OO for GUI Design (contd.)

Questions:

1. Why can't layout management be as simple as placing components at certain fixed pre-determined locations in a window?

2. What about placing components at locations relative to the size of the window?

3. AWT/Swing's BorderLayout manager recognizes how many locations for object placement?

4. What is the resizing behavior exhibited by AWT/Swing's FlowLayout manager?

5. To create spatial arrangements that are visually pleasing, BoxLayout allows us to insert varying degrees of empty space between components.

   What are the three different kinds of empty space one can have with BoxLayout?

6. True or false: The GridLayout manager allows you to have an arbitrary number of rows and an arbitrary number of columns in a layout?

7. Since AWT/Swing's CardLayout manager displays multiple components in the same space, what additional feature does it need to help the user select the component to be seen?

GridBagLayout:

`GridBagLayout` is the most versatile (and also the most difficult to master) of the layout managers that come with the Java platform.

Like `GridLayout`, it divides the display area into an array of cells. But, it goes way, way beyond what `GridLayout` can do.

`GridBagLayout` gives you the following in a row-column presentation of components:

- The display area assigned to a component can span multiple rows, multiple columns, or both multiple rows and multiple columns.

- Each row can be of a different height, and each column of a different width.

- As a window is resized, the resizing behavior of the display area allocated to each component in relation to that of the display areas assigned to the other components can be tuned on an individual basis.

- Within the display area allocated to each component, how the component expands or shrinks as a window is resized can again be controlled on an individual basis.

All of this is done with the help of the `GridBagConstraints` class whose various data members provide control over the various aspects of the display.

One first creates an instance of `GridBagContraints`, sets its data members to desired values, and then informs the `GridBagLayout` manager which component is to be impacted by those constraints.

Data members of `GridBagConstraints` that can be set to provide control over a display:

gridx, gridy These decide which cell in the array a component goes into.

gridwidth, gridheight These control the number of cells the display area assigned to a component can occupy in width and in height. The default value for each is 1.

fill This control the resizing behavior of a component within its allocated display area when the top-level window is resized.

ipadx, ipady Through these variables, you can make the rows of unequal width and the columns of unequal height. These, referred to as the padding variables, have a default value of 0.

insets This specifies the extent of the empty space between the boundary of the component and the edges of its display area.

anchor If a component is smaller than its allocated display area, this controls where in the display area the component will be placed.

weightx, weighty These have a major bearing on the resizing behavior of the display area assigned to a component as the top-level window is resized manually.

To use `GridBagConstraints`:

First, you'd create an instance of the layout manager and an instance of `GridBagConstraints`, as illustrated by

```
Container contentPane = frame.getContentPane();
GridBagLayout gridbag = new GridBagLayout();
contentPane.setLayout( gridbag );
GridBagConstraints constraints = new GridBagConstraints();
```

Now suppose you wanted all your components to fill out their assigned display areas and you also wanted the display areas to expand and shrink proportionately as the top-level window is resized, you could say at the very outset

```
cons.fill = GridBagConstraints.BOTH;
cons.weightx = 1.0;
cons.weighty = 1.0;
```

Now let's say you'd like to add a wider-than-usual `JButton` at the top-left corner of the display. You would say:

```
button = new JButton( "Button 1" );
constraints.gridx = 0;
constraints.gridy = 0;
constraints.ipadx = 100;
gridbag.setConstraints( button, constraints );
contentPane.add( button );
```

After using the `GridBagConstraints` object for one component, you have the option of using the same constraint object for the next component or creating a new constraint object.

If you use the same constraint object, you have to remember to reset the values of those constraints that were set previously.

```
button = new JButton( "Button 2" );
cons.gridx = 1;
cons.gridy = 0;
cons.ipadx = 0;
gridbag.setConstraints( button, cons );
contentPane.add( button );
```

Since, in relation to the previous button, we want this button to appear at its regular width, we reset `ipadx` to 0.

```
//GridBagLayoutDemo.java

import java.awt.*;             // for Container, BorderLayout
import java.awt.event.*;       // for WindowAdapter
import javax.swing.*;


public class GridBagLayoutDemo {

    public static void main( String[] args ) {

        JButton button;

        JFrame f = new JFrame( "GridBagLayoutDemo" );
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        Container contentPane = f.getContentPane();
        GridBagLayout gridbag = new GridBagLayout();
        contentPane.setLayout( gridbag );
        GridBagConstraints cons = new GridBagConstraints();
//        cons.fill = GridBagConstraints.HORIZONTAL;
//        cons.fill = GridBagConstraints.VERTICAL;
        cons.fill = GridBagConstraints.BOTH;
        cons.weightx = 1.0;
        cons.weighty = 1.0;

        // ROW 1:
        button = new JButton( "Button 1" );
        cons.gridx = 0;
        cons.gridy = 0;
        cons.ipadx = 100;
        gridbag.setConstraints( button, cons );
```

```
contentPane.add( button );

button = new JButton( "Button 2" );
cons.gridx = 1;
cons.gridy = 0;
cons.ipadx = 0;
gridbag.setConstraints( button, cons );
contentPane.add( button );

button = new JButton( "Button 3" );
cons.gridx = 2;
cons.gridy = 0;
gridbag.setConstraints( button, cons );
contentPane.add( button );

// ROW 2:
button = new JButton( "Button 4" );
cons.gridwidth = 2;
cons.gridx = 0;
cons.gridy = 1;
cons.ipady = 50;
gridbag.setConstraints( button, cons );
contentPane.add( button );

button = new JButton( "Button 5" );
cons.gridwidth = 1;
cons.gridx = 2;
cons.gridy = 1;
gridbag.setConstraints( button, cons );
contentPane.add( button );

// ROW 3:
button = new JButton( "Button 6" );
cons.gridwidth = 3;
cons.gridx = 0;
cons.gridy = 2;
```

```
        cons.ipady = 0;
        gridbag.setConstraints( button, cons );
        contentPane.add( button );

        f.pack();
        f.setLocation( 200, 300 );
        f.setVisible( true );
    }
}
```

Controlling the Spatial Layout in C++ Qt

QHBox
QHBoxLayout
QVBox
QVBoxLayout
QGrid
QGridLayout
QLayout

```
// hboxdemo.cc
#include <qpushbutton.h>
#include <qhbox.h>
#include <qapplication.h>


class MyHBox : public QHBox {
public:
    MyHBox();
};


MyHBox::MyHBox()
{
    setSpacing( 5 );
    setMargin( 10 );

    new QPushButton( "button1", this );
    new QPushButton( "button2", this );
    new QPushButton( "button3", this );
}

int main( int argc, char* argv[] )
{
    QApplication a( argc, argv );

    MyHBox* hb = new MyHBox();
    hb->show();
    a.setMainWidget( hb );

    return a.exec();
}
```

```
g++ -o hboxdemo hboxdemo.cc -I$QTDIR/include -L$QTDIR/lib -lqt
```

```
//hboxlayout.cc
#include <qdialog.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qapplication.h>


class MyDialog : public QDialog {
public:
    MyDialog();
};


MyDialog::MyDialog()
{
    QPushButton* b1 = new QPushButton( "button1", this );
    b1->setMinimumSize( b1->sizeHint() );
    QPushButton* b2 = new QPushButton( "button2", this );
    b2->setMinimumSize( b2->sizeHint() );
    QPushButton* b3 = new QPushButton( "button3", this );
    b3->setMinimumSize( b3->sizeHint() );

    QHBoxLayout* layout = new QHBoxLayout( this );
    layout->addWidget( b1 );
    layout->addWidget( b2 );
    layout->addWidget( b3 );
    layout->activate();
}


int main( int argc, char* argv[] )
{
    QApplication a( argc, argv );

    MyDialog* dlg = new MyDialog();
    dlg->show();
```

```
    a.setMainWidget( dlg );

    return a.exec();
}
```

```
void QBoxLayout::addWidget( QWidget* widget,
                            int stretch = 0,
                            int alignment = 0 );
```

```
#include <qpushbutton.h>
#include <qgrid.h>
#include <qapplication.h>


class MyGrid : public QGrid {
public:
    MyGrid( int );
};


MyGrid::MyGrid( int cols ) :  QGrid( cols )
{
    setSpacing( 5 );
    setMargin( 10 );

    new QPushButton( "button1", this );
    new QPushButton( "button2", this );
    new QPushButton( "button3", this );
    new QPushButton( "button4", this );
    new QPushButton( "button5", this );
}

int main( int argc, char* argv[] )
{
    QApplication a( argc, argv );

    MyGrid* mg = new MyGrid( 2 );
    mg->show();
    a.setMainWidget( mg );

    return a.exec();
}
```

```
#include <qdialog.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qapplication.h>


class MyDialog : public QDialog {
public:
    MyDialog();
};


MyDialog::MyDialog()
{
    QPushButton* b1 = new QPushButton( "button1", this );
    b1->setMinimumSize( b1->sizeHint() );
    QPushButton* b2 = new QPushButton( "button2", this );
    b2->setMinimumSize( b2->sizeHint() );
    QPushButton* b3 = new QPushButton( "button3", this );
    b3->setMinimumSize( b3->sizeHint() );
    QPushButton* b4 = new QPushButton( "button4", this );
    b4->setMinimumSize( b4->sizeHint() );

    QGridLayout* layout = new QGridLayout( this, 2, 3 );
    layout->addWidget( b1, 0, 0 );
    layout->addWidget( b2, 0, 1 );
    layout->addWidget( b3, 0, 2 );
    layout->addWidget( b4, 1, 1 );
    layout->activate();
}


int main( int argc, char* argv[] )
{
    QApplication a( argc, argv );
```

```
    MyDialog* dlg = new MyDialog();
    dlg->show();
    a.setMainWidget( dlg );

    return a.exec();
}
```

`GtkHbox`
`GtkVBox`
`GtkTable`

```
GtkWidget* gtk_hbox_new( gboolean homogeneous, gint spacing )
```

The second argument is the number of pixels of empty space around each child widget.

If `homogeneous` is set to TRUE, each child widget will occupy an equal amount of space horizontally.

This means that the amount of horizontal space alloted to a button with a short label will the same as to a button with a long label.

As to the sizes of the individual buttons (and how that size would depend on the length of the button label) and as to whether the button would expand or shink within its alloted space, those are controlled by by the third and the fourth arguments supplied to the `gtk_box_pack_start()` function that inserts a child widget into the layout.

The prototype of this function

```
void gtk_box_pack_start( GtkBox* box,
                         GtkWidget* child,
                         gboolean expand,
                         gboolean fill,
                         guint padding );
```

If the `expand` parameter is set to TRUE, the button would be made just large enough to fit the label (the label length has to exceed a threshold for this to be true).

If the `fill` parameter is set to TRUE, the button will occupy as much space as possible (taking into account the value assigned to `padding`) within the horizontal space alloted to the button.

```
#include <gnome.h>

gint eventDestroy( GtkWidget* widget, GdkEvent* event, gpointer data );


int main( int argc, char* argv[] )
{
    GtkWidget* window;
    GtkWidget* hbox;
    GtkWidget* button;

    gnome_init( "HorizBoxDemo", "1.0", argc, argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );

    gtk_signal_connect( GTK_OBJECT( window ),
                        "destroy",
                        GTK_SIGNAL_FUNC( eventDestroy ),
                        NULL );

    gtk_container_set_border_width( GTK_CONTAINER( window ), 25 );

    hbox = gtk_hbox_new( TRUE, 10 );

    button = gtk_button_new_with_label( "Hi" );
    gtk_box_pack_start( GTK_BOX( hbox ), button, FALSE, TRUE, 0 );
    button = gtk_button_new_with_label( "Hello" );
    gtk_box_pack_start( GTK_BOX( hbox ), button, TRUE, FALSE, 0 );
    button = gtk_button_new_with_label( "Hi There" );
    gtk_box_pack_start( GTK_BOX( hbox ), button, FALSE, FALSE, 0 );
    button = gtk_button_new_with_label( "Hello There" );
    gtk_box_pack_start( GTK_BOX( hbox ), button, TRUE, TRUE, 0 );

    gtk_container_add( GTK_CONTAINER( window ), hbox );

    gtk_widget_show_all( window );
```

```
    gtk_main();
    exit( 0 );
}


gint eventDestroy( GtkWidget* widget, GdkEvent* event, gpointer data ) {
    gtk_main_quit();
    return 0;
}
```

```
GtkWidget* gtk_table_new( guint rows,
                          guint columns,
                          gboolean homogeneous );
```

```
void gtk_table_attach_defaults( GtkTable* table,
                                GtkWidget* widget,
                                guint left_attach,
                                guint right_attach,
                                guint top_attach,
                                guint bottom_attach );
```

```
button = gtk_button_new_with_label( "Hello There" );
gtk_table_attach_defaults( GTK_TABLE( table ), button, 0, 3, 1, 2 );
```

```
 void gtk_table_attach( GtkTable* table,
                        GtkWidget* widget,
                        guint left_attach,
                        guint right_attach,
                        guint top_attach,
                        guint bottom_attach,
                        GtkAttachOptions xoptions,
                        GtkAttachOptions yoptions,
                        guint xpadding,
                        guint ypadding );
```

```
typedef enum {
  GTK_EXPAND = 1 << 0,
  GTK_SHRINK = 1 << 1,
  GTK_FILL   = 1 << 2,
} GtkAttachOptions
```

```
#include <gnome.h>

gint eventDestroy( GtkWidget* widget, GdkEvent* event, gpointer data );


int main( int argc, char* argv[] )
{
    GtkWidget* window;
    GtkWidget* table;
    GtkWidget* button;

    gnome_init( "TableDemo", "1.0", argc, argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );

    gtk_signal_connect( GTK_OBJECT( window ),
                        "destroy",
                        GTK_SIGNAL_FUNC( eventDestroy ),
                        NULL );

    gtk_container_set_border_width( GTK_CONTAINER( window ), 25 );

    table = gtk_table_new( 2, 3, TRUE );  //rows, cols, homogeneous

    button = gtk_button_new_with_label( "Hi" );
    gtk_table_attach_defaults( GTK_TABLE( table ), button, 0, 1, 0, 1 );

    button = gtk_button_new_with_label( "Hello" );
    gtk_table_attach_defaults( GTK_TABLE( table ), button, 1, 2, 0, 1 );

    button = gtk_button_new_with_label( "Hi There" );
    gtk_table_attach_defaults( GTK_TABLE( table ), button, 2, 3, 0, 1 );

    button = gtk_button_new_with_label( "Hello There" );
    gtk_table_attach_defaults( GTK_TABLE( table ), button, 0, 3, 1, 2 );
```

```
    gtk_container_add( GTK_CONTAINER( window ), table );

    gtk_widget_show_all( window );
    gtk_main();
    exit( 0 );
}


gint eventDestroy( GtkWidget* widget, GdkEvent* event, gpointer data ) {
    gtk_main_quit();
    return 0;
}
```