The JGoodies Forms Framework

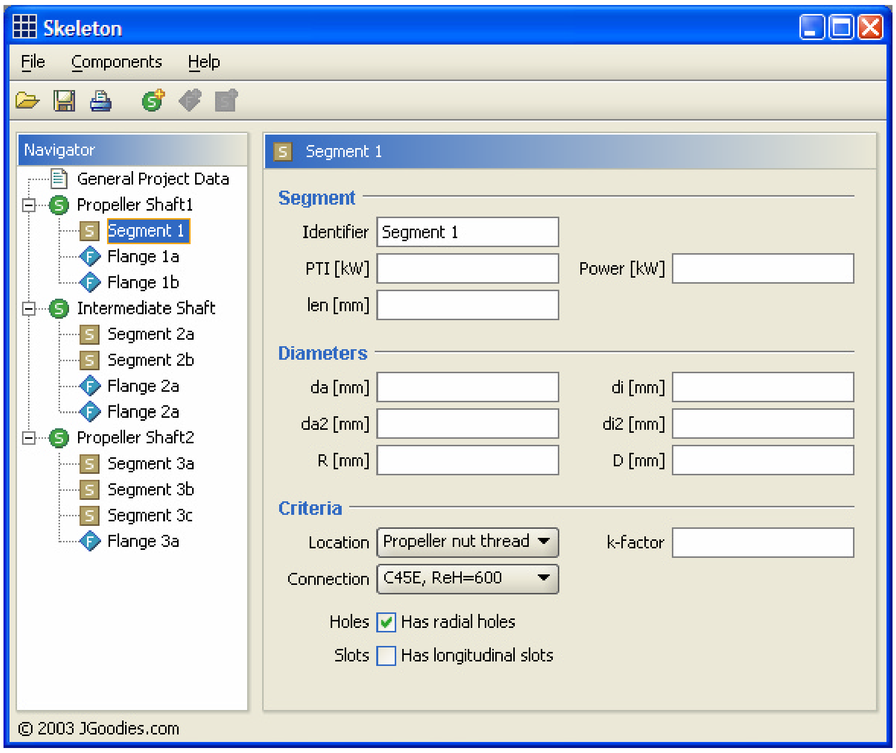
Karsten Lentzsch, April 25, 2003

1 Introduction

The JGoodies Forms framework helps you layout and implement elegant Swing panels consistently and quickly. It aims to make simple things easy and the hard stuff possible, the good design easy and the bad difficult

This document introduces the Forms framework, analyzes weaknesses of existing layout systems, presents design goals, explains how these have been addressed, aquaints you with the Forms layout model and API, and compares Forms with other layout systems.

Forms focuses on form-oriented panels much like the "Segment" panel in the example below. Nevertheless, it is a general purpose layout system that can be used for the vast majority of rectangular layouts.



Identifier	Segment 1		
PTI [kW]		Power [kW]	
len [mm]			
da [mm]		di [mm]	
da2 [mm]		di2 [mm]	
R [mm]		D [mm]	
Location	Propeller nut thread	k-factor	
Connection	C45E, ReH=600		
Holes	✓ Has radial holes		
Slots	☐ Has longitudinal slots		

Screenshot 1: A Form-Oriented Application

Contents

Weaknesses of Existing Layout Systems

I have found that many developers face the following problems with the existing layout systems: they are difficult to understand; hard to work with; they do not represent the human mental layout model; the implementation does not separate concerns, e.g. a layout manager is used to specify layout *and* to fill in components.

Sources of hand-coded panels are often hard to read, they take many lines of code and you can't get a mental design from the code easily.

Some layout managers make the hard stuff impossible. Mostly due to a weak layout specification. For example, most layout managers use pixel sizes and so don't scale with the resolution, font, font size, and look&feel. Some layout systems are extensible but don't handle common layouts and common problems out-of-the-box. Since only a few developers dig into details much potential remains unused.

Most layout systems make the simple things difficult. They don't assist developers in building frequently used layouts. And so, you end up writing the same code again and again. Let's say you want to build a panel with 2 columns: one for labels another for the components; all rows have the same height except titles that group rows. How long does it take you to layout and implement such a design? Should it take more than 10 minutes? How long does it take you to implement the design of screenshot 1?

Some layout systems make it hard to implement a visual builder on top. Or the builders generate weak layouts and don't increase the productivity.

Last but not least, some layout manager implementations are hard to read, so that even an experienced developer cannot verify its correctness.

Design Goals

We want to build form-oriented panels quickly. The framework shall cover 90% of all panels used in a data-oriented desktop application. Results shall be consistent over multiple panels, applications and teams. It shall help novice users achieve good results and experts users save time.

The resulting source code shall be easy to read and easy to understand. The layout system shall work well with visual builders that either increase the productivity or increase the result quality or both. The UI construction process shall be easy to understand

The framework shall ship with well designed examples and all required parts out-of-the-box so that users are immediately productive and have no need to learn the implementation or extend the code.

How Forms Addresses the Design Goals

The Forms framework uses the following principles:

1. Use a grid for a single layout, grid systems for many layouts.
2. Separate concerns.
3. Provide a powerful layout specification language.
4. Allow string representations to shorten layout source code.
5. Provide abstractions above the layout manager.

1) Grid systems are a powerful, flexible and simple means to layout elements. Professional designers use grids to find, balance, construct, and reuse good design in both artwork and everyday media, see [1], [7], [8]. And I've found that many user interface developers use grids implicitly to layout or align components. This works well with paper and pencil and to a lesser extent with todays visual builders that often stiffle creativity more than they assist in finding and constructing good design.

2) Layout managers often combine many features in a single class: specify a layout, fill a panel with components and set component bounds. On the other hand most layout systems lack support for frequently used layouts and provide no means to reuse common design. These problems can be fixed if we separate concerns, use classes that specialize in different tasks and allow to combine them freely. The Forms framework uses a class to describe a form's grid, non-visual builders to fill a form, and uses the layout manager only for one job: set component bounds in a container.

3) With the FormLayout manager you describe a layout before you fill a panel and before the layout manager sets the component bounds. You can specify the form's grid in an expressive language so that readers can easily determine a panel's layout from your code and modify it. And a single specification can be applied to many components; for example, you can specify that all labels in a column are right-aligned.

4) To further improve the code readability we allow to specify the grid in a human readable and concise language using string representations. Even a complex form layout can now be expressed in a few lines of code.

5) The Forms framework provides abstractions above the layout manager: non-visual builder classes that assist you in filling different panel types, and factories that answer prepared panels. The builder classes 'drive' the FormLayout manager. They help you traverse a grid and fill a panel with components. For example, the ButtonBarBuilder specializes in building button bars and knows about default gaps and minimum button sizes. The DefaultFormBuilder builds form-oriented panels with label columns, component columns that are grouped as paragraphs and split into equally sized subsections. Factories utilize these builders and answer prebuilt panels. For example, the ButtonBarFactory can create a default bar for OK, Cancel that honors the platform's button ordering: Mac vs. Windows.

We end up with three code layers:

1. layout classes: layout manager, sizes, alignments, constraints
2. non-visual builder classes to fill panels
3. factory classes that vend prepared layout

In addition to the code we provide:

4. a demo application
5. a tutorial
6. well designed example applications

FormLayout is a powerful, flexible, and precise layout manager that aligns components vertically and horizontally in a dynamic rectangular grid of cells, with each component occupying one or more cells. To define a form layout you specify the form's *columns, rows* and optionally *column groups* and *row groups*. Everything that applies to columns applies to rows too, just with a different orientation. FormLayout uses the same API, algorithms, and implementation for columns and rows.

2.1 Column and Row Specifications

The column specifications define the default alignment, the size and the resizing behavior; the same applies to row specifications. Columns and rows can be specified by instances of ColumnSpec and RowSpec or via string representations that will be parsed to create the associated instances of ColumnSpec and RowSpec.

```
new FormLayout("right:pref, 10px, left:pref:grow",  // 3 columns
               "pref, 4px, pref, pref:grow");       // 4 rows
```

Code Example 1: A FormLayout Specification

In example1 we layout a form with 3 columns and 4 rows. All components in the first column will be aligned to the right hand side by default and the column size is the maximum of the preferred widths of the contained components. The second column is a gap of 10 pixels. The third column is left aligned and its size starts with the maximum component width and grows to fill all available horizontal space.

ColumnSpec supports the following alignments: *left, center, right, fill*; RowSpec supports *top, center, bottom, fill.* These are default alignments that can be overriden for individual components.

A column's or row's size is defined by a constant size, component size, or bounded size. Constant sizes are specified by a value plus unit that is one of: *Pixel, Points, Inches, Millimeter, Centimeter* and *Dialog Units* abbreviated as *px, pt, in, mm, cm,* and *dlu*. Component sizes are one of: *min, pref, default.*

The resize behavior is defined by a non-negative resize weight. The syntax for the string representation for columns and rows is as follows:

```
columnSpec       ::=   [columnAlignment:] size [:resizeBehavior]
rowSpec          ::=   [rowAlignment   :] size [:resizeBehavior]

columnAlignment  ::=   LEFT | CENTER | RIGHT  | FILL | L | C | R | F
rowAlignment     ::=   TOP  | CENTER | BOTTOM | FILL | T | C | B | F

size             ::=   constantSize | componentSize | boundedSize
componentSize    ::=   MIN | PREF | DEFAULT | M | P | D
constantSize     ::=   <integer>integerUnit | <double>doubleUnit
integerUnit      ::=   PX | PT | DLU
doubleUnit       ::=   IN | MM | CM
boundedSize      ::=   MIN(constantSize;componentSize)
                     | MAX(constantSize;componentSize)

resizeBehavior   ::=   NONE | GROW | GROW(<double>) | G(<double>)
```

Spec 1: Column and Row String Specification Syntax

Examples for column string specifications: `“10px”, “4dlu”, “min”, “pref”, “default”, “left:6px”, “right:6dlu”, “left:pref:grow”, “pref:grow(0.5)”, “l:m:g(0.8)”, “left:max(50dlu;pref)”`

Component Sizes

Component sizes give a column or row the maximum size of its contained components, where components can be measured by their minimum or preferred size. You can specifiy the first by the `min` size, the latter by `pref`. The `default` size is like the preferred size but shrinks to the minimum size if space is scarce. See table 1 for the sizing and resizing behavior.

Dialog Units

A layout shall retain proportions when the font size or resolution changes. For example, a button gap of 4 pixels is fine when used with an 8pt font on 96dpi; if used with a 12pt font on 120dpi the same gap is too small. Consequently you should specify sizes of gaps, borders, etc. in a unit that honors the font, font size and resolution; that’s what *Dialog Units* are for. Dialog Units are based on the pixel size of the dialog font and so, they grow and shrink with the font and resolution.

Only in rare cases you want to use pixel sizes. For example, if you align components with non-scalable pixel data, e. g. an image.

Bounded Sizes

Bounded sizes allow to specify lower and upper bounds for a column’s or row’s start size (before resizing). It can be used to ensure a minimum or maximim column width. For example, in button bars that comply with the Microsoft user interface style guide [6] a command button shall have a minimum size of 50 dialog units. In FormLayout you can specifiy such a column with: `new ColumnSpec(“max(50dlu;pref)”);`

Bounded sizes are also useful to guarantee that columns have equal size over multiple panels that can be switched, for example in tabbed panes or panels that use the card layout manager.

Resizing Behavior

Columns and rows can grow if the layout container becomes larger than the preferred size. By default, columns and rows won’t resize. The extra space is distributed over the columns and rows that have a resize weight larger than 0.0, where each column gets a space proportional to its weight.

Layout Task	`constant`	`min`	`pref`	`default`
Get minimum size	constant	minimum	preferred	minimum
Get preferred size	constant	minimum	preferred	preferred
Layout (size <= min)	constant	minimum	preferred	minimum
Layout (min < size < pref)	constant	minimum	preferred	minimum+ available space
Layout (pref <= size)	constant + resize portion	minimum + resize portion	preferred + resize portion	preferred + resize portion

Table 1: Sizing Behavior of Constant and Component Sizes

2.2 Column and Row Groups

Column and row groups specifiy that a set of columns or rows will get the same width or height. This is an essential feature to implement symmetric, or more generally, balanced design. In the following example it is ensured that columns 2 and 4 get the same width, rows 1 and 4 get the same height as well as rows 2 and 3:

```
FormLayout fl = new FormLayout(“p, d, p, d”, “p, p, p, p”);
fl.setColumnGroups(new int[][]{ {2, 4} });
fl.setRowGroups   (new int[][]{ {1, 4}, {2, 3} });
```

2.3 Cell Constraints

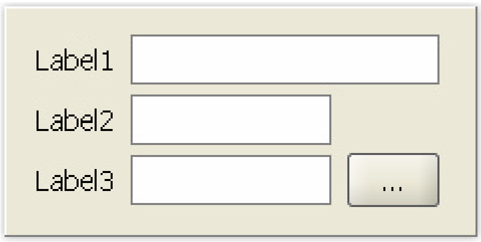
Each component managed by a FormLayout is associated with an instance of CellConstraints. The constraints object specifies where a component should be located on the form's grid and how the component should be positioned.

```
constraints ::=   column, row [, colSpan, rowSpan][, hAlign, vAlign]
column      ::=   <integer>
row         ::=   <integer>
colSpan     ::=   <integer>
rowSpan     ::=   <integer>
hAlign      ::=   LEFT | CENTER | RIGHT  | DEFAULT | FILL
                | L    | C      | R      | D       | F
vAlign      ::=   TOP  | CENTER | BOTTOM | DEFAULT | FILL
                | T    | C      | B      | D       | F
```

Spec 2: Cell Constraints Syntax

2.4 Form Layout Example

We want to build the following layout using the FormLayout manager:



Screenshot 2: A simple panel

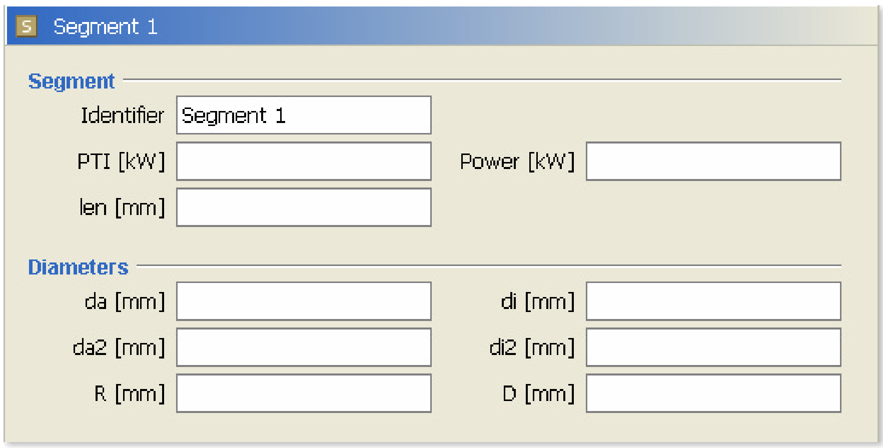
```
FormLayout layout = new FormLayout(“r:p, 6dlu, 50dlu, 4dlu, d”,
                                   “p, 2dlu, p, 2dlu, p”);
layout.setRowGroups(new int[][]{ {1, 3, 5} });

CellConstraints cc = new CellConstraints();
JPanel panel = new JPanel(layout);
panel.add(new JLabel(“Label1”),    cc.xy  (1, 1));
panel.add(new JTextField(),        cc.xywh(3, 1, 3, 1));
panel.add(new JLabel(“Label2”),    cc.xy  (1, 3));
panel.add(new JTextField(),        cc.xy  (3, 3));
panel.add(new JLabel(“Label3”),    cc.xy  (1, 5));
panel.add(new JTextField(),        cc.xy  (3, 5));
panel.add(new JButton(“…”), cc.xy  (5, 5));
```

Code Example 2: A simple panel

Layout managers have been designed to talk to a container – not a human. The Forms framework separates concerns: the layout task from the layout specification and the panel building process. Therefore, Forms provides a set of non-visual builder classes that assist you in building panels and that can shield your from details of the layout manager.

When constructing a panel you talk to a builder which in turn talks to the layout manager. This leads to a smaller layout manager API and to more flexibility. Builders can provide a cursor to keep track of the location where the next component will be added.



Screenshot 3: A common form-oriented layout

```
FormLayout layout = new FormLayout(
    “right:max(50dlu;p), 4dlu, 75dlu, 7dlu, right:p, 4dlu, 75dlu”,
    “p, 2dlu, p, 3dlu, p, 3dlu, p, 7dlu, ” +
    “p, 2dlu, p, 3dlu, p, 3dlu, p”);

PanelBuilder builder = new PanelBuilder(layout);
builder.setDefaultDialogBorder();
CellConstraints cc = new CellConstraints();

builder.addSeparator(“Segment”,      cc.xywh(1,  1, 7, 1));
builder.addLabel(“Identifier”,       cc.xy  (1,  3));
builder.add(new JTextField(),        cc.xy  (3,  3));

builder.addLabel(“PTI [kW]”,         cc.xy  (1,  5));
builder.add(new JTextField(),        cc.xy  (3,  5));
builder.addLabel(“Power [kW]”,       cc.xy  (5,  5));
builder.add(new JTextField(),        cc.xy  (7,  5));

builder.addLabel(“len [mm]”,         cc.xy  (1,  7));
builder.add(new JTextField(),        cc.xy  (3,  7));

builder.addSeparator(“Diameters”,    cc.xywh(1,  9, 7, 1));
builder.addLabel(“da [mm]”,          cc.xy  (1, 11));
builder.add(new JTextField(),        cc.xy  (3, 11));
builder.addLabel(“di [mm]”,          cc.xy  (5, 11));
builder.add(new JTextField(),        cc.xy  (7, 11));

builder.addLabel(“da2 [mm]”,         cc.xy  (1, 13));
builder.add(new JTextField(),        cc.xy  (3, 13));
builder.addLabel(“di2 [mm]”,         cc.xy  (5, 13));
builder.add(new JTextField(),        cc.xy  (7, 13));

builder.addLabel(“R [mm]”,           cc.xy  (1, 15));
builder.add(new JTextField(),        cc.xy  (3, 15));
builder.addLabel(“D [mm]”,           cc.xy  (5, 15));
builder.add(new JTextField(),        cc.xy  (7, 15));
```

Code Example 3: Implementation of Screenshot 3

Example 3 uses the PanelBuilder to set a default border and to add labels and separators. Columns *and* rows are defined before the components are added to the panel. The next step is to append rows to the layout dynamically. Example 4 uses a sophisticated builder to add rows dynamically to the layout. It also uses a convenience method to add a label and associated component in one step. It leads to concise building code:

```
FormLayout layout = new FormLayout(
    "right:max(40dlu;p), 4dlu, 80dlu, 7dlu, "   // 1st major column
  + "right:max(40dlu;p), 4dlu, 80dlu",          // 2nd major column
    "");                                        // rows

DefaultFormBuilder builder = new DefaultFormBuilder(layout);
builder.setDefaultDialogBorder();

builder.appendSeparator("Segment");
builder.append("Identifier",  new JTextField());
builder.nextLine();

builder.append("PTI [kW]",    new JTextField());
builder.append("Power [kW]",  new JTextField());

builder.append("len [mm]",    new JTextField());
builder.nextLine();

builder.appendSeparator("Diameters");
builder.append("da [mm]",     new JTextField());
builder.append("di [mm]",     new JTextField());

builder.append("da2 [mm]",    new JTextField());
builder.append("di2 [mm]",    new JTextField());

builder.append("R [mm]",      new JTextField());
builder.append("D [mm]",      new JTextField());
```

Code Example 4: Rewrite of Example 3

The `DefaultFormBuilder` used in Example 4 adds gaps between lines and paragraphs as needed. Other builders specialize in building button bars and stacks, map resource keys to internationalized strings, etc.

4 Form Factories

The Forms framework provides a set of factory classes that can create frequently used layouts, panels, and button bars quickly and consistent. Whenever possible you should use these factories to create subpanels.

The factories utilize the form builders or the underlying form layout.

Form Factory Example

```
private JComponent buildInterface() {
    JButton helpButton    = getHelpButton();
    JButton backButton    = getBackButton();
    JButton nextButton    = getNextButton();
    JButton finishButton  = getFinishButton();
    JButton canceltButton = getCancelButton();
    ...
    return WizardBarFactory.createHelpBackNextFinishCancelBar(
        helpButton, backButton, nextButton, finishButton,
        cancelButton);
}
```

5 Comparisons with Other Layout Managers

In this section we compare the FormLayout with other popular, flexible, and powerful layout managers: TableLayout, HIGLayout, GridBagLayout, and SpringLayout. The Forms framework has been designed to supercede TableLayout and HIGLayout and can often replace GridBagLayout and SpringLayout when building form-oriented panels.

None of the layout managers mentioned above supports dialog units or a similar sizing system out-of-the-box. However, SpringLayout can be extended to provide dialog unit springs, and almost every layout manager can be wrapped with a mapper from dialog units to pixels.

TableLayout

TableLayout [2] differs from FormLayout in that it 1) uses a weaker layout specification, 2) encodes form specifications with number types and 3) provides no means to give columns or rows the same size [2].

1) TableLayout lacks the FormLayout's `default` component size that shrinks components from preferred size down to minimum size if the container space is scarce. Columns and rows have constant or component size *or* grow; you cannot combine these options in TableLayout and cannot specify resize weights. Also, TableLayout cannot specify a column or row default alignment and doesn't support cell insets.

2) Column and row specifications are encoded with numbers that are categorized by their type to express different specification types whereas FormLayout uses implementations of Size or a string encoding.

Both, TableLayout and FormLayout care about rounding errors when distributing extra space.

I've found that novice and experienced developers can work well with the TableLayout. The grid-based layout model seems to be easy to learn and easy to work with. And it seems to me that TableLayout works well with grid-oriented visual form editors, for example, the free Radical editor.

HIGLayout

HIGLayout [5] is much like TableLayout but allows to give columns and rows the same size. It differs from FormLayout in that it 1) uses a weaker layout specification and 2) encodes form specifications with number types.

1) HIGLayout lacks the FormLayout's minimum component size and the default component size that shrinks components from preferred size down to minimum size if the container space is scarce. HIGLayout cannot specify a column's or row's default alignment.

2) HIGLayout overloads integers to specify column and row sizes: constant sizes, component preferred size, and group specification.

HIGLayoutConstraints provides a cursor that assist you in traversing the form while building a panel. In contrast, the Forms framework uses separate builder classes to traverse a form.

GridBagLayout

GridBagLayout [9] arranges components in a rectangular grid that is specified implicitly by component constraints. GridBagLayout has no means to give columns or rows equal sizes.

The GridBagLayout supports logical panel building directions for left-to-right and right-to-left. In Forms, builders traverse a form, not the layout manager. Currently Forms provides only builders that are optimized for left-to-right and top-to-bottom directions.

Changing a grid dynamically is a one-step process in GridBagLayout, because the GridBagConstraints implicitly specify the layout. FormLayout requires two steps: change the grid and modify the form's contents.

The NetBeans and Sun Forte IDEs come with a popular visual editor that works well with GridBagLayout. The editor doesn't support dialog units and has almost no support for consistent design or compliance with style guides. And from my perspective, this editor requires many steps to construct even simple design.

In GridBagLayout you need to check all constraints to determine the grid and how components will be arranged. Even if you reuse constraints, readers cannot easily infer the layout from the building code. Sources for panel building are often cluttered by the GridBagConstraints configuration – even if you use convenience methods to create or reuse constraints.

SpringLayout

SpringLayout [11] ships with the J2SE 1.4. It defines relationships between component edges using a spring concept. This way it sets the position and size of components and ties together components, their position and resizing behavior. SpringLayout has been designed to be extendable and can delegate the layout task to constraints that in turn can implement a layout strategy. It comes with convenience spring implementations.

With little effort, SpringLayout can give components equal size and it can be extended with custom springs to support dialog units. Unlike most other layout managers, SpringLayout may be able to specify inter-panel layout constraints; I haven't verified if this works.

SpringLayout needs more code than the FormLayout to express simple but frequently used form design. From my perspecitive, the main problem with SpringLayout is, that the layout specification language (Java code) doesn't represent the human mental layout model well. You can hardly *see* the layout by just looking at the panel building code. FormLayout's layout specification language has been designed to express how many people think and talk about layout. How many lines of code do you need to build the form in Example 3 with the SpringLayout?

FormLayout favors an expressive layout specification over extensibility. It has been designed to be powerful and flexible enough to layout almost every panel that I've designed during the last decade. It covers all panels in the JGoodies tools and seems to be able to layout all panels in large apps like the Eclipse JDT and the NetBeans IDE. And so, I doubt that there's a need to extend the FormLayout – at least it isn't urgent.

Future Directions

The current implementation of the JGoodies Forms is a good starting for a productive UI construction process. I plan to write new builders and enhance the factories.

I would like to extent the FormLayout to allow inter-panel constraints to provide a stable layout if you switch panels in a tabbed pane or cards. For example, to give a label column the maximum width of all labels in the tabbed panel set. Currently we handle this with bounded sizes.

Precise layout that complies with style guides should distinguish layout bounds and perceived bounds. If components are rendered with a pseudo 3D effect they often need asymmetric gaps on top and bottom, left and right to be perceived as symmetric.

And most likely I will write a visual form editor for the FormLayout.

UI Template Method

I recommend to use abstract dialog and frame classes that complement the Forms framework to comprise what you do again and again when building dialogs and frames: build the content pane, pack it, resize to gain aesthetic aspect ratios, locate on screen, provide actions for OK, Cancel, Apply, Help, etc. If you refactor a larger GUI application and form template methods (see refactoring #345, [4]), such an abstraction is a natural result.

Project State

The FormLayout is stable since December 2002; the Size implementations and Builders are very stable since the beginning of 2003 I have added only a few methods to the published builder classes.

Recently, I've added some classes that are work in progress to an extras package in source code only: the `DefaultFormBuilder` that has been used to implement Example 4; an `I15dPanelBuilder` that asssists you in building panels that used internationalized (i15d) labels; a `FormDebugPanel`, that can paint the form's grid; the `FormDebugUtils` that an print debug information to the console: column and row specs, cell constraints, and grid bounds.

I'm in the process of writing a Forms tutorial and a Forms Demo that provides "live" panels for the examples in the tutorial.

Feedback

Your comments and suggestions regarding the form layout, the builder, factories and this article are welcome and will help me improve this library.

Acknowledgements

The form-oriented building has been inspired by the grid layout system as described by Kevin Mullet and Darrel Sano in [7]. The FormLayout's API has roots in the APIs of TableLayout by Daniel Barbalace, HIGLayout by Daniel Michalik and GridBagLayout by Doug Stein. The concept of dialog units has been taken from Microsoft user interface implementations.

Many thanks to Johannes Riege of Riege Software International for supporting me in writing this layout system.

References

[1] Alexander, Christopher (1964)
Notes on the Synthesis of Form, Cambridge: Harvard University Press

[2] Apple Computer, Inc. (2002)
Aqua Human Interface Guidelines, www.apple.com/developer/

[3] Barbalace, Daniel (2001)
TableLayout, java.sun.com/products/jfc/tsc/articles/tablelayout/

[4] Fowler, Martin et al (1999)
Refactoring: Improving the Design of Existing Code, Reading: Addison Wesley

[5] Michalik, Daniel (2001)
HIGLayout, www.autel.cz/dmi/tutorial.html

[6] Microsoft Corporation (2002)
Design Specifications and Guidelines – Visual Design, msdn.microsoft.com

[7] Mullet, Kevin and Sano, Darrel (1995)
Designing Visual Interfaces, Prentice Hall

[8] Müller-Brockmann, Josef (1988)
Grid Systems in Graphic Design, Stuttgart: Verlag Gerd Hatje

[9] Stein, Doug and Sun Microsystems (1995)
GridBagLayout, J2SE 1.4, java.sun.com/j2se/1.4.1/docs/

[10] Willberg, Hans Peter and Forssmann, Friedrich (1997)
Lesetypographie, Mainz: Verlag Herrmann Schmidt

[11] Winchester, Joe and Milne, Philip (2001)
SpringLayout, java.sun.com/j2se/1.4.1/docs/