優れたGUIを構築する

Javaレイアウト・マネージャーによるGUI設計・開発プロセスの改善

Joe Winchester
Software developer
IBM

Renee Schwartz (rsch@us.ibm.com)
Software designer
IBM

2001年 10月 01日

率直に言って、ソフトウェアの設計者と開発者の見解が一致することはめったにありません。この現象が最も顕著に現れるのはGUI (グラフィカル・ユーザー・インターフェース) の開発プロセスです。そこでは設計者主導になりますが、開発者が自分のビジョンを実現するために本当に必要とする情報を設計者が提供してくれることはあまりありません。この記事で、執筆者のRenee SchwartzとJoe Winchesterは、Javaレイアウト・マネージャーを使用してGUIの設計プロセスと開発プロセスを一致させる方法を説明し、チーム内のこの両サイドが共同作業するための共通のフレームワークを提示しています。この手法を使えば、彼らの間の摩擦を緩和できるだけでなく、よりよい総合的GUI設計を行い、より堅固な最終製品を作り出すことができます。

アプリケーションのGUIを作成するプロセスでは、開発者と設計者が緊密に協力し合って所定の目標を達成しなければなりません。最もよく編成されたプロダクション・チームの場合でも、共有する用語がないために、このプロセスがしばしば阻害され、そのあげくは、誤解が生じ、単純な作業が繰り返されてお粗末な結果になります。

この記事では、GUI開発プロセスにJavaレイアウト・マネージャーを取り入れることで、設計者と開発者がどのようにして利益を享受できるかということについて説明します。Swingクラス・ライブラリーには、非常に簡単な`FlowLayout` マネージャーから、複雑で柔軟な`GridBagLayout` マネージャーまでのいくつかのレイアウト・マネージャーが含まれています。経験上、レイアウト・マネージャーを利用することで、設計プロセスと開発プロセスを一致させることができるということが分かっています。それは、各レイアウト・マネージャーが固有な定義済みの設計モデル提供し、開発チームはそれらを簡単にインプリメントできるからです。

これから、サンプルGUIの検討に入ります。ここでは、GUIの最も簡単な形式 (つまり、GUIの作成にレイアウト・マネージャーを使用しない) から、最も複雑な形式 (つまり、拡張レイアウト・マネージャーを使用して制御機構 (control) や列を操作したり、GUIウィンドウ内の余白の割り振

りを操作する) までを取り上げます。読者がこの記事を読み終えるころには、レイアウト・マネージャーが、GUI設計だけでなく、GUI開発プロセスに対してもプラスの影響を与えることができるということを十分に理解できているはずです。

設計・開発プロセス

一般的なGUI設計・開発プロセスの場合、設計者はGUIの各画面を表す一連のイメージを作成します。通常これらのイメージは、ホワイトボードや用紙に描かれた後、PhotoshopやVisual Basicなどのペイント・ツールでモデル化されます。設計者は、満足のいく結果が得られるまで、インターフェース内の制御機構のフォント、カラー、レイアウトなどを検討します。次に、このモデル化されたGUIは仕様に組み込まれて、希望のGUIをインプリメントする開発者または開発チームに渡されます。

多くの場合、設計者のモデルは開発者や開発チームにとって多くの問題を抱えていますが、その問題の大半は、間違った想定が原因になっています。それではまず、普通に発生するこれらの間違った想定を検討し、次に、Javaレイアウト・マネージャーがこうした問題の解決にいかに役立つかを見ていきましょう。

欠陥設計の原因になるよくある想定

まず、設計者は、ボタンやラベルなどの制御機構に含まれているストリング・リテラルのサイズには一貫性があるものと想定しているかもしれません。しかし、アプリケーションがさまざまな国のユーザーに展開されることを考えた場合、画面に表示するストリングはそれぞれの国の言語に変換されることになります。よく起こることですが、変換されたたストリングの長さが設計者が指定した長さと違ったりすると、長い方のストリングは制御機構によって切り取られ、短い方のストリングは周囲に余白が入ります。

2番目に、設計者は、ウィンドウ全体のサイズを制御することを想定しているかもしれません。ユーザーはウィンドウ・サイズを変更できたらと思うことがしばしばあります。つまり、ウィンドウ・サイズを拡大して使用領域を広げたり、ウィンドウ・サイズを縮小して一度に表示できるアプリケーション・ウィンドウの数を増やしたりできればよいと考えます。いずれの場合も、GUI制御機構の位置とサイズは、新しいスペースを最適に使用できるように調整する必要があります。ウィンドウの拡大に合わせて制御ボタン (固定長のリテラル・ストリングを収めている) も大きくしたいと思うユーザーはあまりいないと思われますが、リストやテーブルなどの要素を拡大して行数を増やしたり、列の幅を広くしたいと思っているユーザーはいるかもしれません。

3番目に、設計者は、プロトタイプ・マシン上の制御機構の外観をすべてのエンド・ユーザー・マシンに広げることを想定しているかもしれません。Java言語の約束事の1つは、クロス・プラットフォーム互換性です。これを実現するために、Java Foundation Class (JFC) は、異なるプラットフォームやオペレーティング・システムで実行できるGUI制御機構のセットを提供しています。ボタンや入力ボックスなどの基本制御機構の場合、開発者はAbstract Window Toolkit (AWT) クラス・ライブラリーを利用することができます。AWTを利用すれば、Java言語で各システムのネイティブ (またはヘビーウェイト) ・ウィジェットを使用できますが、基本的なユーザー・インターフェースしか使用できません。

ツリー・リスト、テーブル、ツールバー、グラフィックス付きボタン、その他の高機能制御機構などを使用するアプリケーションの場合、開発者は、JFCやSwingクラス・ライブラリーの使用を

考えることができます。Swingは、移植性を実現するために、キャンバスを作成し、低レベルのペイントおよびマウスAPI呼び出しを使って各制御機構を実際に作成します。ネイティブ・ウィジェットは使用されないため、Swingはライトウェイト またはエミュレート型ウィジェット・ツールキットと呼ばれます。各制御機構の実際の描画は、ルック・アンド・フィール と呼ばれるオブジェクトが作成されるまで据え置かれます。このルック・アンド・フィールは、ネイティブ制御機構の外観と特性をできる限りエミュレートしようとするので、ユーザーは、ネイティブ・アプリケーションからJavaアプリケーションに切り替える場合に、ほとんど違いを感じることはありません。

ルック・アンド・フィールが異なると、制御機構の位置や描画もかなり異なるため、Windowsのルック・アンド・フィールにおけるGUIの外観と、MotifやMacintoshのルック・アンド・フィールで実行されるGUIの外観は非常に異なったものになります。これを考慮せずに設計・開発したGUIは、Windowsプロトタイプではすばらしいもの見えるかもしれませんが、LinuxやMacintoshなどの別のオペレーティング・システムで実行した場合は、かなり劣って見える可能性があります。

これらの取り違えの解決策

最初の3つの想定を回避するには、サイズ変更できないウィンドウと固定したルック・アンド・フィールを持つ単一言語のアプリケーションを作ることです。しかし、この表示設定はまったく別の問題を生じさせます。この設定は、アプリケーションを展開するマシンとプロトタイプ・マシンでは、異なる構成になる可能性があります。たとえば、フォントやフォント・サイズは、ユーザーのアクセシビリティに関した好みに合わせてユーザー側でシステム・ワイドに設定することができます。この場合、画面に表示されるストリングのサイズだけでなく、入力を受け入れる制御機構 (入力ボックスなど) も、設計時のものとは異なってきます。

制御機構やウィンドウ、フォントなどはシステムごとに異なるため、GUIを作成する場合は、これらの可変要因を考慮に入れておく必要があり、また変更が生じた場合は適切に調整する必要があります。Webアプリケーションの設計者や開発者が処理しなければならない最も重要な概念は、流動性です。静的モデルに基づいてJava GUIを設計するのは自滅的です。むしろ我々は、制御機構の位置とサイズが多くの可変要因によって決定されるフレームワークの中で作業しなければなりません。

以下のセクションでは、1つのウィンドウといくつかの制御機構からなる簡単なGUIを取り上げます。まず、レイアウト・マネージャーを使用しないGUI設計を検討します。

レイアウト・マネージャーを使わない場合の制御機構の操作

GUIは、いくつかの制御機構を内部に含む最上位のウィンドウからなっています。各制御機構は、`java.awt.Component` のサブクラスであるJavaクラスのインスタンスです。制御機構の例としては、入力ボックス、ラベル、ボタン、リストなどがあります。最外部の制御機構は、タイトル・バーと「最小化」および「最大化」ボタンからなる、`java.awt.Window` のサブクラスです。すべての制御機構の位置付けは、そのx位置、y位置、幅、高さが含まれた長方形を割り当てることで行われます。各制御機構は、`Window` そのものを除き、親の制御機構を持っています。

図1は、この記事で使用するGUIの最初の形式を示しています。

図1. 簡単なGUI

図1に示されたウィンドウを作成するコードをリスト1に示します。1行目は、最外部のウィンドウに使用するクラス`Frame`の作成を示しています。次の行は、このフレームのレイアウト・マネージャーをヌルに設定しています。レイアウト・マネージャーを使用していないので、制御機構は自分自身でフレーム上に位置付けしなければなりません。次に、2つのボタンが作成され、フレームに追加されて、位置と寸法が割り当てられます。この位置は、xとyが、それぞれ10,30、70,30です。x座標は左から右へ、y座標は上から下へ進みます。

リスト1. 2つのボタンを持つ簡単なGUI

```
Frame frame = new Frame();
frame.setLayout(null);
frame.setBounds(100,100,150,70);
Button button1 = new Button("Next");
frame.add(button1);
button1.setBounds(10,30,50,25);
Button button2 = new Button("Previous");
frame.add(button2);
button2.setBounds(70,30,50,25);
frame.setVisible(true);
```

固定座標の問題

絶対座標に基づいて制御機構を位置付ける (絶対位置付け と呼ばれる) コードの問題は、ラベル内のストリングが別の言語に変換されたり、ユーザーがウィンドウのサイズを変更したりする可能性があることです。いずれの場合も、ボタンは元の位置に固定されています。その結果が図2に示されています。図2は、GUIが英語からフランス語に変換された場合や、ウィンドウのサイズが変更された場合にどういう結果になるかを示しています。

図2. 簡単なGUIで固定座標を使用した場合の結果

この問題を解決するためには、最も簡単なレイアウト・マネージャーであるフロー・レイアウト・マネージャーを利用します。

簡単なレイアウト・マネージャー

ウィンドウなどの制御機構は、内部に子制御機構を持つことができる、`java.awt.Container`のサブクラスです。すべてのコンテナーはレイアウト・マネージャーを備えており、各レイアウト・マネージャーは自分のコンテナーの制御機構を位置付けます。最も簡単なレイアウト・マネージャー`FlowLayout`は、制御機構を左から右へ配置します。

これを実行するコードがリスト2に示されています。リスト2では、フレームのレイアウトを`FlowLayout`のインスタンスになるように設定し、2つのボタンをフレームに追加しています。2

つのボタンの境界を設定してその位置とサイズを決定する行が削除されていることに注意してください。レイアウト・マネージャーを使用する場合は、境界設定による制御機構の個別位置付けが行われなくなりました。すべての子制御機構の位置とサイズを計算するのは、レイアウト・マネージャーの仕事です。

リスト2. フロー・レイアウト・マネージャーを使った場合の簡単なGUI

```
Frame frame = new Frame();
frame.setLayout(new FlowLayout());
frame.setBounds(100,100,160,70);
Button button1 = new Button("Suivante");
frame.add(button1);
Button button2 = new Button("Precedente");
frame.add(button2);
frame.setVisible(true);
```

図3に示されているように、フロー・レイアウト・マネージャーは、2つのボタンを空きスペースの中心に位置付けます。それぞれのボタンには、ラベル用の十分なスペースが割り振られ、両側に適切な量の空白が設けられています。

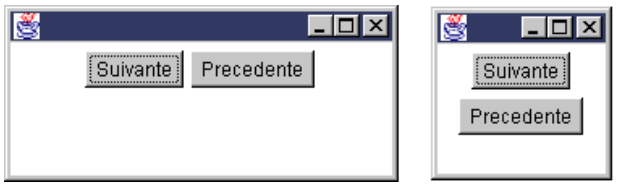
図3. FlowLayoutによる左から右への制御機構の配置

リスト2には、ボタンのサイズを設定するコードはありません。このため読者は、フロー・レイアウト・マネージャーがどのようにして各ボタンの適切な幅を計算しているのか、不思議に思われるかもしれません。この計算は、`Component` クラスの`getPreferredSize()` メソッドを使って行われます。推奨サイズは、幅と高さを含む`Dimension` のインスタンスです。`Button` クラスを作成して、そのボタンのラベルと、フォントの寸法を調べ、ラベル全体を表すために必要な幅を計算します。ラベルが拡大すると推奨サイズも変わるため、レイアウト・マネージャーは新しい推奨サイズを使用します。図4は、英語からフランス語に変換されたGUIを示したもので、ボタンのサイズが、新しいストリングを収容できるように拡大されています。

図4. FlowLayoutによるボタンの推奨サイズの調整

各制御機構に推奨サイズを問い合わせた後、レイアウト・マネージャーは、空きスペースを調べて、各制御機構の位置を設定します。制御機構は、互いに重なり合わないように位置付けられます。ウィンドウが拡大されると、`FlowLayout` は制御機構を空きスペースの中心に位置付けし直します。制御機構を横並びに配置する余裕がない場合は、レイアウト・マネージャーは、図5に示すように、制御機構を上下に配置します。

図5. FlowLayoutによる制御機構の上下の配置



GUIウィンドウの動的サイジング

上のリスト1とリスト2では、ウィンドウの実際のサイズが、幅160、高さ70になるようにコード化されています。これには問題があります。というのは、制御機構そのもののサイズと位置は、それ自体に含まれているテキストに基づいて、拡大したり縮小したりするからです。制御機構がウィンドウに比べて大きくなり過ぎると切り取られ、小さくなり過ぎると周囲に余白が入ります。その結果が図6に示されています。図6では、ストリングがウィンドウに比べて大きくなり過ぎています。

図6. 静的なウィンドウ・サイズを使用した結果

`Window` コンポーネントは、この問題を解決するための`pack()` メソッドを提供します。ウィンドウに`pack()` が送られると、ウィンドウは、フレーム内のすべての制御機構を表示するために十分なスペースを確保できるように、自分自身のサイズを設定します。リスト3は、`pack()` メソッドを書き込んだ簡単なGUIを示しています。

リスト3. pack() メソッドを使った簡単なGUI

```
Frame frame = new Frame();
frame.setLayout(new FlowLayout());
Button button1 = new Button("Advance Forward");
frame.add(button1);
Button button2 = new Button(" Revert to Previous");
frame.add(button2);
frame.pack();
frame.setVisible(true);
```

`pack()` を使用した場合、`Window` は、制御機構が切り取られることがないように、自分自身のサイズを設定します。

メソッドを使用した推奨サイズの計算

上の例から分かるように、GUI画面を設計するときに固定サイズを指定するのは危険です。また、制御機構の位置をx位置とy位置で参照したり、幅と高さで参照したりしてはなりません。各制御機構には、推奨サイズを動的に計算するメソッドが含まれており、これらのメソッドは、GUIの流動性が最大になるように設定する必要があります。表1は、制御機構のサイズを計算するためのいくつかのメソッド例を示しています。

表1. 制御機構のサイズを計算するためのメソッド

制御機構	説明	メソッド

ボタン	現在のラベル・テキストを示す	setLabel(String) またはsetText(String)
テキスト	文字数を示す	setColumns(int)
ラベル	ラベルのテキストを示す	setLabel(String) またはsetText(String)
入力域	文字の行と列の数を示す	setRows(int) とsetColumns(int)
リスト	行数を示す	追加した行の数に基づいて計算される

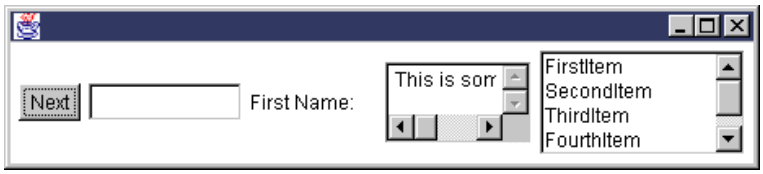
リスト4は、簡単なGUIにおける上記各メソッドの構造を示しています。

リスト4. 制御機構の推奨サイズを計算するためのメソッドを備えた簡単なGUI

```
Frame frame = new Frame();
frame.setLayout(new FlowLayout());
Button button1 = new Button("Next");
frame.add(button1);
TextField text1 = new TextField();
text1.setColumns(10);
frame.add(text1);
Label label1 = new Label("First Name:");
frame.add(label1);
TextArea textArea1 = new TextArea("This is some text in a text area");
textArea1.setRows(2);
textArea1.setColumns(10);
frame.add(textArea1);
List list1 = new List();
list1.add("FirstItem");
list1.add("SecondItem");
list1.add("ThirdItem");
list1.add("FourthItem");
list1.add("FifthItem");
frame.add(list1);
frame.pack();
frame.setVisible(true);
```

図7は、GUIに対する上記メソッドの影響を示しています。

図7. 推奨サイズ設定メソッドを使用した結果



拡張レイアウト・マネージャー

ここまでは、GUIの中で最も簡単なものだけを扱ってきました。レイアウト・マネージャーを使用しなくてもGUIを作成できる方法を紹介しました。この方法の場合、各制御機構は、`setBounds(Rectangle)` メソッドを介して、固定された位置とサイズを受け取ります。また、最も簡単なレイアウト・マネージャー`FlowLayout` の使用方法も紹介しました。この方法では、`getPreferredSize()` メソッドと個々の制御メソッドを展開して、各制御機構のサイズと位置を設定します。`FlowLayout` の場合は、コンポーネントを配置する順序のみを制御します。もっと細かく制御するには、制約レイアウト・マネージャー という機能を使用しなければなりません。

制約レイアウト・マネージャーとは、制約オブジェクトが関連付けられているレイアウト・マネージャーのことです。コンポーネントをコンテナーに追加すると、必ず制約オブジェクトと関

連付けられるので、これをレイアウト・ヒントと考えることができます。制約レイアウト・マネージャーの例としては、制約オブジェクト`GridBagConstraints`を持つ`GridBagLayout`があります。`GridBagLayout`を使用すれば、画面を一連の論理行と論理列に分割することができます。

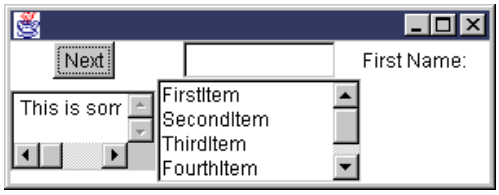
リスト5は、`FlowLayout`の例で使用した5つの各制御機構に`GridBagLayout`を追加する方法を示しています。`GridBagConstraints`オブジェクトは、各コンポーネントをフレームに追加する`add(Component,Object)`メソッドの2番目の引数です。

リスト5. 推奨サイズを計算するための制御メソッド

```
Frame frame = new Frame();
frame.setLayout(new GridBagLayout());
GridBagConstraints constraints = new GridBagConstraints();
Button button1 = new Button("Next");
frame.add(button1 , constraints);
TextField text1 = new TextField();
text1.setColumns(10);
constraints.gridx = 1;
frame.add(text1 , constraints);
Label label1 = new Label("First Name:");
constraints.gridx = 2;
frame.add(label1 , constraints );
TextArea textArea1 = new TextArea("This is some text in a text area");
textArea1.setRows(2);
textArea1.setColumns(10);
constarints.gridx = 0;
constraints.gridy = 1;
frame.add(textArea1 , constraints );
List list1 = new List();
list1.add("FirstItem");
list1.add("SecondItem");
list1.add("ThirdItem");
list1.add("FourthItem");
list1.add("FifthItem");
constraints.gridx = 1;
frame.add(list1 , constraints );
frame.pack();
frame.setVisible(true);
List list1 = new List();
list1.add("FirstItem");
list1.add("SecondItem");
list1.add("ThirdItem");
list1.add("FourthItem");
list1.add("FifthItem");
constraints.gridx = 1;
frame.add(list1 , constraints );
frame.pack();
frame.setVisible(true);
```

図8は、GUI画面上で`GridBagLayout`を使用して制御機構を列に分離した結果を示したものです。

図8. 制御機構にGridBagLayoutを使用した結果



制御機構の配置と列のサイジング

「Next」ボタンは、gridxが0、gridyが0の左上のセルに配置されています。入力フィールドのgridx制約が1に設定されているため、入力フィールドはこのボタンの横の列に配置されます。gridxが2のラベルは、次 (つまり3番目) の列に配置されます。

次に、入力域が0列に配置されていて、この列にはボタンがあることが分かります。`GridBagLayout` は、各列で最も大きな制御機構の推奨サイズを使用してその列のサイズを設定します。こうすれば、幅が足りない列や、高さが足りない行に制御機構を入れても、切り取られることはありません。

ボタンと入力域の場合は、どちらも同じ列に入り、2つのうち、入力域がその大きい方になります。したがって、列は入力域が収まるようにサイズ変更されます。つまり、ボタンの周囲に余白が生じることになります。上の例では、小さい方の制御機構を列の中心に配置することでこれを処理しています。

ボタンを列の左側に配置すれば、もっと望ましい効果が得られるかもしれません。同様に、入力ボックスは、列の左または右の壁に固定することができます。

制御機構を列の壁に固定すれば、列の拡大に伴って制御機構も拡大され、ユーザーはそのスペースを使用できるようになります。制御機構を固定するには、`GridBagConstraints` オブジェクトのanchorフィールドを設定します。この値は、制御機構を固定したい列の端の方位点です。たとえば、ボタンを左側に固定したい場合は、次のようにします。

```
constraints.anchor = GridBagConstraints.WEST;
```

このコマンドは、制御機構の左端を列の左端に固定します。固定された制御機構と同じ行内のすべての制御機構の推奨高がその制御機構より大きい場合は、デフォルトにより、その制御機構は垂直方向の中心にも位置付けられます。制御機構を特定の隅に固定したい場合は、次の例のように、方位点の組み合わせを使用することができます。

```
constraints.anchor = GridBagConstraints.NORTHWEST
```

制御機構を左に固定し、空きスペースをすべて埋め込むには、次のようにfillプロパティーを使用します。

```
constraints.fill = GridBagConstraints.HORIZONTAL;
```

水平、垂直の両方向に埋め込みたい場合は、`Both` 値を使用する必要があります。

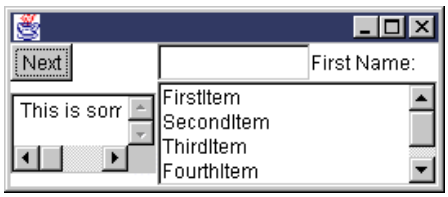
```
constraints.fill = GridBagConstraints.Both
```

同じGUI例を使ってもうひとつやってみたいのは、ラベルの下にある列をリスト・ボックスのために使用することです。この方法を使用すれば、リストを2つの列に拡張することができます。この効果を作り出すパラメーターは、次のような`GridBagConstraints` の`Gridwidth` フィールドです。

```
constraints.gridwidth = 2
```

`gridheight` フィールドを使用すれば、複数の列を使用できるだけでなく、複数行の使用も指定することができます。これらのすべてのフィールドをコードに組み込んだ結果を、図9に示します。

図9. GridBagConstraintsを制御機構に追加した結果



次に、新しいフィールドをすべて組み込んだコードを示します。

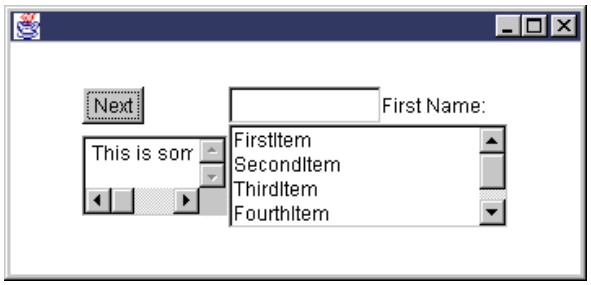
リスト6. GridBagConstraintsを追加した簡単なGUI

```
Frame frame = new Frame();
frame.setLayout(new GridBagLayout());
GridBagConstraints constraints = new GridBagConstraints();
Button button1 = new Button("Next");
constraints.anchor = GridBagConstraints.WEST;
frame.add(button1 , constraints);
TextField text1 = new TextField();
text1.setColumns(10);
constraints.gridx = 1;
constraints.fill = GridBagConstraints.HORIZONTAL;
frame.add(text1 , constraints);
Label label1 = new Label("First Name:");
constraints.gridx = 2;
frame.add(label1 , constraints );
TextArea textArea1 = new TextArea("This is some text in a text area");
textArea1.setRows(2);
textArea1.setColumns(10);
constraints.gridx = 0;
constraints.gridy = 1;
frame.add(textArea1 , constraints );
List list1 = new List();
list1.add("FirstItem");
list1.add("SecondItem");
list1.add("ThirdItem");
list1.add("FourthItem");
list1.add("FifthItem");
constraints.gridx = 1;
constraints.gridwidth = 2;
frame.add(list1 , constraints );
frame.pack();
frame.setVisible(true);
```

ウィンドウ内の余白の処理

ここまで、制御機構と列を操作するためのメソッドを検討してきましたが、次に、もう1つのエレメントについて検討します。それは、ウィンドウ内のスペース全般と、GUIの設計・開発におけるこのスペースの利用法です。図10は、スペースを考慮することがGUIの設計でいかに重要であるかを示しています。図10では、制御機構がデフォルトにより位置付け (つまり中央に配置) され、ウィンドウ・サイズの流動性が考慮されていない簡単なGUIが示されています。

図10. スペースの利用が十分でないGUI

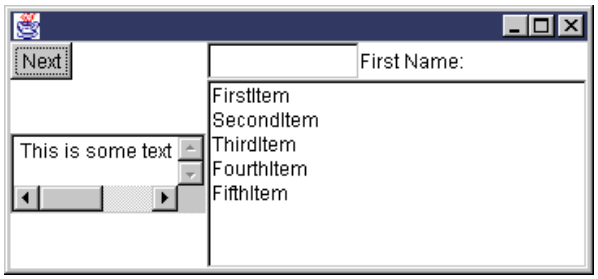


余白を無駄にするのではなく、それをうまく操作して制御機構を拡大することができます。ボタンやラベルなどの制御機構は拡大しても意味がないので、デフォルトでは、制御機構はウィンドウと共に拡大しません。しかし、入力ボックスやリスト・ボックスは、このように拡大できると便利です。

ウィンドウ・サイズが拡大した場合に制御機構も拡大するように指定するには、`GridBagConstraints` でweightxフィールドとweightyフィールドを使用します。これらのフィールドは0～1.0の範囲の値を取り、デフォルトは0です。水平方向に余白がある場合は、`GridBagLayout` クラスは各制御機構にweightxフィールドを問い合わせ、それぞれのweightxに比例してその余白をすべての制御機構間で分割します。

この簡単なGUIの例を使用して、`TextField` に0.5のweightxを割り当て、リスト・ボックスに1のweightxを割り当てます。こうすると、余白は1:2の比率で入力フィールドとリスト・ボックス間で分割されます。リスト・ボックスのweightyを1.0にすると、図11に示されているように、垂直方向の余白をすべて使用するように、リスト・ボックスを拡大することができます。

図11. weightxフィールドとweightyフィールドを使用した結果



weightxとweightyを使用した場合、余白が割り当てられるのは、指定した制御機構ではなく、列です。制御機構が余白を使用するかどうかは、制御機構がどのように固定され、埋め込まれているかによって決まります。列に割り振った余白を図9 のリスト・ボックスで使用するためには、`fill = BOTH` を指定して、水平方向と垂直方向に埋め込みを行い余白を使い切る必要がありました。

リスト7に、簡単なGUIの最後のコードを示します。この簡単なGUIは、移植性と拡張性を考えて設計されています。拡張レイアウト・マネージャーを利用して、制御機構を重要な可変要素 (ストリング長やウィンドウ・サイズの変更など) として動的に位置付けし、サイズを設定しています。

リスト7. weightxパラメーターとweightyパラメーターが追加された簡単な最終GUI

```
Frame frame = new Frame();
```

```
frame.setLayout(new GridBagLayout());
GridBagConstraints constraints = new GridBagConstraints();
Button button1 = new Button("Next");
constraints.anchor = GridBagConstraints.WEST;
frame.add(button1 , constraints);
TextField text1 = new TextField();
text1.setColumns(10);
constraints.gridx = 1;
constraints.fill = GridBagConstraints.HORIZONTAL;
frame.add(text1 , constraints);
Label label1 = new Label("First Name:");
constraints.gridx = 2;
frame.add(label1 , constraints );
TextArea textArea1 = new TextArea("This is some text in a text area");
textArea1.setRows(2);
textArea1.setColumns(10);
constraints.gridx = 0;
constraints.gridy = 1;
constraints.weightx = 0.5;
frame.add(textArea1 , constraints );
List list1 = new List();
list1.add("FirstItem");
list1.add("SecondItem");
list1.add("ThirdItem");
list1.add("FourthItem");
list1.add("FifthItem");
constraints.gridx = 1;
constraints.gridwidth = 2;
constraints.weightx = 1.0;
constraints.weighty = 1.0;
constraints.fill = GridBagConstraints.BOTH;
frame.add(list1 , constraints );
frame.pack();
frame.setVisible(true);
```

結論

この記事では、品質の劣るGUI設計や、失敗する開発プロセスなどの原因になり得るよくある間違いや想定を指摘しました。また、GUI設計・開発用の共通で、さらに柔軟性のあるフレームワークを提供するだけで、Javaレイアウト・マネージャーが、いかにこれらの間違いや想定の訂正に役立つかも示しました。レイアウト・マネージャーをGUI設計プロセスのベースとして使用すれば、設計者は、プロトタイプGUIのスクリーン・ショットを開発チームに提供できるだけでなく、レイアウト・マネージャー設定を各制御機構ごとにより正確に指定することができます。

このように正確に指定しておけば、設計者が、たとえば、フォントを変えたり、ウィンドウ・サイズを変更したり、アプリケーションを別のオペレーティング・システムで実行したりする際に、GUIをどのように振る舞わせるつもりなのかを、開発チームは考えなくて済みます。それどころか、設計者と開発者は協力し合って、異なる実行時条件下でのGUIの振る舞いを分析し把握することができます。結局、レイアウト・マネージャーを使用することで、設計者が設定した当初の仕様をより満たし、また開発チームが設定した実行時要件もより満たすGUIを作り出すことができます。

著者について

Joe Winchester

Joe Winchesterは、ノースカロライナ州にあるIBM Research Triangle Park研究所のソフトウェア・デベロッパーで、WebSphere用開発ツールを担当しています。現在は、JavaBeans同士を接続することで豊富なGUIアプリケーションを作成できるようにする開発者向けビルダーを担当しています。連絡先はjoewin@us.ibm.com です。
Renee Schwartzは、ノースカロライナ州にあるIBM Research Triangle Park研究所のソフトウェア・デザイナーです。IBM WebSphereソフトウェア製品ファミリーのユーザー・インターフェース設計を幅広く担当しており、設計技法とコーディング方針を組み合わせることに興味を抱いています。連絡先はrsch@us.ibm.com です。

Renee Schwartz

Renee Schwartzは、ノースカロライナ州にあるIBM Research Triangle Park研究所のソフトウェア・デザイナーです。IBM WebSphereソフトウェア製品ファミリーのユーザー・インターフェース設計を幅広く担当しており、設計技法とコーディング方針を組み合わせることに興味を抱いています。連絡先はrsch@us.ibm.com です。