

The NestedSwinger Manual (Version 1)

Andrew Davison
Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112
Thailand
ad@fivedots.coe.psu.ac.th

1. Introduction

NestedSwinger translates a simple GUI textual notation into a Java application. The notation is considerably easier to write than Java GUI code since it hides details about GUI component initialization, layout managers, and listener code. It does this by treating a GUI as a series of widgets inside nested containers.

The generated Java code is simple to read and change because the NestedSwinger translation restricts itself to using basic GUI controls, standard layout managers, simple listener methods, all formatted to be human-readable.

NestedSwinger isn't the last word in automatic GUI creation. It's aimed at quickly prototyping fully functioning GUIs employed by novice or intermediate Java users. In particular, the programmer will almost certainly need to edit the resulting listener code to add or change features.

2. An Example

Let's assume that we need to create a GUI application for controlling a printer, which has to look something like Figure 1.

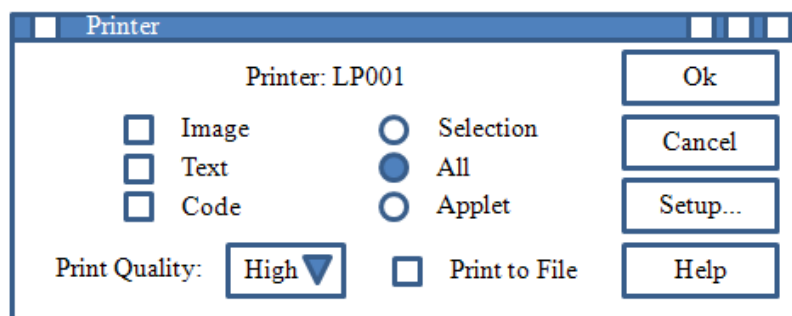


Figure 1. A Printer Application.

The first step is to identify the Swing controls (widgets), as shown in Figure 2.

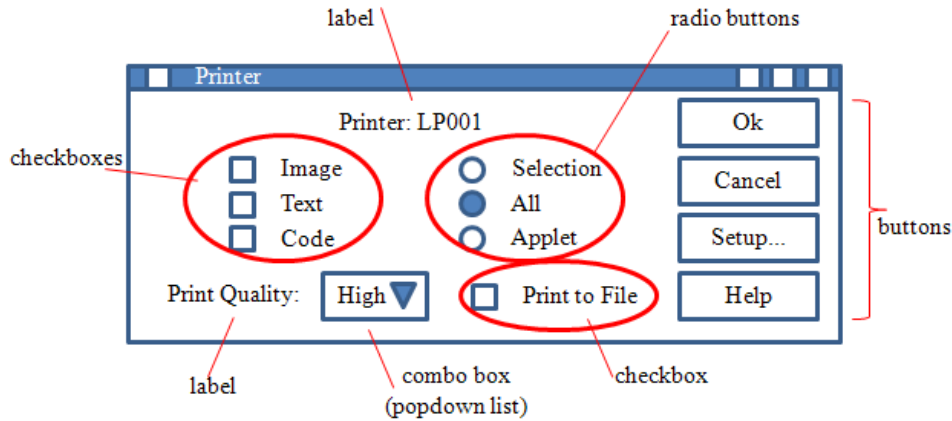


Figure 2. Highlighted Swing Controls in the Printer Application.

The next step is to start grouping the widgets into containers. A container is a group of widgets (or other containers) organized using one of five layout managers. These are: flow, border, grid, horizontal box, and vertical box, as illustrated in Figure 3.

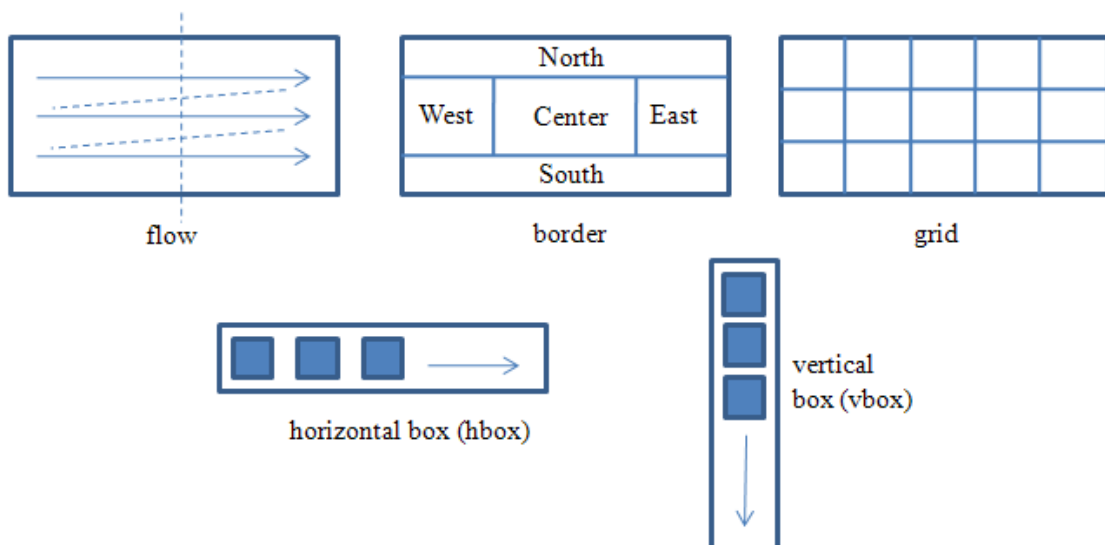


Figure 3. The Five Containers.

These correspond to the Java layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, `horizontal BoxLayout`, and `vertical BoxLayout`.

The widgets in Figure 2 must be collected into containers. One possible initial grouping is shown in Figure 4.

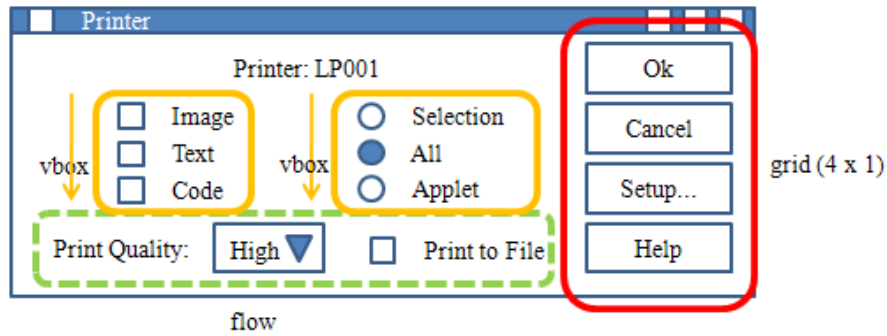


Figure 4. Stage 1 of Grouping the Widgets.

The three checkboxes and three radio buttons have been placed inside vertical box containers (vboxes), the four buttons in a 4 x 1 grid container, and the bottom row of widgets inside a flow container.

We're not done yet, since we have to keep grouping containers and widgets until the GUI is represented by a single container. A possible second stage is to group the two vboxes into their own flow container, resulting in Figure 5. The widgets already inside containers have been left out of Figure 5 to make it easier to understand.

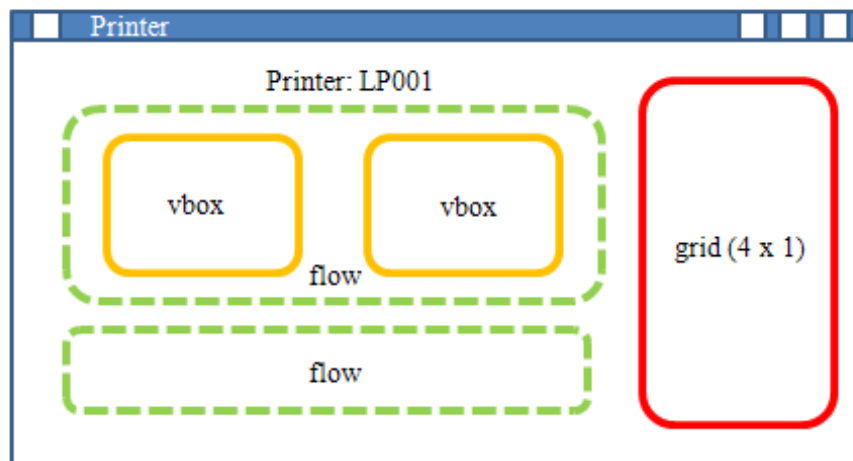


Figure 5. Stage 2 of Grouping the Widgets and Containers.

Stage 3 gathers up the printer label and the two flow containers into a vertical box, as in Figure 6.

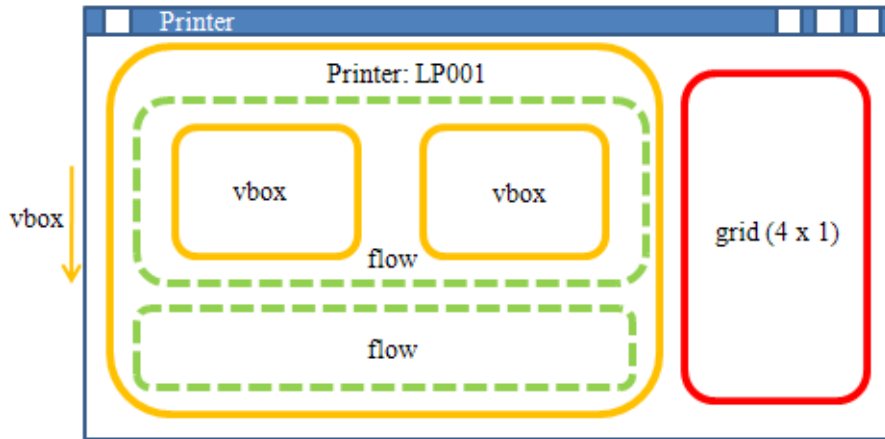


Figure 6. Stage 3 of Grouping the Widgets and Containers.

The application now consists of two containers – a vbox on the left and a grid on the right. These can be placed inside a single container in various ways. One approach is to use a border container, with the vbox in the center, and the grid to the east, as in Figure 7.

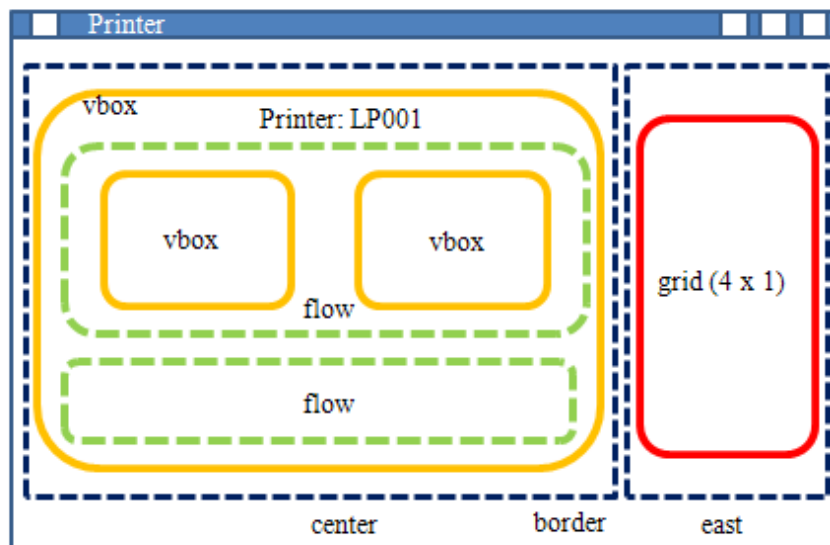


Figure 7. Final Stage of Grouping the Widgets and Containers.

This nesting of containers and widgets can now be written down using NestedSwinger notation, as shown below:

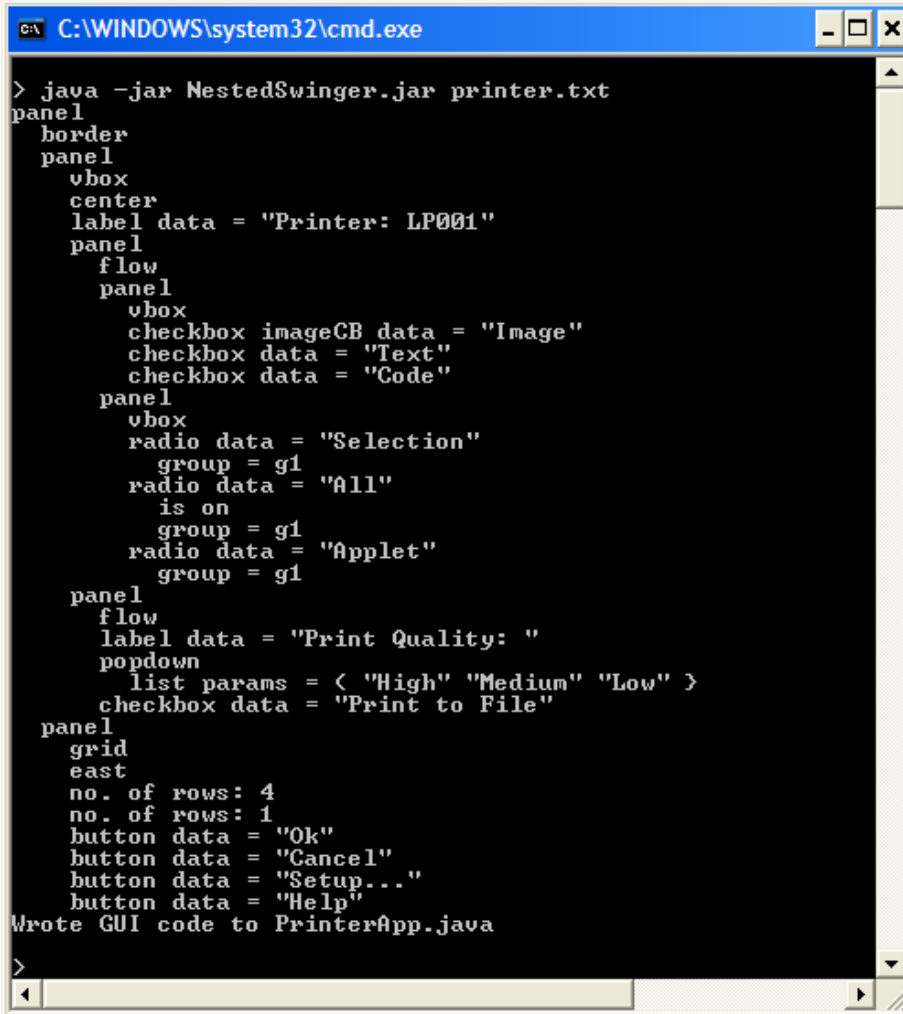
```
border {
  center: vbox {
    label "Printer: LP001"

    flow {
      vbox {
        checkbox "Image"
      }
    }
  }
}
```

```
        checkbox "Text "  
        checkbox "Code "  
    }  
    vbox {  
        radio "Selection"  group=g1  
        radio "All" on     group=g1  
        radio "Applet"    group=g1  
    }  
}  
  
flow {  
    label "Print Quality: "  
    popdown { "High" "Medium" "Low" }  
    checkbox "Print to File"  
}  
}  
  
east: grid 4 1 {  
    button "Ok"  
    button "Cancel"  
    button "Setup..."  
    button "Help"  
}  
}
```

I'll explain the details of the syntax in the next section, but the mapping from Figure 7 to the code should be fairly obvious. The containers are border, flow, two vboxes, and grid, with their nesting represented by { ... } blocks. The widgets inside the containers are labels, radio buttons, checkboxes, popdown, and buttons.

This text is stored in printer.txt, and passed to the NestedSwinger translator, which is shown in action in Figure 8.



```
C:\WINDOWS\system32\cmd.exe
> java -jar NestedSwinger.jar printer.txt
panel
  border
  panel
    vbox
      center
        label data = "Printer: LP001"
      panel
        flow
          panel
            vbox
              checkbox imageCB data = "Image"
              checkbox data = "Text"
              checkbox data = "Code"
            panel
              vbox
                radio data = "Selection"
                  group = g1
                radio data = "All"
                  is on
                  group = g1
                radio data = "Applet"
                  group = g1
            panel
              flow
                label data = "Print Quality: "
                popdown
                  list params = < "High" "Medium" "Low" >
                  checkbox data = "Print to File"
          panel
            grid
              east
                no. of rows: 4
                no. of rows: 1
                button data = "Ok"
                button data = "Cancel"
                button data = "Setup..."
                button data = "Help"
        Wrote GUI code to PrinterApp.java
  >
```

Figure 8. Calling NestedSwinger with printer.txt

The processing includes a print-out of the parse tree, and the Java code is saved into PrinterApp.java.

PrinterApp.java is a complete Java application, which includes simple listeners for the checkboxes, radio buttons, popdown list, and buttons. Since only standard Java controls and layouts are used, the application can be compiled and executed without the addition of any third-party libraries, as in Figure 9.

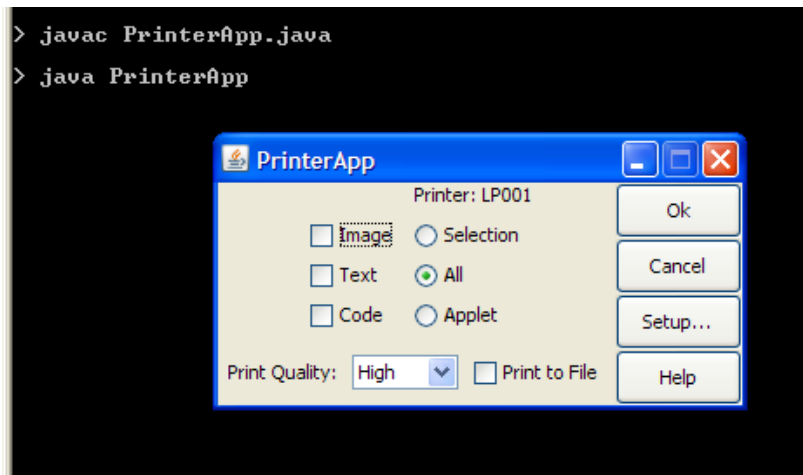


Figure 9 PrinterApp Executing.

Clicking on the controls causes their listeners to print simple messages to stdout. Figure 10 shows some typical output.

```

> javac PrinterApp.java
> java PrinterApp
Selected "Image"
Deselected "All"
Selected "Applet"
Selected "Medium"
Pressed "Setup..."
Pressed "Ok"
Pressed "Help"

```

Figure 10. Listener Output from PrinterApp.

The PrinterApp Code

NestedSwinger is a GUI prototyping tool, and the resulting code, while fully functional, will probably not be ideal. In particular, the programmer will need to edit PrinterApp.java to make its listeners more useful. This is not a difficult task since the generated code is neatly formatted, and only uses standard anonymous listeners.

PrinterApp.java is a JFrame, which calls makeGUI() to create the GUI:

```

public PrinterApp()
{
    super("PrinterApp");
    makeGUI();

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    setResizable(false);
    setLocationRelativeTo(null); // center the window
    setVisible(true);
}

```

```
} // end of PrinterApp()
```

Each container in NestedSwinger notation becomes a JPanel with an associated layout manager. For example, the first two levels of printer.txt are:

```
border {
  center: vbox {
    : // more notation
  }
}
```

In makeGUI(), they are represented by two JPanels:

```
JPanel panel__0 = new JPanel();
panel__0.setLayout( new BorderLayout());

JPanel panel__1 = new JPanel();
panel__1.setLayout( new BoxLayout(panel__1, BoxLayout.Y_AXIS));

: // more Java code

panel__0.add(panel__1, BorderLayout.CENTER);
getContentPane().add(panel__0);
```

Note that the top-most panel (the border container) is added to the content pane for the JFrame.

Each NestedSwinger widget is mapped to a Swing component, usually with the addition of a simple anonymous listener. For example, the checkbox line in printer.txt:

```
checkbox "Image"
```

is translated to the following code in makeGUI():

```
JCheckBox checkbox__5 = new JCheckBox("Image", false);
panel__4.add(checkbox__5);
checkbox__5.addItemListener( new ItemListener() {
  public void itemStateChanged(ItemEvent e)
  { if (e.getStateChange() == ItemEvent.SELECTED)
    System.out.println("Selected \"Image\" ");
    else
    System.out.println("Deselected \"Image\" ");
  }
});
```

Since the checkbox was declared in printer.txt without a name, NestedSwinger generates one itself (checkbox__5). It's also possible to name widgets, like so:

```
checkbox = imageCB "Image"
```

The resulting Java code will use the variable name imageCB. For instance:

```
JCheckBox imageCB = new JCheckBox("Image", false);
panel__4.add(imageCB);
```



```
imageCB.addItemListener( new ItemListener() { . . . } )
```

The code added to `makeGUI()` uses only locally defined variables (e.g. the `imageCB` `JCheckBox` is not a global). A common programmer change to generated listener code is to add references to other GUI elements, such as setting the text in a textfield when the checkbox is selected. This will require the movement of the definitions of those variables to a global scope, so they can be accessed inside the listener methods.

3. The NestedSwinger Syntax

I'll use a BNF grammar to define the NestedSwinger syntax: uppercase words are non-terminals, lowercase words and quoted symbols are terminals. The starting non-terminal is `CONTAINER`.

```
CONTAINER ::= LAYOUT [ edged [ STRING ] ]
            '{' ( [BORDER_POS] ( CONTAINER | WIDGET ))+ '}'

LAYOUT ::= flow | grid rowNo colNo | border | vbox | hbox

BORDER_POS ::= ((north | east | south | west | center) ':' )

WIDGET ::= LABEL | BUTTON | RADIO | CHECKBOX | TEXTFIELD | PASSWORD |
          TEXTAREA | POPDOWN | LIST | SPINNER | SLIDER | SPACE |
          CANVAS

LABEL ::= label [ '=' NAME ] STRING
BUTTON ::= button [ '=' NAME ] STRING
RADIO ::= radio [ '=' NAME ] STRING [ on ] [ group '=' NAME ]
CHECKBOX ::= checkbox [ '=' NAME ] STRING [ on ]
TEXTFIELD ::= textfield [ COLS ] [ '=' NAME ] STRING
PASSWORD ::= password [ COLS ] [ '=' NAME ]
TEXTAREA ::= textarea [ROWS COLS] [ '=' NAME ] STRING
POPDOWN ::= popdown [ '=' NAME ] '{' STRING+ '}'
LIST ::= list [ '=' NAME ] '{' STRING+ '}'
SPINNER ::= spinner [ '=' NAME ] '{' STRING+ '}'
SLIDER ::= slider [up] [ '=' NAME ]
SPACE ::= space [ NUM_SPACES ]
CANVAS ::= canvas [ROWS COLS] [ '=' NAME ]

COLS ::= INTEGER
ROWS ::= INTEGER
NUM_SPACES ::= INTEGER
```

I haven't defined the meaning of `NAME`, `STRING`, and `INTEGER`, which are standard.

The only recursive rule is `CONTAINER`, which groups other containers and widgets, and includes a `LAYOUT` definition.

The mapping from widgets to Swing components is fairly obvious in most cases, but is summarized in Table 1.

Widget	Swing Component
label	JLabel
button	JButton
radio	JRadioButton <i>and</i> ButtonGroup
checkbox	JCheckBox
textfield	JTextField
password	JPasswordField
textarea	JTextArea inside a JScrollPane
popdown	JComboBox
list	JList
spinner	JSpinner
slider	horizontal or vertical JSlider
space	horizontal or vertical Box strut
canvas	CanvasPanel, a subclass of JPanel

Table 1. NestedSwinger Widgets and Their Equivalent Swing Components.

The choice of widgets in NestedSwinger shows that it's intended for novice or intermediate Java users, who don't require more complex components such as JTable or JTree. Several basic components, such as JMenu and JProgressBar, are also not present, mainly because there seemed no simple widget equivalent. This highlights one aim of NestedSwinger which is to offer a *simplified* GUI interface, leaving out many Swing options. For example, there's no way inside NestedSwinger to add labels or tick marks to a slider (see Example 5 below).

Listener code is generated for all widgets except labels and spaces, and use standard interfaces such as ActionListener and ItemListener.

radio: The radio widget may have an optional `group` argument, as in:

```
radio "Selection" group=g1
```

This will be translated into a JRadioButton *and* a ButtonGroup :

```
JRadioButton radio__8 = new JRadioButton("Selection", false);
ButtonGroup g1 = new ButtonGroup();
g1.add(radio__8);
```

NestedSwinger keeps a record of group names, and so any future use of group `g1` will be translated into a call to `g1.add()`.

space: The space widget can be translated into a horizontal or vertical Swing Box strut. The choice of orientation is out of the programmer's hands, depending instead on the layout of the enclosing container. If the layout is flow, border, grid, or hbox, then a horizontal strut is created; if the layout is vbox then the space is translated into a vertical strut (see Example 2 below).

canvas: The canvas widget is translated into a specialized JPanel called CanvasPanel which catches mouse drag events and draws them onto the panel at each repaint. It also reports mouse presses and releases. and has a white background and blue border.

For example, `test10.txt` uses a canvas:

```
border {
  west : vbox {
    button "pong"
    button "ping"
  }
  center: canvas
}
```

It is translated into the Test10App application shown in Figure 11.



Figure 11. Test10App: a Canvas Widget Example.

As the user drags the cursor over the canvas, a series of black dots are drawn onto it.

CanvasPanel is an extension of JPanel which stores the cursor points in an array, and draws them onto the panel at each repaint.

The canvas widget shows how it's possible to include 'new' widgets in NestedSwinger which have no immediate equivalent in Swing.

4. More Examples

The following five examples illustrate various aspects of the NestedSwinger syntax, and include screenshots of their equivalent Java applications.

Example 1: Widget Names and Comments

test1.txt uses widget names and comments:

```
// test1.txt

border {
  north: label = j1 "enter name:"
  center: button = b1 "press me"
  south: radio "hi there" on // group=g1
}
```

The generated code for the label and button use their names:

```
// in makeGUI()
JPanel panel__0 = new JPanel();
panel__0.setLayout( new BorderLayout());

JLabel j1 = new JLabel("enter name:");
panel__0.add(j1, BorderLayout.NORTH);

JButton b1 = new JButton("press me");
panel__0.add(b1, BorderLayout.CENTER);
b1.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e)
  { System.out.println("Pressed \"press me\" "); }
});
```

Test1App.java is shown executing in Figure 12.

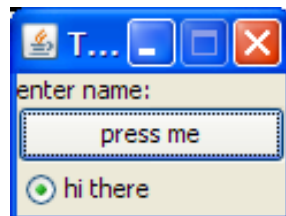


Figure 12. Test1App Executing.

Example 2: Image Label, Labeled Edges, and Spacing

test3.txt employs a label widget with a filename, which at execution time is treated as an image file to be loaded into the corresponding JLabel. test3.txt also uses a container edge label and two kinds of space.

```
border {
  north: label "duke.png"
  center:
    vbox edged "Alignment" {
      hbox {
        label "Horizontal"
        space
        popdown { "left" "center" "right" }
      }
      space 40
      hbox {
```

```

        label "Vertical"
        space
        popdown { "top" "center" "bottom" }
    }
}

```

The label translation uses an ImageIcon instance:

```
JLabel label__1 = new JLabel( new ImageIcon("duke.png"));
```

The edged argument of the vbox container causes a titled border to be added to its JPanel:

```
JPanel panel__2 = new JPanel();
panel__2.setLayout( new BorderLayout(panel__2, BorderLayout.Y_AXIS));
panel__2.setBorder( BorderFactory.createTitledBorder("Alignment"));
```

The space widgets inside the hbox are treated as horizontal spaces, but the one in the vbox becomes a vertical space of 40 pixels. The resulting Test3App.java is shown in Figure 13.

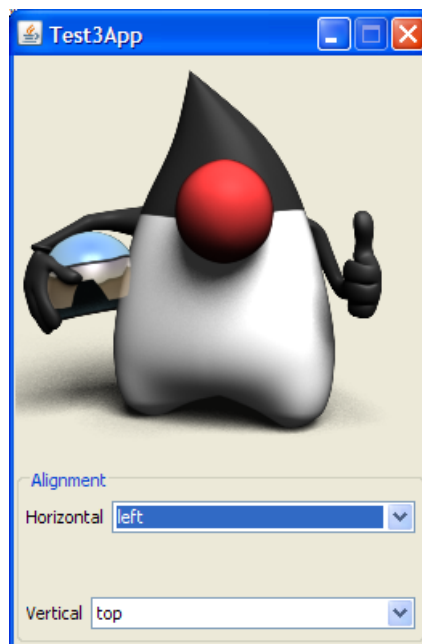


Figure 13. Test3App Executing.

Example 3: Text Area Processing

test4.txt contains a textarea widget with a large amount of text:

```
border {
    north: textarea "Call me Ishmael. Some years ago -- never mind how
long precisely -- having little or no money in my purse, and nothing
particular to interest me on shore, I thought I would sail about a
little and see the watery part of the world. It is a way I have of
driving off the spleen, and regulating the circulation. Whenever I
find myself growing grim about the mouth; whenever it is a damp,
drizzly November in my soul; whenever I find myself involuntarily
```

pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off -- then, I account it high time to get to sea as soon as I can."

```

center:
    flow edged "Wrap Options" {
        checkbox "Wrap" on
        radio "Wrap Words" on group=g1
        radio "Wrap Characters" group=g1
    }
}
}

```

The textarea is translated into a JTextArea inside a JScrollPane so all the text can be seen, along with a DocumentListener:

```

JTextArea textarea__1 =
    new JTextArea("Call me Ishmael.... can.", 5, 10);
textarea__1.setLineWrap(true);
JScrollPane scroll__2 = new JScrollPane(textarea__1);
panel__0.add(scroll__2, BorderLayout.NORTH);
textarea__1.getDocument().addDocumentListener(
    new DocumentListener() {
    public void changedUpdate(DocumentEvent e)
    { System.out.println("Changed textarea__1 at " +
        e.getOffset() ); }
    public void insertUpdate(DocumentEvent e)
    { System.out.println("Inserted into textarea__1 at " +
        e.getOffset() ); }
    public void removeUpdate(DocumentEvent e)
    { System.out.println("Removed from textarea__1 at " +
        e.getOffset() ); }
    });

```

Test4App.java is shown running in Figure 14.

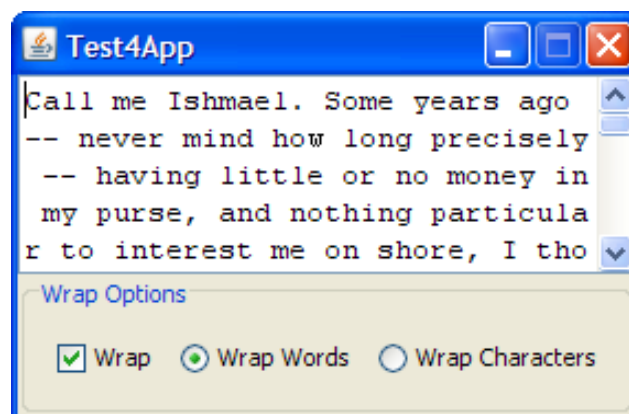


Figure 14. Test4App Executing.

Example 4: Password and Spinner

test6.txt utilizes password and spinner widgets:

```
flow {
  label "Enter Password: "
  password =pw1
  spinner =s1 { "usa" "uk" "france" "germany" "thailand" "china" }
}
```

These are converted into JPasswordField and JSpinner components, as shown in Test6App.java in Figure 15.



Figure 15. Test6App Executing.

When the user types into the password field, the text is obscured.

Example 5. Sliders and Grid Container

test8.txt employs three sliders in a grid container:

```
border {
  north: label "Show Colors"

  center:
    grid 3 2 edged "Choose Colors" {
      label "Red"   slider
      label "Green" slider
      label "Blue"  slider
    }
}
```

This example looks a little different from earlier ones since I've placed two widgets (label and slider) on a line. This is irrelevant to the positioning of the components in the Java executable which fills the grid in a left-to-right top-down order (see Test8App in Figure 16).

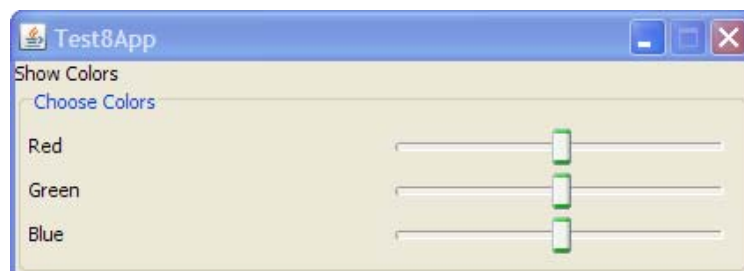


Figure 16. Test8App Executing.

A slider widget can include an “up” argument to switch it to a vertical slider, but there’s no other way to modify it, such as adding labels or tick marks.

5. Implementation

NestedSwinger translates a text file of NestedSwinger notation into a Java program. The three main compilation stages are illustrated by Figure 17.

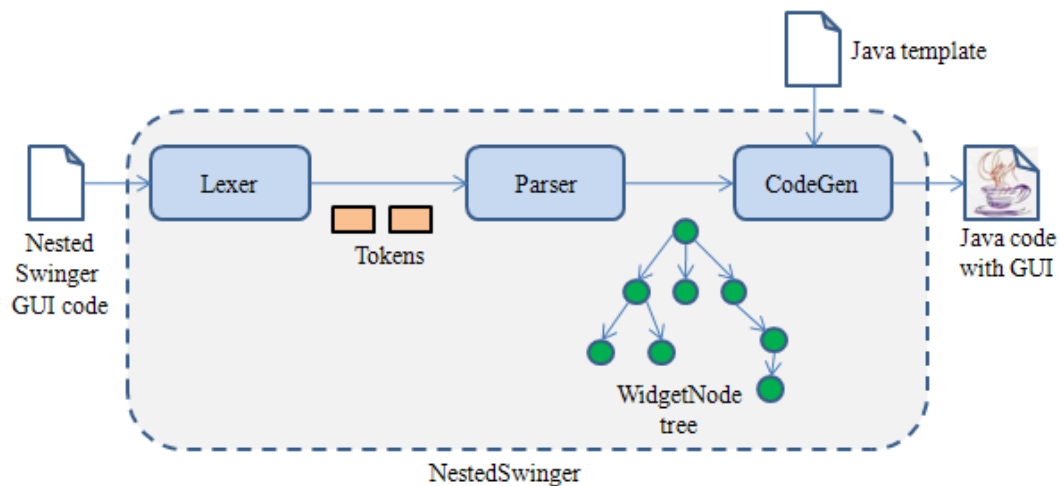


Figure 17. NestedSwinger’s Implementation.

Lexer reads in NestedSwinger text as a stream of characters, and converts it to a stream of Token objects, which are made available to the Parser class. Internally, Lexer uses Java’s StreamTokenizer.

Parser is a top-down recursive descent parser based on the BNF grammar described earlier. As Parser consumes tokens, it builds a tree of WidgetNode nodes. Each node represents a particular NestedSwinger container or widget, with the containers appearing as branches, and the widgets as leaves.

CodeGen traverses the WidgetNode tree, generating the Java text which will appear inside the makeGUI() method of the output file. The rest of the Java code for the application is read from a template file, called TemplateForGUI.txt, which has the form:

```
// #NAME.java
// #DATE

/* Generated with NestedSwinger
*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```



```

public class #NAME extends JFrame
{

    public #NAME()
    {
        super("#NAME");
        makeGUI();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setResizable(false);
        setLocationRelativeTo(null); // center the window
        setVisible(true);
    } // end of #NAME()

    private void makeGUI()
    {
        #GUI_CODE
    } // end of makeGUI()

    // -----

    public static void main(String[] args)
    {
        if(!setNimbusLaF())
            setSystemLaF();
        new #NAME();
    }

    private static boolean setNimbusLaF()
    {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");
            return true;
        }
        catch(Exception e)
        { return false; }
    } // end of setNimbusLaF()

    private static boolean setSystemLaF()
    {
        try {
            UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
            return true;
        }
        catch(Exception e)
        { return false; }
    } // end of setSystemLaF()

} // end of #NAME class

```

TemplateForGUI.txt looks like a Java file except for three macros: #NAME, #DATE, and #GUI_CODE. CodeGen replaces all occurrences of #NAME with the name of the application (e.g. Test1App), #DATE is changed to the current time and date, and #GUI_CODE becomes the GUI text generated by CodeGen.

The template file can contain any Java text. For example, the UIManager.setLookAndFeel() code in TemplateForGUI.txt could be changed or deleted.