリは、あらゆるインスタンスがこのアクセサを使用して適切に機能できるように、開かれていなくてはなりません。IBM ILOG Views Studio のグループの詳細情報から、[アトリビュート] タブを選択し、さらに [編集] > [継承] を選択すると、プロトタイプ・アクセサをプロトタイプに追加できます。

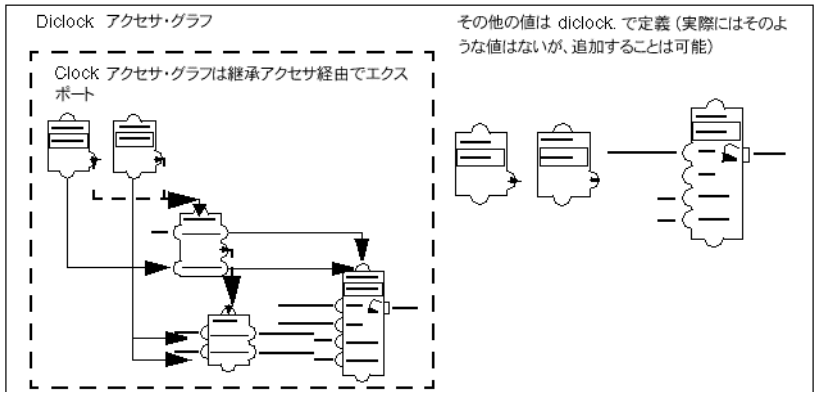
パラメータ

- **プロトタイプの名前** : そのアクセサを継承するプロトタイプの名前。

例

このアクセサは、現在のアクセサ・グラフのコンテキストにエクスポートされたすべての値を表示するサブグラフを表しています。

`sources` ライブラリの `diclock` プロトタイプは、`clock` プロトタイプのアクセサすべてをカプセル化してエクスポートします。`clock` プロトタイプとまったく同じように振る舞いますが、グラフィック表示は異なります。


Diclock アクセサ・グラフ
Clock アクセサ・グラフは継承アクセサ経由でエクスポート
その他の値は diclock. で定義 (実際にはそのような値はないが、追加することは可能)

索引