

The JGoodies Forms Framework

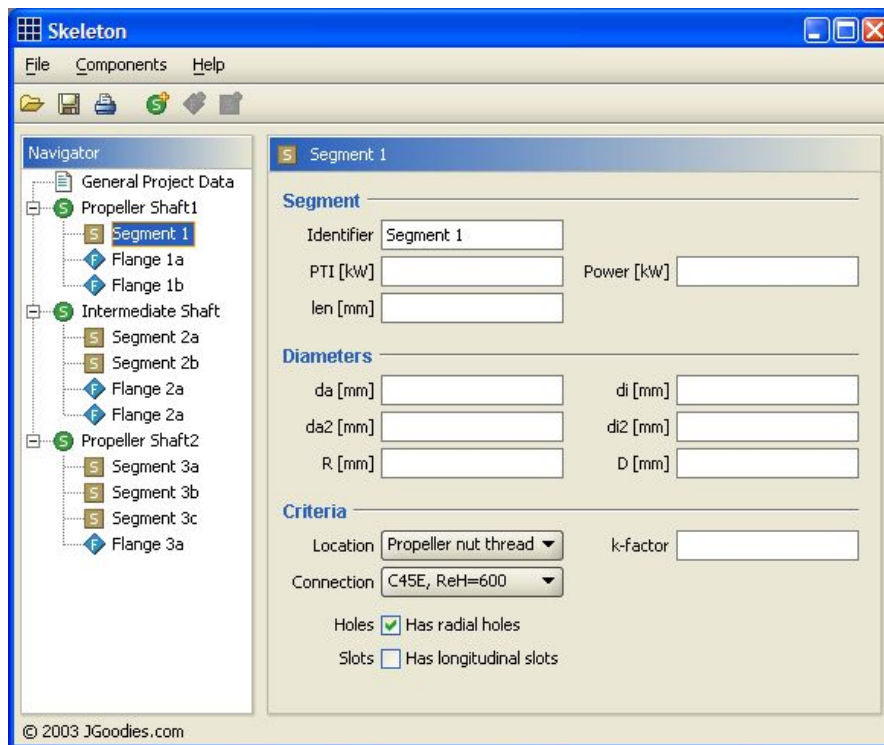
Karsten Lentzsch, August 5, 2003

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, acquaints 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.



Screenshot 1: A Form-Oriented User Interface

Contents

1. Introduction
2. Form Layout
3. Form Builder
4. Form Factories
5. Comparison with other Layout Managers
6. Misc

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 hardly infer a mental layout from the code.

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 retain proportions if you change the font or resolution. 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 poor layouts or decrease your productivity.

Last but not least, some layout manager implementations are hard to read, so that even an experienced developer cannot verify the 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 expert 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 follows five principles:

1. Use a grid for a single layout, use grid systems for many layouts.
2. Separate concerns.
3. Provide a powerful layout specification language.
4. Allow string representations to shorten layout code.
5. Provide developer guidance on top of 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 artwork and everyday media, see [1], [7], [8], [11]. 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 today's visual builders. The latter often stifle 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: compute and set component bounds.

3) With the `FormLayout` 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 infer a panel's layout from your code quickly and modify it easily. 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. This way even complex layouts can be expressed in a few lines of code.

5) Forms provides abstractions on top of the layout manager that lead to consistent UIs and assist you in style guide compliance. Non-visual builder classes 'drive' the `FormLayout`. They help you traverse the grid and seed it with components. For example, the `ButtonBarBuilder` specializes in building button bars and knows about default gaps and minimum sizes. The `DefaultFormBuilder` builds form-oriented panels with label columns and 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 layouts and prebuilt panels

In addition to the code we provide:

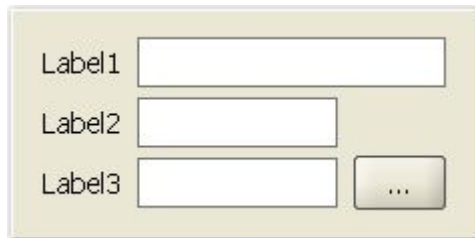
4. a demo application
5. a tutorial
6. well designed example applications

2 Form Layout

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.

An Example

Let's jump into an example that demonstrates basic FormLayout features. We're going to learn the details step by step in the following sections and will learn how to write more complex layouts with less code.



Screenshot 2: A simple panel

```
1: FormLayout layout = new FormLayout(  
2:     "pref, 4dlu, 50dlu, 4dlu, min",           // columns  
3:     "pref, 2dlu, pref, 2dlu, pref");          // rows  
4:  
5: layout.setRowGroups(new int[]{{1, 3, 5}});  
6:  
7: JPanel panel = new JPanel(layout);  
8:  
9: CellConstraints cc = new CellConstraints();  
10: panel.add(new JLabel("Label1"), cc.xy(1, 1));  
11: panel.add(textField1, cc.xywh(3, 1, 3, 1));  
12: panel.add(new JLabel("Label2"), cc.xy(1, 3));  
13: panel.add(textField2, cc.xy(3, 3));  
14: panel.add(new JLabel("Label3"), cc.xy(1, 5));  
15: panel.add(textField3, cc.xy(3, 5));  
16: panel.add(detailsButton, cc.xy(5, 5));
```

Code Example 1: a 'raw' implementation of screenshot 2

Before we started to implement the example, we identified the panel's grid – including the columns and rows for the gaps between the components.

In lines 1 to 3 we construct a FormLayout and specify the columns and rows using a string representation, where *pref* is for 'preferred size', *min* is for 'minimum size', and *dlu* is a size unit that scales with the font. In line 5 we just say that all component rows (1, 3 and 5) shall get the same height. In line 7 we create the layout container, a JPanel. In line 9 we obtain a CellConstraints object used to specify the component locations in the grid. And finally, in lines 10 to 16 we fill the panel with the components, where labels are created on-the-fly.

2.1 Column and Row Specifications

The columns and rows are specified by three parts: a mandatory size, an optional default alignment and an optional resizing behavior. You can use instances of `ColumnSpec` and `RowSpec` or 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 2: A `FormLayout` Specification

In `example2` 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 overridden 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 Encoding Syntax

```
"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)"
```

Code Example 3: Column String Representations

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 specify 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 a non-scaled 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 maximum column width. For example, the MS user interface style guide [6] recommends a minimum width of 50 dialog units for command buttons. In `FormLayout` you can specify such a column with:

```
new ColumnSpec("max(50dlu;pref)");
```

Bounded sizes are also useful to avoid the tabbed panels seem to 'jump' to the left and right, or up and down; they same applies to the cards in a `CardLayout`. To make the design stable just reuse the same bounded sizes in all panels that are switched.

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 specify 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 layout = new FormLayout("p, d, p, d", "p, p, p, p");  
layout.setColumnGroups(new int[][]{ {2, 4} });  
layout.setRowGroups    (new int[][]{ {1, 4}, {2, 3} });
```

2.3 Cell Constraints

Each component managed by a `FormLayout` is associated with an instance of `CellConstraints` that specifies a component's display area and alignment. The column and row origins are mandatory, but as we will see later, often a non-visual builder will automatically create the `CellConstraints` for you.

By default the column and row span is just 1, and the alignments are inherited from the related column and row. If possible you should specify the alignment for the column and row, not for the component; this way you can reduce the amount of alignment specifications significantly.

`CellConstraints` objects can be constructed in different ways using a mixture of ints, objects and strings. I recommend to specify the origin and span using ints and the alignment with strings – just to increase the code readability. You can reuse `CellConstraints` objects because they are cloned internally by the `FormLayout`.

```
// 1) Creation methods intended for use by humans
CellConstraints cc = new CellConstraints();
cc.xy(2, 1); // second col, first row
cc.xy(2, 1, "right, bottom"); // aligned to right and bottom
cc.xy(2, 1, "r, b"); // abbreviated alignment
cc.xywh(2, 1, 4, 3); // spans 4 cols, 3 rows
cc.xywh(2, 1, 4, 3, "right, bottom");
cc.xywh(2, 1, 4, 3, "r, b");

// 2) Constructors intended for builders
new CellConstraints(); // first col, first row
new CellConstraints(2, 1);
new CellConstraints(2, 1, 4, 3);
new CellConstraints(2, 1, CellConstraints.RIGHT,
    CellConstraints.BOTTOM);

// 3) Constructors intended for building UIs from XML
CellConstraints cc = new CellConstraints();
new CellConstraints("2, 1");
new CellConstraints("2, 1, r, b");
new CellConstraints("2, 1, 4, 3");
new CellConstraints("2, 1, 4, 3, r, b");
```

Code Example 4: CellConstraints Construction

Example 4 demonstrates different ways to obtain a `CellConstraints` object. The methods in section 1) are the typical and recommended style to create `CellConstraints`: specify the column, row, column span and row span with ints and the optional alignment as an encoded String. As seen before, this code style often takes a single line of code to add a component to a panel.

The style in section 2) uses objects instead of Strings and gains compile time safety but requires more written code which may clutter your panel building code. It is recommend for use by non-visual builders, where you favor safety and efficiency over code readability.

The constructors in section 3) convert a String representation into a `CellConstraints` object. They minimize the effort required to build a UI from a String representation, as decribed by the following syntax:

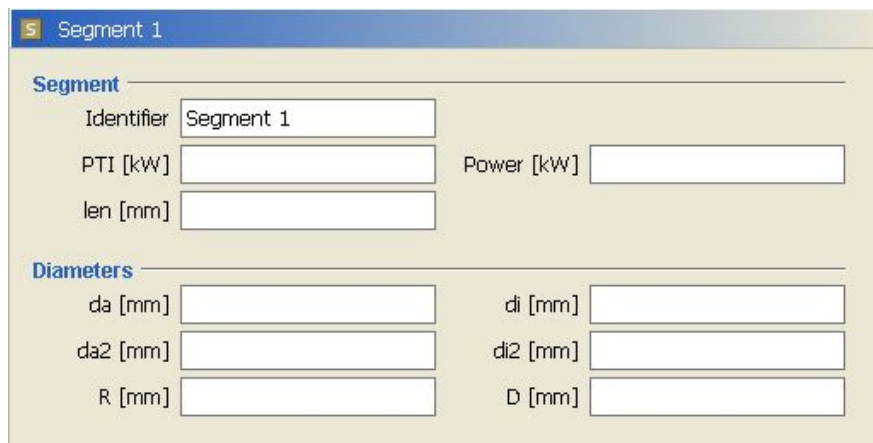
```
constraints ::= column, row [, colSpan, rowSpan][, hAlign, vAlign]
column      ::= <integer>
row         ::= <integer>
colSpan     ::= <integer>
rowSpan     ::= <integer>
hAlign      ::=
| LEFT | CENTER | RIGHT | DEFAULT | FILL
| L    | C     | R    | D      | F
vAlign     ::=
| TOP  | CENTER | BOTTOM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

Spec 2: Cell Constraints String Encoding Syntax

3 Form Builder

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 you 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, can create frequently used components, can assist in style guide compliance, etc.



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(idField, cc.xy(3, 3));

builder.addLabel("PTI [kW]", cc.xy(1, 5));
builder.add(ptiField, cc.xy(3, 5));
builder.addLabel("Power [kW]", cc.xy(5, 5));
builder.add(powerField, cc.xy(7, 5));

builder.addLabel("len [mm]", cc.xy(1, 7));
builder.add(lenField, cc.xy(3, 7));

builder.addSeparator("Diameters", cc.xywh(1, 9, 7, 1));
builder.addLabel("da [mm]", cc.xy(1, 11));
builder.add(daField, cc.xy(3, 11));
builder.addLabel("di [mm]", cc.xy(5, 11));
builder.add(diField, cc.xy(7, 11));

builder.addLabel("da2 [mm]", cc.xy(1, 13));
builder.add(da2Field, cc.xy(3, 13));
builder.addLabel("di2 [mm]", cc.xy(5, 13));
builder.add(di2Field, cc.xy(7, 13));

builder.addLabel("R [mm]", cc.xy(1, 15));
builder.add(rField(), cc.xy(3, 15));
builder.addLabel("D [mm]", cc.xy(5, 15));
builder.add(dField, cc.xy(7, 15));
```

Code Example 5: Implementation of Screenshot 3

Example 5 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 6 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
    "");                                     // add rows dynamically

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

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

builder.append("PTI [kW]", ptField);
builder.append("Power [kW]", powerField);

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

builder.appendSeparator("Diameters");
builder.append("da [mm]", daField);
builder.append("di [mm]", diField);

builder.append("da2 [mm]", da2Field);
builder.append("di2 [mm]", di2Field);

builder.append("R [mm]", rField);
builder.append("D [mm]", dField);
```

Code Example 6: Rewrite of Example 5

The `DefaultFormBuilder` adds lines and line gap rows as needed. And he creates label, titles, separators and accepts pairs of label and component, which shrinks the panel building code significantly. Other non-visual 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 cancelButton = getCancelButton();
    ...
    return WizardBarFactory.createHelpBackNextFinishCancelButton(
        helpButton, backButton, nextButton, finishButton,
        cancelButton);
}
```

5 Comparisons with Other Layout Managers

In this section we compare `FormLayout` with `TableLayout`, `ExplicitLayout`, `GridBagLayout` and `SpringLayout` – all which are flexible and powerful.

The Forms framework has been designed to supercede `TableLayout` and it can often replace `GridBagLayout` and `SpringLayout` when building form-oriented panels. `ExplicitLayout` offers more flexibility in the location and size of components than `FormLayout`.

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.

ExplicitLayout

`ExplicitLayout` [9] is a very powerful general purpose layout manager. It provides many features to specify the location and size of a component, almost all things a designer may need – even for non-rectangular layouts. Constraints are defined by quite simple but highly flexible expressions.

Beyond the flexibility, the `ExplicitLayout` offers other useful features: styles, external UI specification and layout reuse. Styles are similar to the Forms builder classes; they assist developers in making simple layouts easy. The current style set provides a level of layout abstraction that is higher than the abstractions provided by the Forms builders and layout factories. Very simple forms can be constructed with a single line of code, with the downside of less flexibility.

`ExplicitLayout` is licensed under the LGPL and ships with a user guide and well chosen examples that demonstrate its flexibility. The 3.0 release lacks support for non-pixel sizes (like dlu) and logical sizes as provided by the Forms layout styles. Anyway, these features could be added as well as a richer set of styles (or builders).

I'd say the `ExplicitLayout` is the first choice for non-rectangular layouts and other complex non-form-oriented designs. Due to its flexibility it is a powerful weapon – at least in the hands of experienced UI designers. In contrast `FormLayout` focuses on form-oriented design and reduces the flexibility to guide developers in using good design – and avoiding the bad.

TableLayout

`TableLayout` [3] 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.

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.

GridBagLayout

`GridBagLayout` [11] 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` [12] 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 perspective, 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.

6 Misc

7 Future Directions

The JGoodies Forms is a good starting point for producing high-fidelity Java UIs. I plan to further simplify the UI construction process and provide guidance to good and consistent design: write new builders, enhance the factories, add logical sizes, etc.

I would like to extend the `FormLayout` to allow inter-panel constraints to provide a stable layout if you switch panels with 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; the same applies to drop shadows.

And I plan to assist vendors in writing a productive 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` manager is stable since December 2002, and the API is stable since March 2003. Tutorial sources demonstrate the `FormLayout`, builder classes, different builder styles and layout problems that you can hardly solve with the JDK layout managers. The Forms Demo is a web startable application that uses the tutorial sources and visualizes the layout problems. English and German presentations around the JGoodies Forms and general Java layout issues are available; a tutorial will be available soon. Some classes that are work in progress ship the extras source package: the `DefaultFormBuilder` used in Example 6 and the `I15dPanelBuilder` class that assists in building internationalized (i15d) panels.

Debugging

Forms ships with the `FormDebugUtils` that can print debug information to the console: column and row specs, cell constraints, and grid bounds. And a special `FormDebugPanel` paints the form's grid.

Feedback

Your comments and suggestions regarding the `FormLayout`, the builders, 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[5] by Daniel Michalik and GridBagLayout by Doug Stein. Dialog units stem from Microsoft user interfaces. Many thanks to Johannes Riege and Sun Microsystems for supporting me in writing and opening the Forms.

Copyright Notes

All rights reserved. No part of this article may be reproduced in any form by any means without prior written authorization of Karsten Lentzsch.

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] Prayle, Alex (2001)
ExplicitLayout, www.zookitec.com/explicitlayout.html
- [10] Stein, Doug and Sun Microsystems (1995)
GridBagLayout, J2SE 1.4, java.sun.com/j2se/1.4.1/docs/
- [11] Willberg, Hans Peter and Forssmann, Friedrich (1997)
Lesetypographie, Mainz: Verlag Herrmann Schmidt
- [12] Winchester, Joe and Milne, Philip (2001)
SpringLayout, java.sun.com/j2se/1.4.1/docs/