Demystifying the *GridBagLayout* Layout Manager

Peter Haggar (IBM Network Computing Software)

April 14, 1999

Abstract

The *GridBagLayout* layout manager is the most powerful and complex of the layout managers provided in the Java libraries. Because of its power and design, it is the most useful layout manager provided. However, due to the intricate nature of *Gridbag*, many people are apprehensive to use it. With the proliferation of visual builder tools available today in the marketplace, many developers resort to using them to build their complex interfaces. These tools typically use the *GridBagLayout* and shield the developer from having to know the details contained in this paper. It is fine to use a visual builder tool, but I also think it is very useful to understand this class and be able to debug the generated code if and when necessary. This paper will explore this perplexing class and explain exactly how to use this layout manager to develop custom user interfaces.

1 Introduction

I often refer the *GridBagLayout* layout manager as "The mother of all layout managers" due to its complexity. Part of the problem for Java developers using the *GridBagLayout* is the lack of good, accurate information on how to use it. Many resources, frankly, do a very poor job at explaining the details of *GridBagLayout* sufficiently for developers to work with it effectively. Taking advantage of the *GridBagLayout* is typically necessary to produce anything but extremely simple graphical user interfaces. My experience has shown when developing a functional user interface, you will almost always wind up using the *GridBagLayout*. The other layout managers, (*FlowLayout*, *BoxLayout*, *BorderLayout*, *CardLayout*, *GridLayout*) although useful, typically are not able to provide the functionality or flexibility needed to develop a suitable user interface. I have found that you will use each of the other layout managers by incorporating them into your *GridBagLayout*. Therefore, you are not using *GridBagLayout* at the expense of the other layout managers, but in addition to them.

This requires you to understand and know how to use all of them. Various books usually do a good job of explaining all but the *GridBagLayout* layout manager. This paper will attempt to demystify the *GridBagLayout* layout manager by providing useful and accurate information about how it works and how to program it.

2 The Basics

The *GridBagLayout* lays out its components in a grid. Each component that is added is associated with an object called the *GridBagConstraints* object. This object specifies the size, position, and behavior of the component in the grid. Before moving on to the details, it is important to first define some terms I will be using throughout the description of *GridBagLayout*. It is imperative to understand these terms and differentiate them from one another in order to understand how to use *GridBagLayout*.

2.1 Grid Cells

Grid cells are shown in Figure 1. These are logical cells that the *GridBagLayout* layout manager uses when placing components within its grid.

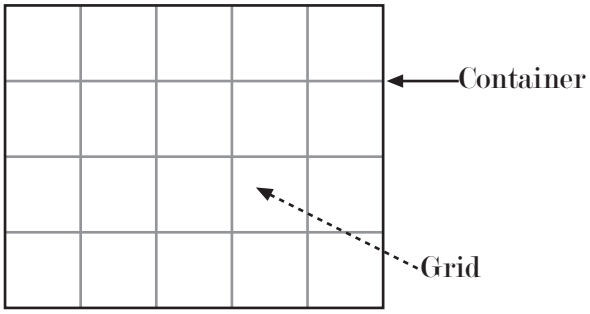

Figure 1: The basic grid of *GridBagLayout*

2.2 Component Cells

Component cells are areas of the basic grid which components occupy. Component cells can occupy multiple grid cells. These multiple grid cells are called the component cell and is specified with the *GridBagConstraints* object, which we discuss in full detail below. Figure 2 gives the visual representation of component cells within a grid.

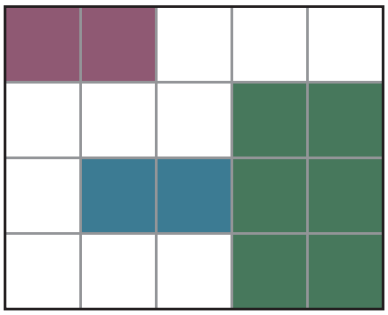

Figure 2: Component cells within the grid

2.3 Components Within Component Cells

Figure 3 shows components within component cells within multiple grid cells. It is important to understand the difference between these three terms as I will be referring to each (grid cells, component cells, and components) during the rest of the discussion of *GridBagLayout*. These details are important as we examine how to create one of these layouts and get it to look and behave properly.

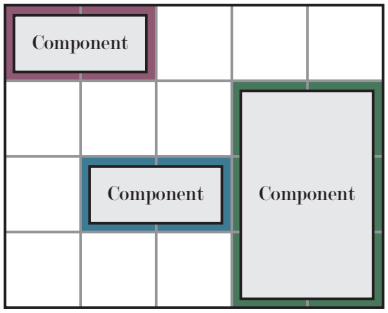

Figure 3: Components within component cells

3 The GridBagConstraints Object

Programming the *GridBagLayout* requires an in-depth understanding of the *GridBagConstraints* class. This class is used to specify many things about the components in a *GridBagLayout*. Each component added to a *GridBagLayout* must have a corresponding *GridBagConstraints* object. This object defines, among other things, the size and position of the component cell, the size of the component, and how the component behaves in the layout as its container changes size. The *GridBagConstraints* object consists of the following fields:

- `gridx`
- `gridy`
- `gridwidth`
- `gridheight`
- `fill`
- `anchor`
- `weightx`
- `weighty`
- `ipadx`
- `ipady`
- `insets`

Let's examine each one of these separately. As I am explaining each field of the *GridBagConstraints* object, I will be referring to grid location, component cell, and component. See Figures 1, 2, and 3, and the associated discussion of these terms in *The Basics* section above if you are not sure of the differences between them.

3.1 `gridx` and `gridy`

These are used to specify the location of the component cell in the grid. You can specify integer values for `gridx` and `gridy` or a predefined constant value `RELATIVE`. Integer values denote the starting location for the component cell. The grid in a *GridBagLayout* is zero based. The constant value `RELATIVE` tells the *GridBagLayout* to place this component cell just to the right (`gridx`) or just below (`gridy`) the previously added component.

3.2 `gridwidth` and `gridheight`

These are used to specify the size of the component cell in the grid. Notice from Figure 2 that the component cell can cover multiple grid locations. You can specify integer values or two predefined constant values: `RELATIVE` and `REMAINDER`. The integer values denote the size of the component cell in grid locations. `REMAINDER` indicates that this component cell is the last one in its row (`gridwidth`) or column (`gridheight`). `RELATIVE` indicates that this component cell is the next to last one in its row (`gridwidth`) or column (`gridheight`).

3.3 `fill`

When a component does not take up all of the space of its component cell, the `fill` constraint is used to specify how the component will occupy the extra space. The valid values for fill and their affect on the component are:

`HORIZONTAL` The component in the component cell will be sized such that it fully occupies the width of the component cell. The height of the component will be the component's preferred height.

`VERTICAL` The component in the component cell will be sized such that it fully occupies the height of the component cell. The width of the component will be the component's preferred width.

`BOTH` The component in the component cell will be sized such that it fully occupies the width and height of the component cell. The preferred width and height of the component will be ignored.

`NONE` The component in the component cell will be sized to be its preferred size. The GridBagLayout manager will not attempt to fill the component cell with this component either horizontally or vertically.

3.4 `anchor`

When a component does not take up all of the space of its component cell, the `anchor` constraint is used to specify where the component will be anchored in the component cell. The valid values for anchor are:

- `CENTER`
- `NORTH`
- `NORTHEAST`
- `EAST`
- `SOUTHEAST`
- `SOUTH`
- `SOUTHWEST`
- `WEST`
- `NORTHWEST`

The default value is `CENTER`.

The component cell is logically broken up into nine locations. The nine anchor constants place the component in one of these locations.

3.5 `weightx` and `weighty`

When the container the *GridBagLayout* is in is sized, `weightx` and `weighty` are used to determine how the extra space will be allocated to the components in the layout. The value is not a straight percentage. For example, setting the weight of a component to 50 and two others to 25 will not necessarily result in one half the space being given to the one component while the other two split the remainder. The way weight works in a *GridBagLayout* is fairly straightforward once you understand what is going on.

When the *GridBagLayout* layout manager increases or decreases its size based on the sizing of its container it performs the following calculation:

For each column, the largest `weightx` for all components in a column is determined. For purposes of discussion, we will call this `columnWeight`. Then, all column weights are totaled for a total weight. We will call this `totalWeight`. `numPixes` refers to the total number of pixels to be added or removed from the layout. The *GridBagLayout* layout manager then uses the following formula to determine the number of pixels(`numPixes`) to add or subtract to or from a particular column:

$$\frac{columnWeight * numPixels}{totalWeight}$$

The formula for rows is similar. Simply substitute 'row' for 'column' above.

By default, `weightx` and `weighty` have the value 0. When all component's `weightx` and `weighty` values are 0, no components get any extra space. In addition, all components are positioned in the *GridBagLayout* relative to the center of their container.

One of the things that is tricky about weight in *GridBagLayout* is that a component can have a weight of 0, but still change its size. This occurs when other components in that components row or column have weights other than 0. Remember that the formula for determining how many pixels to apply to

each row/column determines a weight for the entire row/colum. Therefore if a component has a weight of 0, but another component in that same row or column has a non zero weight, all components in that row/column will be affected.

What happens when a component, which has a weight assigned, is in more than one row or column? Which row/column does the weight apply to? You may think that the weight applies to all the rows and columns the component spans. However, the weight only applies to the last row and the last column the component occupies. The weight is not a factor in the other rows and columns the component may occupy.

3.6 `ipadx` and `ipady`

These two constrains simply grow the component by the specified amount in the specified direction. For example, an `ipadx` of 5 will add five pixels to the width of the specified component.

3.7 `insets`

This constraint specifies margins to be added to the inside edge of the component's cell. These are similar to container insets in that components will not draw on the insets. The use of `insets` is a way to provide a border around a component in a *GridBagLayout*.

4 Creating a GridBagLayout

The listing below shows the code used to create the layout in Figure 3. I'll use buttons for the arbitrary components.

```
class GridBagTest extends Frame
{
    private Button b1 = new Button("b1");
    private Button b2 = new Button("b2");
    private Button b3 = new Button("b3");

    public GridBagTest()
    {
        super();

        GridBagLayout gbl = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();  // 1
        setLayout(gbl);                                     // 2

        gbc.fill   = GridBagConstraints.HORIZONTAL;         // 3
        addToGridBag(b1, gbl, gbc, 0, 0, 3, 1, 1, 1);

        gbc.anchor = GridBagConstraints.WEST;
        addToGridBag(b2, gbl, gbc, 2, 2, 2, 1, 1, 1);

        gbc.anchor = GridBagConstraints.SOUTHEAST;
        gbc.fill   = GridBagConstraints.BOTH;
```

```
        addToGridBag(b3, gbl, gbc, 4, 1, 2, 3, 1, 1);

        pack();
    }

    private void addToGridBag(Component c,
        GridBagLayout gbl, GridBagConstraints gbc,
        int x, int y, int w, int h, int wx, int wy)         // 4
    {
        gbc.gridx      = x;
        gbc.gridy      = y;
        gbc.gridwidth  = w;
        gbc.gridheight = h;
        gbc.weightx    = wx;
        gbc.weighty    = wy;
        gbl.setConstraints(c, gbc);
        add(c);
    }

    public static void main(String[] args)
    {
        GridBagTest gbt = new GridBagTest();
        gbt.show();
    }
}
```

At `// 1` we are creating the *GridBagConstraints* object we will need. We can reuse it each time since the object data is copies internally, but remember to update any constraints the next time you add another component if you want them to be different. At `// 2` we are setting our layout to be the *GridBagLayout*. At `// 3` and the lines of code following, we are setting some constraints and then adding the three buttons to the layout. Since there are so many constraints to set, it is often useful when working with *GridBagLayout* to create a helper function as we have done at `// 4`.

As can be seen, the actual mechanics of getting components displayed in a *GridBagLayout* are not too difficult. Understanding what you need to do to get a proper looking layout can be difficult since you will be mixing the different constraints.

It is important to remember that the behavior of components in a *GridBagLayout* change as other components are added with different constraints. This makes understanding what will happen given a set of constraints fairly difficult. This is because the constraints for a component in a row/column can affect all of other components in that row/colum. There are two examples of *GridBagLayout* code at http://www.ibm.com/java/education/javalman/javalman.html including a program called *GridBagLayoutTest.java*. This is a program which helps you understand how all of the different *GridBagConstraints* work together and affect one another in a layout. To understand *GridBagLayout*, I highly recommend people use and play with this program.

5 Tips for laying out a GridBagLayout

The following are some tips that I have found through my experience with *GridBagLayout* to be useful when attempting to code one.

5.1 Draw a Grid

I have found that one of the most useful techniques for creating the code for a particular view in *GridBagLayout* is to first draw the layout on paper. Then draw vertical and horizontal lines between your components. This will give you a picture of your grid so you know what `gridx`, `gridy`, `gridwidth`, and `gridheight` values to use. Figure 4 shows an example of this.

From Figure 4, you can see that it is easy to determine the grid positions and sizes. I use this technique when designing my own *GridBagLayout*s. The following listing contains the Java code that builds and displays the dialog in Figure 4. It uses the same `addToGridBag()` method used and defined in the previous listing.

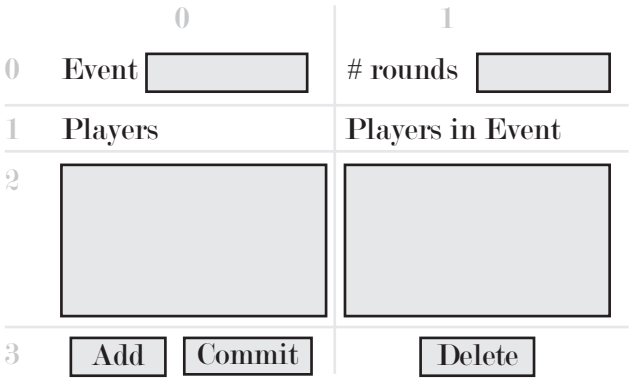

Figure 4: Lines drawn through a drawing of a dialog

```
class SomeDlg extends Dialog...
    // Use the GridBagLayout for this dialog.
    GridBagLayout gbl = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbl);

    Panel p = new Panel();
    p.add(lEvent);
    p.add(tfEvent);
    p.add(lNumRounds);
    p.add(tfNumRounds);

    addToGridBag(p, gbl, gbc, 0, 0, 2, 1, 1, 1);
    addToGridBag(lPlayers, gbl, gbc, 0, 1, 1, 1, 1, 1);
    addToGridBag(lPlayersInEvent, gbl, gbc, 1, 1, 1, 1, 1, 1);
```

```
addToGridBag(playersList, gbl, gbc, 0, 2, 1, 1, 1, 1);
addToGridBag(playersInEventList, gbl, gbc, 1, 2, 1, 1, 1, 1);

Panel p2 = new Panel();
p2.add(addButton);
p2.add(commitButton);
addToGridBag(p2, gbl, gbc, 0,3, 1, 1, 1, 1);

Panel p3 = new Panel();
p3.add(deleteButton);
addToGridBag(p3, gbl, gbc, 1, 3, 1, 1, 1, 1);
```

5.2 Start Small

When developing a *GridBagLayout* it is important to start small and build up. Don't write all of the code for the layout and then try to debug it when your layout doesnt work properly. In addition, I recommend to initially set all weights to 0, and only modify `gridx`, `gridy`, `gridwidth`, and `gridheight`. Once you have your layout looking somewhat like you want should you start modifing the other *GridBagConstraint* parameters. *GridBagLayout* is very complicated and it is much easier to deal with if you draw a picture of your layout as above, draw a grid between the items and try to code it to get it on the screen in parts. Smaller parts are always easier to debug than larger parts.

Be patient when working with the *GridBagLayout*. It sometimes takes many tries to get it right. It if very difficult to code one right the first time. The layout usually requires quite a bit of fiddling to get it to look right.

6 Summary

The *GridBagLayout* layout manager is the most powerful, yet difficult to use layout manager provided in JAVA. JAVA developers doing any GUI work need to know how to use this layout manager properly. Many people shy away from it because of its complexity. I have presented some information that properly explains all of the *GridBagConstraint* options and how they work. Hopefully, this information will save time when developing a *GridBagLayout*.

7 References

Core Java Cornell, Hortsmann

Graphic Java Geary, McClellan

Java 1.1 Developer's Handbook Heller, Roberts, Seymour, McGinn

8 About the Author

Peter Haggar is an Advisory Software Engineer with IBM in the NETWORK COMPUTING SOFTWARE division in Research Triangle Park, North Carolina.

Peter currently works on emerging JAVA and Internet technology with a focus on embedded JAVA and real-time operating systems. Peter is also writing a forthcoming book on JAVA to be published by Addison-Wesley. Peter has worked for IBM since 1987 and has been working with graphical user interface design and implementation since 1989 and object oriented development and technology since 1991. For the past 9 years, Peter has worked on multiple OO and GUI development projects. For 4 years he worked on the IBM OpenClass C++ class libraries for the VisualAge for C++ product. Before that he worked on many GUI controls for OS/2. Peter received a B.S. in Computer Science from Clarkson University in New York in 1987. Peter can be contacted at <haggar@us.ibm.com>.