Get to Know GroupLayout

Dissecting the Matisse Core

Tomas Pavek
GroupLayout is a new layout manager that was developed as a Swing Labs project in conjunction with Matisse, the new GUI builder in NetBeans 5.0, and is now part of the Mustang project. Though the layout manager was originally designed to suit the GUI builder needs, it is also quite handy for manual coding. This article will help you get up to speed with how GroupLayout works and shows you how you can start building GUIs using it, whether you choose to use NetBeans GUI Builder or write your own code.

In this article, I'm going to cover some of the theory behind GroupLayout and create a complete example with detailed explanation.

**Design principle: independent dimensions**

The first thing you need to know about GroupLayout is that it works with horizontal and vertical layout separately. This is not that uncommon, but unlike other layout managers GroupLayout does not use a single constraints object or method to completely specify a component's layout; the layout is defined for each dimension independently.

This might seem a bit unusual at first, but it actually makes things easier because the definition is simpler. When defining the horizontal layout, you don't need to worry about the vertical dimension, and vice versa. The layout along the horizontal axis is quite independent of the layout along the vertical axis. By focusing just on one dimension at a time you only have to solve half the problem; the other dimension can be solved later. The downside of this approach is that each component needs to be defined twice in the layout. (You'll find out if you forgot to do this, because GroupLayout will generate an exception.)

This dimension independence is quite a powerful concept. It is similar to SpringLayout, because it provides flexibility other layouts can't offer. We'll get back to this topic later; but first let's see what makes GroupLayout different from SpringLayout and other layout managers.

**Layout organization: hierarchical groups**

GroupLayout uses two types of arrangements – sequential and parallel, combined with hierarchical composition. These principles are quite basic and well known from Swing.

1. With **sequential** arrangement, the components are simply placed one after another. Just likeBoxLayout or FlowLayout would do along one axis. The position of each component is defined as being relative to the preceding component. This is important for platform-independent layout.

2. The second way places the components in **parallel**, on top of each other in the same space, and aligned along a common reference point. For example, the components can be right-aligned along the horizontal axis, or baseline-aligned along the vertical axis, etc.

Usually, components placed in parallel in one dimension are in a sequence in the other, so they don't overlap. See the examples below.

What makes these two principles powerful is that they can be combined (nested) hierarchically. For this purpose GroupLayout defines **layout groups**. A group is either sequential or parallel and may contain components, gaps and other groups. The size of a sequential group is the sum of the sizes of the contained elements, and the size of a parallel group corresponds to the size of the largest element.

Defining a layout means defining how the components should be grouped by combining the sequential and parallel arrangements. This resembles nested panels with BoxLayout, but the groups are quite lightweight compared to panels. There is also a difference in the independent dimensions as described above. Panels do nesting in both dimensions at once, while groups can be nested as needed, for each dimension separately.
That’s enough theory for now. Let’s take a look at how it works in practice with a simple example.

**Simple example**

Let’s start with something really simple, just three components in a row, as shown in Figure 1.

We would like to express this layout using groups. Starting with the horizontal axis it’s easy to see there is a **sequential group** of three components arranged from left to right. Along the vertical axis there is a **parallel group** of the same three components (at the same coordinate); let’s say they are aligned along a baseline. See Figure 2.

In pseudo code, the layout specification might look like this (the real code is shown later):

```plaintext
horizontal layout = sequential group { c1, c2, c3 }  
vertical layout = parallel group (BASELINE) { c1, c2, c3 }
```

Note this illustrates a principle I mentioned earlier: components grouped sequentially in one dimension usually form a parallel group in the orthogonal dimension.

Now let’s add one more component, C4 (Figure 3). Along the horizontal axis the new component forms a parallel group with C3 (because it occupies the same horizontal space as C3), let’s say we want the components left aligned. Along the vertical axis C4 forms a sequential group with the original parallel group of the three components (see Figure 4).

In pseudo code, the layout specification now looks like this:

```plaintext
horizontal layout =  sequential group {  
  c1, c2,  
  parallel group (LEFT) { c3, c4 }  
}  
vertical layout = sequential group {  
  parallel group (BASELINE) { c1, c2, c3 }, c4 }
```

Now that you understand the principle of groups, you know the most important thing about designing layouts with ** GroupLayout**. There are some more details to explain, though: how to add gaps, how to define resize behavior, how to write real code, etc.

**Gaps**

A gap can be thought of as an invisible component of a certain size. Gaps of arbitrary size can be added to groups just like components or other groups. Using gaps you can precisely control the distance between components or from the container border.

**GroupLayout** also defines a symbolic `default gap` that corresponds to a **preferred distance** between neighboring components (or between a component and container border). The size of such a gap is not defined by an explicit number, but computed dynamically based on the look-and-feel the
Get to Know GroupLayout

application is using (the LayoutStyle class is used for this).

There are two advantages to using preferred gaps: you don’t have to specify the pixel sizes of the gaps, and they automatically adjust to the environment the UI runs in, reflecting the actual platform guidelines.

GroupLayout distinguishes between (a) the preferred gap between two components and (b) the preferred gap between a component and the container border. There are corresponding methods in the GroupLayout API for adding these gaps (addPreferredGap() and addContainerGap()). There are three types of component gaps: related, unrelated and indented (see Figure 5). The LayoutStyle class defines corresponding constants (to be used as the first parameter of addPreferredGap()): RELATED, UNRELATED and INDENT. The difference between related and unrelated gaps is just in size (the distance between unrelated components is a bit bigger). Indented represents a preferred horizontal distance of two components, where one of them is positioned underneath the second with an indent.

To make things easier, GroupLayout can insert gaps automatically. If you don’t add your own gaps explicitly, it adds the related preferred gaps for you. This is not the default behavior, you have to turn this feature on by invoking setAutocreateGaps(true) and setAutocreateContainerGaps(true) on the layout. Then you’ll get correct spacing almost for free!

How to write code

Now, let’s take a look at the actual code to create the layout described above. We’ll assume we have a container named panel and four components (c1, c2, c3, c4) which are already set up. First, we create a new GroupLayout object and associate it with the panel:

```java
GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);
```

We specify automatic gap insertion:

```java
layout.setAutocreateGaps(true);
layout.setAutocreateContainerGaps(true);
```

Finally, we define groups and add the components. We establish a root group for each dimension using setHorizontalGroup() and setVerticalGroup(). Groups are created via createSequentialGroup() and createParallelGroup(). Components are added to groups by using a variant of the add() method.

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(c1)
    .add(c2)
    .add(layout.createParallelGroup(GroupLayout.LEADING)
        .add(c3)
        .add(c4))
);
```

Figure 4
Left aligning components

Figure 5
Types of component gaps
```java
setLayoutVerticalGroup(layout.createSequentialGroup()
    .add(layout.createParallelGroup(GridLayout.BASELINE)
        .add(c1)
        .add(c2)
        .add(c3))
    .add(c4)
);
```

Note that default alignment must be specified for parallel groups. It can be one of the following constants defined in the `GridLayout` class: LEADING, TRAILING and CENTER. These constants are used for both dimensions; in the horizontal dimension LEADING means “left”, while in the vertical dimension it means “top”. Similarly TRAILING maps to “right” or “bottom”. The BASELINE alignment is valid only in the vertical dimension.

Some notes about the code:

- Components are not added to the container directly, they are added to groups. `GridLayout` adds the components to the container automatically for you.
- Note the chained calls of the `add()` method used to fill the groups. `add()` always returns the group on which it is called. Thanks to this you don’t need to use local variables to hold the groups.
- It is a good idea to indent the code so it is easy to see the hierarchical structure of the groups. Give each component a new line, add one level of indent for each new group in the hierarchy. A decent source editor will help you with pairing the parenthesis to close the `createXXXGroup()` methods. By following these simple rules it is easier to add a new component, or remove an existing one.

## Size of components and resizability

The size of each component in a `GridLayout` is constrained by three values; minimum size, preferred size and maximum size. These sizes control how the component resizes within the layout. The `GridLayout.add(....)` method allows the size constraints to be specified. There is no limit on the number of resizable components in a layout.

If not specified explicitly, the layout asks the component for its default sizes (by using the component’s `getMinimumSize()`, `getPreferredSize()` and `get MaximumSize()` methods). Thus you don’t need to specify anything for most of the components, e.g. to make `JTextField` resizable or `JButton` fixed, because the components have the desired resizing behavior as default. On the other hand, you can override the default behavior. For example you can make a `JTextField` fixed or `JButton` resizable.

`GridLayout` defines constants that provide precise control over resize behavior. They can be used as parameters in the `add(Component comp, int min, int pref, int max)` method. Here are two examples:

1. To force a component to be resizable (allow shrinking and growing):

   ```java
   group.add(component, 0, GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE) ...
   ```

   This allows the component to resize between zero size (minimum) to any size (Short.MAX_VALUE as maximum size means “infinite”). If we wanted the component not to shrink below its default minimum size, we’d use `GroupLayout.DEFAULT_SIZE` instead of 0 in the second parameter.

2. To make a component have a fixed size (suppress resizing):

   ```java
   group.add(component, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE) ...
   ```

   In these examples the initial size of the component is not altered; its default size is the component’s preferred size. If we wanted a specific size for the
component, we would specify it in the second parameter instead of using GroupLayout.DEFAULT_SIZE.

Resizable gaps

Specifying size and resizability applies to gaps as well, including the preferred ones. For example, you can specify a preferred gap between two components that acts like a spring pushing the components away from each other (to the opposite sides of the container). The preferred distance of the two components is only used as the minimum size of the gap. See the following snippet:

```java
layout.createSequentialGroup()
    .add(c1)
    .addPreferredGap.LayoutStyle.RELATED,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
    .add(c2);
```

Justified layout

Resizable elements placed in a parallel group are stretched to fill the space of the group determined by the largest element in the group; so they end up aligned with the same size. GroupLayout also provides control over whether the enclosing parallel group itself should resize. If group resizing is suppressed, it prevents the contained elements from growing over the preferred size of the group. This way you can make a block of components align on both sides, or constrain individual components to have the same size.

Let’s try to achieve the same size for two components from our example (c3 and c4 in the horizontal dimension):

```java
layout.createParallelGroup(GroupLayout.LEADING, false)
    .add(c3, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
    .add(c4, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE);
```

The underlying mechanism works as follows:

1. The size of the parallel group is set to the preferred size of the largest element; so to the preferred size of c4 in our example.
2. Resizable elements are stretched to the size of the group. In our example, only c3 is effectively stretched, the size of c4 already corresponds to the size of the group.

As a result, c3 and c4 would have the same width. The components would not resize further because the parallel group itself is not resizable (see Figure 6).

A question for attentive readers: Why do we define both components in the parallel group as resizable in this example? It seems enough to have just c3 resizable since c4 is not stretched anyway...

The answer is simple: because of platform and localization independence. Otherwise we would have to rely on the fact that the c4 component is always bigger than c3. But this may change when the application runs on a different platform or is translated to another language. By having both components resizing, they adjust to each other, no matter which one is bigger at the moment.

Same component size

The previous case is special because the components are in the same parallel group. But what if we wanted unrelated components to have the same size? Clearly, the same size can’t always be ensured by grouping. The OK and Cancel buttons in a row at the bottom of a dialog are a good example. For this purpose GroupLayout provides a linkSize method. This method allows the size of arbitrary components to be linked, regardless of where they are placed. The resulting size of...
the linked components is set according to the largest component. For example:

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(label)
    .add(layout.createParallelGroup(GroupLayout.LEADING)
        .add(textField)
        .add(layout.createSequentialGroup()
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(caseCheckBox)
                .add(wholeCheckBox))
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(wrapCheckBox)
                .add(backCheckBox))))
    .add(layout.createParallelGroup(GroupLayout.LEADING)));
```

Note that in this example the size is linked selectively for the horizontal dimension.

Next I will show how to create a layout for a sample dialog, with detailed explanation of each step, illustrations and code samples.

**A real example**

Having provided the theory behind GroupLayout, now it's time to try it on a real example. So let's create a layout for a sample dialog, shown in Figure 7.

A good way to analyze the layout is to use top-down decomposition. I'll describe the procedure that proved useful to me; step by step to make it clear. Once you get used to “group thinking”, you'll be able to proceed much faster.

**Horizontal layout**

Examining the horizontal dimension from left to right, we can see there are three groups in a sequence. The first one is actually not a group, just a component – the label. The second one is a group containing the text field and the checkboxes (we'll decompose it later). And the third is a group of the two buttons. As illustrated in Figure 8.

Let's sketch out the sequential group in code. Note that GroupLayout.LEADING corresponds to left alignment in the horizontal dimension. Also note we don't specify gaps, assuming the gap auto-insertion feature is turned on.

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(label)
    .add(layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createParallelGroup(GroupLayout.LEADING)));
```

Now let's decompose the group in the middle. This is the hardest one. There's a text field in parallel with a sequence of two parallel groups each containing two checkboxes. See Figure 9.

Let's add the corresponding code:

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(label)
    .add(layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createParallelGroup(GroupLayout.LEADING)
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(txtField)
                .add(layout.createParallelGroup(GroupLayout.LEADING)
                    .add(layout.createParallelGroup(GroupLayout.LEADING)
                        .add(caseCheckBox)
                        .add(wholeCheckBox))))
            .add(layout.createParallelGroup(GROUP_LAYOUT.LEADING)
                .add(layout.createParallelGroup(GroupLayout.LEADING)
                    .add(wrapCheckBox)
                    .add(backCheckBox))));
```

Note we want the text field to be resizable, but that happens automatically since JTextField returns the right maximum size by default.

The remaining group on the right is trivial: it contains just two buttons. We don't even need the picture. Here's the code:
Get to Know GroupLayout

```java
layout.setVerticalGroup(layout.createParallelGroup(GroupLayout.BASELINE)
    .add(layout.createSequentialGroup()
        .add(layout.createParallelGroup(GroupLayout.BASELINE)
            .add(label)
            .add(textField)
            .add(findButton))
    .add(layout.createParallelGroup(GroupLayout.LEADING));
```

Now let's look at the bottom group. Note the Cancel button is not on a shared baseline with the checkboxes; it is aligned at the top. So the second parallel group comprises the button and a sequential group of two baseline groups with checkboxes (see Figure 11).

The corresponding code looks as follows:

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(layout.createParallelGroup(GroupLayout.BASELINE)
        .add(label)
        .add(textField)
        .add(findButton))
    .add(layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .add(layout.createParallelGroup(GroupLayout.BASELINE)
                .add(caseCheckBox)
                .add(wrapCheckBox))
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(caseCheckBox)
                .add(wholeCheckBox)
                .add(backCheckBox)))
    .add(cancelButton));
```

And finally we'd like the buttons to have always the same size, so let's link them:

```java
layout.linkSize(new Component[] {findButton, cancelButton},
    GroupLayout.HORIZONTAL);
```

Now we are done with the horizontal dimension. Let's switch to the vertical dimension. From now we'll only need to think about the y axis.

**Vertical layout**

In the vertical dimension, we examine the layout from top to bottom. We definitely want all the components on the first line aligned on the baseline. So along the vertical axis there is a sequence of the baseline group, followed by a group of the remaining components. See Figure 10.

Let's sketch out the code. First, we need to define two parallel groups. Note that GroupLayout.LEADING corresponds to the top alignment in the vertical dimension.

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(layout.createParallelGroup(
        GroupLayout.LEADING)
    .add(label)
    .add(textField)
    .add(findButton))
    .add(layout.createParallelGroup(
        GroupLayout.LEADING)
    .add(layout.createParallelGroup(
        GroupLayout.LEADING)
    .add(caseCheckBox)
    .add(wholeCheckBox))
    .add(layout.createParallelGroup(
        GroupLayout.LEADING)
    .add(caseCheckBox)
    .add(wholeCheckBox))
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE))
    .add(cancelButton));
```

We can fill the baseline group right away:

```java
layout.setHeightGroup(layout.createSequentialGroup()
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(label)
    .add(textField)
    .add(findButton))
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(caseCheckBox)
    .add(wholeCheckBox))
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(caseCheckBox)
    .add(wholeCheckBox))
    .add(cancelButton));
```

Now let's look at the bottom group. Note the Cancel button is not on a shared baseline with the checkboxes; it is aligned at the top. So the second parallel group comprises the button and a sequential group of two baseline groups with checkboxes (see Figure 11).

The corresponding code looks as follows:

```java
layout.setVerticalGroup(layout.createSequentialGroup()
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(label)
    .add(textField)
    .add(findButton))
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(caseCheckBox)
    .add(wholeCheckBox))
    .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
    .add(wholeCheckBox))
    .add(cancelButton));
```

So, that's it. We've created a complete layout including resize behavior without specifying a single number in pixels. That's a true cross platform layout! Note that we don't need to specify gaps between components; we get correct spacing automatically and according to the look-
import java.awt.Component;
import javax.swing.*;
import org.jdesktop.layout.*;

public class Find extends JFrame {
    public Find() {
        JLabel label = new JLabel("Find What:");
        JTextField textField = new JTextField();
        JCheckBox caseCheckBox =
            new JCheckBox("Match Case");
        JCheckBox wrapCheckBox =
            new JCheckBox("Wrap Around");
        JCheckBox wholeCheckBox =
            new JCheckBox("Whole Words");
        JCheckBox backCheckBox =
            new JCheckBox("Search Backwards");
        JButton findButton = new JButton("Find");
        JButton cancelButton = new JButton("Cancel");
        // remove redundant default border of
        // check boxes - they would hinder
        // correct spacing and aligning (maybe
        // not needed on some look-and-feels)
        caseCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        wrapCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        wholeCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        backCheckBox.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 0));
        GroupLayout layout = new GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setAutocreateGaps(true);
        layout.setAutocreateContainerGaps(true);
        layout.setHorizontalGroup(
            layout.createSequentialGroup().
                add(label).
                add(layout.createParallelGroup(GroupLayout.LEADING).
                    add(textField).
                    add(layout.createSequentialGroup().
                        add(caseCheckBox).
                        add(wholeCheckBox)).
                    add(layout.createParallelGroup(GroupLayout.LEADING).
                        add(wrapCheckBox).
                        add(backCheckBox)).
                add(findButton)).
            layout.createParallelGroup(GroupLayout.LEADING).
                add(findButton).add(cancelButton));
        layout.setVerticalGroup(
            layout.createSequentialGroup().
                add(layout.createParallelGroup(GroupLayout.BASELINE).
                    add(label).add(textField).add(findButton)).
                add(layout.createParallelGroup(GroupLayout.LEADING).
                    add(layout.createParallelGroup(GroupLayout.BASELINE).
                        add(caseCheckBox).add(wrapCheckBox)).
                    add(layout.createParallelGroup(GroupLayout.BASELINE).
                        add(wholeCheckBox).add(backCheckBox))).
                add(cancelButton));
        setTitle("Find");
        pack();
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(
            new Runnable() {
                public void run() {
                    try {
                        UIManager.setLookAndFeel(UIManager.
                            getSystemLookAndFeelClassName());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                    new Find().setVisible(true);
                }
            });
    }
}

Conclusions

Now you should know enough about GroupLayout to start using it! The easiest way to test GroupLayout is to use it with NetBeans 5.0 where it is bundled (just make sure “Swing Layout Extensions” library is added in the libraries of the working project). The layout manager and related extensions are hosted on swing-layout.dev.java.net as one of the Swing Labs projects.

And-feel guidelines. Listing 1 shows the complete code for the Find dialog (it is available for download in the magazine’s website).

You can compile this file and run it against the Swing Layout Extensions library. Try resizing the dialog horizontally to see how the layout automatically adjusts to the new size.

Tomas Pavek (tomas.pavek@sun.com) is a software engineer at Sun Microsystems who has been working on the NetBeans IDE since 2000. He is the lead developer of the GUI builder in NetBeans, now primarily focused on the project Matisse. Tomas lives in Prague, Czech Republic.
NetBeans is Everywhere at JavaOne 2006

Don’t miss out on Labs, Sessions and BOFs covering IDE features, Matisse, Plug-in Modules, Wireless Development, Enterprise pack, success cases and more.

<table>
<thead>
<tr>
<th>NetBeans-Related Hands-On Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>May 16</strong></td>
</tr>
<tr>
<td>2:00 PM LAB-8105 NetBeans Enterprise Pack: Basic UML Modeling</td>
</tr>
<tr>
<td><strong>May 17</strong></td>
</tr>
<tr>
<td>2:45 PM LAB-5250 Building Production Quality GUI Applications using NetBeans 5.0 GUI Builder (Matisse)</td>
</tr>
<tr>
<td><strong>May 18</strong></td>
</tr>
<tr>
<td>9:45 AM LAB-4255 AJAX: Riding the Web Application Horse a Little Further</td>
</tr>
<tr>
<td>11:30 AM LAB-5106 NetBeans 5.0: Plug-in Development</td>
</tr>
<tr>
<td>2:45 PM LAB-8125 NetBeans Enterprise Pack: BPEL</td>
</tr>
<tr>
<td><strong>May 19</strong></td>
</tr>
<tr>
<td>10:45 AM LAB-5120 Application Monitoring and Management with NetBeans 5.0</td>
</tr>
<tr>
<td>3:45 PM LAB-6205 Mobile Applications: Visualize it, Build it, Deploy it to Many Devices</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NetBeans-Related Sessions and BOFs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>May 16</strong></td>
</tr>
<tr>
<td>5:45 PM TS-3361 Java EE 5 Platform: Even Easier With Tools</td>
</tr>
<tr>
<td>9:30 PM BOF-2496 Building Development Tools on Top of the NetBeans IDE</td>
</tr>
<tr>
<td>10:30 PM BOF-0678 Meet the “Java Posse”</td>
</tr>
<tr>
<td><strong>May 17</strong></td>
</tr>
<tr>
<td>11:00 AM TS-4255 MHP/OCAP iTV Applications In a Nutshell</td>
</tr>
<tr>
<td>12:15 PM TS-4916 Creating Professional Swing UIs Using Matisse GUI Builder</td>
</tr>
<tr>
<td>1:30 PM TS-1293 Best Practices for Building Optimized Wireless Solutions for Web Services</td>
</tr>
<tr>
<td>2:45 PM TS-4589 Good Morning, Buenos Dias, Dobry Den: Mobile Internationalization in Action</td>
</tr>
<tr>
<td>8:30 PM BOF-2340 Creating NetBeans Plug-ins for Integration With JavaServer Faces, Hibernate, Spring, and EJB™ 3.0 Technology</td>
</tr>
<tr>
<td>9:45 PM TS-1387 Twelve Reasons to Use NetBeans Software: Episode 2</td>
</tr>
<tr>
<td><strong>May 18</strong></td>
</tr>
<tr>
<td>9:45 AM TS-1278 Creating and Deploying Custom Jackpot Queries and Transformers</td>
</tr>
<tr>
<td>1:30 PM TS-1878 Debugging Across Tiers: Advanced Techniques</td>
</tr>
<tr>
<td>2:45 PM TS-1549 Debugging and Profiling J2EE /Java EE 5 Platform-Based Applications</td>
</tr>
<tr>
<td>7:30 PM BOF-2559 Discovery and Dependency Injection Patterns in Modular Architectures</td>
</tr>
<tr>
<td>7:30 PM BOF-2807 Java Persistence API in the NetBeans IDE</td>
</tr>
<tr>
<td>8:30 PM BOF-2417 Memory Leaks in Java Technology-Based Applications: Different Tools for Different Types of Leaks</td>
</tr>
<tr>
<td>9:30 PM BOF-2835 Creating an IDE for your favorite web framework</td>
</tr>
<tr>
<td>9:30 PM BOF-2461 Use the Tools to Generate Mobile Business!</td>
</tr>
<tr>
<td>10:30 PM BOF-2031 Experience Self-Organized and Ad Hoc Collaborative Development With the NetBeans IDE</td>
</tr>
</tbody>
</table>