SKETCHI - FROM SKETCHES TO GUIS

Johnathan Tunnicliffe

Supervisor: Dr Simon Colton

Individual Project Report for MEng Computing, Imperial College London

jdt05@doc.ic.ac.uk

June 2009

Abstract

Despite the dominance of Graphical User Interfaces (GUIs) in both consumer and business applications, there is still no simple, straightforward way of producing them. Manual coding requires a programmer to do a designer's job whilst drag-and-drop tools produce verbose, difficult-to-read code.

Various tools and techniques have been created over the years to aid developers in producing GUIs whilst reducing some of the prerequisite knowledge required. However, there is still a wealth of expertise needed regarding the specific widget toolkit to use these tools and produce a good quality interface for an end user.

This report discusses a new, more natural tool that developers can use to build GUIs particularly for Swing in Java. We explore transforming sketches and natural input into the code for user interfaces. We have implemented a new GUI design tool that removes the learning curve in creating GUIs, produces concise code and requires minimal user intervention and prior toolkit knowledge.

The results show that we can produce quality GUIs from natural input with minimal widget toolkit knowledge. Furthermore, the code created is more concise and of a higher quality than existing tools. The final finding is that we can abstract the toolkit so a single GUI sketch can produce code for multiple toolkits and programming languages.

Acknowledgements

I would firstly like to thank my supervisor Dr Simon Colton for his excellent project idea as well as his help throughout the duration of the project. I would also like to thank my family, girlfriend and friends for helping me through university and pushing me over the finish line after four long years.

Table of Contents

1 Introduction

Despite Graphical User Interfaces (GUIs) being utilised in the majority of modern day programs, creating them can still be tedious and frustrating. Existing techniques include coding by hand in a specific toolkit, declaring GUIs in a markup language and using graphical GUI building tools to drag and drop components onto a screen. Although programmers are spoilt for choice when deciding how to build their GUI, the existing techniques all have some form of shortcoming.

1.1 Motivation

1.1.1 Knowledge Barrier

When programming a GUI in Java there is choice to make regarding the toolkit implement in. Whether we use the Standard Widget Toolkit (SWT) or Swing for coding our GUIs should not matter for basic forms; a button is a button and a list is a list in any widget toolkit. This same choice has to be made many languages.

A major hurdle with GUI coding is the toolkit specific knowledge required. Understanding package structures and toolkit-specific interaction paradigms should not be a major hurdle in GUI programming but it is. Each toolkit uses its own paradigms and this introduces an unnecessary learning curve for the programmer. Here are two snippets of code that compare how to respond to a button click in Java Swing and C# WinForms.

```
Button btnDemo = new Button();
btnDemo.Click += new EventHandler(this.btnDemo_Click);

. . . // Other code

private void btnDemo_Click(object sender, EventArgs e)
{
    // process button click
}
```

Figure 1.1: C# WinForms event response example.

```
JButton btnDemo = new JButton("Demo Button");
btnDemo.addActionListener(new MyActionListner());

. . . // Other code

private class MyActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        // process button click
    }
}
```

Figure 1.2: Java Swing event response example.

As you can see, the two look similar in that you register some sort of listener to the object. In C# WinForms you must register an event handler with a method delegate on an action of the widget (`btnDemo.Click` above). In Java Swing, you assign a specific

implementation of a handler to a widget. The end result is the same, in that you detect the button click and run code in response, but the paradigms used by the toolkits are different. For a comparison of various toolkits see section 2.4.

1.1.2 Achieving Good Layout

Poor layout management in a GUI can destroy the quality of an application. Whilst a developer may be able to position and size widgets pixel perfect using an absolute layout, a user resizing the application will destroy the layout. This is why layout managers were created. A layout manager controls the position and size of widgets on the screen given certain constraints. Unfortunately, creating these constraints is not an easy task. To get the desired layout involves repeatedly tweaking the constraints, recompiling and rerunning the GUI. Anyone who's tried to write a GUI that scales as planned when it's window is resized knows this can take as long as writing the application code. The other point to add is that some layouts just can't be achieved with the standard layout managers. For instance, .NET has terrible layout managers and Java supplies a handful of layout managers that need to be nested in order to achieve anything remotely complex. However, there are two reasons why nesting layout managers should be avoided:

- *Default gaps* – Layout managers in Java use different default gap sizes, thus combining layout managers can lead to unexpected behaviour of GUI components.
- *Treatment of components* – Different layout managers will treat the components differently depending on parameters such as preferred size, minimum size and layout constraints. The same component added to different layout managers will not behave comparably and this makes visualising layout complicated.

A single global layout manager is a far better solution but there are not many available with the versatility to handle complex layouts. Fortunately there exist various third party layout managers that can handle complex layouts (see section 2.4.2 for details) but of course these have to be learnt by the programmer.

1.1.3 Quality of Generated Code

Whilst using a tool such as NetBeans helps with the visual aspect of designing a GUI form, it introduces other problems. NetBeans employs a drag-and-drop system for fast GUI creation, but the code it produces is verbose. For a simple form with two text fields, two buttons and two text labels, NetBeans produces 36 lines of layout code. The same form coded with the popular MigLayout (see section 2.4.2 for an overview of MigLayout) layout manager requires only 6 lines of layout code[i]. Even for a simple form, the code produced is complex and would require a much larger amount of work from a programmer to modify at a later date when compared to a manually coded version.

[i] MigLayout only requires one line of layout code per GUI widget.

1.2 Contributions

This project outlines a new idea for designing GUIs that aims to solve of the above problems. The end result is an intuitive application written in Java that can be used to visually build GUIs whilst removing the knowledge barrier and producing clean, concise code for the user. In particular, we use natural input from a touch screen so a user can draw their GUI on the screen and have the GUI built for them. The application is composed of three main components:

- *Classification of sketched widgets* – The user draws their user interface on screen using preconfigured symbols that look like the real interface widgets. This removes the knowledge barrier and helps abstract the toolkit by using natural input. This segment of the application is aimed at recognising what the user has drawn and classifying it as a user interface widget. We use a Case-Based Reasoning (CBR) implementation with multiple features to find a nearest match to a known widget and hence classify the input as a widget (see section 4.2 for implementation details of the widget classifier). We show that with clever features and a small number of examples from a user, recognition rates approaching 100% can be achieved (see section 8.1 for recognition rate results).
- *Automatic generation of layout* – This involves taking the sketch with the classification of the widgets and generating correct layout code. We take the bounds of sketched widgets and attempt to infer the correct number of rows and columns in the layout, as well as widget constraints for the layout manager. We generate multiple layouts and give the user the choice of which layout to proceed with (see section 5 for implementation details of the layout generator).
- *Code generation* – We implement an abstract and extensible code generator so that new code generators can be implemented to generate GUIs for other languages. This means the same GUI can be deployed across multiple toolkits with no extra work from the user (see section 6 for implementation details of the code generator).

These three components together remove much of the prerequisite knowledge required to create a GUI. The sketching element abstracts the widget toolkit and removes the knowledge barrier. The automated layout generation makes it easy to get well behaved, intelligently resizing forms and the code generator produces minimal, clean, code.

1.3 Report Structure

The remainder of this report is structured as follows:

- *Background* – The background chapter, found in section 2, gives details of some possible implementations that can be used in achieving our goals. We look at existing tools and techniques for GUI creation, different approaches for input recognition, a background on layout management and a detailed view of What You See Is What You Get (WYSIWYG) GUI designers.
- *Design Considerations* – The design considerations chapter, found in section 3, states the aims of the project and formulates a design to fulfil these goals. We give a critique of the existing tools and use this to design our own solution. We also discuss the design choices we have made and give an overview of our application, Sketchi.
- *Input Detection & Classification* – The input detection and classification chapter, found in section 4, includes details on our input classification and detection systems. We give implementation details of our CBR system and show how we classify a user's sketch. We also outline the algorithm we use for finding the individual widgets in the user's sketch.
- *Automated Layout Generation* – The automated layout generation chapter, found in section 5, looks at our algorithm for automatically finding layout constraints for widgets in the sketched GUI. Using these constraints, we can layout the user's interface in a window that intelligently resizes with no further interaction from the user being required.
- *Code Generation* – The code generation chapter, found in section 6, discusses the implementation of our abstract code generator. In particular, we show how its structure allows us to provide multiple implementations for various toolkits and languages.
- *Experimental Design* – The experimental design chapter, found in section 7, outlines the experiments we carried out to evaluate the software. We give reasons for the experiments we undertook and explain how the experiments were used to evaluate our software.
- *Results & Analysis* – The results and analysis chapter, found in section 8, presents the results of the experiments and explains what they mean in the context of our application.
- *Conclusion & Future Work* – The conclusion and future work chapter, found in section 9, highlights the successes and failures of the project and suggests new possible directions.

2 Background

2.1 GUI Basics

2.1.1 What is a GUI?

A Graphical User Interface (GUI) is a type of user interface that allows a user to interact with graphical icons, command buttons and other visual indicators to achieve tasks on an electronic based system. Regarded as a more intuitive and user friendly method of interacting with a system than it's text-based counterpart (Microsoft 2007), the command-line interface (CLI), the GUI has revolutionized personal computing and made it accessible to the consumer market. Although GUIs are prevalently found in personal computing, they have since found themselves shifted on to handheld devices such as phones, MP3 players and gaming devices as well as other consumer electronics.

Early GUIs implemented the Windows, Icons, Menu, Pointing device (WIMP) style of interaction, which was developed at Xerox PARC in 1973 for the Xerox Alto personal computer that later influenced both Microsoft and Apple's operating systems. Related commands are accumulated in menus and accessed with the pointing device. These menus can be explored without the user committing to a particular command, which can often be reversed. This encourages exploration by the user and helps them learn. Most applications share the same form so knowledge can be transformed between them making them more accessible to first time users whilst also presenting shortcuts for power users.

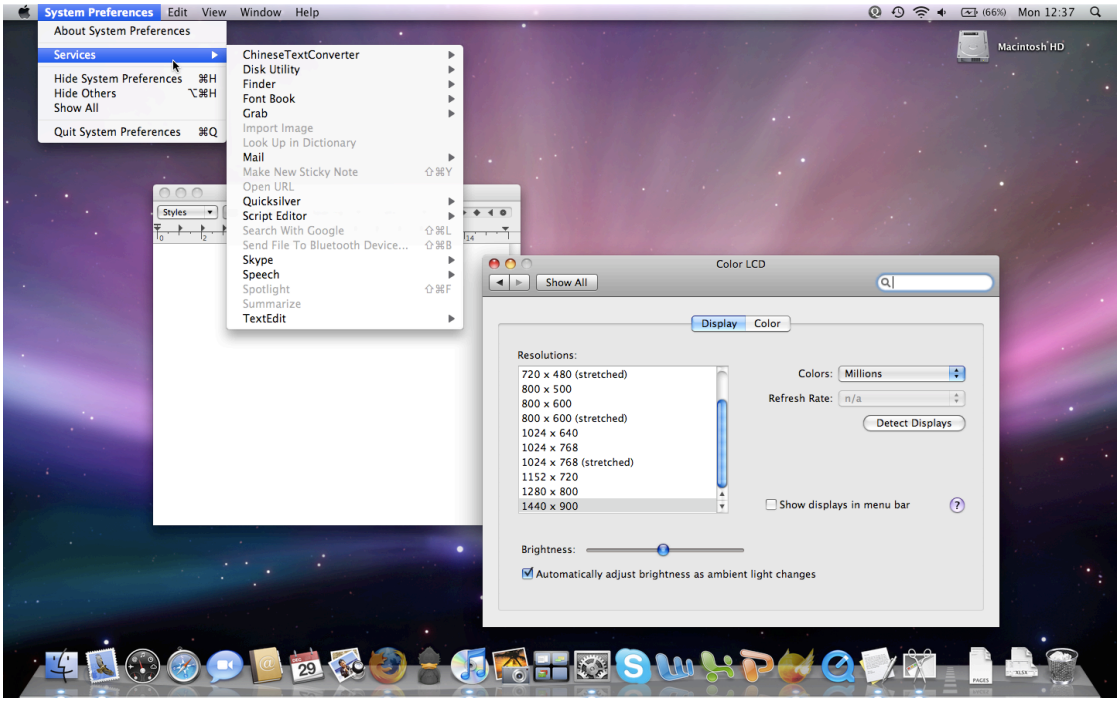


Figure 2.1: The Mac OS X Interface uses WIMP.

2.1.2 Human Interface Guidelines

Human Interface Guidelines (HIGs) are software development guides, put in place by the creators of the operating system (OS), that help application developers give a consistent visual and behavioural experience across applications and the OS. The documents outline fundamental design principles, the interface components available and guidelines on how to use and implement them on the particular platform. Applications that follow the guidelines for the particular platform will help their users as well as their applications in a number of areas such as:

- *Speed of learning the application* - Interface elements will be consistent between applications and common shortcuts will behave in the same way.
- *Look that integrates with the desktop* - Using the standard interface elements and guideline look and feel will help the application blend into the platform making the user's interaction seamless.
- *Users with special needs* - HIGs outline the platform specific technologies that help users with special needs use the system. Employing these technologies in your applications will make them more accessible to these users.

Guidelines are created for most major platforms and quite often contradict each other making cross-platform development more difficult. In this situation the developer is left to make the best decision to make the application as accessible to as many users as possible. Frequently the backend code at the core of the application will be shared and separate interfaces will be built for different platforms.

There are HIGs for Mac OS X, Windows Vista, GNOME Desktop, KDE Desktop, Java and others. Whilst all these platforms share common interface elements such as buttons, text boxes, scroll bars, tabbed panes, etc. major differences between the guidelines are found in recommended layout of the elements and in particular dialog box construction.

2.1.3 GUI Toolkits

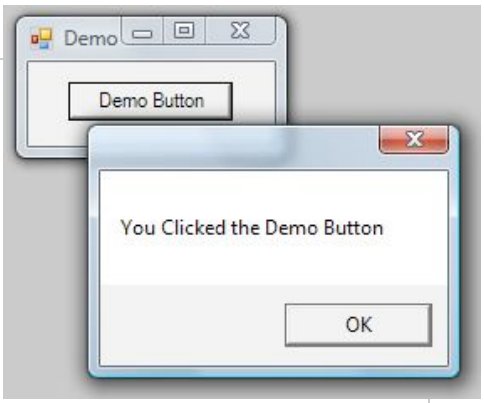
A GUI toolkit, also known as a widget toolkit, is a set of interface elements with an Application Programming Interface (API) that allows developers to build GUIs. The toolkits themselves contain widgets with a consistent look and feel that is transferred to the developer's interface. Some of the toolkits can decouple the look and feel of the widgets from the behaviour of the widgets and so can be skinned with pluggable look and feels[i]. The API provided by the toolkit generally allows handling of user events that cover common interaction with the widgets. Different platforms have different GUI toolkits with their own design paradigms and a lot of platforms have multiple toolkits available for them. The subsequent sections show small descriptions and comparisons of some popular toolkits:

[i] Discussed later in the descriptions of WinForms, Swing and SWT.

.NET WinForms – Microsoft Windows

With the creation of the .NET framework, Microsoft created the WinForms GUI toolkit to provide access to native Microsoft Windows interface elements. Although WinForms is a technology built for the Windows platform, a project called Mono provides a multi-platform implementation of the core .NET framework API that includes WinForms 2.0 (Mono Project 2008). All .NET compatible languages can use the WinForms API including C#, C++ and VB .NET.

WinForms uses an event-driven paradigm that uses event emitters and event consumers. An event is normally triggered by user actions such as a mouse click or a key press and any assigned consumers can then act in response to the event. WinForms uses the native Windows widgets which are governed by the theme in the host OS. For that reason it does not support pluggable look and feel. Figure 2.2 is a standard example of some WinForms code written in C# to show some features of the toolkit and the code required to build a simple window.




```
using System;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Demo
{
    public partial class Demo : Form
    {
        public static void Main()
        {
            Application.Run(new Demo());
        }
        private Button btnDemo = new Button();

        public Demo()
        {
            btnDemo.Location = new Point(24, 12);
            btnDemo.Size = new Size(98, 23);
            btnDemo.Text = "Demo Button";

             // register method to event handler
            btnDemo.Click += new EventHandler(this.btnDemo_Click);

            this.Text = "Demo";
            this.ClientSize = new Size(147, 50);

            Controls.Add(this.btnDemo);
        }

        private void btnDemo_Click(object sender, EventArgs e)
        {
            Button original = (Button)sender;
            MessageBox.Show("You Clicked the " + original.Text);
        }
    }
}
```

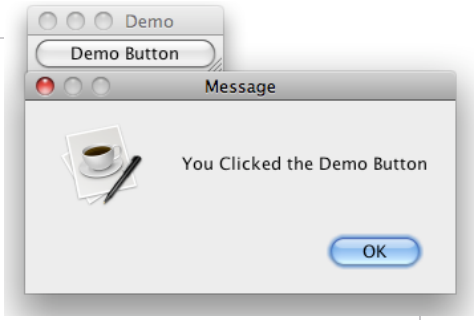
Figure 2.2: WinForms code example using C#.

Swing - Java

Swing is the widget toolkit included as part of the Java Foundation Classes (JFC). This means that it is included in the Java 2 Platform, Standard Edition (JSE) as of JSE 1.2 (Sun Microsystems 1998) so that it doesn't have to be downloaded separately for use in applications. Swing has a number of key features:

- *Platform independence* – Swing widgets are implemented in Java and use non-native universal rendering. Java draws the widgets using the Java2D primitives that are implemented for all platforms the Java Runtime Environment is available on.
- *Pluggable Look and Feel (PLAF)* – This allows Swing components to emulate the appearance of native components on the current platform without sacrificing platform independence.
- *Customizability* – Due to Swing implementing a twist of the Model-View-Controller (MVC) design pattern, sometimes called the separable model architecture, individual parts of a Swing component can be modified or changed completely without a rewrite of the entire component such as the model or the view.

The interaction paradigm employed by Swing uses the observer pattern in order to trigger code from user actions with the widgets. Figure 2.3 is the demo application coded in the previous section using Java and the Swing toolkit:




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

public class Demo extends JFrame
{
    public static void main(String args[])
    {
        JFrame demoFrame = new Demo();
    }
    public Demo()
    {
        JButton btnDemo = new JButton("Demo Button");

        // add the event handler
        btnDemo.addActionListener(new MyActionListner());

        this.add(btnDemo);
        this.setTitle("Demo");
        this.setSize(147, 50);
        this.setVisible(true);
    }
    private class MyActionListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e) {
            JButton original = (JButton)e.getSource();
            String msg = "You Clicked the " + original.getText();
            JOptionPane.showMessageDialog(null, msg);
        }
    }
}
```

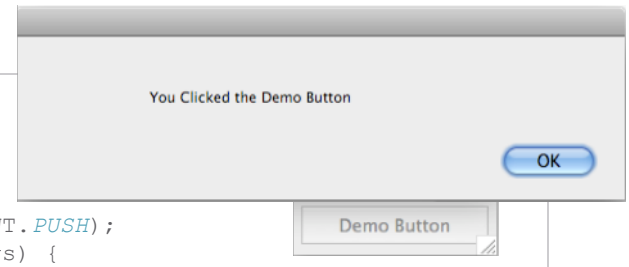
Figure 2.3: Java Swing code example.

SWT – Java

The Standard Widget Toolkit (SWT) is a competing widget toolkit to Swing for the Java platform developed by IBM for their Eclipse Integrated Development Environment (IDE). Unlike Swing which provides Java implemented versions of GUI widgets, SWT uses the Java Native Interface (JNI) to access the host OS's native widgets just as a native API call would. This means that applications using SWT have a completely native look and feel with deep platform integration, which is meant to give high performance. But with the native technique used by SWT comes certain compromises:

- *Customizable Look and Feel* – As SWT is merely a thin wrapper around native interface widgets, customizing the look and feel of the applications is very difficult, especially when compared to Swing.
- *Portability* – Running SWT requires an external library, not included in the JSE, specific to each platform SWT is running on which must be downloaded separately to the JSE.

Using the SWT also means that Java cannot use automated garbage collection to free up the memory of the interface widgets since the OS controls them. Instead the `dispose()` method has to be explicitly called which is synonymous to `free` in C.




```
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
public class Demo {
    Display display = new Display();
    Shell shell = new Shell(display);
    Button button = new Button(shell, SWT.PUSH);
    public static void main(String[] args) {
        new Demo();
    }
    public Demo() {
        button.addListener(SWT.Selection, new MyListener());
        button.setText("Demo Button");
        button.setBounds(0, 0, 150, 40);
        shell.pack();
        shell.open();

        // Set up the event loop.
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }
    private class MyListener implements Listener {
        public void handleEvent(Event event) {
            Button original = (Button)event.widget;
            MessageBox messageBox = new MessageBox(shell, SWT.OK );
            String msg = "You Clicked the " + original.getText();
            messageBox.setMessage(msg);
            messageBox.open();
        }
    }
}
```

Figure 2.4: Java SWT code example.

2.2 Current GUI Building Methods

The rest of this background looks at existing technologies and tools used to develop GUIs and discusses what's wrong with these current techniques. After this we can start to explore how to solve these problems and we look into specific methods for recognising natural input and inferring correct layout management code.

2.2.1 Coding by Hand

Coding GUIs by hand is the classic technique used to build user interfaces. It involves creating instances of the widgets and specifying their properties, including size and location if a layout manger isn't in use. The design of the GUI is purely code based with no visual representation of the GUI available until the code has been compiled and run. Most modern GUI toolkits provide access to image resources such as bitmaps to be incorporated in the interface through an API. As most GUI toolkits are event driven, the majority of widgets can have code blocks associated with certain actions performed with the widget. Figure 2.5 shows the code required to attach an event listener to a component in Swing.

```
JTextField textField = new JTextField();
textField.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
    }
});
// Add the textfield to the frame and display the frame
this.getContentPane().add(textField);
this.pack();
this.setVisible(true);
```

Figure 2.5: Adding an event listener to a widget in Swing.

The above figure shows a code block being associated with a Swing text field. When the default action is performed, which is when the return key is pressed for a text field, the action command text is printed to the console. Multiple code segments can be associated with each type of event triggered by the widget and using this, complex interfaces can be built up.

2.2.2 GUI Builders: WYSIWYG

Below we discuss what a GUI builder is, the typical functions provided by GUI builders and the current GUI builders available to create interfaces in various GUI toolkits.

What is a GUI builder?

GUI builders were invented to simplify the process of creating interfaces. They typically implement a drag-and-drop style design principle that allows the developer to arrange the interface widgets in a WYSIWYG editor. This gives the developer a live view of the interface they are designing without the need to recompile and run the application.

Typical functions provided by a GUI builder include:

- *Designer Screen* – The most important part of a GUI builder is the designer screen. This is typically designed to look like a window or a panel on which interface widgets can be dragged onto. Once on the designer screen, the user can often drag edges of controls to resize them or click and drag the body of the widget to move it. The user can specify the layout manager to be used in each panel and the user can often make a composite of containers to achieve complex layouts (although these days this is discouraged as resizing of the window doesn't always happen in the desired way). The designer screen provides a live preview of how the interface will look once the application is running although some previews don't act as the compiled running interface.

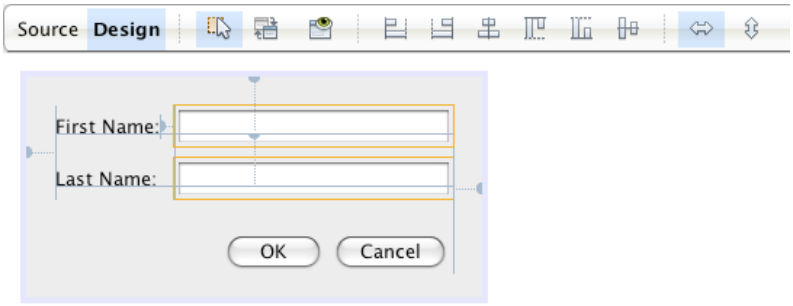


Figure 2.6: The NetBeans Designer Screen.

Figure 2.6 shows the NetBeans designer screen. The screen looks like a panel and allows the interface widgets to be dragged around to the desired location. NetBeans also includes a toolbar to quickly align controls, view the source code and preview the interface.

- *Widget Palette* – The palette is the location users can find the set of common controls from the GUI toolkit. Users can often customize the palette by removing, reorganizing and adding widgets to the sub-groups of the palette. Users click and drag the widgets from the palette over onto the designer screen where they can then position and resize it as necessary.

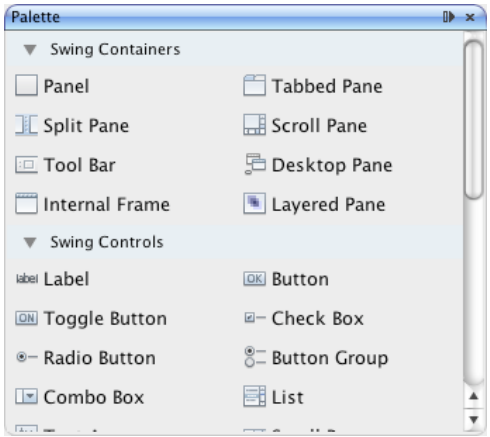


Figure 2.7: The NetBeans Widget Palette.

- *Property Editing* – Property editors are available to a developer to change a number of the attributes a widget can posses. This is often a docked table showing the selected widget's attributes with values. These values are editable and the results of changing these properties are reflected in the designer screen in real-time. Some editors provide limited inline editing from the designer screen. Simply double-clicking on a JLabel in NetBeans allows you to inline edit the text it displays, but it is not available for many widgets.

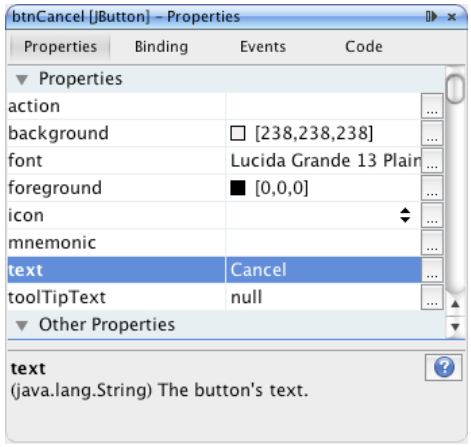


Figure 2.8: The NetBeans property editor.

- *Layout Hints* – Modern GUI builders provide hints to the user whilst interface widgets are being moved and resized in the designer screen. They generally appear as dotted lines in an overlay style that show how the layout will change if the user were to release the mouse button at that particular point. These overlay hints show alignment hints and often anchoring hints if the layout manager supports it. Figure 6 shows the layout hints active on the designer screen for the two text fields. It shows the alignment hints as well as anchor hints for the sides of the panel.
- *Code linking* – Code linking is an integral part to a GUI builder's function. This involves automatically generating boilerplate code for interacting with widgets. Java Swing follows an event-driven model, as most GUI toolkits do, so the auto-generation of code will be to link events on a widget to implemented methods. These events could be anything available for the widget such as a mouse over, focus gained or a mouse click.
- *Two-way Design* – Two-way design is the ability to have both generated code updated when the design screen is modified and have the design screen updated if the developer modifies the underlying source code to the GUI. This involves the GUI builder having the ability to parse the Java code and not work from an internal representation or external helper file as NetBeans does[i]. The necessity of two-way design is a debated topic in the GUI design field. Some see two-way design critical to the design process (Grev 2008) whilst others believe that GUI code should simply belong to the GUI designer. Whilst one can understand the want to be able to tweak the GUI code manually, if the tool is good enough the manual tweaking shouldn't be necessary.

Next we discuss particular GUI builders that are available for the various GUI toolkits along with their unique functionality and features.

[i] NetBeans creates a .form file that is discussed in more detail in the next section.

Visual Studio - .NET

Visual Studio (VS), currently available in a 2008 edition, is a closed source, non-free IDE developed by Microsoft to help develop on the Microsoft Windows platform and particularly with .NET technologies.

VS includes a code editor, source debugger and a forms designer to give a WYSIWYG view of the user interface you are designing. The IDE contains a property editor for the currently selected widget and double-clicking a widget generates boilerplate code for the default event fired by the widget. The .NET platform provides a limited number of layout management widgets by default that includes a panel, split container, flow layout and a table layout panel. In order to achieve a good layout that scales with the window, the programmer must visually nest these building-block components and use anchors and docking to get the desired output.

VS has a designer that employs one-way design, which means it cannot parse source code back into the designer. The main problem with this method is after creating the interface with the designer, the generated code cannot be modified without being overwritten when anything is changed in the designer. Whilst this may seem like a negative side effect, it follows the principle that the GUI code should belong to the GUI builder, which could be seen as more desired than two-way design.

As with most GUI builders that automatically generate code, the code generated may seem bloated compared to code produced from manual coding. Obviously this isn't a problem if the GUI code is left completely to the GUI builder but maintaining the GUI code without the designer at a later date could become more challenging. Figures 2.9 and 2.10 show a comparison of the form initialization of a very simple form created with the WinForms designer and a manual coding of the same form:

```
private void InitializeComponent() {
    this.btnClickMe = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // btnClickMe
    //
    this.btnClickMe.Dock = System.Windows.Forms.DockStyle.Fill;
    // Location is irrelevant when the button is docked in line above
    this.btnClickMe.Location = new System.Drawing.Point(0, 0);
    // Size is irrelevant when the button is docked
    this.btnClickMe.Size = new System.Drawing.Size(250, 77);
    // This is the only control in form so tab index is 0 by default
    this.btnClickMe.TabIndex = 0;
    this.btnClickMe.Text = "Click Me!";
    this.btnClickMe.Click += new EventHandler(this.btnClickMe_Click);
    //
    // frmDemo
    //
    this.ClientSize = new System.Drawing.Size(250, 77);
    this.Controls.Add(this.btnClickMe);
    this.Text = "Demo";
    this.ResumeLayout(false);
}
```

Figure 2.9: VS generated form code with added comments.

```
private void InitializeComponent() {
    btnClickMe = new Button();
    btnClickMe.Dock = DockStyle.Fill;
    btnClickMe.Text = "Click Me!";
    btnClickMe.Click += new EventHandler(btnClickMe_Click);

    ClientSize = new Size(250, 77);
    Controls.Add(btnClickMe);
    Text = "Demo";
}
```

Figure 2.10: Simple form manually coded assuming correct imports.

As can be seen from the code examples above, manual code leads to more concise code that in turn is easier to read and maintain. With manual coding, imports can be added to the file so fully qualified class references are not required on each line. Manual coding also allows redundant lines to be removed. In figure 9, location and size are specified for the button but are irrelevant to this particular form since the button is docked in the form.

JFormDesigner – Swing

JFormDesigner is a closed source, non-free GUI builder for Java Swing. Karl Tauber, who has more than 15 years of experience in creating GUIs, created it due to the lack of powerful visual GUI builders for Java Swing. He started the development of JFormDesigner in 2003 (FormDev Software n.d.) and it is currently at version 4.0.2.

JFormDesigner, although not free, is in my opinion the best Swing GUI builder available. Though not a fully-fledged IDE, it is available as a stand-alone application or as a plugin for the Eclipse, JBuilder and IntelliJ IDEs. What makes JFormDesigner so good is it's easy of use, tight integration with various third party layout managers and ability to plugin into various popular Java IDEs.

In terms of output, JFormDesigner can generate Java source code in a one-way design fashion like VS does but developers can also output a JFormDesigner XML file that describes the designed form. Developers then use an open-source runtime library to load the XML files into their applications. A major feature that is available in JFormDesigner, and not VS, is the ability to add boilerplate code for events other than the default event. JFormDesigner uses JavaBeans technology to find information about the components in the form at runtime that also permits third party or custom components with BeanInfos to be used. All this combined with a fast, responsive feedback system for laying out components make JFormDesigner a great choice for designing Swing forms.

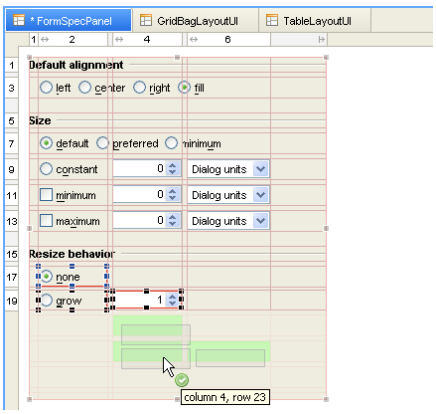


Figure 2.11: JFormDesigner design view.

Figure 2.11 shows the main JFormDesigner design screen with the visual feedback in action. The red lines are showing alignment of the controls to the grid in the layout manager. The green boxes are showing where the dragged controls will be placed in the layout before you release them.

NetBeans - Swing

NetBeans is an open source IDE sponsored by Sun Microsystems currently at version 6.5. As of version 5.0 NetBeans included a GUI builder formerly known as Project Matisse to bring an integrated GUI builder to the IDE. With Project Matisse came a new layout manager, Group Layout, that is now included in JSE 6.0. The GUI builder along with Group Layout lets you lay out components freely, providing visual guidelines for optimal spacing between components and alignment of components. The NetBeans GUI builder infers the appropriate resizing behaviour and more, freeing the developer from the complexities of Swing layout managers. You can just use the intuitive visual form builder to produce a professional GUI easily – in the background, the IDE produces the correct implementation using a layout manager and other Swing constructs (NetBeans n.d.).

NetBeans generates a .form file as well as the Java code responsible for the GUI. Without this .form file the GUI cannot be modified with the GUI builder. This is akin to JFormDesigner, which relies on a separate file to represent the GUI internally. This all means one-way design is employed. In fact NetBeans goes one step further in protecting its GUI code. Whereas in JFormDesigner you can modify the source code it has generated when it's integrated in an IDE but not updated in the live view, NetBeans' IDE protects the GUI code completely and doesn't allow you to modify it. In the NetBeans IDE this is known as a blue guarded block.

A very good feature that NetBeans does provide is that of inline editing for certain components in the GUI builder. Instead of having to use the property editor or writing Java code to perform the same actions, users can click a button to quickly edit text in a label, button or a menu item amongst other components.

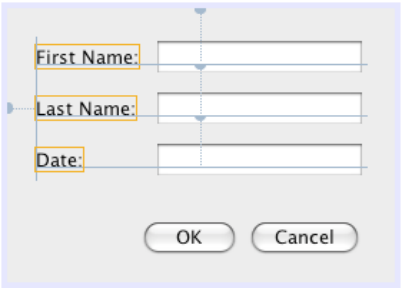


Figure 2.12: NetBeans anchor guides.

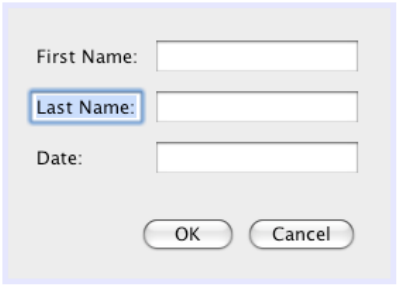


Figure 2.13: NetBeans inline editing of a JLabel's text.

With the current state of other Java GUI builders, NetBeans provides the best open source, free IDE and GUI builder combination available.

Other GUI Builders

Outlined above are just three of the popular GUI builders available for developing interfaces but it is by no means an exhaustive list. In fact, there are as many GUI builders as there are GUI toolkits and in a lot of cases there are multiple GUI builders for each toolkit. Below are some small descriptions of other GUI builders that are available:

- *Jigloo* – This is an Eclipse plug-in that allows WYSIWYG design of both Java Swing and Java SWT interfaces. Jigloo is closed source but is free for non-commercial use. It provides many unique features such as two-way design, conversion from Swing interfaces to SWT interfaces and it is one of the only GUI builders to support the highly popular, free and natural MigLayout layout manager from MiG Infocom.

- *Visual Editor (VE)* – This is the original GUI builder plug-in developed for Eclipse to develop Java Swing and Java SWT GUIs. It is an open source project but support for the project seems to have died. The last version of Eclipse officially supported by VE is 3.2 and Eclipse is currently at version 3.4. Some unofficial patches have floated around the Internet[i] that allows use of VE in Eclipse 3.4.
- *Interface Builder* – Interface Builder is part of Apple's XCode development tools for Mac OS X and allows simple construction of both Cocoa and Carbon applications. It differs to other GUI builders in that it doesn't generate GUI code that is then compiled and run. Instead it works with live components that the user can manipulate in Interface Builder that are then encoded to what are called freeze-dried objects (Anguish, Buck and Yacktman 2002). These objects are live instances of interface widgets that were once live in memory and have been written out to disk. In the developer's OS X application, the archived file is loaded and the freeze-dried components are restored to the state they were in at the point of encoding.
- *Glade* – Glade is an interface designer for the GTK+ toolkit, which is a toolkit primarily used in the GNOME window manager for Linux. Again it differs from other interface designers in that it doesn't generate code directly. Glade creates Glade XML files that represent the interface in an XML file. An external library known as ***libglade*** can then be used to dynamically load up interfaces at runtime in the multiple languages that are supported by ***libglade***. Specifying GUIs in XML is something discussed in the next section.

2.2.3 Using a Markup Language to Specify GUIs

Specifying GUIs in a markup language isn't a new idea but it is one that isn't well implemented. There have been lots of different markup language specifications for specifying user interfaces but none have been widely adapted. Here we will discuss different markup languages that can be used to describe user interfaces and consider the various implementations of user interface specifications with their supporting tools.

XML

Extensible Markup Language (XML) is a specification developed by the World Wide Web Consortium (W3C). XML was particularly designed for web documents and is considered platform independent. It allows designers to create their own customized tags, enabling the definition, transmission and validation of data between applications and platforms. It is hierarchical in nature and has two basic nodes, an element and element attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Sample XML file -->
<people>
    <person id="person1">
        <name>Johnathan Tunnicliffe</name>
        <gender>male</gender>
        <age units="years">21</age>
    </person>
</people>
```

Figure 2.14: Simple XML example.

The above XML example shows an encoding of a person's data. The document has the following form:

[i] http://wiki.eclipse.org/VE/Installing shows how to install VE in Eclipse 3.4 but nothing official has been announced.

- *The XML declaration* – This defines the XML version and the encoding used.
- *A comment* – An example comment.
- *XML body* – `people` is the root node that can contain `person` child elements. `person` has an attribute `id`. `name` and `age` are child elements of the root node `person`. `units` is another attribute but this time for the `age` attribute tag. The data for the attributes appears between the attribute tags.

This is just a simple example of what XML looks like. It can be used to express a wealth of constructs and its flexibility allows it to be used in most data representation instances. XML has been extended to include much more functionality such as the ability to refer to a particular section of an XML document using XPath or to search and manipulate XML data using XQuery like SQL can with relational databases.

The flexibility of XML also allows it to be used to describe GUIs and many specifications have been proposed with none becoming standard. These are described later in this section. Although these specifications have been released to the community, GUI building tools tend to use their own internal specifications that use XML for describing user interfaces. JFormDesigner uses JFormDesigner XML[i] files, Glade uses Glade XML and CookSwing uses it's own specification as well. To show a basic example of a user interface defined in XML, CookSwing's XML specification to define a simple Swing interface is used:

```
<frame title="Demo Frame">
    <borderlayout>
        <constraint location="Center">
            <panel>
                <borderlayout>
                    <constraint location="North">
                        <button text="North" />
                    </constraint>
                    <constraint location="South">
                        <button text="South" />
                    </constraint>
                    <constraint location="East">
                        <button text="East" />
                    </constraint>
                    <constraint location="West">
                        <button text="West" />
                    </constraint>
                    <constraint location="Center">
                        <button text="Center" />
                    </constraint>
                </borderlayout>
            </panel>
        </constraint>
    </borderlayout>
</frame>
```

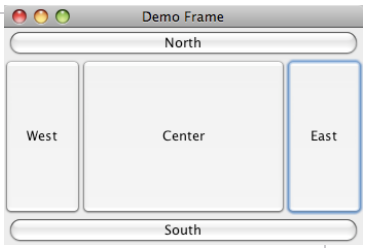


Figure 2.15: CookSwing XML user interface example.

Figure 2.15 shows the use of buttons and a layout manager in CookSwing's XML specification.

[i] JFormDesigner can generate java code as well as use a runtime library to create GUIs from it's internal XML files http://www.jformdesigner.com/doc/help/runtime_library.html

YAML

YAML™ (rhymes with "camel") is a human-friendly, cross language, Unicode based data serialization language designed around the common native data types of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing (Ben-Kiki, Evans and Net 2008). YAML's key goals are to be human readable and to provide support for serializing arbitrary native data structures.

YAML is quite often compared to XML but the two were designed with very different goals in mind. XML was designed to support structured documentation whereas YAML was designed to serialize data. Although XML can be used to describe a data's structure and represent data, YAML's definition of the same data is more human readable. Another key difference between XML and YAML is the lack of a schema system for YAML and this means it's not possible to validate YAML documents. Of course a proprietary validation system could be created for internal use within an application but there is currently no formal validation mechanism. Figure 2.16 is the example from the XML section defined in YAML[i]:

```
people:
  Johnathan Tunnicliffe: &person1
    gender: male
    age:    21
```

Figure 2.16: Simple YAML example.

As can be seen, the YAML version is more succinct and more human readable without the angle brackets everywhere and doesn't require quotation marks for strings. YAML doesn't contain the notion of attributes so the `age` element has lost its `units` attribute. But this is a separation of meta-data (describing the units) describing the data and the data itself, the former of which YAML doesn't attempt to provide.

From the description provided above it would seem that YAML would be ideal for the description of user interfaces and there is one current up-to-date project called javabuilders that looks to do this. The javabuilders projects define user interfaces declaratively in YAML. Users use the required javabuilders library to load the YAML file that returns a user interface. There are implementations (or part implementations in progress) for a number of GUI toolkits for Java with the most mature implementation being for Java Swing. This implementation has support for all the standard Swing components, support for adding event handling methods, data binding and multiple layout managers including the popular third party manager MigLayout (see page *_* for details). In addition to using standard layout manager constraints, the developers of javabuilders have developed something called Layout DeScription Language (Layout DSL)[ii]. Layout DSL provides a more natural way to layout interface objects in a purely text based manner and from their relative alignments and number of rows. The Layout DSL in reality is a layer of indirection on top of the MigLayout constraints albeit a useful one.

[i] The full YAML version 1.2 specification can be found here: http://yaml.org/spec/1.2/
[ii] Layout DSL information: http://code.google.com/p/javabuilders/wiki/LayoutDSL

Figure 2.17 shows the CookSwing XML user interface example in YAML code according to the javabuilders specification:

```
JFrame(title=Demo Frame):
    content:
        - JButton(name=north,text=North)
        - JButton(name=south,text=South)
        - JButton(name=east,text=East)
        - JButton(name=west,text=West)
        - JButton(name=center,text=Center)
        - MigLayout:
            constraints:
                - north: north
                - south: south
                - east: east
                - west: west
                - center: center
```

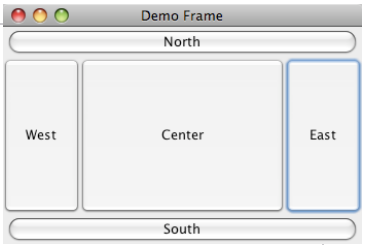


Figure 2.17: javabuilders YAML user interface example.

As can be seen from Figure 2.17, the YAML version of the interface is far less verbose whilst still producing the required GUI. The code above doesn't use Layout DSL instead it uses standard MigLayout constraints, as Layout DSL doesn't yet have support for docking as MigLayout manager does (although it can be simulated in a grid using Layout DSL).

XAML

Extensible Application Markup Language (XAML) is a markup language based on XML created by Microsoft principally to support its Windows Presentation Foundation (WPF) technology in .NET. WPF is the graphical technology included in the latest .NET 3.5 framework and included with Windows XP SP2 and Windows Vista. WPF puts a clear emphasis on separation of business logic from the user interface.

XAML is a user interface markup language that simplifies creating a UI for the .NET Framework programming model. XAML allows declaration of visible UI elements and then allows you to separate this UI definition from the runtime logic using code-behind files, joined to the XAML by partial class definitions.

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Demo Frame" Height="300" Width="400">
    <Grid>
        <DockPanel>
           <Button Height="35" DockPanel.Dock="Top">North</Button>
           <Button Height="35" DockPanel.Dock="Bottom">South</Button>
           <Button Width="75" DockPanel.Dock="Left">East</Button>
           <Button Width="75" DockPanel.Dock="Right">West</Button>
           <Button>Center</Button>
        </DockPanel>
    </Grid>
</Window>
```

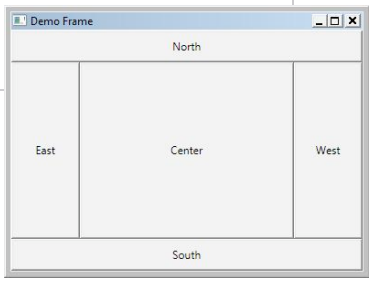


Figure 2.18: WPF XAML user interface example.

Figure 2.18 shows the XAML definition for WPF technology of the simple test interface we have been using. It is very readable but then again this is a simple test application. Compared to the YAML example the XAML code above is longwinded and a lack of good layout managers in the .NET framework means that panels have to be nested in order to create more complicated layouts. Even in this example the DockPanel component is nested inside a Grid container.

XAML files themselves can be edited using a standard text editor or Visual Studio, which offers a live WYSIWYG designer with a drag-and-drop interface or code editor with a live preview. Another tool produced by Microsoft is Expression Blend that allows graphic designers to create the user interfaces. Projects can then be linked to Visual Studio so designers can work on user interfaces and programmers can work on the code-behind files that provide functionality.

As the XAML specification is available under Microsoft's Open Specification Promise (OSP), others can implement uses of it without any legal repercussions. One company to see the use of XAML in other languages is Soyatec, which has released a product known as eFace[i]. eFace is a tool for converting XAML files to Java user interfaces and currently supports converting XAML to Swing and SWT. eFace describes itself as "a platform-independent and technology-neutral presentation framework", much like WPF but with Java as the backend technology. eFace provides a plug-in for Eclipse that allows modification of XAML files with a live preview of the interface although it does not allow manipulation of the preview directly (two-way design). The key advantage to using eFace is having a common resource of XAML defined interfaces that can be shared between .NET and Java developers. It also removes the need to learn specifics about Swing and SWT although knowledge of XAML is required.

Other Markup Language Specifications

XAML isn't the only specification that has been proposed to specify GUIs using markup languages. In fact there have been many attempts to standardize a declarative language for user interfaces but none have made a big impression on the industry. Below is a small list of other XML based markup languages used to describe user interfaces:

- *XUL* – XML User Interface Language (XUL) is a markup language developed by the Mozilla project to help specify cross-platform interfaces such as Firefox and other web applications. It is based on existing standards such as XML, HTML, CSS, DOM and JavaScript so for should be easy to learn for existing web developers (Mozilla 2007).
- *UIML* – User Interface Markup Language (UIML) is an abstract markup language for specifying GUIs. The look and feel of the application isn't specified but instead what interface items need to be shown and how they should behave. The main goal behind UIML is to provide a markup language that is device-independent and user interface metaphor independent(UIML.org n.d.).

[i] More on eFace can be found here: http://www.soyatec.com/eface/

2.3 Input Recognition

Here we discuss how input recognition in the application could possibly work. This would allow the user to directly draw the user interface using a mouse, touchscreen or graphics tablet onto a canvas that the GUI code can be generated from.

2.3.1 Artificial Neural Networks

An artificial neural network (ANN) is an information-processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. ANNs consist of layers of artificial neurons that can be interconnected to work together to solve the specific problem they have been trained for. A neuron can be thought of as an individual calculating unit with multiple possible inputs and often has multiple outputs as well. The inputs into a neuron are typically real-valued numbers that the neuron can perform it's own internal calculation on. This produces a value that can be below or above the neuron's threshold value and this decides if the neuron fires or not (produces an output or not). Figure 2.19 shows the general architecture of an ANN:

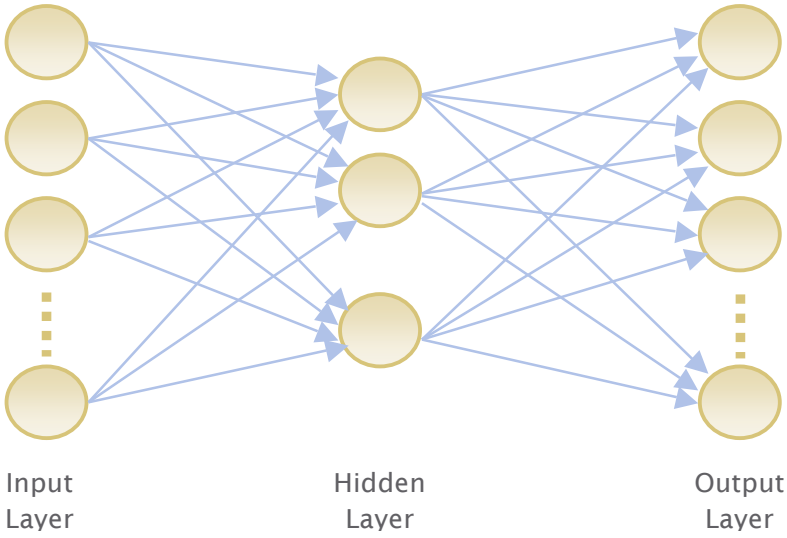


Figure 2.19: Artificial Neural Network architecture.

The nodes represent the neurons and the arrows represent connections between them. Note how highly connected it is, this is to simulate the synapse connections in biological neural networks (BNN). The input layer contains the neurons attached to the outside world, the hidden layer performs calculations on the inputs and the output layer presents the results to the outside world. To increase the complexity of the ANN the number of hidden layers can be increased.

Each connection has an associated ***weight (w)*** that is applied to the ***input (x)*** value flowing along it into the neuron. Therefore the collective of i inputs into a neuron can be represented as:

$$z = \sum_i w_i x_i$$

With this value the neuron can now apply it's ***activation function*** on z, the result of which will determine the value outputted by the neuron. The values of w are determined through a process known as learning (or training). Learning is carried out using a set of training data

that is a collection of input examples with known desired output values. ANNs use eager learning as their learning method. This means that a general, explicit description of the target function based on the provided training examples is constructed. The most popular learning technique is a method called backpropogation, which is used in Feed Forward Networks (networks without loops) and can be outlined in the following steps (Colton 2004):

- *Training input* – One of the training samples is passed into the neural network.
- *Output comparison* – The output of the neural network is compared to the desired output for that training sample. The error in the output for each output neuron is calculated.
- *Hidden layer error calculation* – These calculated errors are used to calculate the errors from the neurons in the hidden layer of the network by propagating the errors from the output back through the network.
- *Modify weights* – With errors calculated, the weights can be adjusted to match the desired output for that training example.
- *Repeat* – This is repeated for all the training data, possibly multiple times, until there are no misclassifications of input or the error in classification is below a threshold.

So long as training data is available for training, neural networks can often be used to solve problems for which there is no algorithmic solution.

Artificial Neural Networks for Pattern Recognition

Pattern recognition is the identification of shapes, forms or configurations by some automatic means. It often involves three stages:

- *Observation* – This involves collecting the initial information. If using pattern recognition on an image this would be loading the picture from a camera or, if static analysis is being used, from a storage medium. If using pattern recognition on natural input as this project aims to do then it will be collecting the user input information.
- *Feature extraction* – This is the process of recognising individual parts of the input to be classified. This can be computing metrics on the symbolic features of the input.
- *Classification* – The final step of the pattern recognition process is to classify the object in the input based on the features discovered.

ANNs are very good at pattern recognition due to their resilience against distortions in the input data and their capability to learn. Once the ANN has been created and trained appropriately through training data, it can recognise pre-learnt patterns with minor distortions. Next we walk through an example of producing an ANN to recognise a small set of characters for an optical character recognition (OCR) application.

Imagine a grid with dimensions 5 by 6 to be used for drawing characters onto. Each grid cell will represent a pixel that can be coloured white or black. In our example we attempt to recognise the characters ‘A’ and ‘H’.

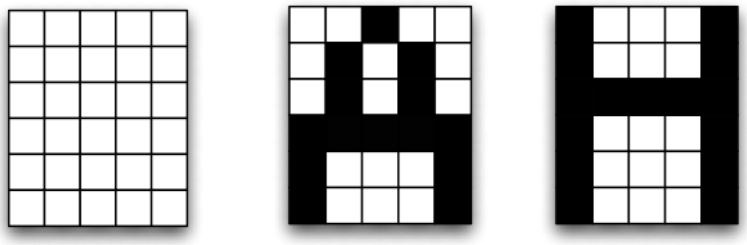


Figure 2.20: Character recognition example.

These character representations can be converted into input data for a neural network by serializing the pixels. If a pixel is black then the input data will be a 1 and if the pixel is white then the input data will be a 0. Since we are using a grid with dimensions 5 by 6, we have 30 pixels and thus we have 30 bits of data to represent our input to the ANN. Our ANN will be constructed of a single layer for input with 30 neurons, a single hidden layer consisting of 30 neurons and a single layer for the output consisting of two neurons.

For the ANN to recognise the characters, it must be trained. The training data will be the bit patterns for the characters together with the desired output pattern. In our case the output pattern could be 0,1 for the letter 'A' and 1,0 for the letter 'H'. The neural network is then trained with this data so the weights between the neurons are adjusted to give the desired output. After training the network, the encoded characters can be passed into the network as input and desired results produced.

The method described uses pixel locations to determine the input values. It should be noted that there are other ways of encoding the character data before it is input to a neural network. One such method is the receptor technique discussed later in the section (see section 2.3.3).

From the example we can see that training an ANN to recognise a set of characters is a relatively easy solution to the task of OCR. The only problem with this technique is the matter of scaling the input to a 5 by 6 grid. The bigger this grid, the more neurons required in the input layer and this would increase the time to train the network dramatically. It can also be seen that the problem of recognising a character set can be likened to that of recognising interface widgets as the set size is similar and the patterns to learn are of comparable complexity. A rectangle that represents a text field is no more complicated that the letter 'O' whilst a grid with two columns and rows is no more complicated than the letter 'H'. Below we explore the available frameworks for implementing ANNs in Java.

Joone

Joone is a neural network framework to create, train and test artificial neural networks. The aim is to create a powerful environment both for enthusiastic and professional users, based on the newest Java technologies.

Joone is composed of a central core engine that is at the heart of all applications developed with Joone. Joone's neural networks can be built on a local machine, be trained on a distributed environment and run on any device that supports JSE. Included with the Joone distribution is the ability to create a wealth of ANN architectures including Feed Forward Networks, Recursive Neural Networks, Modular Neural Networks and Kohonen Self Organizing Maps (SOM) amongst others. It also includes the capacity for supervised and unsupervised learning.

Joone is extensible in that new modules can be built to implement new algorithms or new architectures to include a user's neural network. These modules are implemented as code modules and added to the information flow. Like it's commercial rivals[i], Joone includes a GUI component that allows ANN to be visually produced in a WYSISYG manner though it is a very primitive interface that is overshadowed by the framework's focus on code based ANNs.

[i] Commercial rivals include Synapse (http://www.peltarion.com/products/synapse/) and NeuroSolutions (http://www.neurosolutions.com/).

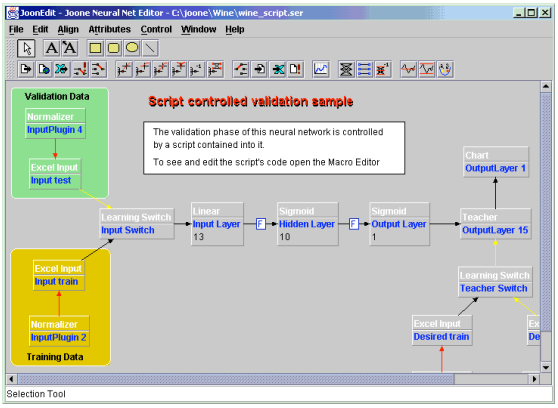


Figure 2.21: The Joone GUI editor.

Unfortunately at the time of writing the Joone project seems to have been abandoned with the last update to the code produced in mid 2007[i] and finding anyone currently working on the project is difficult.

Encog

Encog is a relatively new framework used to create neural networks and automated bots produced by Heaton Research. Encog can be used independently either to create neural networks or HTTP bot programs. Encog also includes classes that combine these two advanced features. Encog contains classes for Feedforward Neural Networks, Hopfield Neural Networks, and self-organizing maps. Training can be accomplished using backpropogation, simulated annealing, and genetic optimisation. Additional classes are provided for pruning neural networks (Heaton Research n.d.).

The Encog project (currently version 1.1) provides frameworks for Java and C# and its latest addition is a GUI builder called the Encog Workbench. The Encog Workbench is a graphical tool for creating and manipulating .eg files, which is the file format used by Encog to store exported neural networks. The Workbench allows you to change training data, the neural network architecture, train the network, modify the weight matrices and visualise the neural network.

Since Encog is relatively new there are not too many details or articles documenting it. This is to be expected and Heaton Research is in the process of producing a series of tutorials outlining its use.

[i] Data gathered from the Joone homepage at http://www.jooneworld.com/.

2.3.2 Pixel Overlay

The idea used in the pattern recognition example for ANNs can be extracted for use without the need for a neural network and it can be explained in the following steps:

- *Initial storage* – An ***ideal*** bitmap representation of each object to recognise is stored. From this the locations of the drawn (black) pixels and the undrawn (white) pixels are known.
- *User input capture* – The user draws their representation of their desired object on screen, which produces a bitmap of the input. The smallest rectangle around the input is drawn and the pixel coordinates are normalized relative to the top-left corner of the rectangle.
- *User/**ideal** input scaled* – The bitmap produced from the user's input can be scaled to the same size as the ***ideal*** representation bitmaps or vice versa.
- *Percentage overlay calculated* – The user input is compared to each ***ideal*** representation. The comparison is the percentage of the number of matching coloured pixels that overlay each other between the user input and each ***ideal*** representation.
- *Object classified* – Each object now has a percentage representing its match with the user input. The highest percentage can be selected and placed into the interface.

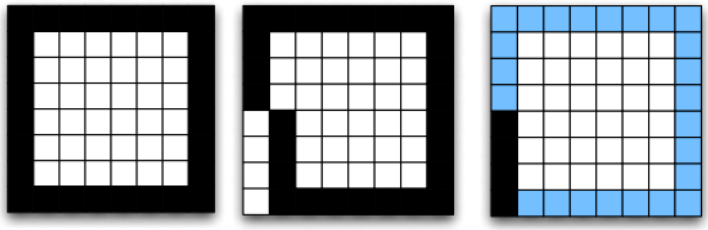


Figure 2.22: Pixel overlay example.

Figure 2.22 shows the pixel overlay technique. The left image is the ideally encoded representation of a text field. The middle image is the user's input; notice how it's slightly malformed. The image on the right shows the pixels from the user's input that match the ideal representation. The ideal representation has 28 coloured pixels and the user's input overlays with 24 of them. This gives a match of 24/28 pixels, which is approximately 86%.

Whilst this technique may seem intuitive and easy to implement, it does have problems associated with it. The first of which is scaling the user input or object representation. This technique relies on the input being a bitmap that is merely a collection of points (or pixels). Although this simplifies the collection method required to implement, scaling the image will remove characteristics of an image and possibly add distortion. This technique does not work well with even a minor amount of distortion in the data, as it is not an intelligent technique. This could be overcome with a more complex method of capturing the user's input. The input could be captured as vectors and this would allow the user input to be easily scaled or the object representations could be stored as vectors to allow simple scaling.

Even when the problem of scaling object representations is overcome there arises the matter of how accurate the user needs to be. A small angle on a line can mean that only a few pixels will overlap the intended object representation that will reduce the accuracy of the technique.

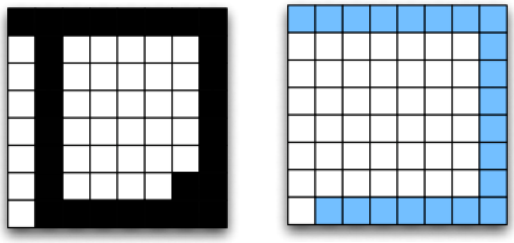


Figure 2.23: Malformed input to pixel overlay technique.

Figure 2.23 shows the malformed input on the left. This is a slightly noisy input that has had the smallest rectangle drawn around it. The right image shows how many pixels overlay. As can be seen, from a single pixel of noise the percentage has reduced from 86% to 75%. On a larger image the accuracy would have to be even greater.

The major advantage of this method other than its simplicity is the learning requirements of the system. Whereas a neural network will need to be pre-trained, deployed and have a neural network framework attached to it, the above stated method needs no training. A single ideal representation of an object can be supplied and software can be used. It has the capacity for a user to add custom representations of objects on the fly in the application without any change to the application or need for repetitive training tasks from the user.

2.3.3 Receptors

The receptor technique (Kirillov 2005) is an adaption of the pixel overlay technique that aims to permit a higher level of tolerance on the accuracy of the user's input whilst removing the problem of scaling. Instead of using the exact pixels found in the bitmap, an ordered set of ***receptors*** is laid over the input. A receptor is simply a short vector line defined within the bounds of the bitmap. Pixels from the bitmap will intersect with different receptors. In the case where a black pixel lays on a receptor, the receptor is considered activated. When no black pixels are found on the receptor line the receptor in considered inactive. Since the receptors are stored in vector form, they can quickly be scaled to match the size of the user input and calculating whether these receptors are active or not is simple. Once the state of all receptors has been calculated we have a pattern of receptors that represents the input (an active receptor is stored as a 1 and inactive is stored as 0).

The following method can be used to classify the object:

- *Receptor set calculation* – The set of receptors to be used is calculated (described in the next section).
- *Initial storage* – The receptor set is laid over each ***ideal*** bitmap representation of each object to recognise and the receptor pattern is calculated for each. From this we have a unique pattern of 1's and 0's the size of the receptor set to represent each. This pattern is then stored in a resource file for later comparison.
- *User input capture* – The user draws their representation of their desired widget on screen, which produces a bitmap of the input. The size of the input can then be calculated by looking at the leftmost, topmost, rightmost and bottommost pixel.
- *Receptor set scaled* – The receptor vectors are scaled to the size of the user input.
- *User input pattern calculated* – The receptor activity pattern is then calculated from the user input bitmap by looking at the intersections with the scaled receptor set.
- *Percentage overlay calculated* – The user input pattern is compared to each ***ideal*** representation. The comparison is the percentage of the number of matching in place

receptors. For example if the user input generates the pattern 1101 then it would have a 50% match with the pattern 1011.

- *Object classified* – Each object now has a percentage representing its match with the user input. The highest percentage can be selected and placed into the interface.

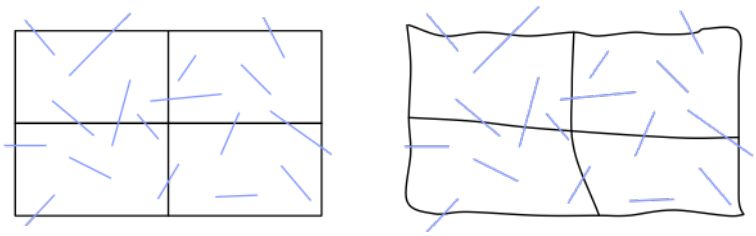


Figure 2.24: Receptor recognition example.

Figure 2.24 shows how the receptor technique's resilience to distortion in the input image. The left image shows receptors (blue lines) overlaying the ideal representation. The image on the right is the user's input and even though it is reasonably distorted, the receptors cross and don't cross in the same places which result in the same object being inferred.

This method, like the pixel overlay method would require no in-application training and would also permit users to add custom patterns for third party controls easily without the need to retrain as a neural network would require. As the receptor technique doesn't require an exact pixel for pixel match like the pixel overlay technique, it reduces the accuracy required by the user. The length of the receptor line determines how accurate the user needs to be but increasing the receptor length too much will reduce the ability to disambiguate between similar objects. Some sets of receptors will perform better than others and we discuss how to find an optimal set in the section.

Finding the Best Set of Receptors

The difficult part of using the receptor method is finding a set of receptors that can differentiate between the various interface widgets and give the user some margin for error. When considering what properties a good receptor needs to have, the following can be concluded:

- *Recognising variants* – A receptor is a good receptor if it can correctly identify different variants of the same object. This means that if it is activated for one variant it should be activated for another and likewise if it is meant to be inactive.
- *Differentiating objects* – The other goal of the receptor is to differentiate between different types of objects. Although a receptor may be able to identify different variants of the same object, it is no good if it active or inactive for every object. Ideally the receptor should be active for 50% of the objects and inactive for the other 50% of objects.

To find these properties of receptors, ***entropy*** from information theory can be used. Entropy is a quantitative measure of the randomness of an event. The more random the event, the higher the entropy. Entropy is defined by the following formula (Shum 2006):

$$\text{entropy} = -\sum_{x \in X} p(x) \log_b p(x)$$

- X = finite set of discrete random variables
- $p(x)$ = probability distribution function

A large set of random receptors can be generated, tested against some training data and then the best receptors can be picked using entropy.

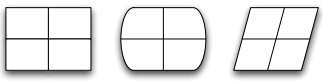
	Receptor 1	Receptor 2	Receptor 3
Text Field 	111	000	000
Button 	001	000	010
Grid 	110	010	111

Table 2.1: Example of receptor pattern encodings.

Table 2.1 shows the three interface objects drawn with three variants with three test receptors. Using this example, we show how entropy calculations can be used to pick the best receptors. We will only check the first receptor, the others can be calculated similarly. Each column represents a different receptor and the receptor patterns in the cells show if the receptor is activated for the corresponding variant of that interface element. For example the first cell shows 111 for receptor 1 with the text field. This means the receptor is active (a 1 in the cell) for all three variants of the text field. Two entropy calculations need to be carried out:

- *Entropy for variance* – Trivially the pattern 111 is good as it shows the receptor recognises every variant of the element. Here is the general form of calculation for the variant entropy

  p_1 = # of 1's in pattern / # of variants per object

  p_2 = # of 0's in pattern / # of variants per object

  $Entropy = -((p_1 * \log_2(p_1)) + ((p_2 * \log_2(p_2))))$

  Solving this equation for the first column gives us 0, 0.276 and 0.276. Here a low number is best.

- *Entropy for differentiation* – This will tell us how good the receptor is at differentiating the different objects. The general formula for this entropy calculation is this:

  p_1 = # of 1's in column / (# of variants per object * # of objects)

  p_2 = # of 0's in column / (# of variants per object * # of objects)

  $Entropy = -((p_1 * \log_2(p_1)) + ((p_2 * \log_2(p_2))))$

Solving this equation for the first column collectively gives the entropy for differentiation of 0.276. In this situation, a value close to 1 is preferred because this means the receptor can differentiate well.

Now, the entropies for variance and the entropy for differentiation of the first column have been calculated, the usefulness of the receptor can be calculated. The usability of the receptor can be found by combining the various entropies and is defined by:

$$Usability = Entropy\ for\ difference * (1 - Average\ entropy\ for\ variance)$$

The receptors with the highest usability should be kept and those with poor usability can be thrown away, leaving us with a set of good receptors that are tolerable to variance and good at differentiating objects[i].

2.3.4 Case-Based Reasoning

A Case-Based Reasoning (CBR) system is an instance base learning technique that solves new problems based on solutions to similar past problems. This mirrors an idea from how humans solve new problems. A CBR contains a case base of past problems together with the solution and in some cases annotations of how to achieve the solution.

When a new problem is encountered that is not found in the case base, the CBR searches for the closest known problem. The solution to the known problem is proposed as the solution to the new problem. This can be accepted or modified by a user of the system. Once a solution is accepted, a link between the problem and the solution is added to the case base. This provides the CBR with new cases that can be matched against in the future. This is an example of lazy learning. Lazy learning is a learning method in which generalization beyond the training data is delayed until a query is made to the system. This means the system can adapt to a changing problem domain. Figure 2.25 visualises the mapping from the problem space to the solution space:

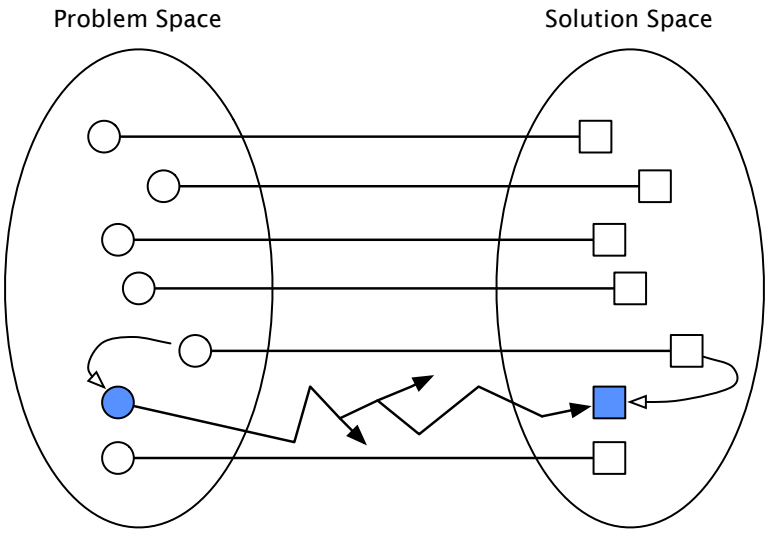


Figure 2.25: Problem solving using CBR.

[i] Method adapted from Andrew Kirillov, "CodeProject: Neural Network OCR," *CodeProject*, 11 04 2005, http://www.codeproject.com/KB/cs/neural_network_ocr.aspx (accessed 01 12, 2009).

The newly learned cases can be used in finding solutions to new problems. This process is called a work cycle and Aamodt and Plaza classify it as the following four steps which is visualised in Figure 2.26 (Aamodt and Plaza 1994):

- *Retrieve* – Given a problem to solve, retrieve the most similar cases from the case base.
- *Reuse* – Take these similar cases and use their known solutions in an attempt to find a solution for the new problem.
- *Revise* – Allow the proposed solution to be changed or tweaked if necessary.
- *Retain* – Add the new problem with the solution as a new case to the CBR.

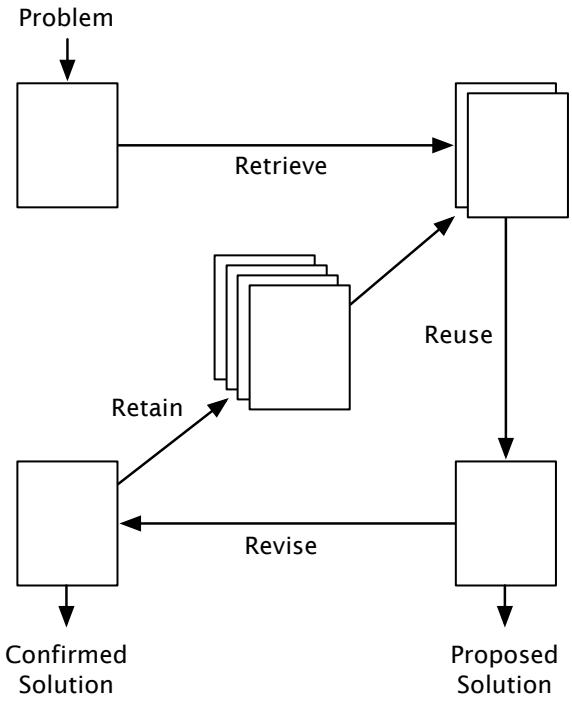


Figure 2.26: CBR work cycle.

These notes on CBR have been adapted from Maja Pantic's course notes titled 'Instance Based Learning' (Pantic n.d.).

Case-Based Reasoning for Pattern Recognition

Pattern recognition, as described on in section 2.3.1, can be achieved with a CBR system. We can use the following steps to achieve pattern recognition with a CBR system:

- *Encode known cases* – We choose an encoding such as the pixel overlay (see section 2.3.2) or receptor (see page 2.3.3) technique to represent our cases. We take each pattern we want to recognise and encode them with one of these techniques. We add this encoding as the problem to our case base and set the solution to be the name of the pattern.
- *Encode unknown case* – Given a pattern as input, we apply the same encoding technique as we did to our known patterns.
- *Look for similar cases* – Now we have the encoding for the input pattern we can look for k-Nearest Neighbours in the case base.
- *Propose a solution* – The nearest neighbour can be proposed as a classification of the pattern but we allow the user to possibly change the classification if need be.
- *Learn the new pattern* – The new pattern can now be added to the case base and proposed as a solution in future problems.

2.4 Layout Management

> "The ultimate metric that I would like to propose for user friendliness is quite simple: if this system was a person, how long would it take before you punched it in the nose?"
>
> Tom Carey

With improper layout management code, things don't work the way users expect and this is frustrating for them. Unfortunately for developers, writing layout code that displays the intended layout of widgets and scales intelligently as the window is resized is not an easy task to accomplish. Here we discuss the importance of layout management code, what can happen if it goes wrong and overview the available layout managers to help get it right.

2.4.1 Basic Layout Techniques

In application development there a few paradigms that developers can employ to layout interface widgets irrespective of the GUI toolkit being used. Here we explore the layout concepts available to developers.

Absolute layout

Absolute layout is the simplest layout formalism by a mile although it requires pixel perfect placement. Using an absolute layout, the developer must specify the size and location of the widget to the pixel. The advantage of this is that you have complete control of where the widgets are placed but the downside is automatic resizing is not available when the window is resized. If using a GUI builder then layout hints can be given to the user at design time to help align widgets but in pure code based form, sizes and locations have to be specified explicitly. Figure 2.27 shows an example of Java Swing using absolute layout (sometimes called `null` layout):


```
import javax.swing.*;
public class Demo extends JFrame {
    public Demo() {
        JButton btnOk = new JButton("OK");
        JButton btnCancel = new JButton("Cancel");

        btnOk.setLocation(5, 5);
        btnOk.setSize(100, 30);
        btnCancel.setLocation(115, 5);
        btnCancel.setSize(100, 30);

        this.setLayout(null);
        this.add(btnOk);
        this.add(btnCancel);
        this.setSize(220, 60);
        this.setResizable(false);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

Figure 2.27: Java Swing absolute layout.

Grid layout

Grid-based designs provide systematic structure to an interface. By structuring each window or view along similar lines, a grid ensures that as users gain more experience with the system, they learn to predict where a particular piece of information will be found (Mullet and Sano 1995). This is why grid-based systems are so good; they help the users. Luckily most modern day GUI toolkits provide a grid based layout manager and there are plenty of third party layout managers that support the grid-based design paradigm. Grid-based layout managers involve a developer adding controls to particular cells in a predefined grid. As the frame is resized, so too do the cells defined in the grid. Figure 2.28 is an example of a grid-based layout in Java Swing:


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

public class Demo extends JFrame {
    public Demo() {
        JButton btnOk = new JButton("OK");
        JButton btnCancel = new JButton("Cancel");
        getContentPane().setLayout(new GridLayout(1,0,10,10));
        this.add(btnOk);
        this.add(btnCancel);
        this.setSize(220, 60);
        this.setResizable(false);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

Figure 2.28: Java Swing GridLayout.

Docking

Docking layouts involve fixing interface widgets on the sides of the frame. Typically the toolkit API offers the user ability to dock a widget north, east, south, west and centre and some layout managers offer the power to do multiple dockings per side. This means widgets can be docked to other widgets. Figure 2.29 is an example of Java Swing's BorderLayout, which gives docking abilities to the developer:

```
import java.awt.BorderLayout;
import javax.swing.*;
public class Demo extends JFrame {
    public Demo() {
        getContentPane().setLayout(new BorderLayout());
        this.add(new JButton("north"), BorderLayout.PAGE_START);
        this.add(new JButton("south"), BorderLayout.PAGE_END);
        this.add(new JButton("east"), BorderLayout.LINE_START);
        this.add(new JButton("west"), BorderLayout.LINE_END);
        this.add(new JButton("center"), BorderLayout.CENTER);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

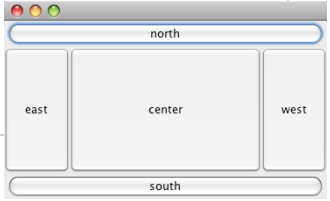


Figure 2.29: Java Swing BorderLayout.

Anchoring

Anchoring is a technique not employed by many layout managers directly in code but is more of an effect seen by the user in the interface. Widgets can be anchored to each other or to a side of their containers so they resize as their counterparts do. GroupLayout has the ability to create these relationships between widgets. Figure 2.30 shows an example of what happens to a text field when it is anchored to the right-edge of its parent container and the container is resized:

Figure 2.30: Anchoring example.

Springs and Struts

Springs and struts are heavily used in the Mac OS X Interface Builder and in some third party layout managers for other GUI toolkits. A strut can be seen as an anchor. If a strut is attached to a child component and it's parent container then when the parent is resized, the child is stretched to maintain the child's size. A spring is exactly the opposite. If the strut were replaced by a spring then the child would remain the same and the spring would just stretch. Figure 2.31 shows an example of a spring being used in a toolbar in OS X:

Figure 2.31: OS X Toolbar spring example.

2.4.2 The Dark Art of Java Layout Management

Whilst Java does offer some basic layout managers in the JSE it is by no means complete and attempting to create a complex user interface is a difficult task. In October 2006, John O'Conner of the Java community proposed a layout manager showdown[i]. He proposed a challenge to the community to use their favourite layout managers to produce an example complex user interface. There are some delicate intricacies of the interface that require very fine control over the layout management. Figure 2.32 shows the testing layout.

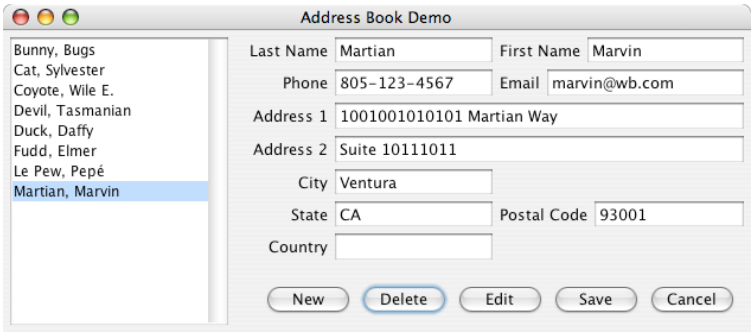


Figure 2.32: Layout Manager Showdown interface.

[i] John O'Conner's layout manager showdown can be found on his java.net blog here at http://weblogs.java.net/blog/joconner/archive/2006/10/layout_manager.html

Achieving this layout with a single existing JSE layout manager is impossible. Using a combination of the JSE layout managers, something similar can be created using nested panels but nesting panels is not recommended. The problem with nesting panels is mixing the different types of layout managers. Because the layout managers were contributed by different people and added to the JSE at different points, they all treat margins, insets and outsets differently between controls and this is where a single powerful layout manager can help.

Existing Java Layout Managers

Here the JSE layout managers are detailed with small examples to show what is available with Java out of the box.

- *Null layout* – Null layout is Java's version of absolute layout as exampled in figure 26. It requires the coder to specify the size and location of each interface widget and provides no automatic relocation or resizing of the widgets.
- *BorderLayout* – BorderLayout is the docking style layout manager as demonstrated in figure 28. It allows components to be docked to the inside edge of its container. This is generally used to dock other panels around a central panel that is used for the main content.
- *BoxLayout* – The BoxLayout layout manager (shown in Figure 2.33) allows components to be stacked vertically on top of each other or aligned in a row horizontally. In this sense it can be seen as a more advanced version of FlowLayout. Users add components to the layout manager and can specify springs and strut sizes between components.


```
import javax.swing.*;
public class Demo extends JFrame {
    public Demo() {
        setLayout(new BoxLayout(getContentPane(), BoxLayout.LINE_AXIS)); //
LINE_AXIS means layout in a row

        // Spring!
        this.add(Box.createHorizontalGlue());
        this.add(new JButton("OK"));
        // Strut!
        this.add(Box.createHorizontalStrut(5));
        this.add(new JButton("Cancel"));
        this.setSize(250, 60);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

Figure 2.33: BoxLayout example.

- *CardLayout* – The CardLayout can be seen as a tabbed pane without the predefined GUI. It lets the developer implement an area that contains different components at different times. It is usually controlled by a combobox that triggers the change of layout. Multiple panels are coded using whichever layout manager is desired and the panels are added to the CardLayout. When the attached event is triggered, the CardLayout switches to the next panel.
- *FlowLayout* – FlowLayout is the original JSE layout manager and is the default for every JPanel. Like the BoxLayout, FlowLayout can layout components in a single row, wrapping onto a new row if the parent container is not of sufficient size.

- *GridBagLayout* – GridBagLayout is the most powerful and flexible layout manager available with the JSE. It positions items in a grid based system where the components can span multiple cells and the column and rows can be of varying widths and heights. Unfortunately compared to some of the third party layout managers the code is verbose and not as intuitive as could be[i].
- *GroupLayout* – GroupLayout is the layout manager used by the NetBeans GUI builder. NetBeans creates very verbose code with GroupLayout that is near impossible to read and very difficult to modify by hand (for an example, see figure 3.1). This is not a problem as it is not designed to be used with manual coding and was created for operability with GUI builders in mind (Sun Microsystems n.d.).
- *SpringLayout* – SpringLayout was also designed with GUI builders in mind and as a result is very low level. It is for this reason that Sun recommends not coding it by hand (Sun Microsystems n.d.). The SpringLayout works by defining the relationships (also called constraints) between the component's edges and the position of each edge is dependent only on another single edge. These constraints are defined in pixels and the layout manager uses the minimum, preferred and maximum sizes of components to calculate positions and sizes of objects. Using these properties of the component (which must be explicitly defined in code), springs are placed between the edges that are free to shrink and expand according to the properties.

Obviously not all layout managers are meant to be able to produce all types of layouts. The old mentality was to nest panels using multiple layout managers to create complex interfaces but as discussed already there are problems with nesting panels. The other problem with the standard JSE layout managers (not including the new GroupLayout) is the matter of units. For all of these layout managers, only pixel references are supported and this doesn't leave room for resolution independence[ii]. Fortunately some powerful third party managers have resolution independence built in.

Third Party Layout Managers

Here we discuss some of the more powerful Java layout managers. These layout managers can create complex interfaces without the need of nesting panels. The same example interface is used in each instance so we can compare the code required.

- *MigLayout* – MigLayout is probably the most popular third party layout manager available. Developed by Mikael Grev, it is open source, flexible and works with both Swing and SWT. MigLayout can replicate the functionality of all the existing JSE layout managers except for FlowLayout as MigLayout uses a grid-based system. It is capable of pure grid-based layout with support for baseline alignment[iii], absolute layouts with relative component constraints and docking all with resolution independence.

  The key strength to MigLayout is the conciseness of the code required for even complex layouts. Global layout constraints, row/column constraints and component constraints are all specified in the constructor of the layout manager. All the component's constraints are specified in one line, inline with adding the

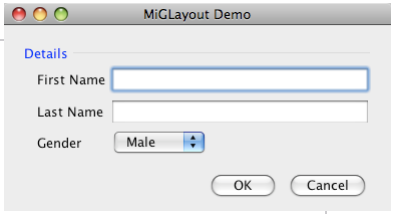
[i] For an example see: http://javatechniques.com/blog/gridbaglayout-example-a-simple-form-layout/

[ii] Resolution independence is the notion that something can be drawn at sizes independent of the pixel grid/screen size being used.

[iii] Baseline alignment was introduced in Java 6 and it aligns the bottom of the text (the baseline) across widgets.

component to the layout. This means only one line is required per widget, which greatly reduces the code size.

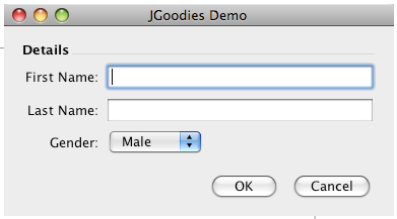
The code is more readable too. MigLayout uses short string-based constraints that give more meaning to the user and are just as accessible for GUI builders. Figure 2.34 shows an example interface coded up in MigLayout that shows just how concise and readable the layout code can be:




```
import java.awt.Color;
import javax.swing.*;
import net.miginfocom.swing.MigLayout;
public class Demo extends JFrame {
    public Demo() {
        // Global constraints
        MigLayout lm = new MigLayout("ins 20",
                       "[para]0[][100lp,fill][60lp][95lp, fill]", "");
        this.setLayoutManager(lm);
        // All other controls
        String[] genders = new String[] {"Male", "Female"};
        JLabel label = new JLabel("Details");
        label.setForeground(Color.blue);
        add(label,                      "split, span");
        add(new JSeparator(JSeparator.HORIZONTAL), "growx, wrap");
        add(new JLabel("First Name"), "skip");
        add(new JTextField(),         "span, growx");
        add(new JLabel("Last Name"),  "skip");
        add(new JTextField(),         "span, growx");
        add(new JLabel("Gender"),     "skip");
        add(new JComboBox(genders),   "wrap");
        add(new JButton("OK"),        "split,right,span");
        add(new JButton("Cancel"),    "wrap");
        pack();
        this.setVisible(true);
        this.setTitle("MiGLayout Demo");
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

Figure 2.34: MigLayout code example.

- *JGoodies FormLayout* – The JGoodies FormLayout was one of the first third party layout managers to achieve a following. It doesn't have the docking and absolute positioning capabilities that MigLayout offers but it was never intended to. Where JGoodies FormLayout proves its worth is creating simple to complex forms interfaces with a grid-based solution. It is simple to learn as the method for building these forms is abstracted away from the user. Instead of adding each control to the layout manager in term and specifying constraints, the developer adds the control to the builder with the location of a cell in the grid they want the control to be placed. Optionally, the number of cells to span vertically and horizontally can be specified. The builder then adds the control to the layout taking care of gaps and precise alignment to produce a professional looking form. Figure 2.35 shows the same screen from figure 33 coded using the JGoodies layout manager:




```
import javax.swing.*;
import com.jgoodies.forms.builder.PanelBuilder;
import com.jgoodies.forms.layout.*;
public class Demo extends JFrame {
    public Demo() {
        // Specify the row and column constraints
        FormLayout lm = new FormLayout(
            "right:pref, 3dlu, pref, 7dlu, right:pref, 3dlu, pref",
            "p, 3dlu, p, 3dlu, p, 3dlu, p, 9dlu, p");
        lm.setColumnGroups(new int[][]{{1, 5}, {3, 7}});
        PanelBuilder builder = new PanelBuilder(lm);
        builder.setDefaultDialogBorder();
        // Obtain a reusable constraints object
        CellConstraints cc = new CellConstraints();
        String[] genders = new String[] {"Male", "Female"};
        builder.addSeparator("Details", cc.xyw(1,  1, 7));
        builder.addLabel("First Name:", cc.xy (1,  3));
        builder.add(new JTextField(),   cc.xyw(3,  3, 5));
        builder.addLabel("Last Name:",  cc.xy (1,  5));
        builder.add(new JTextField(),   cc.xyw(3,  5, 5));
        builder.addLabel("Gender:",     cc.xy (1,  7));
        builder.add(new JComboBox(genders), cc.xyw(3, 7, 2));
        builder.add(new JButton("OK"),     cc.xy (5,  9));
        builder.add(new JButton("Cancel"), cc.xy (7,  9));
        this.add(builder.getPanel());
        this.pack();
        this.setVisible(true);
        this.setTitle("JGoodies Demo");
    }
    public static void main(String[] args) {
        new Demo();
    }
}
```

Figure 2.35: JGoodies code example.

These are two of the more popular third party layout managers available and there are many more with varying features and goals. The aim of this section was to show how complex layouts could be made with a single layout manager in a single panel without nesting panels[i].

2.4.3 Inferring Correct Layout

Now we have a clear grounding of the basics of layout management and the type of code that needs to be generated, we can discuss inferring correct layout management code.

To start with we can look to how existing GUI builders create their layout code and why this tool shouldn't do the same. Current GUI builders integrate tightly with a layout manager. As the user moves a widget around the design screen, the GUI builder is only permitting movement to a valid location as specified by the layout manager and this allows them to generate layout management code on the fly. This restricts where the user can position the interface widgets and often the GUI builder will resize the widgets to fall inline with the layout manager. Wouldn't it be nice if the user could move and size their widgets exactly how they want?

The goal of this project is to provide a more natural and intuitive method to designing GUIs and this can be achieved by removing the constraints of a layout manager at design time. This has two major side effects for the application and the user. Firstly, the user

[i] For a good directory listing of third party layout managers see: http://leepoint.net/notes-java/GUI/layouts/90independent-managers.html

will not get a 100% accurate preview of their application at design-time as they will be aligning items roughly where they want. Showing layout hints and snapping the components to an alignment grid at design-time can improve this. This will allow a completely free design for the user with no restrictions.

The second impact is to how the application generates layout code. An analysis needs to be carried out in an attempt to infer the layout code from the user's positioning of widgets. In a way this is reversing the function of the layout manager. A layout manager takes constraints and calculates the size and location for the object but we need to do the opposite and infer the constraints from the locations and sizes. This is no easy task and it doesn't look to have been tried before without deep integration to a layout manager. Our aim is to infer correct layout code with no input from the user except the location of widgets on the screen.

As interface design is generally grid based and the layout code is grid based we can use this to our advantage. Using a grid based layout manager; the layout manager code is generally specified row-by-row adding components from the top-left of the screen down to the bottom-right. To infer the layout code we can use the following steps:

- *Infer number of rows* – Using the locations Y coordinates of the components we can calculate which components belong to which row of the grid.
- *Infer number of columns* – The same technique can be used with the X coordinates to understand where the columns lie.
- *Infer row/column constraints* – Using the sizes together with the locations of the components we can build an in-memory representation of the layout and build the constraints as a result.

Certain rules will need to be considered when trying to generate the constraints for the components. Following good interface guidelines, only particular widgets will want to be resized in different ways and this will need to be set on a per widget basis. For example, using intuition we can tell that labels and text fields don't need to be vertically resized because they're single line displays and inputs whereas a text field will want to be expanded both horizontally and vertically. Using these intuitive rules we can generate constraints that mimic the desired layout of the user.

3 Design Considerations

3.1 Aims

The core aim of this project is to provide a more intuitive, natural tool for developers to build GUIs. We can achieve this by accomplishing the following four objectives:

- *Capturing natural input* – Let the developer create interfaces more naturally. This involves the developer drawing the interface with a mouse, a graphics tablet or a touchscreen and turning that drawing into a working GUI.
- *Ease layout management* – Remove the knowledge barrier of toolkit specific layout managers from the task of layout management. We aim to automatically generate correct layout code with little to no user intervention. In the event that the perfect layout cannot be generated, we need to let the user tweak and correct it.
- *Abstract the toolkit* – Remove the implementation details of the toolkit from the design process. A window is a window in any toolkit, so the user shouldn't need to know it's a JFrame in Swing but a Shell in SWT. An intermediate representation can be used internally to allow code generation into multiple toolkits.
- *Generate clean, concise code* – Existing GUI building tools generate bloated code for the programmer. They certainly make it easier for users to create GUIs but generating excessive code compromises the long-term maintainability of the code. We aim to solve this using a layout manager that requires minimal layout code and only generating code that is absolutely necessary.

We will now look at the limitations of existing tools and techniques in order to formulate our own solution to GUI building.

3.2 Critique of Existing Techniques

3.2.1 Problems with Manual Coding

Coding GUIs by hand in procedural languages is the traditional method used to create user interfaces but it is not without it's problems. Ben Galbraith is an experienced member of the worldwide computing community, has founded several companies and has worked for companies such as Sun (working on Java) and Mozilla focusing on user interaction with applications. Here is a comment by Ben about designing GUIs by hand:

> "The domain of GUI design is, err, visual, and doing design by writing code either means you've got to get to the point where you can reliably visualise what your layout manager does in your head (emphasis on reliably, folks), or you wind up compiling and executing over and over again doing minor/major tweaking until you finally get it right."

Here Ben points out the key problem with creating GUIs by hand. Unless the developer has a deep level of understanding of the necessary layout manager(s), a cycle of recompiling and rerunning is required to build the interface. This is because unless an IDE with a live preview is in use[i], the user must visualise the interface layout in their head.

Another problem with manual coding is the abstraction of the toolkit. When writing procedural code, the developer needs to understand the paradigms and varying APIs of the interface widgets used in the toolkit. A button in Swing has particular properties that are common to a button in SWT but very different code is needed to produce the same interfaces across the toolkits.

GUI design should be independent of the toolkit in question and this is where declarative languages have an advantage.

3.2.2 Problems with Current GUI Builders

As mentioned in the above section, the domain of GUI design is visual and this is a very good reason to use a GUI builder over coding by hand. But GUI builders still have some major drawbacks that shouldn't be overlooked.

GUI designers often provide a live WYSIWYG view of what is being designed but these aren't necessarily what the compiled interface will look like. For example it is often difficult to understand how the widgets will resize when the frame is resized in the application. This still requires recompiling and rerunning of the generated code although it generally removes the knowledge barrier regarding the GUI toolkit. A user simply drags the item that looks like a button onto the window and places it where they want.

Although the GUI builder can generally abstract the widget toolkit from the developer, it doesn't remove the requirement of layout manager knowledge and this doesn't follow the Low Threshold ideology. The Low Threshold ideology is part of the 'Low Threshold, High Ceiling and Wide Walls' design goals (Resnick, et al. 2005). These goals state an application should be easy for a novice user to start using with little to no training (Low Threshold); the application should have the capacity to satisfy an experienced user's requirements (High Ceilings); and the application should also support and suggest a wide range of explorations (Wide Walls). In the case of Java layout managers, GUI builders are generally geared to work with specific layout managers. Dragging a widget around the window doesn't necessarily mean it will move as expected. The layout manager will control the location and size of the widget and override any actions performed by the user. Not understanding the layout managers can make using most GUI builders a frustrating process. In the case of the .NET WinForms, there aren't a wide variety of layout managers available. This results in the majority of complex layouts being unachievable without nesting multiple panels, which is a bad idea. Different panels with different layout properties respect widget insets in different ways and often don't support resolution independence. The only case where this doesn't apply is when no layout manager is used. Widgets are positioned using absolute coordinates but absolute layouts provide no automated resizing and no automated widget alignment.

In Java, this problem was resolved with the development of the GroupLayout layout manager. GroupLayout integrated with NetBeans provides a very intuitive means to create

[i] Currently only Jigloo (see section 2.2.2 for details) provides a full live preview for Java through an Eclipse plug-in and the WPF designer provides live preview for XAML.

interfaces that scale correctly when resized. The biggest flaw with GroupLayout and NetBeans is the code that is produced. Not only is it guarded, as described in section 2.2.2 regarding NetBeans, but the code produced is far from maintainable. Below is an example of the layout code produced by NetBeans when creating a simple form:

```
private void initComponents() {
    lblName = new JLabel("Name:"); txtName = new JTextField();
    btnCancel = new JButton("Cancel"); btnOk = new JButton("OK");

    // Layout code //
    GroupLayout layout = new GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(layout.createSequentialGroup()
                    .add(lblName)
                    .add(18, 18, 18)
                    .add(txtName, GroupLayout.DEFAULT_SIZE, 167,
                                                    Short.MAX_VALUE))
                .add(GroupLayout.TRAILING,
                    layout.createSequentialGroup()
                      .add(btnOk)
                      .addPreferredGap(LayoutStyle.UNRELATED)
                      .add(btnCancel)))
            .addContainerGap())
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(layout.createParallelGroup(GroupLayout.BASELINE)
                .add(lblName)
                .add(txtName, GroupLayout.PREFERRED_SIZE,
                              GroupLayout.DEFAULT_SIZE,
                              GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(LayoutStyle.RELATED,
                          GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .add(layout.createParallelGroup(GroupLayout.BASELINE)
                .add(btnCancel)
                .add(btnOk))
            .addContainerGap())
     );
     pack();
}
```

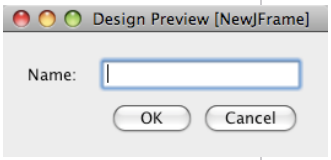


Figure 3.1: NetBeans generated GroupLayout code with the form it represents.

Figure 3.1 shows the auto-generated code from NetBeans to create the simple form with a label, text field and two buttons. Whilst it may behave as expected by the user (the form resizes correctly as expected) the code is verbose and there are much more concise ways of representing the same GUI[i].

GUI builder tools are an improvement but problems still exist. Often, the code produced is bloated (see figure 3.1 for an example of bloated NetBeans code and see figure 2.9 for an example of bloated Visual Studio (VS) .NET code) and an in-depth knowledge of layout managers is needed to work with them. The code produced may be unreadable, or in

[i] For a MigLayout implementation if the same GUI, see page *_*.

the case of the NetBeans IDE, even un-editable (NetBeans will overwrite any changes you make to it's generated GUI code).

3.2.3 Problems with Specifying in Markup Languages

Attempting to describe user interfaces is not an easy task especially in a toolkit independent manner. Different toolkits use different interaction paradigms and designing a specification to suit all toolkits hasn't been accomplished. XAML is the best-supported user interface markup language mainly due to Microsoft's influence in the application development arena. Other attempts at standardizing a markup language for specifying GUIs have largely gone unnoticed.

A key advantage that XAML has over other markup language specifications is good tool support. VS provides a live drag-and-drop designer that can produce XAML code, furthermore it is a roundtrip editor[i]. Other GUI markup languages have little or no good tool support. This still means that designing an interface is purely code based with the recompile and rerun mentality that comes with it.

The verbosity of XML is also a problem. XML is the method of choice for describing user interfaces, but one can argue that this isn't the best choice. Just because XML can be used to describe user interfaces does not necessarily mean that it should. There are other markup languages, with far less verbosity, that may provide a better fit such as YAML and JSON (a subset of YAML).

The last thing to discuss with regard to markup languages is the abstraction of toolkits. The .NET implementation of XAML for WPF still requires knowledge of the toolkit and how various layout components act. The Swing javabuilders project (see section 2.2.3 for details) requires that you know the components available in Swing, but does take two steps in the right direction by using YAML and providing its Layout DeScription Language (Layout DSL) to describe the layout of widgets.

3.2.4 Conclusion of Criticisms

There are many problems with existing GUI building methods that we've outlined above. Although there are numerous good ideas being implemented, certain key problems still exist and a total solution to solve all these problems is missing. Knowing these limitations, we will now design our solution to the GUI building problem. We will attempt to design a solution that overcomes the limitations in existing techniques and achieve the aims we stated in 3.1.

3.3 Design

Our idea is a standalone tool, Sketchi, which takes a user through the complete process of turning a sketch into runnable GUI code. The main aim is to turn a user's sketch into concise GUI code as fast as possible whilst requiring minimal intervention along the way.

[i] Roundtrip means that a change in code will result in a change in the interface and a change in the interface, through user interaction, results in a change in the code.

3.3.1 High Level Architecture

The process of turning a sketch into code consists of four very well defined steps. Sketchi takes the user through each of these stages by letting them correct any errors that may have been made along the way. Figure 3.2 shows the high level design of the flow through the application.

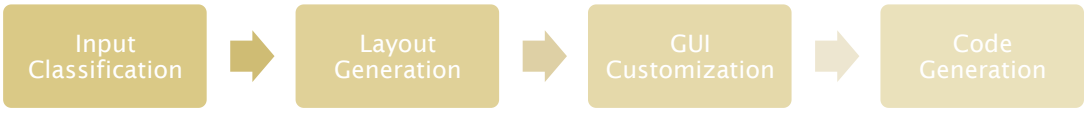


Figure 3.2: Sketchi design flow.

The implementations of each component are detailed in the subsequent chapters of this report. Each module is self-contained with only the inputs and outputs visible to the other components. We initially defined the interfaces between the modules so we were free to implement them as and when chosen.

Such a modular design also provides us with great flexibility for future work. We can easily replace any module with a different implementation that uses other techniques.

3.3.2 Implementation Choices

There were numerous implementation choices to consider when designing Sketchi. Each module in our architecture could be implemented in a number of ways. We state below which we chose and why:

- *Implementation language & toolkit* – We've implemented Sketchi using Java and the Swing widget toolkit. We chose Java so we can provide a platform independent tool. We've used Swing as opposed to other Java widget toolkits for two reasons. Firstly, Swing is part of the Java Standard Edition and so ships with the main Java distribution. Other toolkits need to be downloaded separately. Secondly, Swing is lightweight. This means the rendering is performed entirely in Java 2D, which allows us to provide a high level of customization for our application and give our users an immersive, rich interface.
- *Classification Technique* – When building the input classification subsystem we used a Case-Based Reasoning (CBR) system with receptor patterns as the core feature (see section *_* for details of receptors). We use CBR as opposed to an Artificial Neural Network (ANN) because CBR is the more intuitive method. If we cannot achieve satisfactory results using the CBR technique then we will use an ANN system. Another advantage of CBR over ANN is the ability to instantly learn new data. Since CBR uses lazy learning, we can instantly add new cases to the case base. Using an ANN would require us to retrain the entire network every time a new instance needs to be learnt. We use the receptor pattern as our core feature as it is tolerant to variations in the user drawing (see discussion on receptors in the background section 2.3.3).
- *Layout Manager* – When implementing an automatic layout generation system, we need to base it on a particular layout manager. Unfortunately this isn't very flexible but is necessary in order to generate good layouts for the user's sketch. This is due to layout managers having subtle differences that have large effects on the layout produced. We've chosen the popular third party layout manager MigLayout. We've chosen MigLayout because the layout code required to use it is very concise and

clean (see section 2.4.2 for a comparison of layout managers). Moreover, its string based constraint system is easy to read and intuitive to use. The standard layout managers provided with Java don't meet our requirements of being able to produce clean, concise code whilst handling potentially complex layouts. MigLayout also uses the popular grid-based paradigm that numerous other layout managers use. This means generating code for a toolkit unsupported by MigLayout will only require translating our layout constraints to the constraints of another grid-based layout manager used in the toolkit.

3.3.3 Application Components

Sketchi is written in Java and uses the Swing widget toolkit. We use Swing as opposed to SWT because it is lightweight, so we can provide a customised, rich interface for our users. Swing let's us create custom components so we can tailor the interface to respond well to a touchscreen and immerse the user in the application. It uses a wizard style with clearly defined stages to guide the user through the creation of their GUI. After initially sketching their GUI onscreen, they use the onscreen options to progress through the necessary steps to full code generation. In figure 3.3, we detail the application components and show the stages necessary to produce GUI code from a sketch:

Figure 3.3: Sketchi application hierarchy.

Components can always communicate down through the hierarchy. The only upward communication allowed is straight to the application HUB.

Application HUB

The application HUB is the starting point of the application. It contains the main method to run the application. When Sketchi is started, the HUB loads up any external resources (such as user preferences), sets up the input recognition system with the user's case base and loads up the main window.

Any communication between modules is marshalled through the HUB. It keeps track of the current ***state*** of the application (see application states in section 3.3.4) and is responsible for progressing between the stages in the wizard. Using the application HUB as a mediator maintains independence between the various stages, so we can swap new implementations in and out at a later stage if desired.

Main Window

The main window is the container for all interaction between the user and Sketchi. It consists of the following components:

- *Toolbar* – The toolbar contains access to common functionality such as creation, loading and saving of sketches. It also contains a button to access preferences where the user can customise functionality of the application.

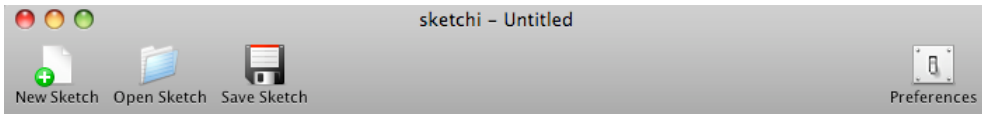


Figure 3.4: Sketchi toolbar.

- *Floating toolbox* – The floating toolbox acts as a single global toolbox for all application states. The contents of the toolbox are specific to each stage in wizard. When a new state is loaded, it passes its toolbox contents to the HUB, which propagates the contents down to the toolbox.

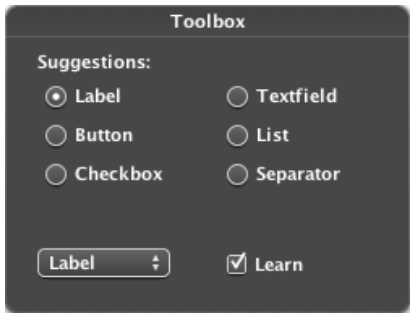


Figure 3.5: Sketchi toolbox.

- *Open File sidebar* – The sidebar lists the currently open files as well as the most recently opened files. It is populated with data from the HUB. When an item from the list is selected, a message is sent up to the application HUB that will switch to a new state and propagate this down to the content pane.

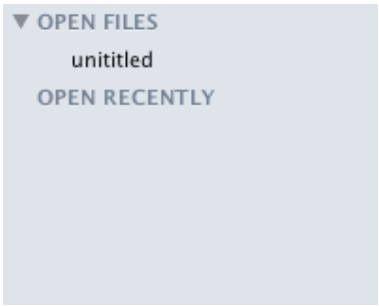


Figure 3.6: Sketchi sidebar.

- *Breadcrumb bar* – The breadcrumb bar shows the current state the user is in. It provides the user with a reminder of the active state in the content pane.

Figure 3.7: Sketchi breadcrumb bar.

- *Content pane holder* – The main window contains a placeholder for the content pane, which is the main point of interaction for the user. Whilst the main window knows the content pane exists, it never knows the current state of the content pane. The application HUB may provide a new state for the main window to load, but this is then delegated down for the content pane to handle.

Figure 3.8: Sketchi content pane.

Content pane

The content pane is the main point of interaction between the user and Sketchi. It acts as a container for various application states (see application states in section 3.3.4).

Training Window

The training window is used to train the input classifier with a user's particular drawing style. The user is asked to draw sketches of the GUI widgets one by one to train the classifier. The sketches they draw are supplied to the HUB, which uses them whilst setting up the input recognition subsystem.

3.3.4 Application States

We split generating a GUI into six major steps and each one of these steps is a possible ***state*** that can be active in the content pane. The six possible states that can be active in the content pane are:

- *Free Design* – The Free Design state is a acts as a virtual paper for the user to draw on. The user sketches their GUI on the paper that is then processed by the next application state. The toolbox for the Free Design state contains instructions on what to do and a button to clear the sketch.
- *Widget Detection* – The Widget Detection state has a similar appearance to the Free Design state but doesn't allow the user to draw on the screen. The Widget Detection

state applies an algorithm to find the widgets in the user's sketch (see input detection and classification in section 4). The user's sketch appears on the page with rectangles around the widgets in the drawing. If the algorithm doesn't arrive at the correct solution, the user can correct it by deleting existing rectangles and selecting the widgets from the sketch themselves. The toolbox contains instructions for doing this, as well as a 'selection' button and an 'eraser' button for activating the two tools.

- *Widget Classification* – The Widget Classification state attempts to classify the widgets the user has drawn. It looks very similar to the previous two states but each rectangle is now labelled with the classification of the widget. If the classifier hasn't classified a widget correctly, then the user can correct it. The user changes the classification by clicking on the object on screen and selecting a new classification from the toolbox. The toolbox contains the top suggested classifications for the currently selected widget. When the user progresses to the next state, any corrections the user has made are learnt by the recognition system (see section 4.2.6 for details of the learning mechanism).
- *Layout Selector* – The Layout Selector state takes the classified drawing and attempts to find a correct layout for the GUI. It is supplied with the widgets' classifications, sizes and locations. From these, it generates possible layouts and displays them all to the user (see the automated layout generation in section 5). The user can live preview a layout by right clicking it[i]. The live preview opens a new window with real GUI controls in it. This allows the user to resize and interact with the layout to see how it acts and if correct, can select it and proceed to the next state. The toolbox contains instructions on how to preview and progress with a chosen layout.
- *GUI Customizer* – The GUI Customizer state allows refinement of the chosen layout and GUI controls. A live preview of the GUI is shown in an internal frame. The user can select any widget in the GUI and customize its properties and layout. Two more floating toolboxes are introduced to allow this. The first toolbox is a component editor, which is specific to the selected component type, and allows properties of the widget to be changed. For example, for a label, we can change the displayed text and foreground colour. When a new component is selected in the preview window, its specific component editor is loaded into this toolbox. The second toolbox is a layout editor. When a component is selected, its layout constraints are loaded into the layout toolbox and the user can tweak them to adjust the component layout. Both the component and layout editors use live binding, which means any changes to the properties in the editors are reflected in real time in the preview window.
- *Code Generator* – The Code Generator state uses a model produced in the GUI Customizer state to produce code in the selected target language and toolkit. The Code Generator state presents the user with code generation options such as class name, file location, target language and target widget toolkit. The user makes their code generation choices and clicks the generate button. The code generator then uses the GUI model to create runnable code for the user (see code generation implementation in section 6).

3.3.5 Component Interaction

The states listed above have a strict order of progression. For example, we can't go straight from a sketch to code generation as widget classification might not be correct and we wouldn't know the layout code to use. To control the progression through the states, we introduce the concept of a ***flow.***

[i] If using a pen the user can use the eraser end or a function button if correctly configured.

The flow has two important functions to perform. It ensures the output of one state is passed as input to the next state. It also acts as a history container for the states so we can trace backwards through the application states. The HUB keeps a reference to the flow objects. When a user wants to progress to the next state, the HUB is notified. The HUB retrieves the next state from the current flow. The current flow acquires the output from the current state and initializes the next state with the output of the previous state. The HUB then propagates this new state down to the Content Pane to be displayed.

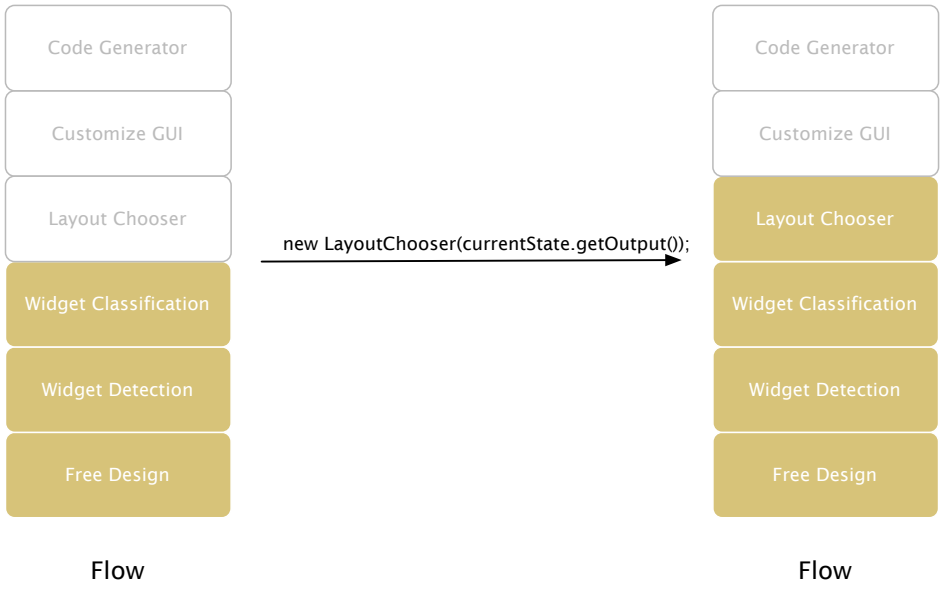


Figure 3.9: Progressing through the flow.

Figure 3.9 shows a visualisation of the `Flow` object. The current state is at the top of the stack. When we need to progress, we call `Flow.progressState()`. This creates the new `State` object, puts it on the top of the stack and returns it to the HUB. When we want to revert back through the states, they are popped off the stack and forwarded to the application HUB.

Using the concept of flow objects gives us another functionality, namely multiple files in the application. The HUB keeps a mapping between flow objects and file names, so if the user wants to switch files, we can immediately load up the state it was last in from its flow object. This is then propagated to the main window's Content Pane so the user can resume where they were with this sketch.

3.3.6 Sketchi Flexibility

Using a HUB as a mediator between application states gives us excellent flexibility. At any time, we can swap or add new stages into the flow without having to change the existing application states. All the application states are self-contained in terms of functionality and all contained within the content pane in terms of interaction with the user. This gives us great flexibility in integrating Sketchi within an IDE at a later stage. The existing content pane can be used and we would only need to write a new HUB to interact with the IDE and marshal the state down to the content pane.

4 Input Detection & Classification

The input detection and classification module takes a user's sketch as input, detects the possible widget's in the sketch and classifies each in turn. We first look at how we detect all the user drawn widgets in a sketch. Once we've identified each widget we can classify them all and progress to the next state.

4.1 Detection

The first part of the input capture stage involves singling out the widgets from the user's sketch. Given a full sketch of a GUI, we need to find the widgets in the drawing so we can perform classification on each individually. To find the objects in the image, we use the algorithm outlined in figure 4.1.

```
proc List<IWidget> findObjects(Set<Point> points) {
    // points is an ordered set of points with the top leftmost at
    // the top of the set
    List<IWidget> result;
    while(!points.isEmpty()) {
        IWidget currentWidget;
        List<Point> workList;

        workList.add(points.first());
        findObject(points, currentWidget, workList);
        removeInsides(points, currentWidget);
        result.add(currentWidget);
    }
    return result;
}

proc void findObject(Set<Point> points, IWidget w, List<Point> wl) {
    if(!wl.isEmpty()) {
        Point cPoint = wl.first();
        wl.remove(cPoint);
        w.add(cPoint);
        // we now test for points within a radius around cPoint
        for(x = cPoint.x - RADIUS; x < cPoint.x + RADIUS; x++) {
            for(y = cPoint.y - RADIUS; y < cPoint.y + RADIUS; y++) {
                if(points.contains(Point(x, y))) {
                    wl.add(Point(x, y)); points.remove(Point(x, y));
                    findObject(points, w, wl);
                }
            }
        }
    }
}
```

Figure 4.1: Algorithm for finding widgets in an image.

The set of points passed into the `findObjects()` method contains a list of the points drawn in a user's sketch. This is an ordered set with the top leftmost point appearing at the front of the set. The algorithm finds a point, looks for points around it within a search radius and traces around until no more points can be found. As it finds new points, it adds them to a work list, then adds them to the widget representation and removes them from the original set of points so that it doesn't loop forever. We use a search radius to tolerate gaps in a user's

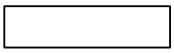
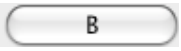
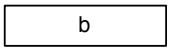
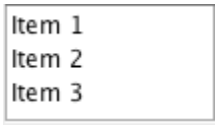
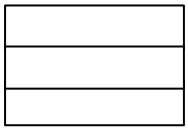
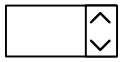
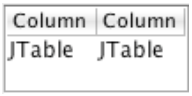
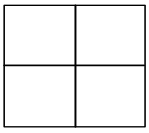
drawing. Once there are no more points in the work list, we have added them all to the widget and we can continue.

Unfortunately, if the drawing is of a widget with details on the inside, such as the 'b' on the button, these details will not be added to the widget representation. To solve this, we call the `removeInsides()` method. This method, which takes the leftover points from the original image and the current widget, adds these internal details to the current widget. We can get the bounds of the current widget from its object and then remove any points from the original image within these bounds. We add these removed points to the current widget representation to complete the process. The algorithm returns a list of the widgets found in the user's drawing.

One obvious shortcoming of this algorithm is that it does not handle nested content. The insides are removed regardless of what they represent. To solve this problem, the detection and classification stages would need to be merged together. This is outlined in future work in section 9.2.

4.2 Classification

For classification, we use a list of known representations of the widgets. This means that the user needs to learn these representations however we make them as close to the real rendering as possible. Here are the representations we are using:

Component Name	Rendered	Hand-drawn
Text Field		
Button		
List		
Spinner		
Table		

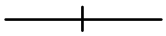
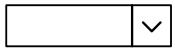
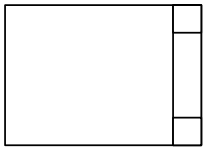
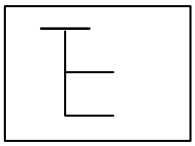
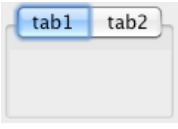
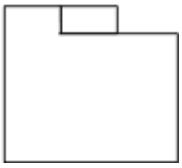
Slider		
Combo Box		
Check Box		
Radio Button		
Label		
Text Area		
Tree		
Progress		
Tabbed Pane		
Separator		

Table 4.1: List of recognisable widgets.

Now that we know what we are trying to classify, we can discuss our implementation of the classifier.

4.2.1 Case-Based Reasoning

To perform our classification of a user's sketch, we employ a case based system. Our Case-Based Reasoning (CBR) system includes a case base of known sketches encoded with various features, which we define below. When a user sketch comes into the system, we encode it with the same features and then look in the case base for the nearest match.

The known examples are represented as bitmap images in a sub directory of the project. The examples consist of the hand-drawn representations shown in table 4.1 and a series of drawings gathered from test users. As well as the bitmap representation of the examples, we need to know what these bitmaps represent. We use a file that uses YAML syntax to store this extra data. Figure 4.2 shows a small excerpt from the this file:

```
textfield: !org.sketchi.inputcontroller.util.YamlWidget
   classname: javax.swing.JTextField
   heuristics:
   -  enumName: RecEncoding
   -  enumName: CrossEncoding
      enumValue: 10
   images:
   -  - textfield.bmp
      - textfield2.bmp
```

Figure 4.2: Known examples YAML file.

The example above shows the entry for the text field widget. It stores the class name that represents the text field, the features to encode on this particular widget with their respective weights (see incorporating other features in section 4.2.5 to understand the use of the weight) and a list of images to use as a representation for this widget. The case based system uses this file to load and encode all known widgets to be used as cases in the case base. Storing the known widgets in this way allows us to easily add new widgets in the future, as well as new representations of the existing widgets.

For reading and writing the YAML file we use an open source library called YAML Beans. YAML Beans makes it simple to serialize and deserialize Java object graphs to and from YAML. We use YAML instead of XML to ease debugging (see section 2.2.3 for a comparison of readability) and to allow simple hand tweaking of the files in the development process. When the CBR is initialized, it loads the YAML file into Java objects and converts them to its own internal representation so they can be used for classification.

Figure 4.3 shows the hierarchies of classes used in the input classification system. We now give a brief overview of the concrete classes in this hierarchy:

- *`KnownWidget`* – The `KnownWidget` represents a known representation of a widget. It is created from the properties in the YAML file and a series of bitmap images. Once initialised, it is passed into the `WidgetEncoder` to have its features encoded on the object.
- *`InputWidget`* – The `InputWidget` is similar to the `KnownWidget`, except that the points added to its model are from the user's sketch rather than from an external image.
- *`WidgetEncoder`* – The `WidgetEncoder` takes an `IWidget` (either `KnownWidget` or `InputWidget`) and encodes the necessary features onto the object.

- *`Classifier`* – The `Classifier` can be seen as the core of the CBR system. It is initialised with the encoded `KnownWidget` objects. When the `getWidgetClassification(IInputWidget)` method is called with an `InputWidget`, it can provide the most likely classification for that particular input.
- *Features package* – The features package contains the implementations of the features. There is dependence from the `WidgetEncoder` because the `WidgetEncoder` instantiates the various features and sets them on the `IWidget` objects. There is dependence from the `Classifier` as it uses the encoded features to calculate the difference between the input widget and the known widget representations to give a classification (see suggesting a classification in section 4.2.3). There is dependence from the widgets package as each widget contains a reference to its encoded features.

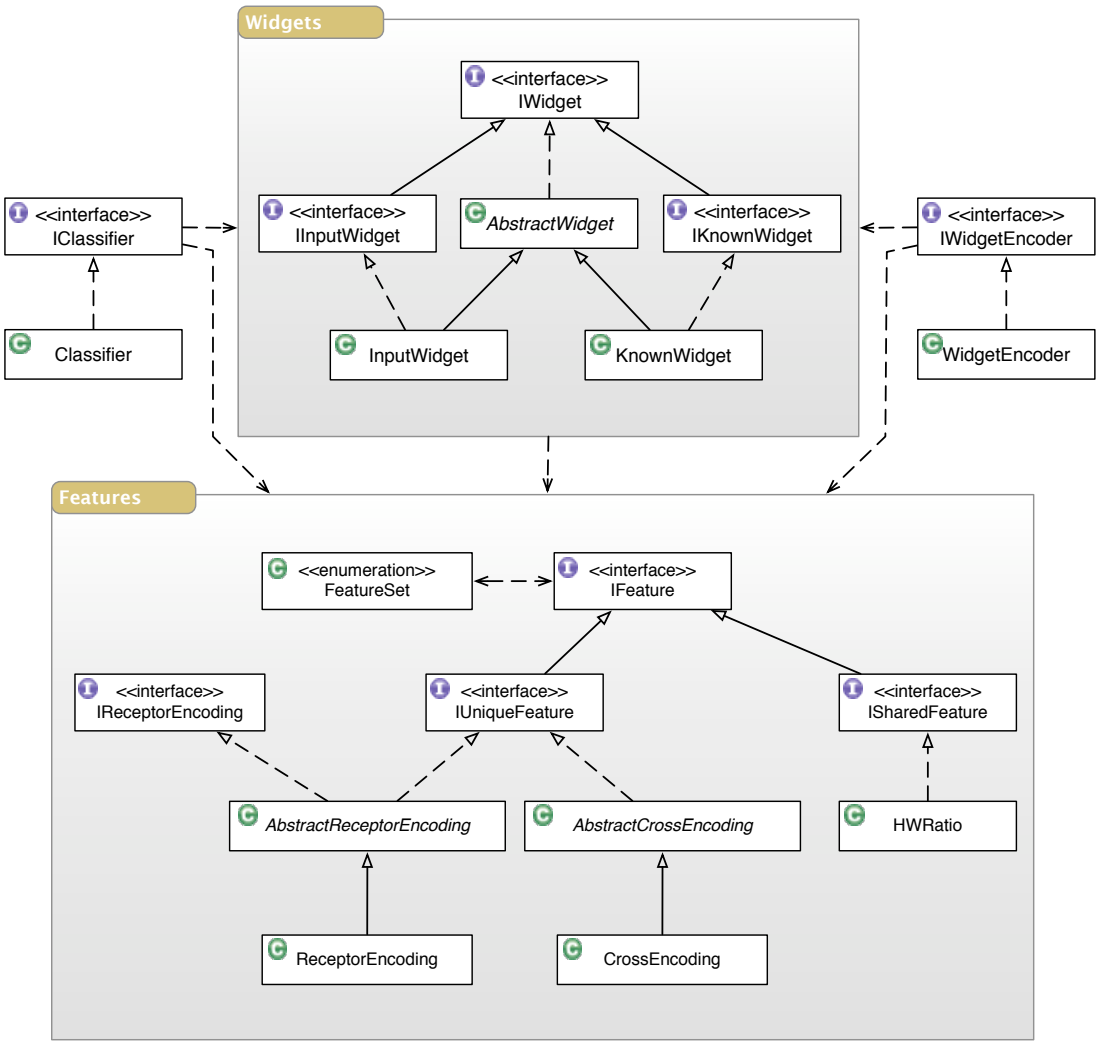


Figure 4.3: UML diagram showing the input classification system.

4.2.2 Receptor Patterns

So far, we haven't discussed the features the CBR encodes on the known examples. The base feature we use is the receptor pattern (outlined in the section *_* of the background). A receptor pattern is simply a pattern of 1's and 0's that represent whether the receptor in the pattern was crossed by a pixel in the image or not. A receptor is a line vector defined in a virtual coordinate space of 100 by 100. Figure 4.4 shows a visualisation of a random set of receptors. We supply two methods for generating these receptors.

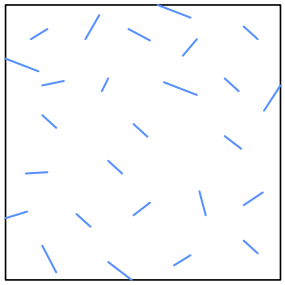


Figure 4.4: Random set of receptors generated in 100 by 100 coordinate space. The blue lines represent the receptors.

Random Receptor Generation

The random receptor generation method is very fast, but provides no guarantee in terms of how well the receptors will recognise the known examples. We randomly generate two points in the virtual coordinate space and check that the receptor isn't too long or too short. We do not test for its ability to recognise a particular object.

Entropy Receptor Generation

This method is outlined in the background section 2.3.3. It is an NP algorithm, so the performance is poor. However, it does guarantee that the receptors generated will be able to recognise the known examples well. This method involves randomly generating the receptor as described above and then calculating its usability measure (see the background section 2.3.3 for the usability method). Every generated receptor is added to a list of candidate receptors and the receptors with a high usability are then used for the encodings. Figure 4.5 shows the pseudo code of the algorithm for generating a entropy-based receptor.

The algorithm generates a list of receptors to be used for classification. The method `calculateUsability()` calculates the usability measure detailed in section 2.3.3 of the background. A performance comparison between the two receptor generation techniques can be found in section 8.6.

```
proc Receptor generateRandomReceptor() {
    int x1; int y1; int x2; int y2; int length;
    while(true) {
        x1 = randomNumber() * VIRTUAL_COORDINATE_SIZE;
        y1 = randomNumber() * VIRTUAL_COORDINATE_SIZE;
        x2 = randomNumber() * VIRTUAL_COORDINATE_SIZE;
        y2 = randomNumber() * VIRTUAL_COORDINATE_SIZE;
        length = sqrt((x1 - x2)^2 + (y1 - y2)^2);

        if(length >= MINLENGTH && length <= MAXLENGTH) {
            return new Receptor(x1, y1, x2, y2);
        }
    }
}
```

```
proc List<Receptor> generateReceptors(Map<String, List<Widget>> ws) {
    int receptorCounter = 0;
    List<Receptor> rs = new List<Receptor>();

    while(receptorCounter < TOTAL_TO_GENERATE) {
        Receptor r = generateRandomReceptor();

        if(calculateUsability(r, ws) > USABILITY_THRESHOLD) {
            rs.add(r);
        }
    }
    return rs;
}
```

Figure 4.5: Entropy receptor generator algorithm.

Encoding a Widget with Receptors

Once we have a list of the receptors, we can encode each known example with a receptor pattern to be used in the classification stage. Each `IWidget` has a field for a `WidgetDescriptor`. These `WidgetDescriptor` objects contain an `EnumMap` mapping the feature name to its encoding. Each feature encoding is an instance of an `IFeature` and the hierarchy of `ReceptorEncoding` can be seen in figure 4.3.

When creating a receptor pattern encoding of a widget representation, we iterate over all the receptors to check if the receptor crosses a drawn pixel in the representation's image. If it does, then the receptor is active, if not, then the receptor is inactive for this representation.

An important point to note is that we must ***scale*** the receptor before we do this. Originally we defined the receptor in a virtual coordinate space but the input representation may not have the same dimensions as this coordinate space. The receptor is defined as a vector and the widget representation as a collection of points, which is why we scale the receptor and not the image. A Point in Java has a fixed size of 1 pixel, and is not something we can easily scale. We would have to calculate the scaling factors and create new points to replace the existing point. Scaling a vector is much simpler and computationally cheaper. Scaling the vectors also means that the size of the user's drawing doesn't affect the receptor pattern. Our implementation of the receptor encoding uses an array of `boolean` values to represent whether the receptors are crossed or not. Figure 4.6 shows the pseudo code for the encoding algorithm for a receptor pattern.

```
proc void encode(IWidget w, List<Receptor> receptors) {
    for(Receptor r : receptors) {

        for(Point p : w.getPoints()) {
            r.scale(w.getHeight(), w.getWidth()); // important!

            if(r.isIntersection(p)) {
                this.set(r.getIndex(), true);
                break;
            }
        }
    }
}
```

Figure 4.6: Receptor Encoding algorithm.

The method `isIntersection(p)` returns true if the point p lies on the receptor line and false otherwise. Using this algorithm, we can encode the receptor pattern for a given widget. The `getPoints()` method on the widget gets a list of pixels that make up the image representation of the widget.

4.2.3 Suggesting a Classification

The CBR system loads the known widgets from the YAML file and encodes the receptor patterns for their representations. A user-supplied sketch of a widget enters the system and is encoded in the same way as a known widget. The classifier then iterates through the known examples and calculates the distance between the receptor patterns on the known cases to the receptor pattern on the input widget. These distances are stored against the name of the known case, and they are normalised to give a value between 0 and 1. The suggestion list is an ordered list of the classifications for an input widget. Those classifications with a 1 are at the top of the list and those with a value of 0 are at the bottom of the list. Figure 4.7 shows the algorithm that produces the classification list (note that the classifier is already initialized with a map of known widgets).

```
proc List<WidgetClassification> getTopClassifications(IInputWidget w)
{
    double maxDistance = 0;
    Set<WidgetClassification> cs;
    for(String curWidget : knownWidgets.keySet()) {
        List<IKnownWidget> variations = knownWidgets.get(curWidget);

        // Each k is a variation of the same widget type
        for(IKnownWidget k : variations) {
            IReceptorEncoding re = w.getReceptorEncoding();
            // difference between the two receptor patterns
            double distance =
                              re.calcDifference(k.getReceptorEncoding);

            // We use Euclidean distance
            distance = sqrt(distance);
            maxDistance = max(distance, maxDistance);
            cs.add(new WidgetClassification(curWidget, distance));
        }
    }
    return normaliseClassifications(cs, maxDistance);
}

proc List<WidgetClassification> normaliseClassifications
                                        (Set<WidgetClassification> cs,
                                                    double maxDistance) {
    List<WidgetClassification> result;

    // We are traversing a treeset which is ordered on the distance
    // field of widget classification
    for(WidgetClassification wc : cs) {
        // Normalise the distance here
        wc.setChance(1 - (wc.getChance() / maxDistance));
        result.add(wc);
    }
    return result;
}
```

Figure 4.7: Calculating feature distance algorithm.

We can see that this returns a list of `WidgetClassification` objects in the order of their distance field. This list is an ordered list of classifications for the user's widget sketch with the first element being the most likely match. The `calcDifference(encoding)` method returns the number of mismatched receptor values between the two encodings.

4.2.4 Enhancing the Receptor Encoding

After some initial testing of the receptor pattern feature, we found certain widgets were frequently being misclassified. Looking at these misclassifications provided us with some insight into what was going wrong.

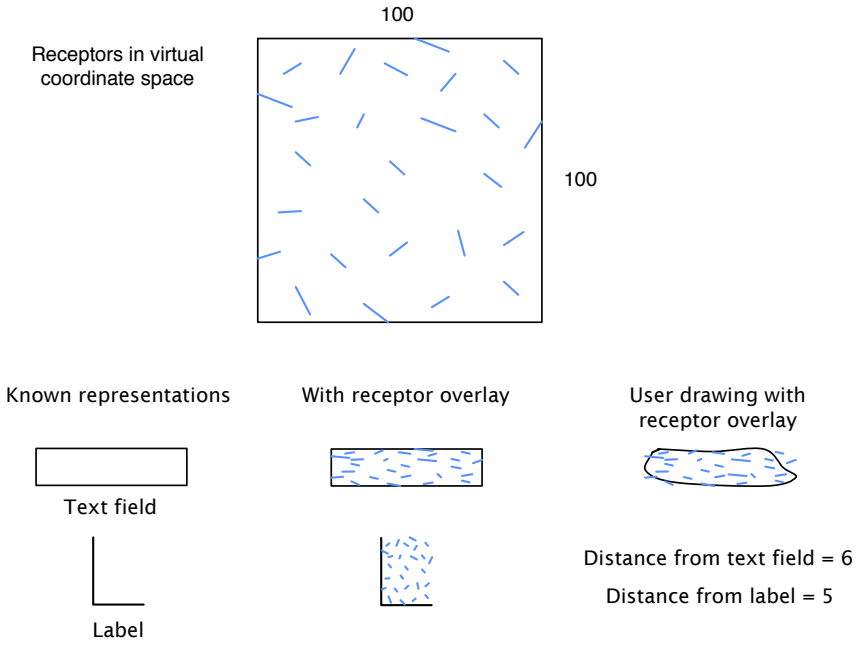


Figure 4.8: Text field to label misclassification.

The blue lines represent receptors used to classify the widget. It can be seen that even if we draw a sketch that looks similar to a text field, the encoding will be very similar to that of the label. In certain instances, the classification system could classify a drawing of a text field as a label as seen in figure 4.8. We would like to increase the distance between the two encodings to be sure that what the user has drawn is definitely not a text field. To overcome this we introduce the concept of a ***receptor weight***.

The idea behind a receptor weight is that certain receptors are more important in the classification of a particular widget than others. A receptor is classed as being more important if it takes on the same value (active or inactive) even after the input image has been distorted by a reasonable amount. This distortion accounts for users drawing wobbly lines instead of perfectly straight lines in their sketch. The intuition is that certain receptors should never be crossed, even if the user draws the widget poorly. If these receptors are crossed, then it's definitely not a widget of this type, so we return a high score for the difference measure. This high return value pushes the widget down the suggestion list in the distance calculation. If the receptor weights for the label in figure 4.8 were set to a value of 15, the calculated distance between the input encoding and the label encoding would increase from 5 to 65. This is due to 4 weighted receptors being mismatched and only 1 unweighted receptor. This increased distance will push the label classification down the suggestion list, so we avoid misclassifying the drawing.

The encoding algorithm is now modified to include weighting of certain receptors. To find the receptors that need to be weighted, we distort the input image and recalculate the receptor intersections with this new distorted image. If the receptor has the same value with

the distorted image as the normal image, then it's tolerant to distortion and is given a larger weight. The modified algorithm is shown in figure 4.9.

```
proc void encode(IWidget w, List<Receptor> receptors) {
    for(Receptor r : receptors) {
        boolean intersection = false;
        for(Point p : w.getPoints()) {
            r.scale(w.getHeight(), w.getWidth());
            if(r.isIntersection(p)) {
                this.set(r.getIndex(), true);
                intersection = true;
                break;
            }
        }
        for(Point p : w.getDistortedPoints()) {
            if(r.isIntersection(p) == intersection) {
                this.setHeavyReceptor(r.getIndex());
                break;
            }
        }
    }
}
```

Figure 4.9: Modified receptor encoding algorithm.

When we calculate the difference between the receptor patterns, we no longer just count the number of mismatches between the two encodings. Where there is a mismatch between two receptors, we used to increment the number mismatches but now we ***add the weight*** of the receptor to the total distance. We change:

```
misses++;
```

Figure 4.10: Old line in calculated difference algorithm.

in the `calcDifference(encoding)` method call to:

```
misses = misses + encoding.getReceptorWeight(index);
```

Figure 4.11: Modified line in calculated difference algorithm.

This change means that a much larger difference is returned where a heavily weighted receptor is crossed when it shouldn't be (`getReceptorWeight(index)` returns 1 if the receptor is unweighted). We can see this as the widget classification being punished and so the widget falls down the suggestion list leaving the correct suggestion at the top.

4.2.5 Incorporating Other Features

After initial user testing it became apparent that just using receptor patterns for classifying wasn't good enough. Too many of the widgets look too similar and some additions to the classification system needed to be made. Fortunately, the CBR system is easily extensible and new features can be added with ease. We just need to provide an implementation of an `IUniqueFeature` or an `ISharedFeature`, add the types to the `FeatureSet` Enum and state which objects should use these features in the widget YAML file.

Cross Encoding Feature

The cross encoding feature is a simple concept that fine-tunes the recognition. We use the following steps to generate a cross encoding for a widget.

- *Overlay grid* – We take the input sketch of the widget and lay virtual grid lines over the sketch.

- *Trace gridlines* – We take each gridline and trace along the line from one side of the grid to the other.
- *Count crossings* – For each gridline we count the number of pixels that we cross on the way to the other side.
- *Combine into an encoding* – Each gridline is given an index in the encoding. Each index in the encoding is assigned the number of pixels that get crossed when tracing the gridline.

The cross encoding provides us with a good feature for differentiating the similar objects that have subtle differences. Where receptor patterns might be close, the cross encoding will act as a differential factor between the two. To calculate the distance between a known example and a user drawn image, we count the difference in the number of pixels crossed for each gridline between the known example and input sketch. Figure 4.12 shows an example of the cross encoding.

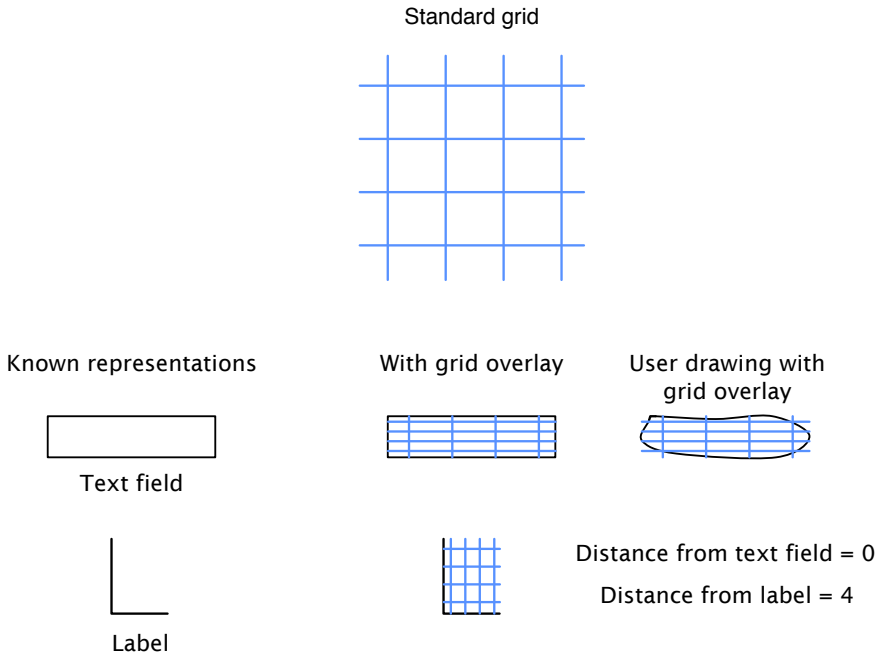


Figure 4.12: Example of cross encoding.

Height to Width Ratio Feature

The height to width ratio is a feature that encodes the height to width ratio of a widget. This feature is encoded on widgets that have typically fixed height to width ratios such as checkboxes and radio buttons. When the KnownWidget objects are created, if the feature is turned on for this widget, its height to width ratio is calculated from the bitmap images. When comparing the height to width ratio of the InputWidget with a KnownWidget, we permit a certain level of tolerance for the user. If the feature is within the tolerance, then the distance between the representations is returned as 0. Otherwise, a large distance is returned to push the known representation down the suggestion list.

Combining the features

There are issues with using multiple features in the CBR and even more so when the known representations use different sets of features. When calculating the distance between two widget encodings, we need to take all features into account, but there is a question mark over how these should be combined. The most intuitive method is to add them all together.

Unfortunately, the distance calculated for some features will overpower others. The result of a cross encoding distance may be small but vital in classifying the widget. But this could easily be nullified by a single heavy weighted receptor being mismatched. Depending on the user's drawing style, this could happen often and result in common misclassifications for the user.

One solution is to weight the features when adding them to the total distance between two widget representations. We change the `getTopClassification()` method in figure *_* to include a loop for comparing all the features on a widget. Figure 4.13 shows the modified code for suggesting the classification of a widget with multiple features.

```
proc List<WidgetClassification> getTopClassifications(IInputWidget w)
{
    double maxDistance = 0;
    Set<WidgetClassification> cs;
    for(String curWidget : knownWidgets.keySet()) {
        List<IKnownWidget> variations = knownWidgets.get(curWidget);

        // Each k is a variation of the same widget type
        for(IKnownWidget k : variations) {
            double distance = 0;

            // Adding the feature loop //
            for(IFeature f : k.getFeatures()) {
                IFeature inputF = w.getFeature(f.getFeatureName());
                distance += inputF.calcDifference(f);
            }
            // We use Euclidean distance
            distance = sqrt(distance);
            maxDistance = max(distance, maxDistance);
            cs.add(new WidgetClassification(curWidget, distance));
        }
    }
    return normaliseClassifications(cs, maxDistance);
}
```

Figure 4.13: Combining features whilst calculating widget classifications.

The distance for a feature is weighted inside the `calculateDistance()` method instead of in the `IFeature` loop itself. This may seem strange, but provides us with two key advantages:

- *Different weights for different widgets* – Certain features will be more important for particular widgets than others and storing the weight with the representation allows us to have varying weights across the known representations.
- *Easy loading* – Storing the weight with the widget allows us to store the weights externally in a widget YAML file. The weight is simply loaded into the `KnownWidget` from the YAML file when the `KnownWidget` objects are created. This means the CBR doesn't have to keep track of all the varying weights.

A testbed was written to test the performance of the assigned weights. The testbed classifies a set of test images, acquired from user testing, using the standard set of known widget representations in the case base. The testbed counts the position in the suggestion list of the correct classification. It displays the percentage of correct classifications and the percentage of times the correct classification appeared in the top three suggestions. Using the testbed we can evaluate different sets of weights. To find these weights, we used two different methods:

- *Manual tweaking* – Initially we set all the weights to the same value and ran the testbed. By looking at the common misclassifications and knowing how the

encodings are calculated, we can understand what contributes to the misclassification and adjust the weights as necessary. It is very difficult to express, in terms of code, what is causing a misclassification and adjust the weights as necessary. If a text field is misclassified just below a button, we have a choice of adjusting the receptor weight or the cross encoding. Whether we decrease the weights of the text field to make the text field appear above the button or increase the weights of the button to make it appear below the text field is a choice. Either choice will remove this misclassification, but adjusting the weights will affect other classifications and may lead to more misclassifications (a worse global solution).

- *Hill climbing* – We implemented a hill climbing algorithm was implemented in an attempt to optimise the weights without having to tweak them by hand. Figure 4.14 outlines the algorithm. We calculate a ***score*** for the current weights using the testbed. We take a collection of training images and classify them. We know which widget the training image represents, and the classification stage gives us an ordered list of classifications. The score for a particular training image is the position of the correct classification in the classification list. All of the training image scores for a particular widget type (button, text field, etc.) are averaged to give a score for the widget type. The minimum value is 1 with the maximum value being the total number of widget types, i.e. a widget is misclassified to the bottom of the list for every training image. A widget with a high score (high is bad) is selected and the weights for its features are adjusted. If the score hasn't improved, then we revert our changes to the weight and pick a new widget or alter the weight by a new amount. If the score remains unchanged for more than the set threshold, then we make a large adjustment to the worst scoring widget by decreasing its weights by a large amount. This will make the widget correctly classified and forces the other weights to be selected for adjustment. We do this adjustment to overcome local maxima.

```
proc void optimiseWeights(Map<String, List<IKnownWidget>> kws) {
    double currentScore = 10000; // any number higher than threshold
    double oldScore = 10000;
    int scoreUnchangedCounter = 0;
    Map<String, Integer> scores;
    radomizeWeights(kws);
    while(currentScore > SCORE_THRESHOLD) {
        scores = calculateScores(kws);
        currentScore = addScores(scores);

        // if we've gotten worse then revert and retry
        if(currentScore < oldScore) {
            oldScore = currentScore;
            scoreUnchangedCounter = 0;
        } else {
            currentScore = oldScore;
            revertLastWeightChange(kws);
            scoreUnchangedCounter++;
        }
        adjustWorstWeights(kws);

        if(scoreUnchangedCounter > UNCHANGED_THRESHOLD)
        {
            randomizeWorst(kws);
            oldScore = 10000; // restart our search with new weights
        }
    }
}
```

Figure 4.14: Hill climbing algorithm for finding optimal weights.

The performance of the different weights obtained from these two methods are compared in the results and analysis section 8.1.

4.2.6 Learning From the User

A core feature of our classification implementation is the ability to learn the user's drawing style. This was a major deciding factor when choosing a CBR system over an ANN system. Fortunately, CBR makes learning from the users very simple. In Sketchi, if a widget is misclassified, we give the user the opportunity to correct its classification. At this point, we can take the encoding of the input sketch and convert it to a `KnownWidget` in the case base. The next time a user draws something similar, this new case becomes the closest match and the correct classification appears at the top of the suggestion list. To see the effects of learning on the input classification look at the results and analysis section 8.1.

We use a user preference file to store the learned widgets for a user. When the CBR is initialized, the user's learned widgets are loaded up and converted to `KnownWidget` objects like any other known representation. This restores the case base to the same state as when the user last exited the application.

4.3 Summary of Input Detection and Classification

This chapter has outlined the implementation of our detection and classification systems. We use a CBR based system and encode the widgets with multiple features. We have enhanced the receptor pattern feature by introducing receptor weights to help distinguish between similar widgets. Finally, we've added the ability to learn from the user in order to adapt to their specific drawing style and improve recognition rates.

5 Automated Layout Generation

The automatic layout generator takes a classified set of widgets as input and attempts to generate a layout that resembles the user's sketch. The aim is to generate layout code that intelligently resizes the GUI without deviating from what the user has drawn. This section outlines the algorithms used for inferring the layout code.

5.1 Layout Generator Architecture

We've implemented an extensible layout generator, so that concrete implementations for other layout managers can be incorporated in the future. It is abstracted in such a way that we could achieve this without changing other components in the overall architecture of the application. Our implementation concentrates on MigLayout (see background section 2.4.2) but many of the algorithms described below can be translated for use with other layout managers.

Architecture

Here we outline the architecture and discuss the classes available in the layout generation process. Figure 5.1 shows the UML diagram of the layout generation system.

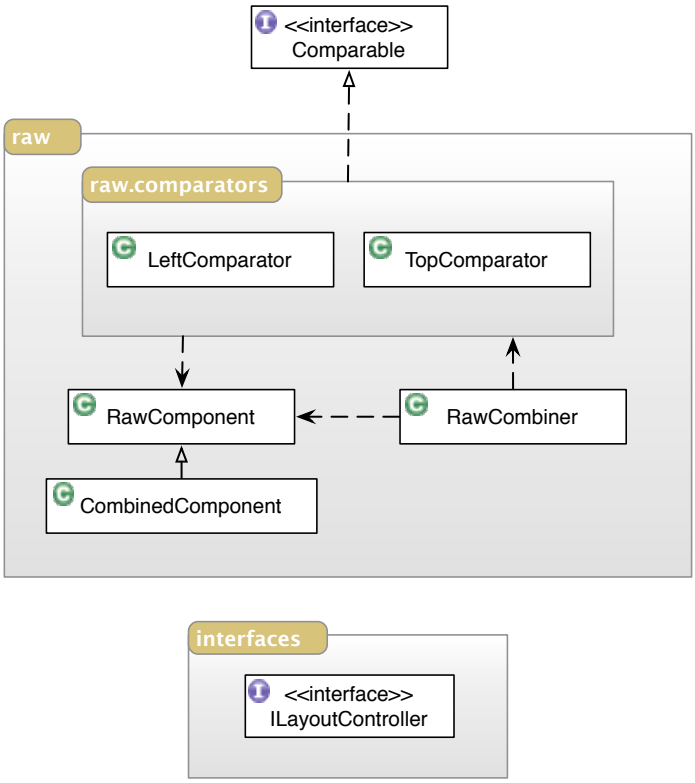


Figure 5.1: UML diagram showing the layout generation system.

We will now describe the elements from figure 5.1:

- *ILayoutController* – Concrete layout generators need to implement this interface to be used with the layout generation component within Sketchi. Its principal method is `getPanel()`. This returns a panel containing the input widgets laid out with the layout manager after analysis has been completed.
- *RawComponent* – The `RawComponent` objects represent the widgets from the user's sketch. The layout controllers are passed a list of `RawComponent` objects that they can analyse to generate a layout. It contains the bounds of the widget and a `JComponent` instance of the widget it represents.
- *CombinedComponent* – The `CombinedComponent` is an object, which is used in our MigLayout implementation but could be useful for other implementations of `ILayoutController`. The `CombinedComponent` represents two or more `RawComponent` objects being combined. It takes on the bounds spanning all `RawComponents` and maintains a reference to the original `RawComponent` objects.
- *RawCombiner* – The `RawCombiner` creates the `CombinedComponent` objects from `RawComponents`. It takes a set of `RawComponent` objects, a tolerance value and two strings representing the types of widgets to combine. If any two objects in the `RawComponent` set have a matching widget type and if they are close enough (subject to the tolerance value passed in) they are merged into a `CombinedComponent`.
- *LeftComparator* – The `LeftComparator` is a comparator for `RawComponent` objects. It orders a set of `RawComponent` objects by their left edge.
- *TopComparator* – `TopComparator` is analogous to `LeftComparator` but objects are ordered by their top edge.

Layout generator implementations can utilise these classes with their own algorithms to generate accurate layouts reflecting the user's sketch.

Visualizer

The visualizer is a small tool created for debugging purposes. It renders the bounds of the `RawComponent` objects to a window so the developer can see what the layout generator is analysing. This window can be generated alongside the output of the layout generator for comparison and to help debug whilst developing a layout generator implementation. Figure 5.2 shows an example of the visualizer.

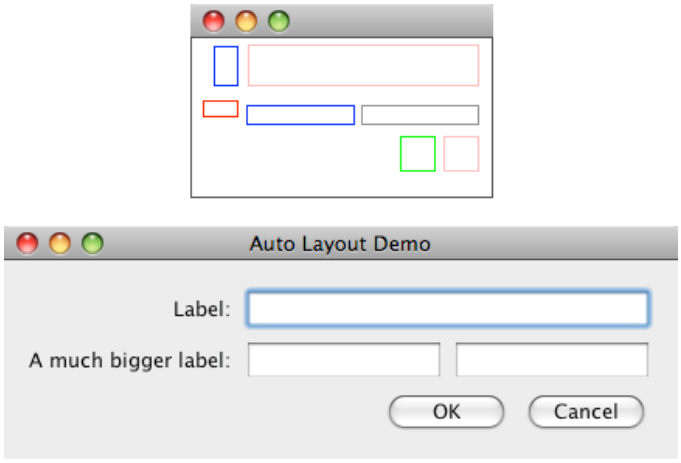


Figure 5.2: Example of layout visualizer next to the auto generated layout.

5.2 MigLayout

We've based our layout generator on MigLayout for the reasons outlined in section 3.3.2. It uses a grid-based paradigm to layout components and we base our algorithms on this concept. Figure 5.3 shows the UML diagram representing our implementation of the layout generator.

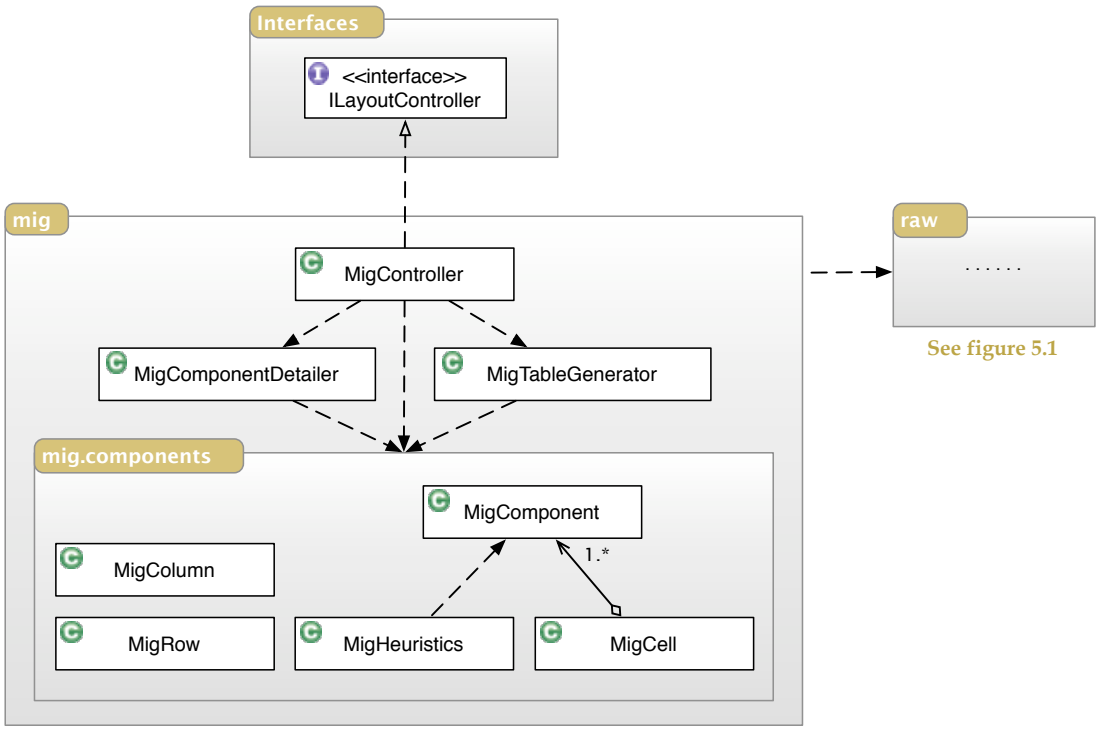


Figure 5.3: UML for MigLayout automatic layout generator implementation.

We now describe the functionality of these classes:

- *`MigController`* – This is our implementation of the layout generator. It takes the set of `RawComponent` objects in its constructor and controls the layout analysis. When the `getPanel()` method is called, it adds the `JComponent` instances to a new panel with their layout constraints and returns the panel.
- *`MigColumn`* – A `MigColumn` represents a column in the layout. It has a left and right field that represent the bounds of the column.
- *`MigRow`* – A `MigRow` is analogous to the `MigColumn` but it represents a row instead of a column with a top and bottom field.
- *`MigComponent`* – The `MigComponent` objects model the layout for each widget. They are created from `RawComponents` and contain fields representing possible layout constraints for the component. These include the row and column the widget can be found in, the alignment of the object within the table cell and many others.
- *`MigCell`* – The `MigCell` represents a cell in the table layout. A cell can contain multiple components and the `MigCell` keeps track of these components in the cell.
- *`MigHeuristics`* – The `MigHeuristics` object adds default layout constraints to widgets. It contains a mapping between the widget type and the layout properties that need to be applied. Given a `MigComponent`, it will supplement the layout constrains with those from the mapping.

- *MigTableGenerator* – This class generates the rows and columns required for the layout. It takes a set of `RawComponent` objects and performs the analysis to find the minimum number of `MigRow` and `MigColumn` objects for the layout.
- *MigComponentDetailer* – The `MigComponentDetailer` class adds extra layout constraints to the `MigComponent` objects. It finds the alignment of a component within a cell, matches the size of widgets and calculates any spanning of the table cells if necessary.

A `MigController` is instantiated with a set of `RawComponent` objects. It then uses the other `mig` classes to produce a layout. Figure 5.4 shows the algorithm that is used to produce the layout.

```
proc MigController(Set<RawComponent> input) {
    // combine components for layout analysis
    Set<RawComponent> combined = combineComponents();
    // Generate the necessary columns and rows
    MigTableGenerator mtg = new MigTableGenerator(combined, input);
    this.columns = mtg.getColumns();
    this.rows = mtg.getRows();
    // Assign components to cells in the table
    this.migcells = assignComponentsToCells(input);
    // Add any default layout constraints, add spannings, add alignments...
    MigComponentDetailer mcd = new MigComponentDetailer(this.migcells,
                                                        this.rows,
                                                        this.columns);
    this.components = mcd.getComponents();
}
```

Figure 5.4: Pseudo-code showing the initialization stages of the MigController.

Combining Widget Representations

To ensure that certain widget types remain together in the layout, we introduce the concept of ***combined components***. An example of this is a row of buttons at the bottom of a window. If the buttons are added to different cells then they may not appear together upon resizing of the window. Figure 5.5 shows this resizing issue.

Figure 5.5: Example showing how layout analysis with un-combined components can cause unwanted gaps in the layout.

If we treat these buttons as a single component, occupying the area of both buttons in the layout generation stage, we can avoid this. This is due to the layout generator ensuring that both components are added to the same cell in the layout table. This results in the desired behaviour as is shown in figure 5.6.

Figure 5.6: Example showing how layout analysis with combined components resolves unwanted layout gap issues.

We will now walk through the algorithms used in the automatic layout generation. Figure 5.7 is a user sketch with two labels and a text field. We will use this as an example and show how the layout code is generated.

Figure 5.7: User sketch example for layout generation.

5.2.1 Inferring Columns & Rows

When generating a table-based layout, we first need to decide how many rows and columns to use. The most intuitive method would be to give every component its own column and row. In the case where the components are vertically aligned, we merge their rows and if they are horizontally aligned, we merge their columns. Unfortunately, this method produces artefacts in the layout. If we take our example drawing from figure 5.7 and use this method we get the artefacts shown in figure 5.8.

Figure 5.8: Layout example demonstrating layout artefact for simple column generation.

In figure 5.7, the two labels are not left aligned and so each is assigned its own column. The text field is also assigned its own column. The top label spans the first two columns and the artefact is produced as a result. The correct solution would be to use two columns for the entire layout. Both labels would be assigned to the first column and the bottom label would need to be right aligned. Running the sketch in figure 5.7 through our implementation yields the result in figure 5.9.

Figure 5.9: Layout example demonstrating no artefact with correct column generation.

The implementations for rows and columns are synonymous, so from here we will only discuss columns. To overcome the layout artefacts seen in figure 5.8, we perform two passes over the components. The first pass, outlined in figure 5.10, generates an initial conservative set of columns. We only add a column in the initial pass if two or more components have a left aligned edge. This ensures that we do not generate more columns than required.

The second pass, outlined in figure 5.12, adds any missing columns. We iterate through each component to check if it is contained within a column. If it is not, then we create a new column that contains the component and we recurse with the new set of columns. This

ensures that we do not generate excess columns since we are retesting with the new columns included.

```
proc void findInitialColumns(Set<RawComponent> leftAlignedComps) {
    // note the components are traversed in order of left edge value
    // and they are a set of combined components
    for(RawComponent first : leftAlignedComps) {
        for(RawComponent second : leftAlignedComps) {
            if(first.equals(second)) {
                continue;
            }
            if(approximatelyLeftsideAligned(first, second)) {
                MigColumn mc = new MigColumn();
                mc.setLefT(first.getLeft());
                mc.setRight(first.getRight());
                this.columns.add(mc);
            }
        }
    }
}
```

Figure 5.10: Algorithm for first pass through column generator.

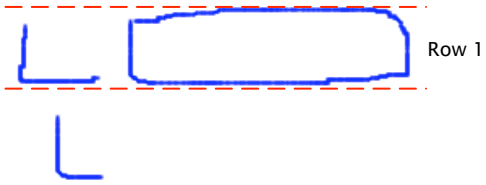


Figure 5.11: Rows and columns identified after first pass.

Figure 5.11 shows that no columns and only a single row have been generated after the first pass. This is due to our conservative method of only adding a column or row when there are aligned components.

```
proc void addMissingColumns(Set<RawComponent> leftAlignedComps) {
    for(RawComponent rc : leftAlignedComps) {
        // Tests if the left edge of the component lies in a column
        if(! leftEdgeInAColumn(rc)) {
            // finds the index in the column list for the new column
            int index = findIndexForNewColumn(rc);
            MigColumn mc = new MigColumn();
            mc.setLefT(rc.getLeft());
            mc.setRight(rc.getRight());
            this.columns.add(index, mc);
            addMissingColumns(leftAlignedComps);
        }
    }
}
```

Figure 5.12: Algorithm for second pass through column generator.

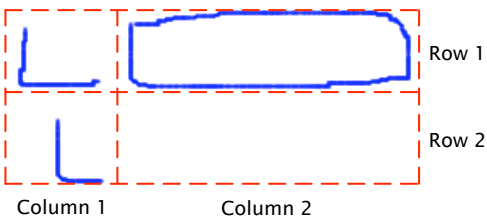


Figure 5.13: Rows and columns identified after second pass..

After the second pass, figure 5.13 shows we have generated the missing columns and rows for the components that previously had none. This results in a correct table generation for our layout and we can now add layout details to the components.

5.2.2 Adding Component Constraints

Once the rows and columns have been defined, we need to allocate the widgets to cells in the table. The method call `assignComponentsToCells(input)` from figure 5.4 performs this step. It iterates through the `RawComponent` objects and finds the column and row for the widget. It returns an array of `MigCell` objects representing the table cells in the layout. Each `MigCell` has a list of `MigComponent` objects that need to be added to the cell.

After the components have been assigned to a cell, we can annotate each component with extra layout information. We create an instance of `MigComponentDetailer` that takes the `MigCells` and adds the extra layout information to the `MigComponents`. Figure 5.14 outlines the steps performed in the initialization stage if the `MigComponentDetailer`.

```
proc MigComponentDetailer(MigCell[][] cells, List<MigColumn> cols,
                                                    List<MigRow> rows) {
    applyDefaultHeuristics(cells);
    calculateSizeGroups(cells.getComponents());
    calculateSpanningWidgets(cells, cols, rows);
    calculateAlignments(cells, cols, rows);
}
```

Figure 5.14: Supplementing widget layouts with a MigComponentDetailer.

Figure 5.15 shows that the widgets from our sketch have been assigned labels and we will now walk through the method calls in figure *_* to see how the layout constraints for each labelled widget change.

Figure 5.15: Example layout generation sketch with added labels.

The first method loads and applies any default layout constraints to the widgets. The method `applyDefaultHeuristics(cells)` instantiates a `MigHeuristics` object, then iterates through the components in the `MigCells` and looks up default layout constraints for the current component. The `MigHeuristics` object loads a default set of layout constraints from file and applies them to a component based on the widget type.

The next stage is to define any ***size groups*** that may exist in the user's sketch. A size group is a group of widgets that are bound to the same size. When a user resizes a window, if two widgets have the same size group, they will maintain the same size. Using size groups gives a more professional look to the layout and the `calculateSizeGroups (cells.getComponents())` method automatically finds these size groups. The algorithm for finding size groups is defined in figure 5.16. The algorithm finds components of approximately the same size and assigns them to the same size group.

```
proc void calculateSizeGroups(Set<MigComponent>components) {
    Set<MigComponent> alreadySeen;
    for(MigComponent first : components) {
        if(alreadySeen.contains(first)) continue;

        Set<MigComponent> sizeGroup;
        for(MigComponent second : components) {
            if(first.equals(second)) continue;

            double widthRatio = max(first.getWidth(), second.getWidth())/
                                min(first.getWidth(), second.getWidth());
            double heightRatio = max(first.getHeight(), second.getHeight())/
                                 min(first.getHeight(), second.getHeight());

            if(widthRatio <= TOLERANCE && heightRatio <= TOLERANCE) {
                sizeGroup.add(second);
            }
        }
        if(! sizeGroup.isEmpty()) {
            sizeGroup.add(first);
            assignUniqueSizeGroup(sizeGroup);
        }
    }
}
```

Figure 5.16: Algorithm for finding size groups.

The next step in the component detailer indentifies widgets that span multiple rows or columns. We check to see if the right or bottom edge of the component lies outside the right or bottom or edge of the column or row. If it does, then the component spans into the next column or row respectively.

The final step of the detailer is to find the component's alignment within the cell. This is a straightforward process, since we know the bounds of the cell as well as the bounds and location of the component. The layout generation is now complete and the end result can be seen back in figure 5.9. Table 5.1 below shows the layout constraints for each of our components after each stage of detailing.

Label	Initial	Heuristics	Size groups	Spanning	Alignments
a	cell 0 0	cell 0 0	cell 0 0	cell 0 0	cell 0 0
b	cell 1 0	cell 1 0, growx	cell 1 0, growx	cell 1 0, growx	cell 1 0, growx
c	cell 0 1	cell 0 1	cell 0 1	cell 0 1	cell 0 1, align right

Table 5.1: Layout constraints for labelled widgets after each component detailing stage.

5.3 Summary of Automated Layout Generation

This chapter has detailed our implementation of an automated layout generator specifically for the MigLayout layout manager. We use combined components and a two-pass algorithm to infer the number of columns and rows required by the layout. Additionally, we have implemented a component detailer to infer component level layout constraints.

6 Code Generation

Code generation is the final stage of converting a sketch to a GUI. We define an abstract code generator that gives us the ability to generate code in multiple languages and toolkits. Our design is based on a combination of visitor pattern and code templates. This combination gives us the flexibility to generate code for multiple toolkits and languages.

6.1 Modelling the GUI

Our GUI model uses the same paradigm as the Swing toolkit. We model the GUI as a container with children. A possible child is a widget component or another container, giving us a composition hierarchy. Figure 6.1 shows the UML diagram representing our model.

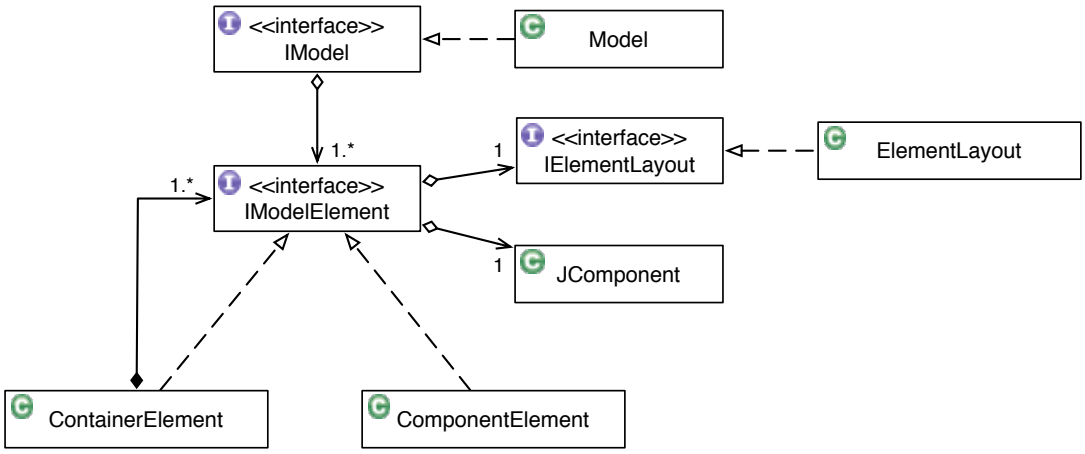


Figure 6.1: UML of our GUI model.

Here is a description of the interfaces and classes from figure 6.1:

- *`IModel`* – This is the interface to our GUI model. The code generator uses it to access the elements from the model.
- *`Model`* – This is our implementation of the `IModel` interface representing our GUI model.
- *`IModelElement`* – These are the elements of our GUI model. They are available to the code generator implementations and provide access to the `JComponent` and `ElementLayout` objects of the model elements.
- *`ContainerElement`* – An implementation of the `IModelElement` interface that represents a container in the GUI. Possible containers include tabbed panes and panels. The container elements can be nested to form deep hierarchies mirroring the Swing component paradigm.
- *`ComponentElement`* – An implementation of the `IModelElement` interface that represents a GUI component that is not a container. This can be anything from a button to a table.
- *`IElementLayout`* – `IElementLayout` is the interface to a component's layout model. It allows the code generator to retrieve layout constraints for the component.

- *ElementLayout* – This is our implementation of the `IElementLayout` interface. It models the layout of our component. It also contains getter and setter methods for the possible layout constraints. These methods are used in the GUI customization stage so the user can fine-tune the layout of a component.
- *JComponent* – The `JComponent` is a reference to the actual instance of the widget we are modelling. Instead of directly modelling each GUI widget accurately, we use an instance of the Swing widget and make any customizations to the object instance itself. When the code generator accesses the `IModelElement`, it reads property values straight from the `JComponent` instance to generate property change code.

We do not directly keep a model of the widgets the user has customized. Instead we borrow from the idea of freeze-dried objects (see section 2.2.2 for details). When a user is customizing the GUI, we do not keep track of the properties that have been changed. In the customization stage, the user is actually modifying the instantiated `JComponent` object representing the widget. This means there is no need to explicitly remember these customizations. The model element maintains a reference to the `JComponent` instance. When we need to generate code, we inspect the instance to get any desired property values. This removes a large amount of complexity from our model.

In Swing, the `JComponent` doesn't keep track of its layout constraints. We directly model a component's layout using the `IElementLayout` interface. In the customization stage we model the possible layout constraints in the `ElementLayout` object of the `IModelElement`. The code generators visit these objects to retrieve the layout constraints for a widget.

6.2 Abstract Code Generation

Our code generator uses a combination of code templates and the visitor pattern to give us a highly adaptable code generator. The code generation screen asks the user for a class name, file location and target toolkit for the code generation. When the user clicks the 'Generate Code' button, the code template is loaded and the correct code generator implementation is initialized. The code generator visits the nodes from the model and injects code into the code template.

Code Templates

We use code templates to simplify code generation and allow user customization of the code. The code templates contain boilerplate code and keywords for code insertion. When the code generator is initialized, the chosen code template is loaded. The keywords in the template are replaced by GUI code from the code generator. Using this technique, the user can supply their own code templates or just use the standard templates supplied with the application. Figure 6.2 shows a simplified flowchart of the code generation system.

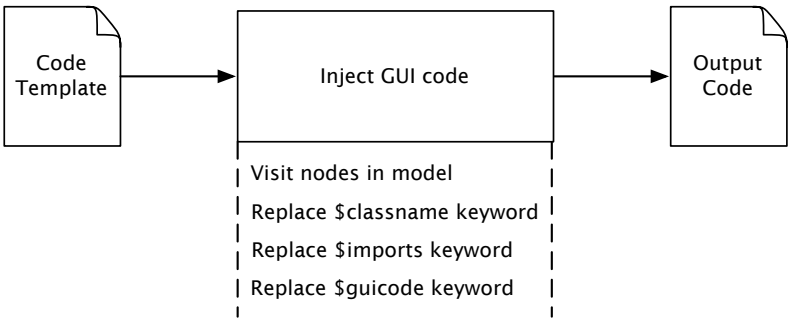


Figure 6.2: Flowchart of code generation process.

The code template contains the following keywords:

- `$classname` – The `$classname` keyword represents the class name entered by the user in the code generation screen. It is injected as the name of the class in the template.
- `$imports` – The `$imports` keyword is replaced by the imports necessary for the GUI to compile and run.
- `$guicode` – The `$guicode` keyword is replaced by all the widget component code. This includes initialization, property change and layout code for each component.

To see the supplied Swing code template, see the appendix section 11.1.

Multiple Visitors

The code required to define a GUI component can be split into four distinctive sections. These include the necessary imports for the component to be used, the code to initialize the component, the code to change the component properties and the layout code. We implement four types of visitor. Each visitor is responsible for generating a single section of the code for every widget. The UML diagram in figure 6.3 shows the classes from our code generator.

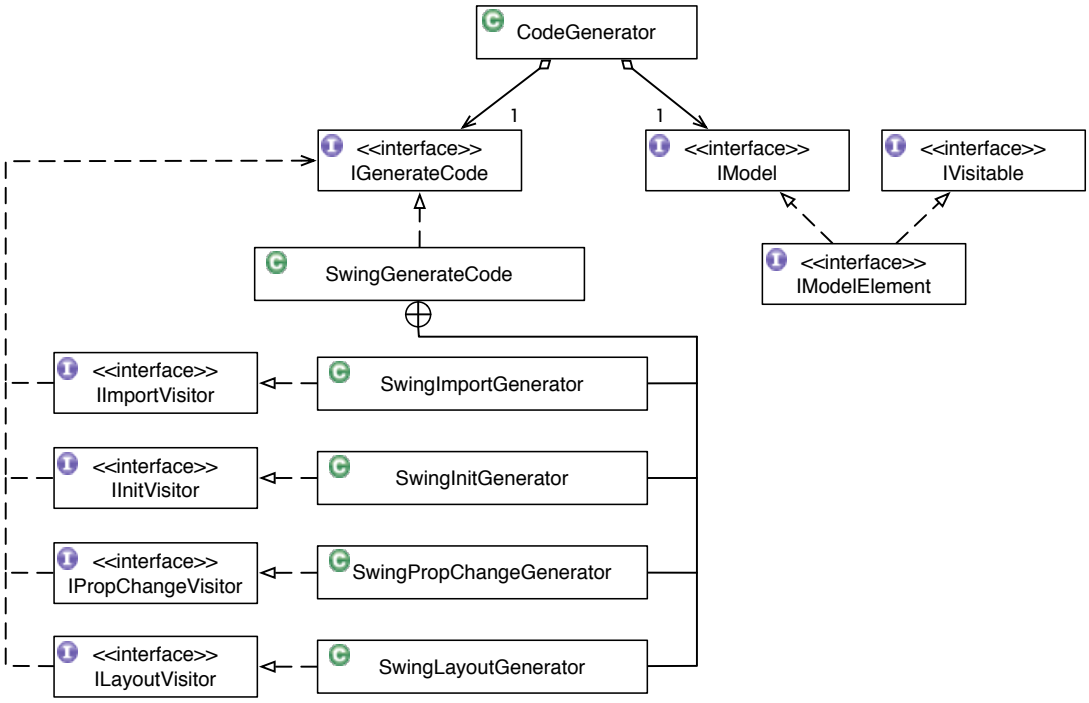


Figure 6.3: UML diagram showing the code generator system.

The elements in figure 6.3 provide the following functionality:

- *CodeGenerator* – This is the class that controls the code generation. It is instantiated with an `IGenerateCode` implementation and an `IModel` implementation. A code template is loaded and the `CodeGenerator` uses both the `IModel` and `IGenerateCode` implementations to generate the code and inject it into the code template.
- *IGenerateCode* – The `IGenerateCode` interface is the interface implemented by the concrete code generators. It provides methods for accessing the visitor implementations to generate toolkit specific code.
- *IImportVisitor* – This is the visitor for generating imports. It visits each node in the model gathering the necessary imports for the GUI components. It contains a `visit` method for accessing the nodes in the model.
- *IInitVisitor* – This is the visitor for generating component initialization code. It contains a `visit` method for accessing the nodes in the model.
- *IPropChangeVisitor* – This is the visitor for generating any property change code a GUI component might require. It contains a `visit` method for accessing the nodes in the model.
- *ILayoutVisitor* – This is the visitor for generating layout code for a GUI component. It contains a `visit` method for accessing the nodes in the model.
- *IVisitable* – The `IVisitable` interface defines `accept` methods for each of the visitors. This allows the four visitors to `visit` each model element in the code generation stage.
- *IModel* – This is the same interface as defined in figure 6.1.
- *IModelElement* – This is the same interface as defined in figure 6.1, but it now extends the `IVisitable` interface. This makes the nodes in our model visitable so that we can generate code from them.

`SwingGenerateCode`, `SwingImportGenerator`, `SwingInitGenerator`, `SwingPropChangeGenerator` and `SwingLayoutGenerator` are our implementations of the code generator interfaces for the Swing toolkit. We embed the visitors as inner classes to keep the code generator implementation in a single file. Using multiple visitors and separating them into interfaces gives us low coupling and high cohesion. We achieve low coupling because the individual visitors are completely independent and we achieve high cohesion by separating the responsibilities of code generation to different classes.

Separating the visitors also helps produce clean code. Every stage produces a separate block of code that can be injected into the code template. This gives us a very clear code structure as a result. Figure 6.5 shows an example of generated code. Figure 6.4 shows the GUI that the code represents.

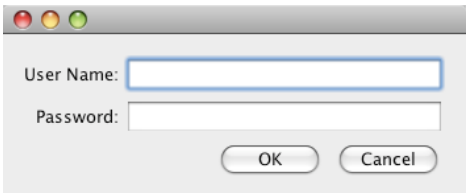


Figure 6.4: GUI window represented by the code found in Figure 6.5.

```
import net.miginfocom.swing.MigLayout;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.Dimension;
import javax.swing.JButton;
import javax.swing.JTextField;
import javax.swing.JLabel;

public class Sketchi extends JFrame
{
    public Sketchi()
    {
        JPanel p = createGUI();
        this.setContentPane(p);
        this.pack();
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setVisible(true);
    }

    /**
    * Creates the GUI controls and adds the layout code
    */
    private JPanel createGUI()
    {
        // Declarations
        JPanel mainPanel = new JPanel(new MigLayout("fill"));
        JLabel firstName = new JLabel("User Name:");
        JTextField txtFName = new JTextField("");
        JLabel lastName = new JLabel("Password:");
        JTextField txtLName = new JTextField("");
        JButton btnOK = new JButton("OK");
        JButton btnCancel = new JButton("Cancel");

        // Property Changes
        mainPanel.setPreferredSize(new Dimension(352, 120));

        // Layout code
        mainPanel.add(firstName, "cell 0 0");
        mainPanel.add(txtFName, "cell 1 0,sizegroup gr1,growx");
        mainPanel.add(lastName, "cell 0 1,align right");
        mainPanel.add(txtLName, "cell 1 1,align right,sizegroup gr1,growx");
        mainPanel.add(btnOK, "cell 1 2,align right,sizegroup gr2");
        mainPanel.add(btnCancel, "cell 1 2,sizegroup gr2");

        return mainPanel;
    }

    /**
    * Standard static void main that creates the JFrame
    *
    * @param args
    */
    public static void main(String args[])
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            @Override
            public void run()
            {
                JFrame mainFrame = new Sketchi();
                mainFrame.setVisible(true);
            }
        });
    }
}
```

Figure 6.5: Example of generated code produced by our Swing code generator.

We give a description of the code in Figure 6.5 below:

- *`main` method* – Uses `SwingUtilities` to create the GUI on the Swing GUI thread.
- *`Sketchi()` constructor* – Creates a new panel using the `createGUI()` method. It sets the returned panel as the content pane for the window and sets up the window to be displayed.
- *`createGUI()` –* This is the bulk of the generated code. We can see it is split into three distinct blocks of code with comments. The first segment represents the initialization code. We the constructor to set default properties accepted by the component's constructor. This reduces the size of the property change code by combining two functionalities into a single line. The next block of code is the property change block. This sets any customised properties the user may have chosen in the customisation stage. The final block of code is the layout code. Each widget in the GUI only requires a single line of layout code.

6.3 Summary of Code Generation

This chapter has outlined an abstract code generator based on the visitor design pattern that can be used for generating code into multiple toolkits and languages. Specifically, we use four types of visitor and delegate different parts of the code generation to each. Furthermore, we have provided an implementation for the Java Swing toolkit that produces clean, concise code.

7 Experimental Design

This chapter details how Sketchi was tested from a usability and software engineering perspective. The results and analysis of these tests can be found in the proceeding chapter.

In the tests outlined below we only use five users ***unless otherwise stated***. This is firstly due to time constraints and secondly because "the best results come from testing no more than five users"(Nielson, Why You Only Need to Test With 5 Users 2000). Nielsen states that after a single test with 5 users, 85% of usability problems will be found.

All tests were carried out using an early 2008 Apple Macbook Pro with a 2.4GHz Intel Core 2 Duo and 4GB RAM. It was running OS X 10.5.7 with Java SE Runtime Environment 1.6.0_07. For user input, we connected a Wacom Cintiq 15x touchscreen monitor.

7.1 Input Recognition Rates

The most important part of turning a sketch into a GUI is the input classification. If the application cannot correctly classify the widgets, it will generate a GUI an incorrect GUI. To test the accuracy of the input classifier, we measured the recognition rates for the GUI widgets. We asked the users to draw each widget 10 times. We recorded the position in the suggestion list of the correct classification. Appearing at position 1 in the list means the widget was the first suggestion and hence correctly classified. We want to test four different modes of operation for the input classifier:

- *Single Feature enabled* – This testing was carried out after the initial receptor feature was implemented (see section 4.2.2 for the receptor implementation details). The user was asked to carry out the test with only the receptor feature enabled in the classifier.
- *All Features enabled* – This test involved using all the features as described in section 4.2.5. The user was instructed to carry out the same test, but this time all features were turned on.
- *Single Feature enabled with learning* – The previous two tests used a default case base with extra instances from the user. This test involved the user drawing each widget five times that were added as cases to the classifier. The user then carried out the standard test of drawing each widget ten times with only the receptor feature turned on.
- *All features enabled with learning* – This test is as above but with all features turned on as well as five instances of learning per widget added to the case base.

To perform the tests, we performed the following steps:

- *Select test users* – A random selection of five users were chosen to test the software. All users had programming experience but their level of experience with GUI creation ranged from complete novice to expert. This concentrates our testing on our target demographic of programmers that don't necessarily have GUI building experience.

- *Give the user the widget table* – The users were given table 4.1 that shows the widget representations.
- *Give instructions to the users* – The users were given instructions on how to carry out the test. In the case where learning was not undertaken, the user was simply asked to draw whatever object they liked so long as all widgets were drawn a total of ten times. In the case of learning, the user was asked to draw each widget five times in turn, so the learning could be applied. After which they were free to draw the widgets in whatever way they wished.
- *Analyse results* – The user drawn images with their widget representations were recorded and analysed after the tests so the user was unaware of the successes or failures of the drawing.

We are interested in the 'top hit' and 'top three hit' percentages. The 'top hit' percentage is the percentage of times the correct classification appears at the top of the suggestion list. The 'top three hit' percentage is the number of times the correct classification appears in the top three items of the suggestion list. We are mainly concerned with the 'top hit' percentage as it represents the recognition rate with no user intervention. We use the 'top three hit' percentage as a metric to further compare the four different tests. The input classifier can be considered usable if the top hit percentage approaches 80%. This would result in only 1 in 5 widgets having to be corrected by the user.

We also use this testing to accumulate test images to compare the performance of different feature weights (outlined in section 8.1). We use the collected test images in a testbed (see section 4.2.5) to compare the recognition rates using the manually tweaked weights against the weights optimised through hill climbing. The results for all of these tests can be found in section 8.1.

7.2 Layout Generation Flexibility

Here we are testing the flexibility of the layout generator. We want to test the layout generator's ability to generate a variety of layouts. We will select a set of interfaces for our test users to draw and see if the correct layouts are generated. We will use table 7.1 as a template for our results.

Attempted interface	Is there a correctly generated layout?	Steps needed to correct layout in Sketchi?

Table 7.1: Blank results table for layout generation flexibility test.

We will show the interface that was being drawn along the user's sketch, whether the generated layout is correct and in the case the layout is incorrect, the steps required to achieve the correct layout in the customization stage. The layout generator can be considered flexible if it can generate all the examples presented. The results of this test can be found in section 8.2.

7.3 Quality of Code Generation

This test is concerned with the quality of the code generation. The quality of code is a qualitative measure as it is an opinion. We will use Sketchi to generate code for a selection of

interfaces. We will also use the NetBeans GUI builder to create the same interfaces. We will then compare the size of the code over all the interfaces. We will also ask a collection of twenty programmers which code samples they feel is more readable without labelling the samples. The code generated can be considered of high quality if the panel of programmers agree that it is more readable in the majority of cases. The results of this test can be found in section 8.3.

7.4 Project Brief Assignment

The project brief assignment is a specific type of usability test. We test the system with three different users whose Swing GUI creation experiences are novice, intermediate and professional respectively. We use three users as it is easy to define their levels of experience into three categories. The users are given a GUI to create as a design brief and are asked to create it using a variety of techniques. We ask the users to create the GUI by manually coding, using NetBeans and using Sketchi. We record the time taken to arrive at the finished product and compare the quality of the code produced as detailed in the preceding section. The test can be considered a success if Sketchi can reduce the time taken to produce a GUI compared to manually coding whilst maintaining high quality of code generated. The results for this test can be found in section 8.4.

7.5 Usability Testing

Ensuring Sketchi has high usability is crucial for it to be successful. If the user cannot quickly and easily create their GUI using Sketchi then it holds no advantages over other methods of GUI building. We perform end-user testing using a variety of techniques described in 7.5.1 and carry out a more technical assessment using Nielson's usability heuristics as outlined in 7.5.2.

7.5.1 End-user Testing

In our end-user testing, the users were asked to use Sketchi to create the code of a simple GUI, which can found in figure 7.2. The users were given Table 4.1 showing the widget representations to use and asked to use the Sketchi to generate the GUI. This ensures that we concentrate on the usability of the Sketchi interface rather than any intricacies of the input classification system. Screen-capture software[i] was used to record the users during the tests, so the results could be analysed afterwards. We also observed the users to see where they struggled with particular elements of the interface.

We recorded the time taken to complete the task and number of user clicks during the test. This does not include the clicks during the drawing of the interface as this is influenced by the user's drawing style and would possibly skew the results. Removing these clicks from our results hones the testing on usability of the application rather than the user's drawing style. To evaluate the user's experience with the software, we use the following two methods:

[i] The screen-capture software used was iShowU HD. There website can be found at http://store.shinywhitebox.com/ishowuhd/main.html.

- *Questionnaire* – The standard test performed in usability testing is to give the users a questionnaire. The users were questioned about what extent they agreed or disagreed with statements regarding the system. When designing a questionnaire, it is important not to lead the users to a particular answer. This will introduce bias to the results. To overcome this, half of the statements in the questionnaire were positively phrased and half were negatively phrased. When designing the scale for the options the user can choose, we ensure that we have an equal number of positive and negative options. This helps remove possible acquiescence bias[i] and allows us to use a Likert scale, which can be analysed in the results. To see the questionnaire, see appendix 11.2.
- *Word List* – David Travis from User Focus[ii] says, "Experience shows that participants are reluctant to be critical of a system, no matter how difficult they found the tasks". The word list technique(Travis 2008) is an adaption of the 'Product Reaction Card' test developed by Microsoft researchers as part of the Microsoft Desirability Toolkit(Benedek and Miner 2002). It is used to gauge user satisfaction whilst minimizing the acquiescence bias. The word list technique involves giving your users a large list, typically over 100, of adjectives to describe your system. They contain an equal number of positive and negative adjectives so not to lead the user on. They are also randomized for each user to remove order bias. The user selects as many adjectives as they wish that they felt apply to the interface. They are then asked to select the five words that best describe their experience with the system. These words are then used in a post-test guided interview with the user. We use these adjectives to direct our questions and find out why the users felt this way about the system. Figure 7.1 shows one of the randomized word lists given to a user.

Simple	Responsive	Familiar	Complex	Too technical
Easy to use	Intuitive	Poor quality	Credible	Entertaining
Time-consuming	Stimulating	Misleading	Engaging	Secure
Consistent	Dated	Slow	Comprehensive	Attractive
Empowering	Approachable	Desirable	Reliable	Incomprehensible
Difficult	New	Relevant	Usable	Bright
Friendly	Non-standard	Frustrating	Insecure	Inconsistent
Simplistic	Awkward	Busy	Motivating	Cluttered
Overwhelming	Business-like	Unpredictable	System-oriented	Illogical
Fast	Organised	Old	Straightforward	Confusing
Convenient	Boring	Counter-intuitive	Unattractive	Fresh
Irrelevant	Contradictory	Powerful	Unrefined	Appealing
Unconventional	Innovative	Ordinary	Intimidating	Compelling
Patronising	Understandable	Exciting	Satisfying	Trustworthy
Flexible	Meaningful	Annoying	Inadequate	Predictable
Effortless	Accessible	Rigid	Advanced	Obscure
Distracting	Vague	Impressive	Ineffective	Effective
Time-saving	Sophisticated	Dull	Energetic	Fun
Controllable	Hard to Use	High quality	Professional	Clear
Stressful	Stable	Cutting edge	Efficient	Expected
Faulty	Ambiguous	Creative	Useful	Clean

Figure 7.1: Example word list given to users.

[i] Acquiescence bias is the fact that people are more likely to agree with a statement than disagree with it - Lee J Cronbach, "Response sets and test validity," *Educational and Psychological Measurements 6*, 1946: 475-494.

[ii] User Focus is a London-based usability consulting and usability training company. Their website can be found at http://www.userfocus.co.uk/.

The results of these two tests will be used to improve and refine the interface after which another round of usability testing can be carried out. The results of the end-user testing can be found in section 8.5.1.

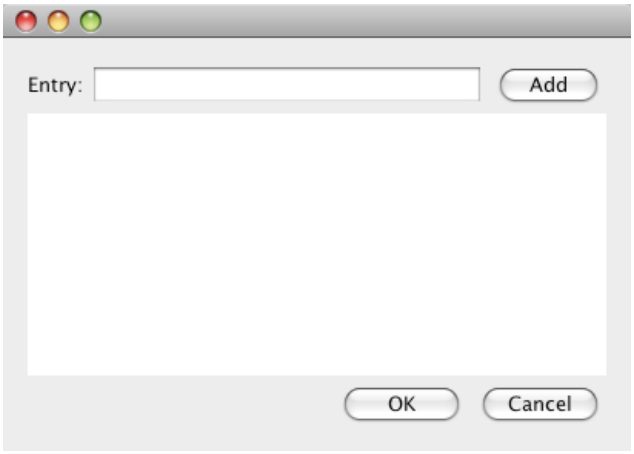


Figure 7.2: Simple user interface that the users were asked to create in the end-user testing.

7.5.2 Nielsen's Usability Heuristics

Another measure of usability is how well a system adheres to Nielsen's Ten Usability Heuristics[i](Nielsen 1994). Sketchi will be assessed in accordance with each of these heuristics to ensure that are met. The results of this test can be found in section 8.5.2.

7.6 Performance Testing

It is important that Sketchi is fast and responsive. In such a user centric application, unnecessary waiting for calculations can frustrate users and degrade the quality of the software. To acquire accurate performance measurements, we used the JProfiler[ii] profiling software. The tests we made to measure the performance of Sketchi, were as follows:

Receptor Generation

In section 3.2.2, two methods for generating receptors were proposed. To compare the two methods, we record the time taken to generate a range of sets of receptors. We then use the test images, acquired from our input recognition rate testing, to compare the quality of the receptor sets at detecting the test images.

Widget Classification

The time taken to classify a user sketch of a widget is an important measure for us to find. Performance bottlenecks in the classification widget will quickly multiply when the user draws multiple widgets in a GUI design. We will measure the time taken to classify sketches containing different numbers of points. For example, an unfilled square of dimension 10 will contain 36 drawn pixels. We will input widgets with varying numbers of drawn pixels and

[i] Jakob Nielsen is a usability engineering expert and has written many papers in the area of usability. His website can be found at http://www.useit.com/.
[ii] JProfiler is a GUI based profiler for finding performance bottlenecks, pin down memory leaks and resolve threading issues. More information can be at http://www.ej-technologies.com/products/jprofiler/overview.html.

measure the time taken to suggest a classification. Ideally we hope to achieve a classification in under 0.1 seconds. This is the limit for having the user feel that the system is reacting instantaneously (Miller 1968) (Card, Robertson and Mackinlay 1991).

Layout Generation

The layout generation performance test times how long it takes a layout to be generated for a variety of inputs. We measure the time taken as we increase the number of widgets that need to be laid out.

Code Generation

To test the performance of the code generation, we need to time how long it takes to generate code for various size input models. We will use input models containing of a range of numbers of widgets and record the time taken to generate the code for our Swing implementation.

The results of all performance tests can be found in section 8.6.

7.7 Robustness & Stability

7.7.1 Monkey Testing

A good way of testing any software system is to use monkey testing. This involves subjecting the software system to random input wherever input is possible and observing the results. To ensure the system is robust, we check that random input does not cause the system to crash and that it is handled in an appropriate manner. Table 7.2 shows the test plan. The results of the monkey testing can be found in section 8.7.1.

Test No.	Test
1	Draw random input onto the drawing pane in Free Design stage
2	Make random selections that do not necessarily contain any part of a drawn widget in the Widget Detection stage
3	Entering a file name that doesn't exist to load a sketch
4	Running with no preference file present
5	Running with no resource files present
6	Classifying no drawing
7	Generating an interface with no widgets
8	Generating code with no widgets
9	Entering non-numerical data in a field expecting only numbers
10	Entering large numbers in number fields
11	Entering random text into the Class Name field of the code generator
12	Entering a non existent folder in the target folder of the code generator
13	Attempt to generate code when the code template is missing

Table 7.2: Monkey testing test plan.

7.7.2 Stress Testing

We will also run very specific tests in an attempt to put the system under excess load. We will perform the following tests:

- Load large images into Sketchi. Users can load bitmap images into the drawing pane of Sketchi. We will attempt to overload the system by loading abnormally large bitmaps.
- Drawing a high number of widgets (50+) in the Free Design stage of Sketchi will result in a stress test of the widget detector and classifier.
- Generating a layout with a large number of widgets will result in a stress test of the layout generator.
- Generating code for an interface that contains a large number of widgets will stress test the code generator. We will attempt to overload the code generator by passing it an interface with over 50 widgets.

The results of the stress testing can be found in section 8.7.2.

8 Results & Analysis

This chapter details and explains the results from the experiments described in the previous chapter.

8.1 Input Recognition Rates

The recognition rates results showed increases for every user as more advanced modes of operation were enabled. Figure 8.1 shows the top hit percentages[i] for our five test users across all four modes of operation.

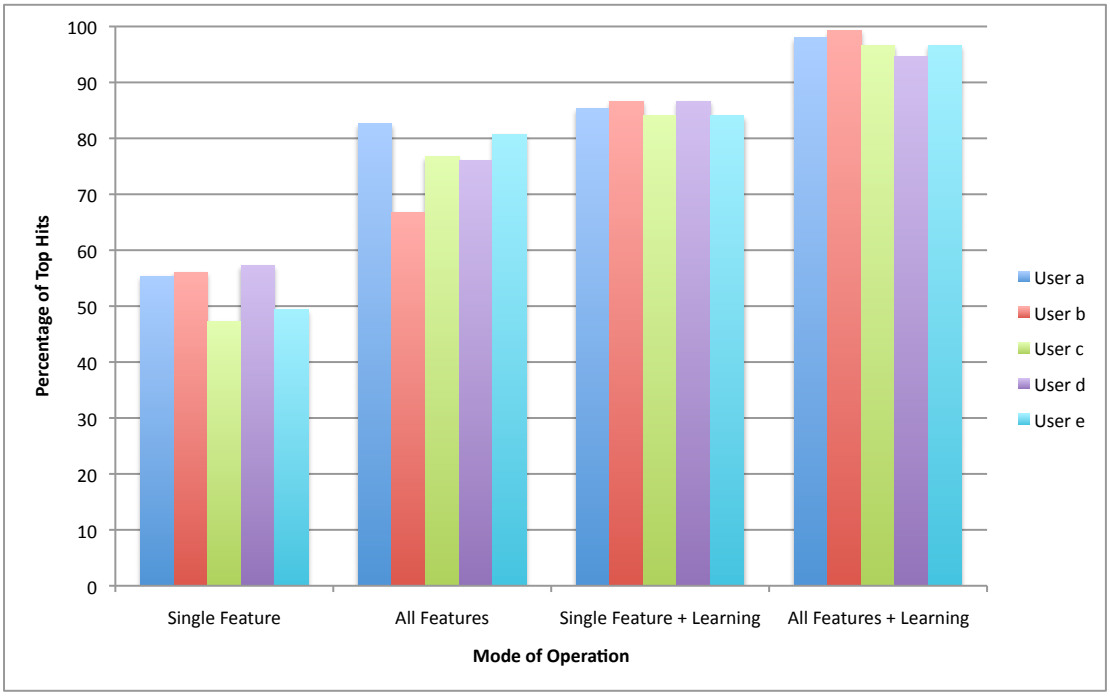


Figure 8.1: The recognition rates for the widget classifier show that, with a small amount of learning, every user can achieve a recognition rate approaching 100%.

We can see that using the extra heuristics gives an improvement in the recognition rates for all users. Whilst the recognition rates are not high enough using only the single feature or all features, the results show that adding a small number of user-specific training examples to the case base improves the recognition rates dramatically. When all the features are enabled, every user was able to achieve a recognition rate of 95% or more.

Figure 8.2 shows the top three percentages for our five test users across all four modes of operation. The results show that with no learning the correct classification is placed in the top three suggestions over 88% of the time. As learning examples are added to the case base and all features are enabled, this percentage reaches 100%. This is a useful fact for designing the suggestion list in the Sketchi application. If it can be assumed that the correct

[i] Top hit percentage is the percentage of times the correct classification was given as the top suggestion to the user.

classification will appear in the top three suggestions, we only need to display these suggestions to the user in the classification stage. This will reduce unnecessary information onscreen and keep the interface clean.

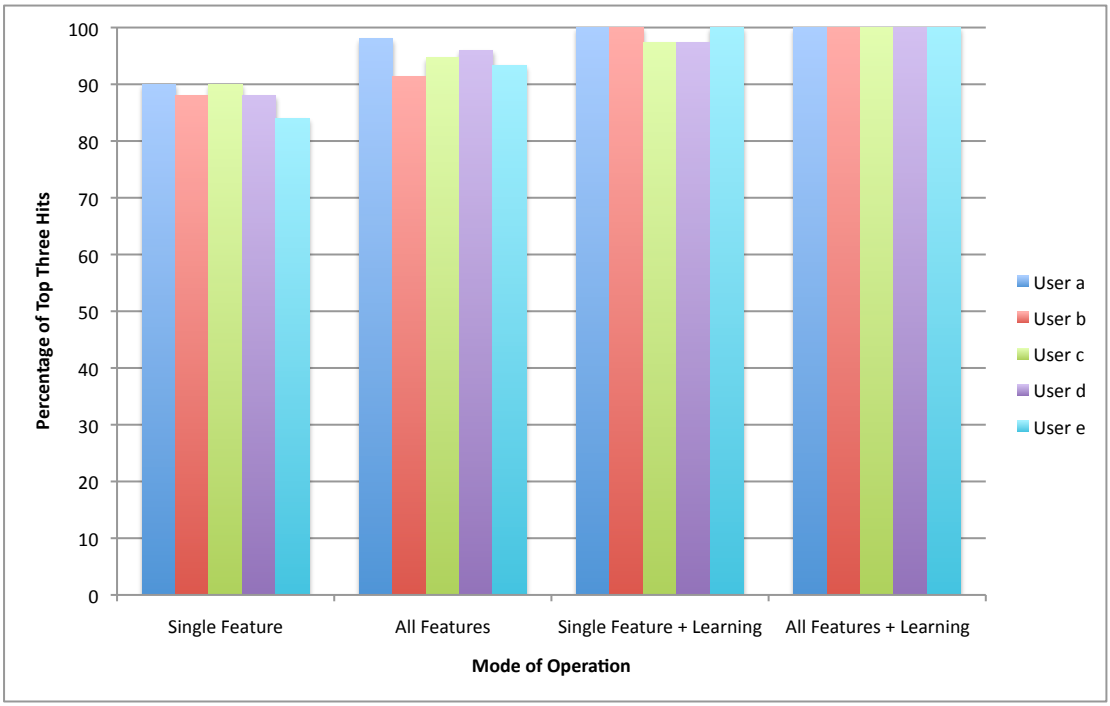


Figure 8.2: The top three hit percentage from recognition rate testing shows we can almost guarantee a top three hit for every user.

Once recognition rate testing was completed we used the images recorded from our users to test the feature weights. Figure 8.3 shows the results of the tests for our manually tweaked weights and weights generated through our hill climbing algorithm (see section 4.2.2).

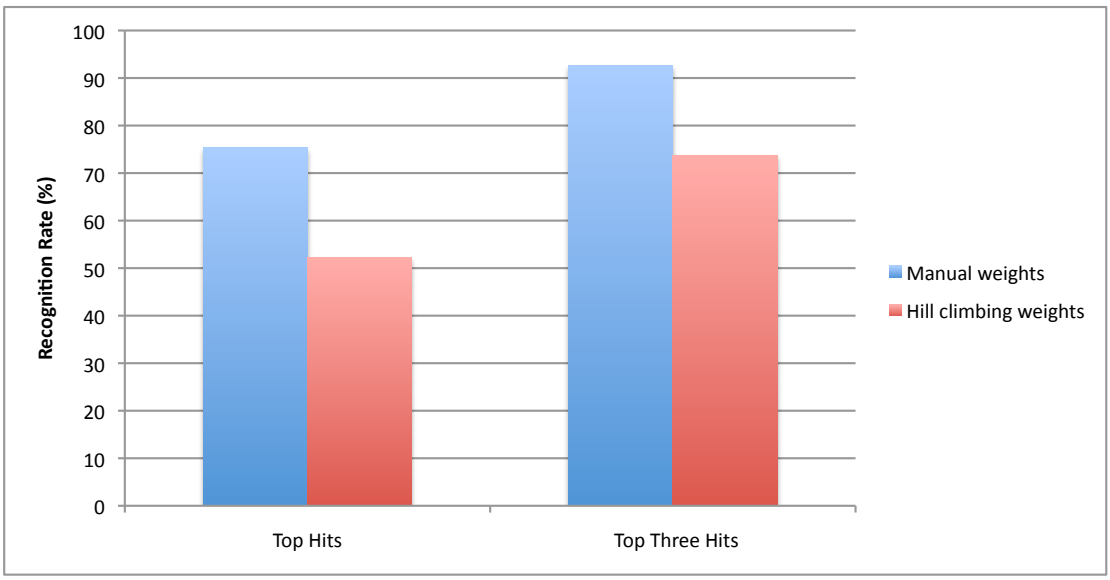


Figure 8.3: Manually tweaked weights performed better than the weights obtained through our hill climbing algorithm.

There are two possible causes for the poor performance of the hill climbing compared to the manually adjusted weights:

- *Overfitting data* – A common problem with machine learning is the possibility of overfitting the training data. Overfitting can occur when learning is undertaken for too long or when the examples in the training set rarely appear in the testing set. To overcome the potential overfitting problem we can shorten the length of our learning process or acquire more example images to learn with. These are suggested in the future work in section 9.2.
- *Poor algorithm* – As described in 4.2.2, adjusting the weights to achieve good results requires intuition. The algorithm outlined in section 4.2.2 is not very advanced and is not sophisticated in how it adjusts its weights. It uses the intuition that if a widget is misclassified then lowering its weights will move it nearer to the top of the suggestion list. Whilst this is true, it does not guarantee an improvement in the global solution only an improvement in the local solution. A better solution may be to increase the weight of the misclassification. This will move the misclassification down the suggestion list, leaving the correct classification as the top of the list. A more advanced technique can be used to achieve better weights and we leave this to the future work in section 9.2.

8.2 Layout Generation Flexibility

In the layout generation flexibility test we take a series of example interfaces and attempt to generate them from user drawings of the respective interface. The interfaces we are attempting to generate with their counterpart user drawing are displayed in figures 8.4 to 8.13.

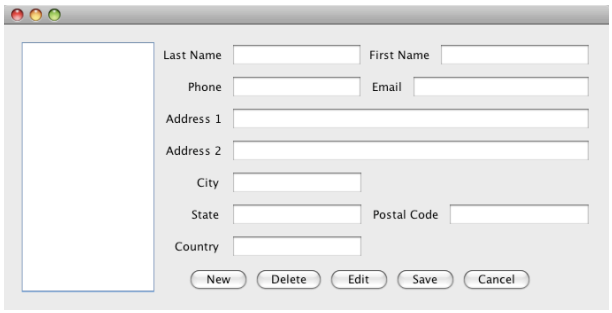


Figure 8.4: Rendered address book window for layout generation testing.

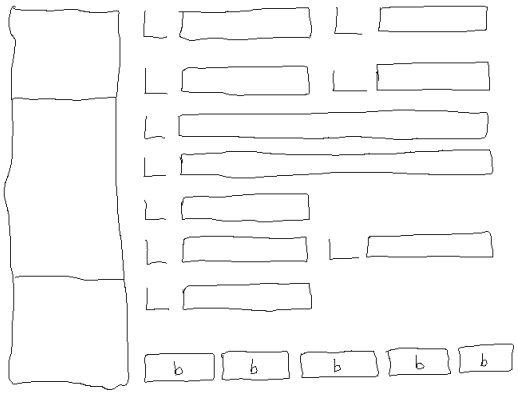


Figure 8.5: User drawn address book window for layout generation testing.

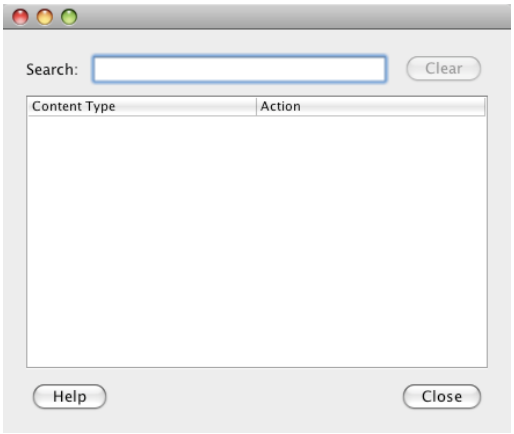


Figure 8.6: Rendered search window for layout generation testing.

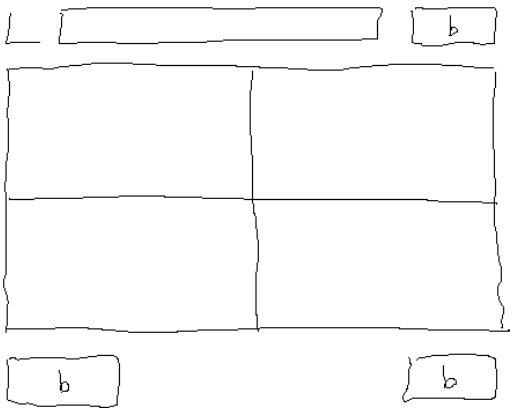


Figure 8.7: User drawn search window for layout generation.

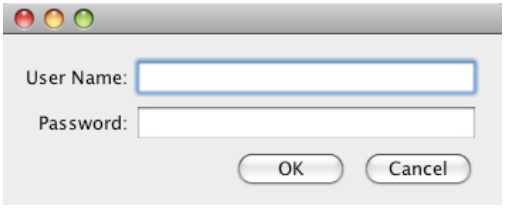


Figure 8.8: Rendered login window for layout generation testing.

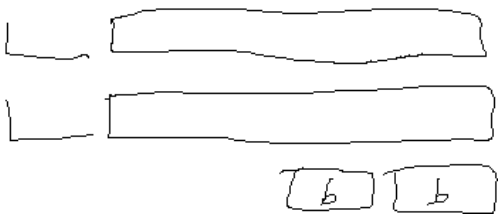


Figure 8.9: User drawn login window for layout generation testing.

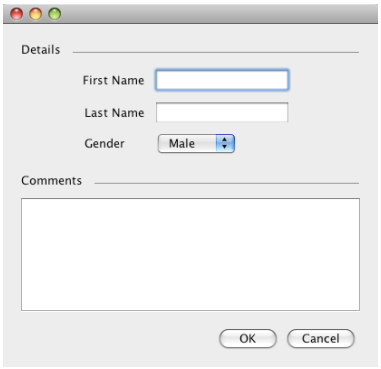


Figure 8.10: Rendered user details window for layout generation testing.

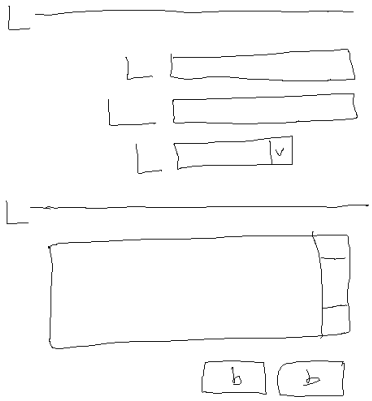


Figure 8.11: User drawn user details window for layout generation testing.

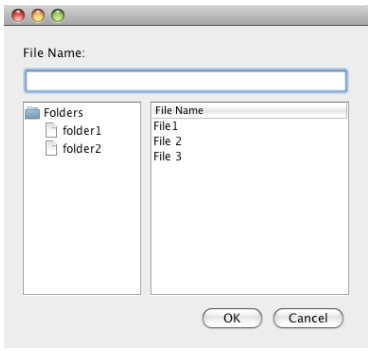


Figure 8.12: Rendered save dialog for layout generation testing.

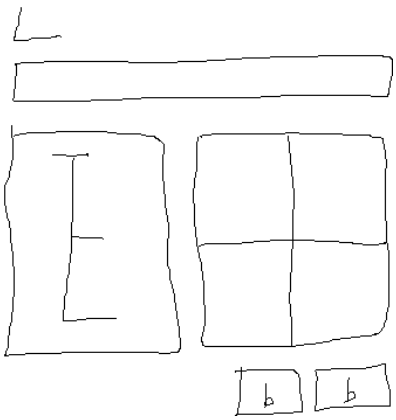


Figure 8.13: User drawn save dialog for layout generation testing.

Table 8.1 shows the results of attempting to generate these layouts.

Attempted interface	Is there a correctly generated layout?	Steps needed to correct layout in Sketchi?
Figure 8.4	No. There is a similar layout that suffers from inconsistent gaps.	To correct the layout, the user must select the 'push gap' option in the layout editor.
Figure 8.6	Yes.	N/A
Figure 8.8	Yes.	N/A
Figure 8.10	Yes.	N/A
Figure 8.12	Yes.	N/A

Table 8.1: Results of layout generation flexibility testing how the majority of interfaces could be generated with no layout alterations by the user.

The results show that Sketchi is able to produce a variety of layouts with minimal user interaction. Figure 8.4 is the interface from the layout manager challenge described in section 2.4.2. This is regarded as the benchmark for how powerful a layout manager is and is expected to be undertaken with manual coding. Figure 8.14 shows the solution generated by Sketchi.

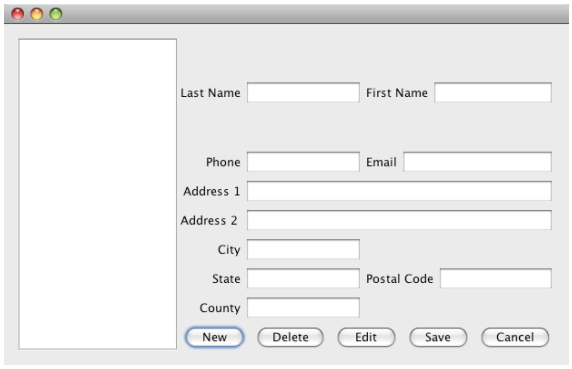


Figure 8.14: Address book generated by Sketchi. Shows the unwanted gap artefact in the layout.

We can see that there is unwanted space above and below the first row. The layout generation code for the top row can be seen in figure 8.15.

```
mainPanel.add(list, "cell 0 0 1 8, growy");
mainPanel.add(lstName, "cell 1 0,align right,");
mainPanel.add(lname, "cell 2 0,growx,");
mainPanel.add(fstName, "cell 3 0,");
mainPanel.add(fname, "cell 3 0,growx,");
```

Figure 8.15: Generated layout code for address book example.

The list is being added to the top row and is being told to grow in the vertical direction. Since the list appears in the first row, it is trying to expand the row, which contains other elements. To overcome this problem we need to add a 'wrap push' layout constraint to an item in the bottom row. This is labelled in the layout editor of the customization stage as 'push gap'. The 'wrap push' constraint pushes the gap below the row of the component it is added to. This will resolve the gap issue. Unfortunately the current layout manager implementation cannot detect this type of issue. The enhancement is left for future work and is detailed in section 9.2.

8.3 Quality of Code Generation

To test the quality of the code generation, we created several interfaces using Sketchi and NetBeans. The code produced by both tools was analysed for code size and layout code size. Figure 8.16 compares the size of code produced by Sketchi and NetBeans as more widgets are added to the interface. To ensure a fair a comparison was made, we removed any blank lines, comments and styled the code identically in terms of brackets. This means the code left was just the application code.

The results show that Sketchi produces much less code than NetBeans. It also shows that as the number of widgets in the interface increases, the size of the generated code increases as expected. This increase occurs at a constant rate and is contributed to the following:

- *Import code* – At most, one extra line of code will be required if the widget's class has not already been imported by another declaration.
- *Initialization code* – One more line of initialization code will be required for any widget added to the interface.
- *Property change code* – The property change code could contribute zero or more lines of code to the generated code depending on how much it is customized by the user in the customization stage of Sketchi.
- *Layout code* – This is the main area where Sketchi will generate less code than NetBeans. An extra widget will require exactly one extra line of layout code whereas this will likely be multiple lines of code in NetBeans. This is due to the verbose nature of the GroupLayout layout manager NetBeans uses in its GUI builder.

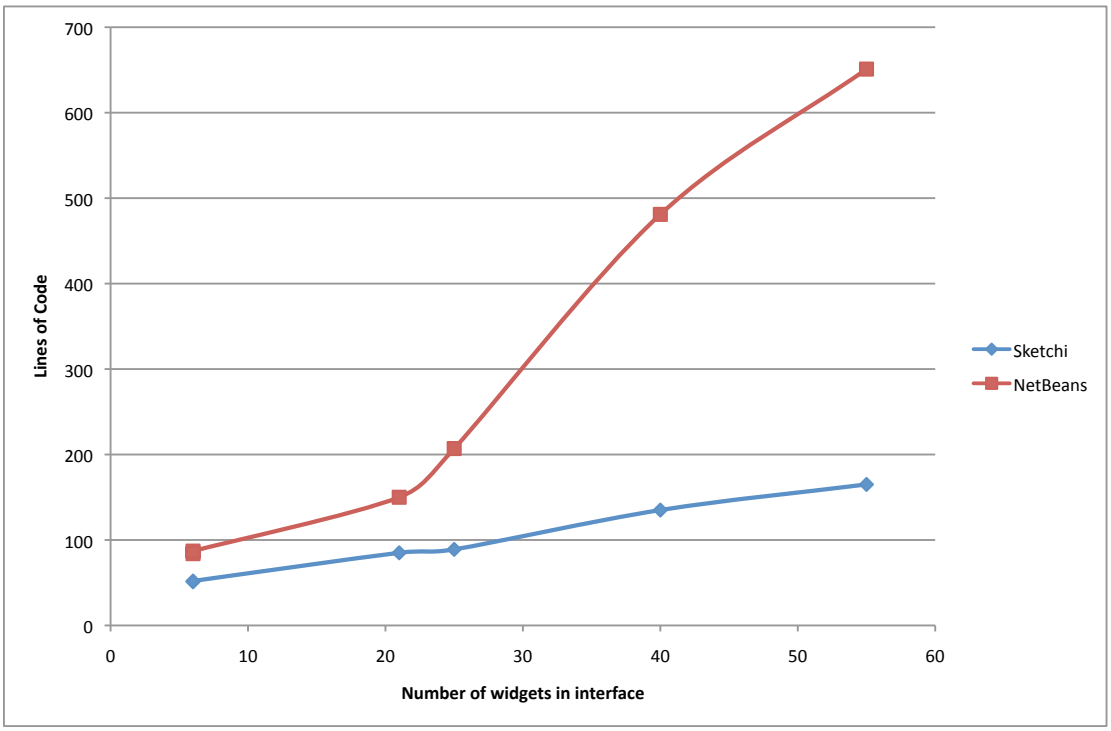


Figure 8.16: Graph showing the total code generated by NetBeans and Sketchi as more widgets are added to the interface.

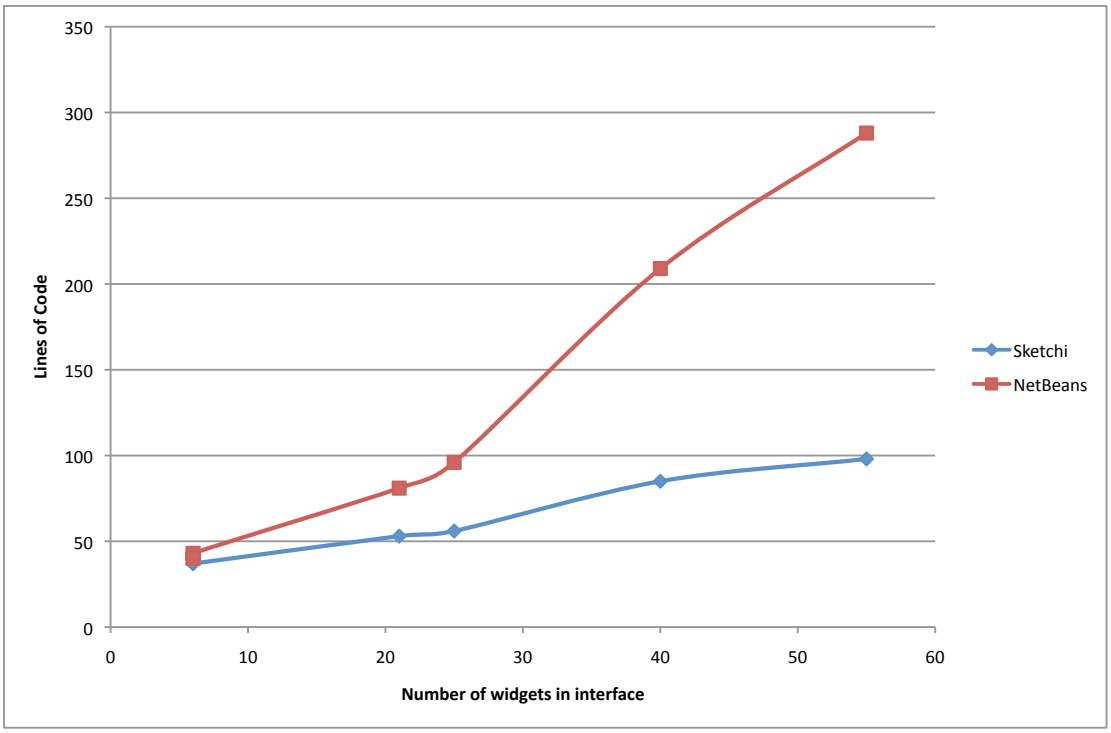


Figure 8.17: Graph showing the code generated by NetBeans and Sketchi without the layout code included in the total.

Figure 8.17 compares the size of code generated without the layout code included in the total. This still shows the Sketchi generated code increasing at a lower rate than NetBeans. To explain the unchanged rate of NetBeans we inspect the generated code. NetBeans declares every widget as a private field. It also uses separate lines of code to initialize these fields and declares properties such as the text for a widget in another line. In contrast, Sketchi declares all of its widgets method local and the variables intelligently by combining property changes into the constructor where possible.

To acquire an unbiased opinion on the quality of the code, we used a panel of programmers. The panel members were each given an unlabelled copy of each file of generated code. In every case, the panel was unanimous in saying Sketchi's code was more readable and subsequently of higher quality. This can be attributed to the verbosity of the GroupLayout layout manager NetBeans uses (see section 2.4.2 for details on the GroupLayout manager). Ideally, other GUI builders would be included in the test but this was not possible due to time constraints and we leave this to the future work in section 9.2.

8.4 Project Brief Assignment

The project brief assignment was undertaken with three users whose Swing knowledge can be considered as novice, intermediate and professional respectively. Figure 8.18 shows a comparison of the times taken to complete the exercise using the three techniques as described in section 7.4.

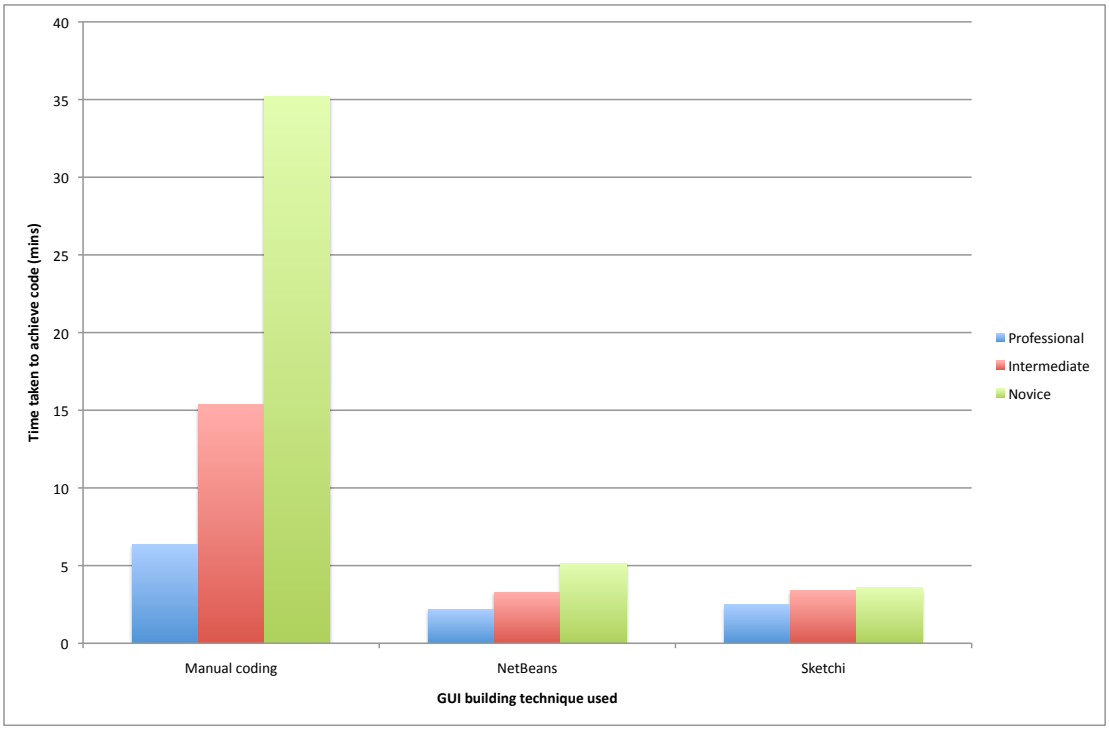


Figure 8.18: Using either Sketchi or NetBeans significantly reduces the time to create a GUI.

Figure 8.18 shows that using either GUI builder significantly speeds up the process of producing the design compared to manual coding. It is also interesting to see that using a GUI builder reduces the time differences between varying expertise levels. In the case of the novice user, using a GUI building tool showed at least a 7 times speed up in the time taken to complete the task. Comparing the times for Sketchi and NetBeans reveals a negligible difference for the professional and intermediate users with NetBeans edging out Sketchi in both cases. This can be explained by both users' prior experience with the NetBeans tool. However, the novice user was able to produce the desired GUI 40% faster with Sketchi than with NetBeans. When observing the user complete the task with NetBeans, it was noticed that the user was initially confused by the interface and found the necessary tools difficult to find. Once the user has familiarized themselves with GUI building tools, the time to completion was comparable to Sketchi. This shows that the Sketchi interface is more intuitive and simpler to use than that of NetBeans.

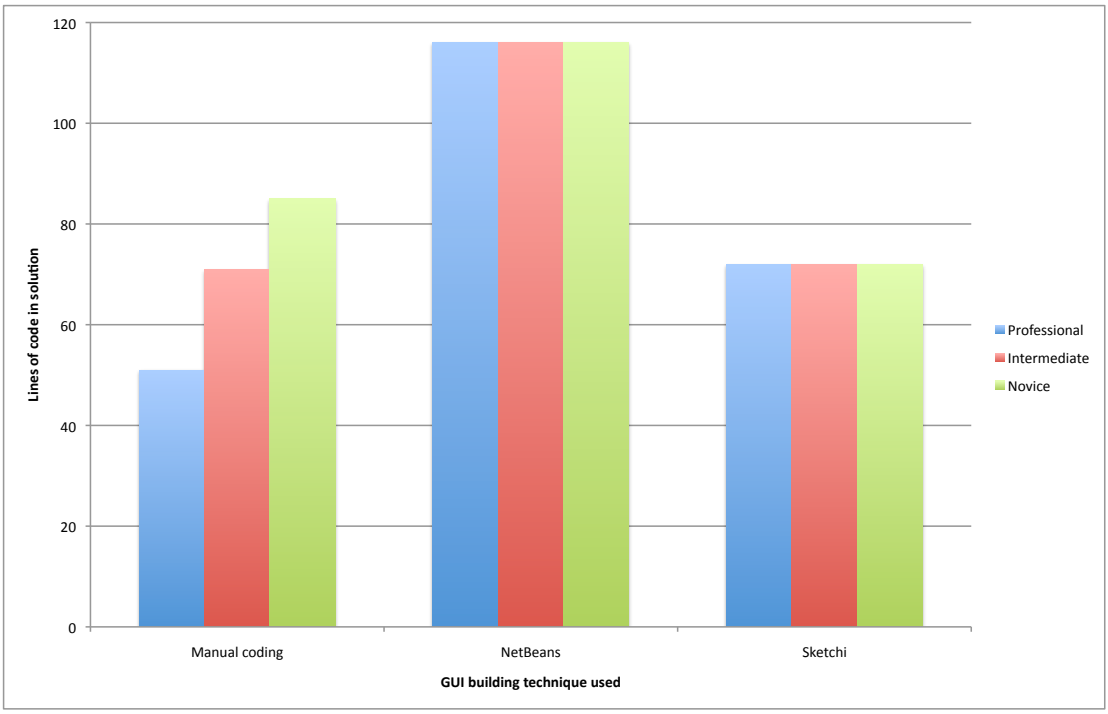


Figure 8.19: Comparison of the size of code produced by manually coding, using Sketchi and using NetBeans.

Figure 8.19 shows the size of the code produced by the users across all three techniques. The code produced by both GUI tools was constant across all three users. This is to be expected, as the code generation is deterministic so the same code will be produced for the same interface. The results reiterate the findings from figure 8.16 that Sketchi produces significantly less code than NetBeans for the same interface. The data shows that code generated by Sketchi was not as concise as the manually coded equivalent.

When analysing all of the code samples produced, it was found that the biggest differences were in the layout management code. Figure 8.20 shows a comparison of the size of layout management code extracted from all the code samples. We can see that Sketchi produces significantly less layout management code than both NetBeans and manually coding. For the intermediate and novice users the generated layout code was 6 times and 7.5 times smaller than the manually coded counterparts respectively. This can be explained by the choice of layout manager used when manually coding as well as the users' experience. Both the intermediate and novice users used the GridBagLayout layout manager (see section 2.4.2 for details). This requires significantly more code than MigLayout for the same user interface. The professional user was able to produce the same number of lines of layout code as Sketchi as the user utilised MigLayout in their manually coded solution.

Overall the results show that using a GUI building tool significantly reduces the time required to build a GUI. They show that the tools reduce the differences in times between the three levels of expertise of users. The times for Sketchi and NetBeans are comparable although for a novice user, the simple interface for Sketchi resulted in a quicker time to complete the task. Finally, although the times are comparable, the quality of code produced by Sketchi is far smaller in size when compared to NetBeans. This is a result of the layout manager it is based on as well as other factors outlined in the analysis of the results of quality of generated code in section 8.3.

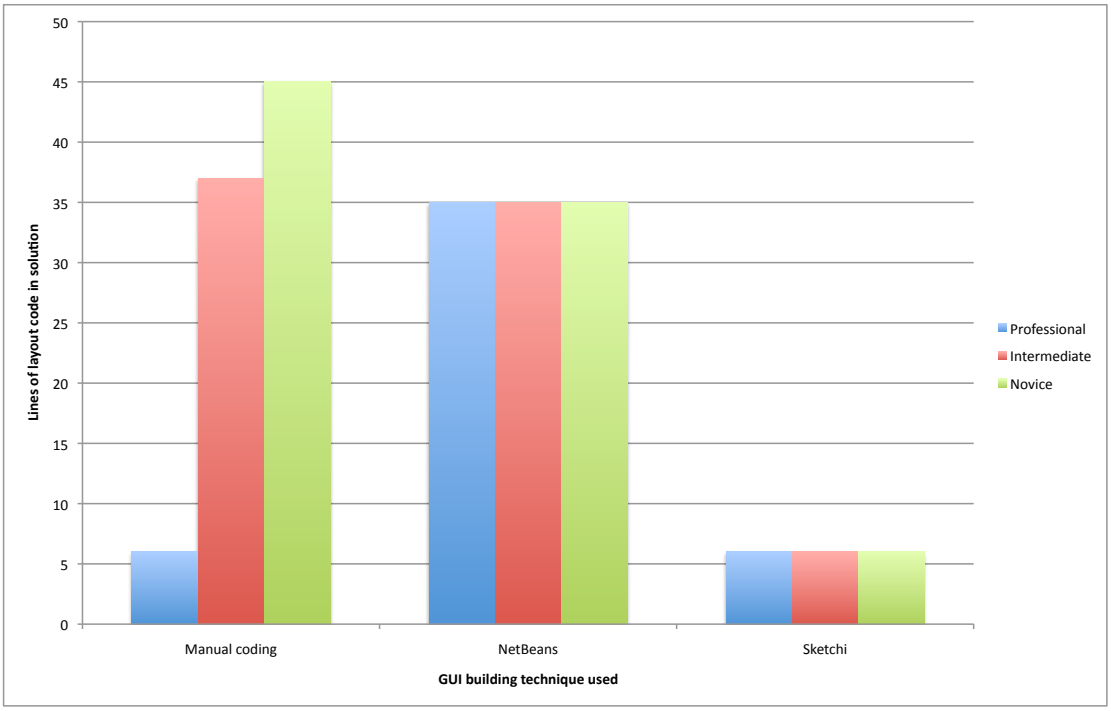


Figure 8.20: Comparison of the layout code produced across all code examples in the project brief test shows Sketchi significantly reduces the amount of layout code when compared to both NetBeans and manual coding.

8.5 Usability Testing

8.5.1 End-user Testing

The user testing revealed several flaws with the Sketchi interface. The time taken to produce the correct interface ranged from 2 minutes 18 seconds to 3 minutes 32 seconds. The number of clicks required to complete the task ranged from 7 clicks to 18 clicks. The higher number of clicks and the higher times correspond with the number of corrections each user had to make at the widget detection and widget classification stage of the software. Figure 8.21 shows the relationship between the number of corrections required by the user and the time taken to complete the task.

Observations made during testing

We observed the users during the testing and discovered key usability flaws experienced by the majority of our users. These problems are outlined below:

- *Navigation issues* – The key cause of confusion in the GUI generation process was navigating through the stages of the system. Users felt there was not enough instruction on how to proceed through the software. This can be contributed to the navigation controls being found in the application toolbar. It was suggested that these should be brought into the content pane area, possibly into a floating panel like the toolbox. The other navigational issue was when a stage required confirmation form the user. A notification panel is displayed to ask the user for confirmation before proceeding to the next stage. Users were attempting to click the navigation buttons whilst the notification panel was active which results in nothing happening. This confused the users.

- *Toolbox usefulness* – The toolbox displays information and tools for the current stage. Users tended not to notice its importance and hence had trouble using the software. After the users realised the toolbox was displaying useful information they showed few further issues. The importance of the toolbox needs to be made more obvious to the users perhaps through subtle animations to draw the users attention.
- *Reclassifying widgets* – The users struggled to understand how to change a widget classification. This is due to a lack of instructions in the toolbox for the classification stage. Once the users understood how to change a classification, some could not find the classification on the panel. The toolbox displays the top six classifications as radio buttons in the window and displays the others in a combo box to maintain a clean interface. Unfortunately, users did not understand the other classifications could be found in the combo box. To resolve this, the combo box needs to be labelled.
- *Interaction issues in the customization stage* – Users did not understand immediately how to use the customization stage of Sketchi. This was caused by a lack of instructions in the toolbox. Multiple users commented on an annoyance they found with having to perform multiple clicks in order to change the properties of a widget. The users suggested that when a widget was clicked, the default field in the property editor should be selected to reduce the need for having to click on it.

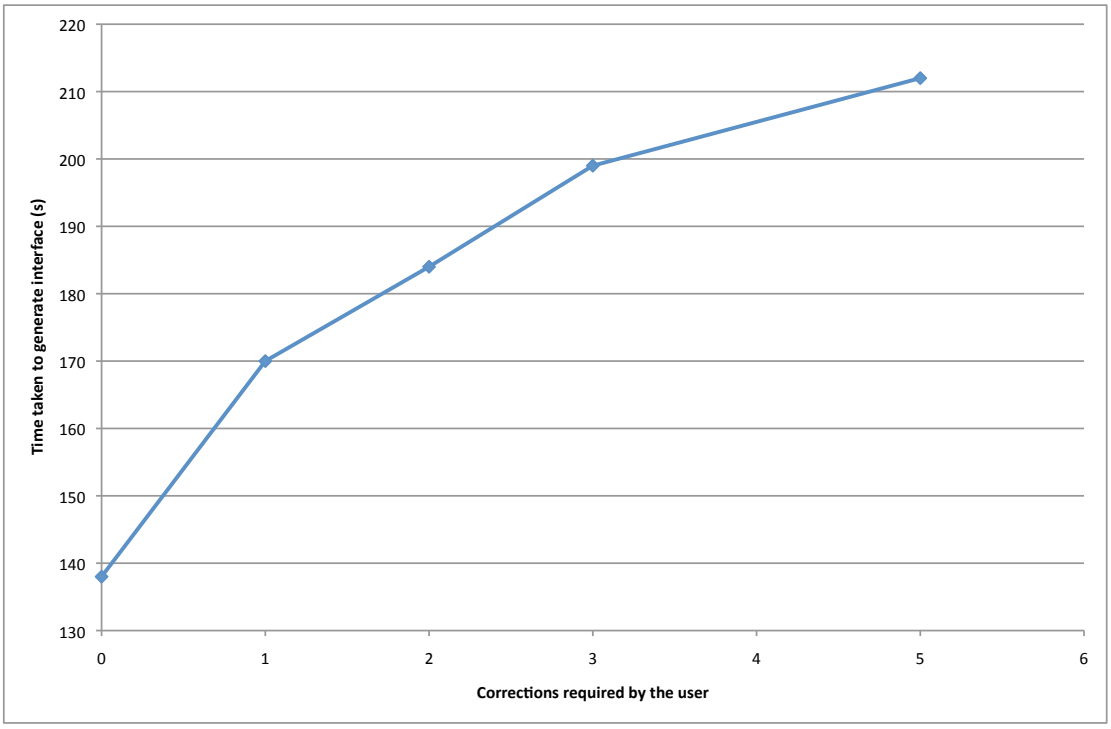


Figure 8.21: Relationship between time taken to complete the task and number of corrections required by the user.

Questionnaire results

The users were asked to complete the questionnaire after they had finished their user test. Table 8.2 shows the average Likert scale ratings for the usability questionnaire.

+/-	Questions	Average Rating
+	I would use this system again	4.8
-	The system was difficult to use	4.4
+	The stages were easy to navigate	4.2
-	I was not in control of the system	4.4
+	The toolbox was useful and informative	3
-	The toolbox got in the way and didn't provide enough information	3.6
+	Sketchi produces comparable results to good manual coding	4.8
-	I feel Sketchi is harder to use than manually coding	4.8
+	The system was fast	4.6
-	The system was slow and unresponsive	5
+	Mistakes were easy to rectify	3.8
-	Mistakes I made were not easy to fix and I wasn't told about them	4.4
+	I feel Sketchi is easier to use than other GUI builders	3.8
-	I think that drag and drop is more intuitive than drawing on a screen	3.4
+	I could easily tweak the layout when I wanted to	4.2
-	I didn't have enough options when customizing the GUIs	4.2
+	Sketchi classified my sketch correctly	4.8
-	I could not produce the GUI I wanted	4.6
+	The code generated was clean and readable	5
-	The code generated was unusable in an application	4.4

Table 8.2: Average Likert scale ratings acquired from user questionnaires.

The average Likert rating for the questionnaire was 86.2 out of a possible score of 100. The Likert rating gives an indication of the usability of a system in the context of that system. Unfortunately it cannot be compared to other Likert ratings from other software unless the same questionnaire is used. Nonetheless, a high Likert rating indicates that users found the system usable in the context of the questions posed.

Analyzing the results from the questionnaires shows that users do not necessarily feel that sketching on a screen is more intuitive than dragging and dropping controls onto a screen with the average rating only achieving 3.4 out of 5. This is most likely due to the user-base used to test the system as 4 out of 5 users rated themselves intermediate to advanced computer users. More savvy users would be more accustomed to using a mouse and the short time they spent with the software would mean they would not have time to adapt to using a touchscreen over a mouse.

The other low rating is regarding the usefulness of the toolbox with a rating of 3 out of a possible 5. This confirms the observation that users did not notice the toolbox during their use of the tool and this should be made a priority when streamlining the interface. Other comments made by the users reaffirmed our initial observations. An extra comment made was regarding the breadcrumb bar that they felt should have been clickable to navigate through the software. This should be incorporated in the next iteration of the interface.

Word list results

To analyse the data we use a word cloud as suggested in the study by Travis(Travis 2008). We count up the number of occurrences of an adjective and display them in a word cloud. The font size of each word is directly proportional to the number of occurrences of the adjective. The most frequently selected adjectives appear the largest to help understand our users sentiments towards the application. Any negative words can be used in the post-test interview to understand why the users felt as they did. Figure 8.22 shows the word cloud obtained from analysing the results.

Figure 8.22: Word cloud produced from results of the word list test. The larger the font size, the more frequently users selected that adjective. Bracketed numbers show the frequency count.

The word cloud shows that users felt very positively about their experience with Sketchi although two words appear that should be followed up with an interview. The one user who used the word 'frustrating' highlighted the navigation issues already observed. The user was upset with unnecessary clicking and confirmations required to achieve the correct interface. The word 'non-standard' also needs to be clarified as it has both negative and positive connotations. When asked, the user clarified that they intended the word in positive manner. They selected 'non-standard' to describe the new way in which you can create GUIs with Sketchi. They had not seen software like this before and so selected the adjective non-standard.

The outlined user interface suggestions are left as future work and the details can be found in section 9.2.

8.5.2 Nielsen's Usability Heuristics

Figure 8.23 shows a screen from the Sketchi user interface.

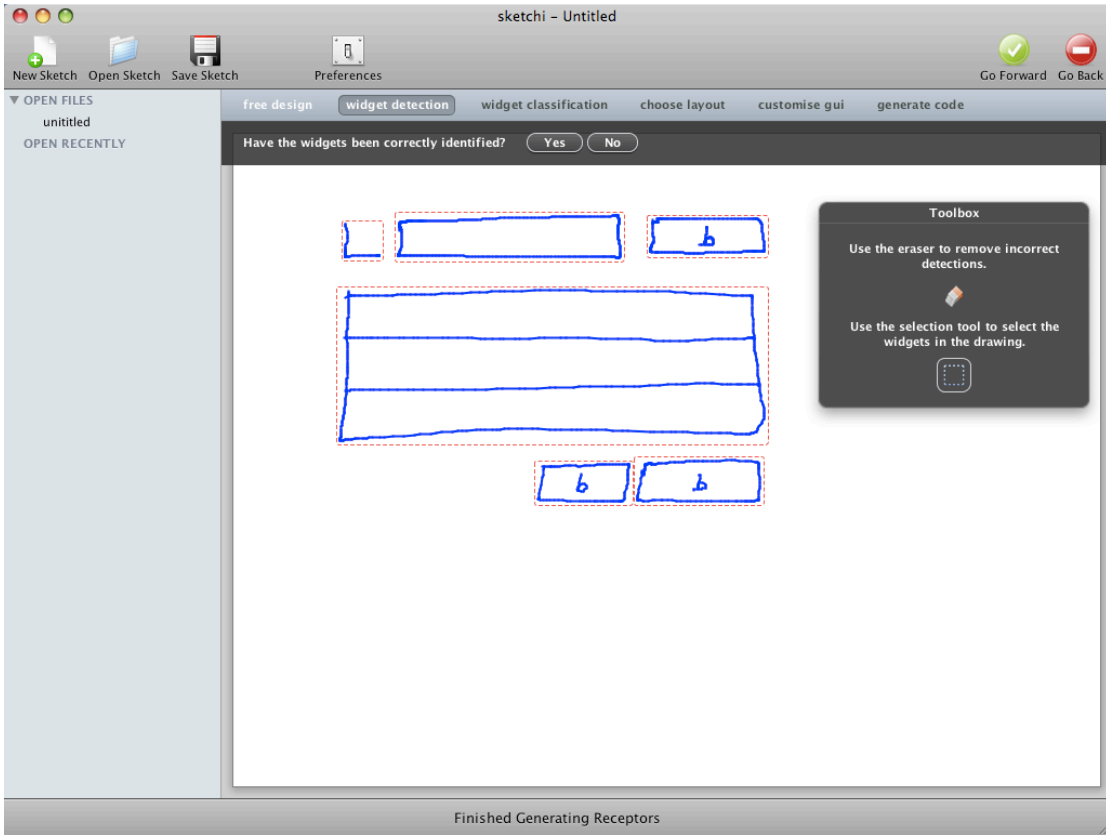


Figure 8.23: The Sketchi user interface complies with Nielsen's 10 Usability Heuristics.

We use to perform a heuristic evaluation using Nielsen's heuristics the results of which are given below:

- *Visibility of system status* – The status of the system is clearly shown at the bottom of the main window in Figure 8.23. The user can see the current state in the blue breadcrumb bar at the top of the content pane.
- *Match between system and real world* – The Sketchi content pane acts as a virtual piece of paper for the user to draw on and the user's sketch is instantly shown onscreen. This mimics a real piece of paper.
- *User control and freedom* – All of the automated tasks carried out by Sketchi can reversed with the back button or corrected using tools from the toolbox.
- *Consistency and standards* – The terminology and style of the application are kept constant across all states of the application with white text on part transparent black background used for notifications and the white background used for creative input from the user.
- *Error prevention* – Validation occurs in every field that accepts input from the user and consistent, relevant validation errors are displayed.
- *Recognition rather than recall* – Information and instructions are clearly visible to the user in every stage of the application. All available tools are displayed to the user without overloading the user with information.
- *Flexibility and efficiency of use* – The nature of the application provides an efficient way to create GUIs with good, clean code as a final product. The architecture of the

application (see section 3.3.1) allows new modules to be swapped in to replace existing modules to provide access to new functionality

- *Aesthetic and minimalist design* – Sketchi is designed with an aesthetically pleasing and minimalistic design. No extra information is displayed so that vital information is not competing for visibility in our dialogs with non-vital information.
- *Help users recognise, diagnose and recover from errors* – Using the notification panel, we confirm the user is happy with their results and if not, then we provide the tools necessary to correct the problems.
- *Help and documentation* – A user guide is supplied with the application and the users are guided on how to train the system if they wish.

The heuristic evaluation carried out above suggests that the Sketchi interface was designed in compliance with Nielsen's Usability Heuristics. The word cloud in Figure 8.22 would suggest that this claim is true as the majority of our users used the word 'usable' to describe Sketchi.

8.6 Performance Testing

The results of the four performance tests outlined in section 7.6 are discussed below.

Receptor Generation

Figure 8.24 shows a comparison of the times taken to generate various size sets of receptors using both random receptor generation and entropy receptor generation.

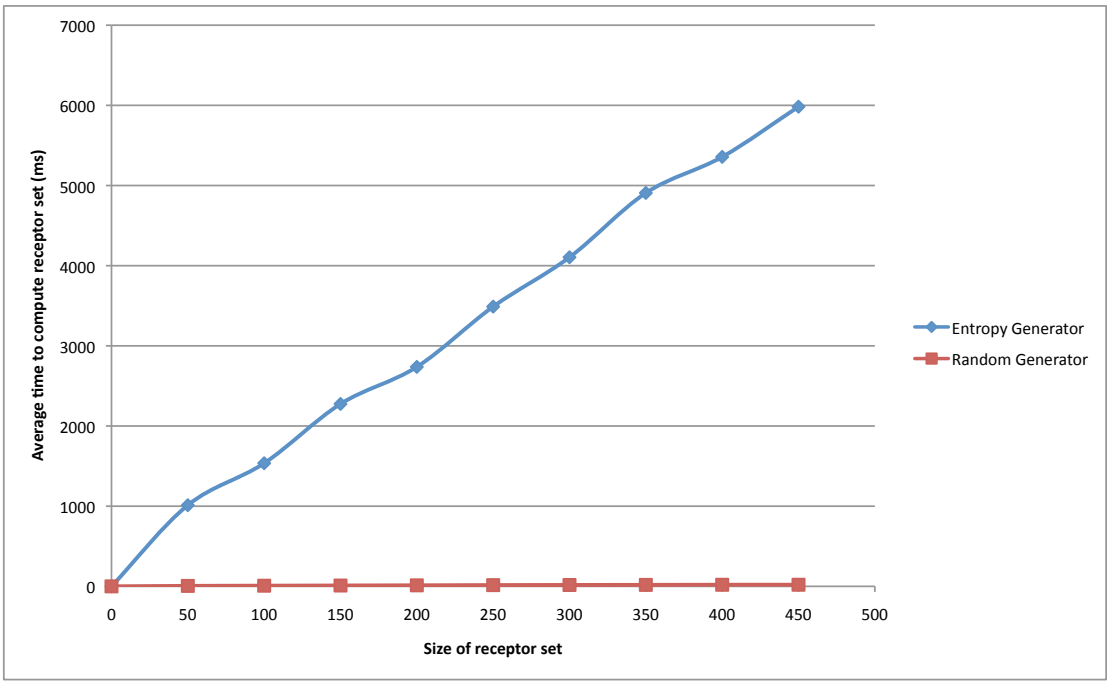


Figure 8.24: Receptor generation comparison. The time to generate sets of receptors using random receptor generation is negligible when compared to entropy receptor generation.

It can be seen that random receptor generation takes a negligible amount of time compared to entropy receptor generation. This is an expected result since the random technique performs no checking of the quality of receptors. The random receptor generator can produce over 15000 receptors per second whereas the entropy-based generator can only generate approximately 70 good receptors per second.

Figure 8.25 compares the quality of the receptors generated by the two techniques using test images acquired in the recognition rate testing.

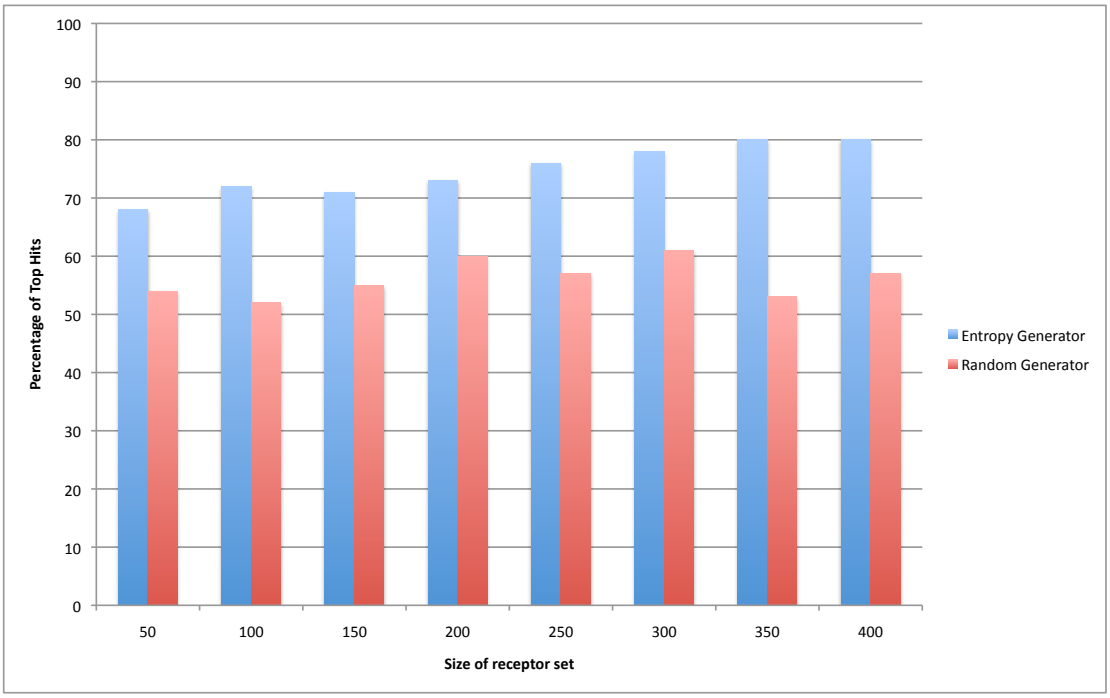


Figure 8.25: Performance comparison between random receptor generation and entropy receptor generation.

The graph shows that the entropy-based receptor generation technique produces consistently better results than the random receptor method. This is an expected result. The entropy technique ensures that the generated receptor is good at distinguishing different objects and good at maintaining the same value across different variants of the same object (see the background section *_* for theory behind entropy technique). The entropy-based generator ensures that the receptors are localised around the distinguishing features of the widget drawing. The converse statement explains the performance for the random receptor generator. As no constraints are placed on the locations of the receptor there is a chance that the receptor is generated where it cannot distinguish between objects and therefore perform misclassifications. Figure *_* visualises this possibility.

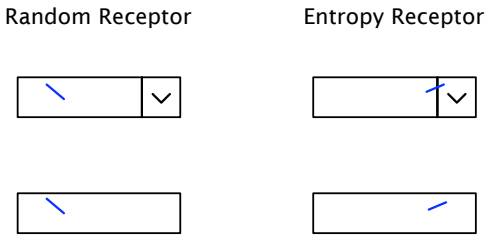


Figure 8.26: Random receptor problem distinguishing two objects.

On left side of figure 8.26 shows that the randomly generated receptor will not be able to distinguish between the two objects. The right side shows the entropy-based receptor is guaranteed to be able to distinguish the objects. Although the entropy-based method takes far longer to generate the set of receptors, the higher accuracy is a feature that is most beneficial in the context of our input classifier. The benefit of the increased classification performance outweighs the detriment of the increased generation time. However, Sketchi is a user-centric application and users will not tolerate long waiting periods. To overcome this problem we

generate a new set of receptors for each new user in the system. These receptors are then persisted to the user's preference file. When the application is launched, the user selects their profile and receptors are instantly loaded into the application with no waiting time experienced by the user. The receptors are not required until the widget classification stage so, when the software is first run by a new user, we can generate the receptors in the background without the user needing to wait.

Widget Classification

Figure 8.27 shows the average time to taken to classify a widget as the number of points in the widget increase.

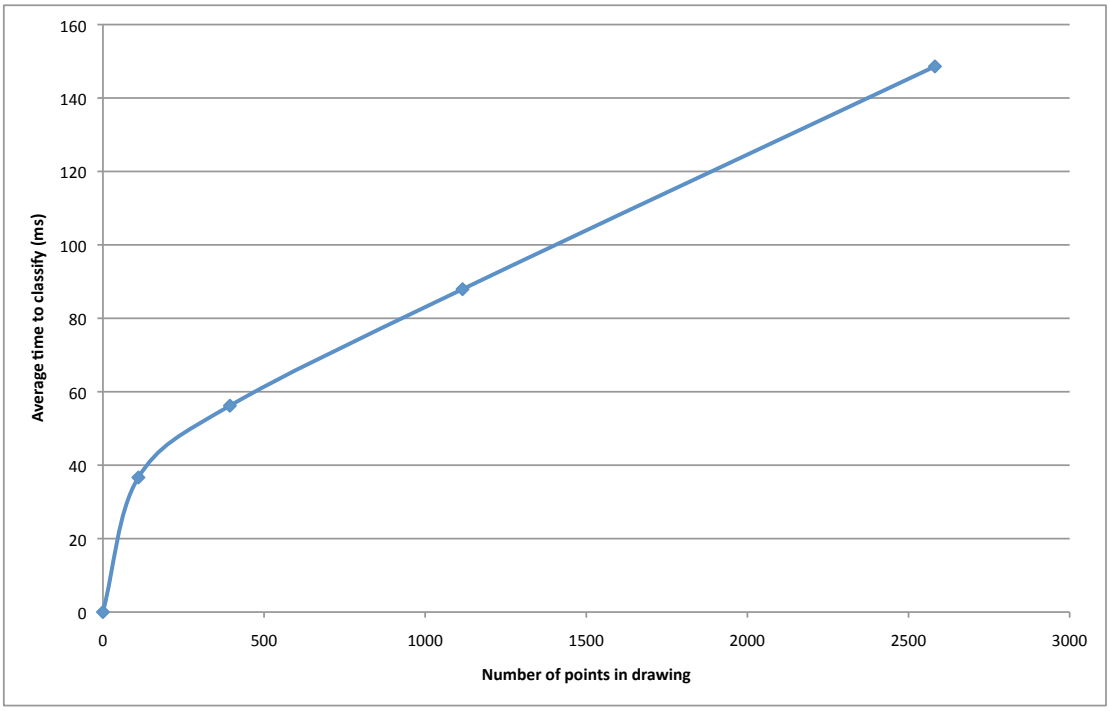


Figure 8.27: Average time to classify a widget as number of points in the input drawing increase. Average time was taken across 10 runs.

The graph shows that the classifier can classify widgets with up to 1400 points within the desired response time of 0.1 seconds. Analysis of the test images collected in the recognition rate testing shows the average number of points in a widget is 519 points. This means our classifier can perform well within the desired response time.

Layout Generation

Figure 8.28 shows the layout generation performance results. The graph shows the time to generate a layout increases at a constant rate as the number of widgets in the layout increases. It also shows that the time taken to generate these layouts is very small. This provides an opportunity to try more advanced techniques to generate layout that may require more time to compute. So long as the time to generate a layout does not exceed the 0.1 seconds threshold it will not be noticeable to the user and we may be able to produce a better layout. This is left to future work and is detailed in section 9.2.

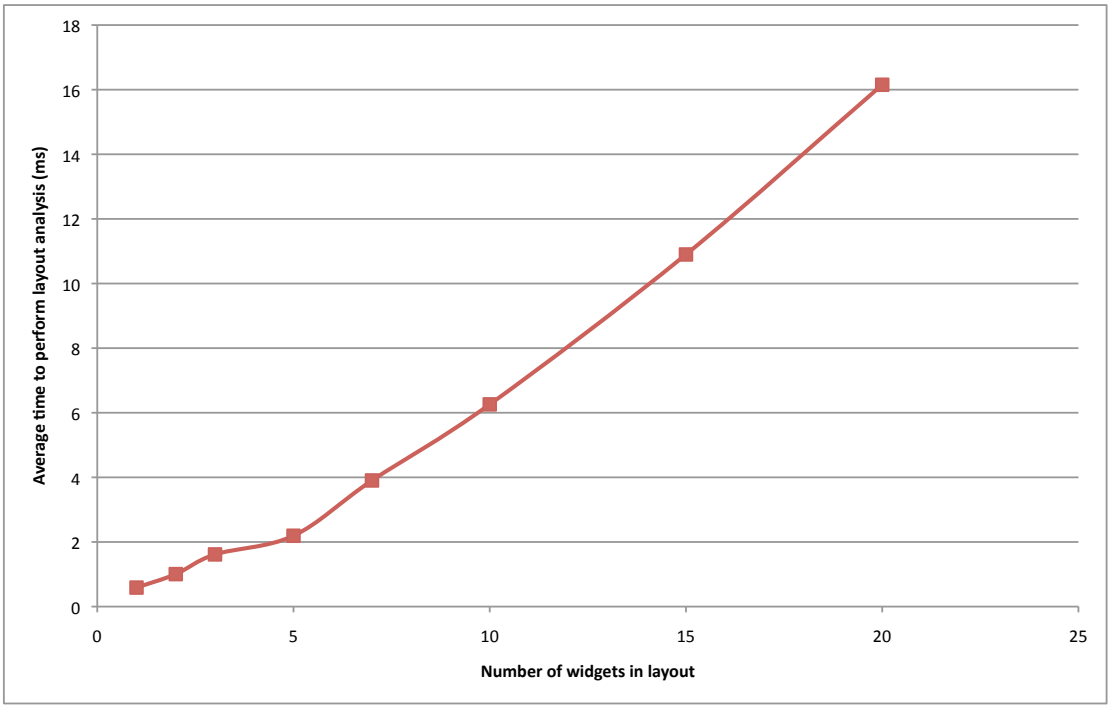


Figure 8.28: Performance graph showing the time taken to generate a layout as more widgets are added to the input.

Code Generation

Figure 8.29 shows the results of the code generation performance test. As the number of widgets in the input model increases so does the time taken to generate the code for the interface. This is explained by the visitor pattern used. The number of visits to each node in the model is constant and this explains the constant rate of increase.

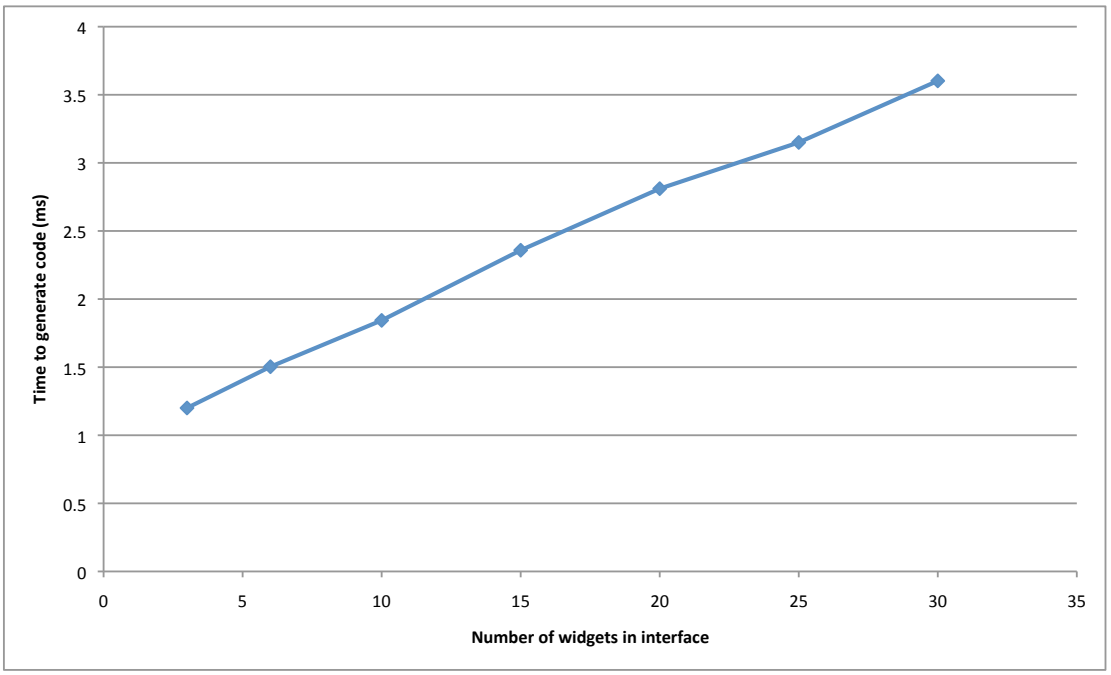


Figure 8.29: Time to generate code increases at a constant rate as the number of widgets in the input model increases.

8.7 Robustness & Stability

8.7.1 Monkey Testing

Table 8.3 shows the results of the monkey testing. The original test plan can be found in table 7.2.

Test No.	Test Result
1	Random input was detected as an object and misclassified as expected. ***No fix required.***
2	Random selections in the detection screen did not lead to any crashes. ***No fix required.***
3	No exception was thrown when a file name did not exist, the request to load the sketch is ignored and the user is not notified of the problem. ***Added notification of the error for the user.***
4	No exception is thrown when running with no preference file. A preference file is created on application start. ***No fix required.***
5	Exception is thrown since Sketchi cannot initialize the input classifier without the external resources. ***Added notification to the user to explain the error.***
6	User is notified in the drawing pane that there is no drawing to classify. ***No fix required.***
7	A null pointer exception is thrown in the layout analysis stage when no widgets are present. ***Exception handled and the layout analysis returns an empty panel as expected.***
8	Test case 7 was fixed before we could test this case. After test case 7 was fixed, code for an empty window was generated as expected. ***No fix required.***
9	Non-numerical characters cannot be entered in any fields that should not contain non-numerical characters. An input handler checks the character being entered and only allows numerical data. ***No fix required.***
10	Excessively large numbers are reverted to the value before editing and a validation message is displayed to the user. ***No fix required.***
11	Class name is validated and validation errors are displayed to the user. ***No fix required.***
12	Folder names are validated and validation errors are displayed to the user. ***No fix required.***
13	Exception is thrown. Sketchi cannot generate code without the code template being necessary. ***Added notification to the user to explain the error.***

Table 8.3: Results of monkey testing.

8.7.2 Stress Testing

The subsequent sections detail the results of the stress testing.

Loading of large images

Sketchi is able to handle large images with no degradation of performance. This can be achieved as Sketchi converts any input image to greyscale before converting it to an internal representation of drawn points. In the case where the image is larger than the drawing pane, the image is placed in the top left corner and cropped.

Drawing a high number of widgets

Drawing an excessive number of widgets on screen degrades the performance of the widget classifier minimally whilst the widget detector is unaffected. A small lag occurs when attempting to classify the widgets on screen. The lag is under 0.4 seconds but it is noticeable

when compared to the standard classification time of under 0.1 seconds. Figure 8.30 shows the drawing from the stress test.

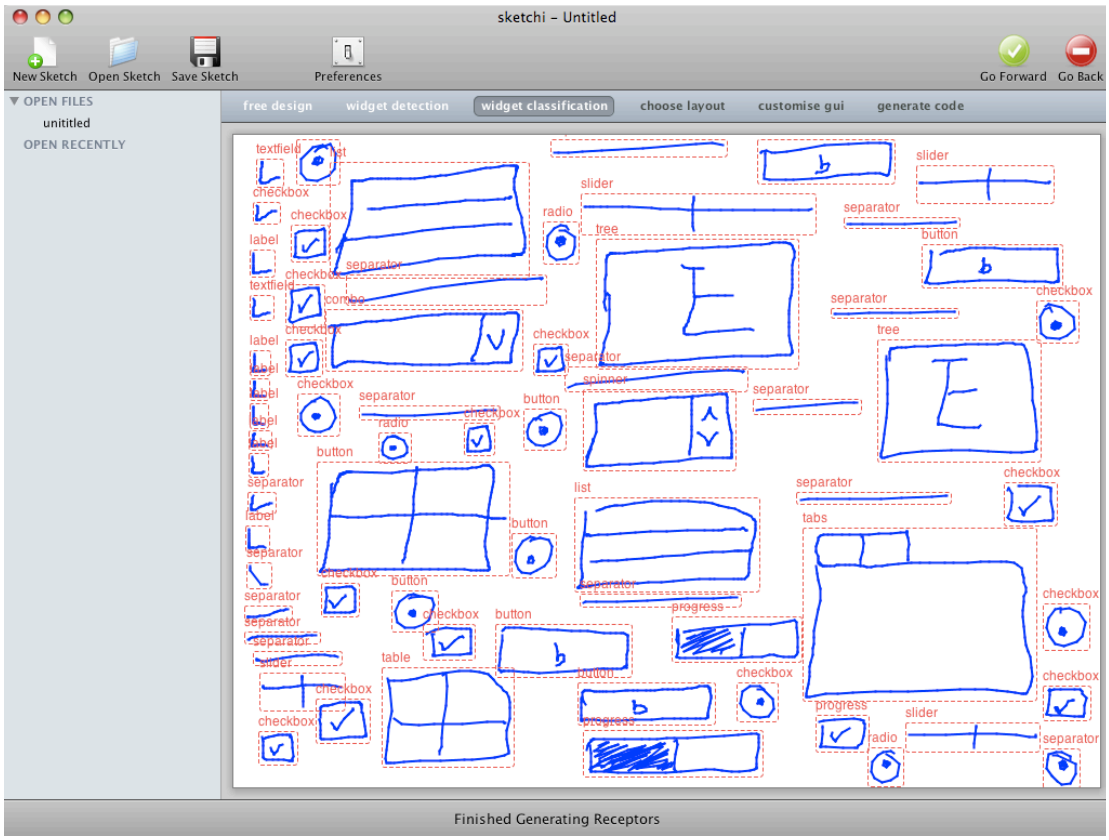


Figure 8.30: Stress testing the widget classifier with 50+ widgets on screen degrades performance minimally.

Generating layouts with a large number of widgets

Passing the sketch in Figure 8.30 into the layout generator produces considerable lag when generating the layout. Upon further investigation, heavy processing occurring on Event Dispatch Thread[i] (EDT) in Swing caused the lag. Performing heavy processing on the EDT makes the user interface unresponsive and the lag issue was solved by delegating the processing to a background thread. The time taken to perform the layout analysis maintained its performance following the constant rate of increase seen in Figure 8.28.

Generating code with a large number of widgets

The code generator performed as expected with the large number of widgets example. It followed the trend seen in Figure 8.29 with no degradation in performance.

[i] Swing uses a single thread to perform all user interface processing. Any other processing on this thread will halt the user interface and the user interface will appear to crash.

8.8 Summary of the Results

Analysis of the test results indicates that Sketchi is a viable tool to intuitively produce GUIs. The CBR based classification system performs well and its predictive ability drastically improves when even a small number of user training cases are added to the case base.

Furthermore, we can see that it greatly reduces the time taken to produce GUIs when compared to manually coding and is comparable to existing GUI builders. However, the code produced by Sketchi is far more readable and concise than code produced by its counterpart GUI builders.

9 Conclusion & Future Work

This chapter deals with our conclusion of the project and puts forward possible directions for future work.

9.1 Conclusion

This project was motivated by a lack of intuitiveness and ease of use in existing GUI building tools. In response, we've created a highly usable, intuitive GUI builder. Below we restate our original aims, as outlined in section 3.1, and show how our project achieves them:

- *Capturing natural input* – We aimed to let a user freely sketch an interface onscreen and have it correctly transformed into a GUI. At the design stage, we chose a Case-Based Reasoning (CBR) system and combined it with a selection of untested feature encodings (see section 3.3.2). The architecture of our CBR allowed new feature encodings to be added easily and after some initial testing it was obvious that more distinguishing features needed to be encoded (see section 4.2.5). We designed an experiment to capture the recognition rates being achieved the classifier (see section 7.1). Analysing these results showed that with only small amounts of classifier training, recognition rates approached 100% for every user tested (see section 8.1).
- *Ease layout management* – We aimed to fully remove the need of manually adjusting layout code. We chose to implement our system on top of the MigLayout layout manager to minimise the amount of layout code generated by Sketchi (see section 3.1). We implemented an algorithm to infer the columns and rows required in the layout and a component analyser to add detailed layout constraints to each component (see section 5). We tested the system by attempting to generate various layouts seen in real applications, including the address book example from John O'Conner's layout manager showdown (see section 2.4.2). Analysis of the results showed that the all except the address book example could be generated with no other user intervention (see section 8.2).
- *Abstract the toolkit* – Abstracting the toolkit is achieved in part by all the modules. The act of drawing an interface rather than writing the code for it completely abstracts the toolkit. To ensure the end result was as abstract as the idea of drawing the GUI, we designed and implemented an abstract code generator that can be extended to generate code in multiple toolkits and languages (see section 6). We provide a Swing implementation of the code generator and leave other generators to be completed as further work (see section 9.2).
- *Generate clean, concise code* – When designing a method of producing concise code, we looked at the problems with existing GUI builder's code generation and attempted to solve their problems. The most obvious problem with the existing tools was the amount of layout code they produced. To combat this, we ensured that we based our code on a layout manager that requires a small amount of layout code. Thus MigLayout was chosen (see section 3.1). To measure the conciseness of the code, we tested the size of code produced by Sketchi with that of NetBeans over a range of interfaces (see section 7.3). The results showed the code being produced was far smaller. We consulted a panel of programmers to investigate which code they felt

was of higher quality. The results were unanimous as every panel member chose Sketchi's code (see section 8.3).

In short, Sketchi was able to satisfy all our initial aims and provide a robust, intuitive solution to the GUI building problem. Most importantly, Sketchi was built with extensibility and flexibility in mind to allow further enhancements and additions at a later stage.

9.2 Future Work

Sketchi has shown that a commercial quality tool can be built to easily turn sketches into full GUI code. Its modular architecture lays the grounds for a wealth of possible future work. Below we detail our ideas for possible future directions:

- *Detecting nested content* – Whilst the current input model supports a nested hierarchy of widgets the application does not utilise it. This is due to the algorithm used by the input detection system (detailed in section 4.1). Our current implementation is not very sophisticated and does not look inside a located widget for other widgets. The problem arises because at the detection stage the type of the widget is unknown. When a possible widget is found, there is no way for the detector to know if it has found a button or a tabbed pane and therefore it cannot decide whether or not to include the points inside the widget's bounds. Figure 9.1 shows the current algorithm attempting to detect nested content. Figure 9.2 shows what a future implementation should hope to achieve. The correct detection method can be achieved by combining the classifier with the detection process. When searching for a widget in the drawing we can attempt to classify it. If the classifier suggests it is a container type widget such as a tabbed pane, we can assume its content will be other widgets and so a hierarchy can be built up.

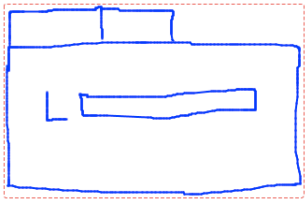


Figure 9.1: Incorrect detection of nested content in a tabbed pane.

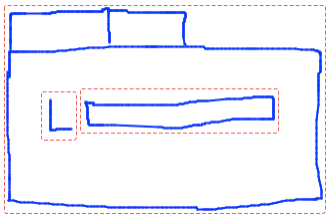


Figure 9.2: Correct detection of nested content in a tabbed pane.

- *Incorporate user interface enhancements* – The test users highlighted several usability flaws in the usability testing. The most prevalent issue was the difficulty users had in navigating through the stages in the application. To resolve this issue, we propose that the navigation functionality be moved from the application toolbar to the toolbox. The notification panel, that caused navigational issues when asking for confirmations can be removed and its functionality incorporated into the toolbox with the navigation. This resolves the other issue of users not recognising the toolbox as a key component in the application flow. Implementing these enhancements will force all navigation to happen through a single point in the application, which should simplify the navigation process. Other small enhancements such as selecting a default field when a widget is clicked in the customization stage, adding instructions to the toolbox in the classification stage and adding a 'Other classifications' label to the combo box in the classification toolbox should be included in the enhancements as well.
- *Investigating other classification techniques* – We chose to use Case-Based Reasoning (CBR) for our input classification and did not experiment with other techniques. We chose CBR as it is the most intuitive method for solving the problem and with thoughtful feature encodings we were able to achieve very good recognition rates. However we did not consider other methods such as using an Artificial Neural Network (ANN). ANNs are used in the field of Optical Character Recognition which is a similar problem to our input classification so could be well suited. Other techniques should be looked at, implemented and compared with our CBR implementation to fully quantify the quality of the current implementation.
- *Comparing with the current state of the art* – The performance of Sketchi was only compared with the GUI builder included in NetBeans due to time constraints. NetBeans was chosen because it is the Swing GUI building solution provided by Sun Microsystems and therefore a good candidate for comparison. However, a better comparison should be made between Sketchi and other GUI builders to truly see where our solution fits into the state of the art.
- *Further code generator implementations* – To enhance the functionality of Sketchi further code generators can be implemented. We provide a Java Swing implementation of the code generator however, using the extensible code generator architecture, it would be simple to provide implementations for other languages and toolkits.
- *Optimising weights in the classifier* – In section 8.1 we analysed the performance of the optimised weights generated by our hill climbing algorithm. Unfortunately, it performed poorly compared to the manually tweaked weights. To achieve good performance with the manual weights, we looked at how the widgets were being misclassified and used our knowledge of how the features were encoded to adjust the weights accordingly. This process is difficult to formalise into code because we cannot represent the intuition required to judge which weight to change and by how much it should be changed. Other optimisation techniques should be looked at in an attempt to achieve better results.
- *IDE integration* – Although Sketchi provides a highly intuitive interface that is ideal for novice users it does not provide any project management control or ability to integrate with existing code bases. A tighter integration with an IDE would let Sketchi integrate with existing code bases and use more advanced code generation techniques provided by modern IDEs. Sketchi's application architecture allows this task to be carried out with minimal modification to core components. The existing content pane can be easily integrated to an IDE and then a new HUB needs to be written to interact with the content pane and its states.

- *More customization options* – In the GUI customization we currently only allow editing of properties on the widgets. To compete with other GUI building tools we should add more functionality such as the ability to add event handlers to widgets. To accomplish this we can add a new editor toolbox to the customization screen to control event handlers. As well as modifying the customization screen, we need to model the event handling in our GUI model and create a new type of visitor to generate the event handler code. The new visitor would visit the new event handling data in the model and generate event-handling code as necessary.
- *Handling input from a scanner or camera* – The current input sketch must be a clean sketch with minimal noise in the image. The Free Design state (see architecture in section 3.3.4) can easily be replaced to allow input from a camera or scanner. An image-processing module would need to be implemented to clean the image or detect the lines in the sketch for further processing by the input detector and classifier.
- *Layout generator enhancement* – To achieve extra accuracy in our generated layouts, a more advanced analysis can be carried out in the layout generator. An issue discovered in section 8.2 shows gap artefacts appearing in the generated layout that are currently unhandled. One possible enhancement is to visualise the generated layout off-screen and compare the generated layout to the drawn image. However, this would be far more computationally expensive and may even lead to a noticeable wait for the user. Other techniques for generating layouts should be explored and experimented with to improve the performance of the layout generator.

9.3 Final Remarks

We have seen how Sketchi can streamline the process of creating GUI code, not just in time taken to produce the GUI but in code size as well. Our receptor pattern implementation combined with our case-based reasoning system, performs excellently in practice and can achieve near perfect recognition with just a small amount of training from the user. Further experimentation and investigation into other classification techniques would provide us with a good benchmark for comparison and it would be interesting to see if our classifier holds up against the competition.

10 BIBLIOGRAPHY

Aamodt, Agnar, and Enric Plaza. "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches." *AI Communications* 7 (1994): 39-59.

Anguish, Scott, Erik M Buck, and Donald A Yacktman. *Cocoa Programming.* Sams Publishing, 2002.

Benedek, Joey, and Trish Miner. "Measuring Desirability: New Methods for evaluating desirability in a usability lab setting." *Microsoft Usability.* Microsoft Corporation. 2002. http://www.microsoft.com/usability/UEPostings/DesirabilityToolkit.doc.

Ben-Kiki, Oren, Clark Evans, and Ingy döt Net. "YAML Ain't Markup Language Version 1.2." *YAML.org.* 11 05 2008. http://yaml.org/spec/1.2/.

Card, Stuart K, George G Robertson, and Jock D Mackinlay. "The information visualizer, an information workspace." *Conference on Human Factors in Computing Systems.* New York: ACM, 1991. 181-188.

Colton, Simon. "Multi-Layer Artificial Neural Networks." *Simon Colton - Department of Computing.* 2004. http://www.doc.ic.ac.uk/~sgc/teaching/v231/lecture13.html (accessed 01 15, 2009).

Cronbach, Lee J. "Response sets and test validity." *Educational and Psychological Measurements 6*, 1946: 475-494.

Evans Data Corporation. "North American Technology Trends." 2005.

FormDev Software. *JFormDesigner - Company.* http://www.jformdesigner.com/company/ (accessed 01 11, 2009).

Grev, Mikael. "Personal Communication." 17 12 2008.

Heaton Research. "About Encog." *Heaton Research.* http://www.heatonresearch.com/encog (accessed 01 13, 2009).

Kirillov, Andrew. "CodeProject: Neural Network OCR." *CodeProject.* 11 04 2005. http://www.codeproject.com/KB/cs/neural_network_ocr.aspx (accessed 01 12, 2009).

Microsoft. "The GUI versus the Command Line: Which is better?" *Technet.com.* 12 03 2007. http://blogs.technet.com/mscom/archive/2007/03/12/the-gui-versus-the-command-line-which-is-better-part-1.aspx.

Miller, Robert B. "Response time in man-computer conversational transactions." *AFIPS Joint Computer Conferences.* New York: ACM, 1968. 267-277.

Mono Project. “Mono 2.0 Release Details.” *The Mono Project.* 06 10 2008. http://www.mono-project.com/Release_Notes_Mono_2.0.

Mozilla. “The Joy of XUL.” *XUL Development.* 09 09 2007. https://developer.mozilla.org/en/The_Joy_of_XUL.

Mullet, Kevin, and Darrell Sano. *Designing Visual Interfaces: Communication Oriented Techniques.* Prentice Hall PTR, 1995.

NetBeans. *NetBeans 5.0 GUI Building Resources.* http://www.netbeans.org/kb/articles/matisse.html (accessed 01 11, 2009).

Nielsen, Jakob. “Heuristic Evaluation.” In *Usability Inspection Methods*, by Jakob Nielsen and Robert L Mack. New York: John Wiley & Sons, 1994.

Nielson, Jakob. *Usability Engineering.* Engineering Press, 1994.

—. “Why You Only Need to Test With 5 Users.” *useit.com: usable information technology.* 19 March 2000. http://www.useit.com/alertbox/20000319.html.

Nielson, Jakob, and Thomas K Landauer. “A mathematical model of the finding of usability problems.” *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems.* New York: ACM, 1993. 206-213.

Pantic, Maja. “Introduction to Machine Learning & Case-Based Reasoning.” *Department of Computing - Home page of Maja Pantic.* https://www.doc.ic.ac.uk/~maja/syllabus-CBR.pdf (accessed 06 10, 2009).

Resnick, Mitchel, et al. “Design Principles for Tools to Support Creative Thinking.” 10 2005: 3.

Shum, Kenneth. “PlanetMath - Shannon's Entropy.” *PlanetMath.* 28 06 2006. http://planetmath.org/?op=getobj&from=objects&id=968 (accessed 01 15, 2009).

Sun Microsystems. “How to use GroupLayout.” *The Java Tutorials.* http://java.sun.com/docs/books/tutorial/uiswing/layout/group.html (accessed 01 14, 2009).

—. “How to use SpringLayout.” *The Java Tutorials.* http://java.sun.com/docs/books/tutorial/uiswing/layout/spring.html (accessed 01 14, 2009).

—. “J2SE 1.2 Press Release.” *Sun delivers next version of the java platform.* 8 12 1998. http://web.archive.org/web/20070816170028/http://www.sun.com/smi/Press/sunflash/1998-12/sunflash.981208.9.xml.

Travis, David. “Measuring satisfaction: Beyond the usability questionnaire.” *User Focus.* 3 March 2008. http://www.userfocus.co.uk/articles/satisfaction.html (accessed June 12, 2009).

UIML.org. *UIML.org - Home.* http://www.uiml.org/ (accessed 01 12, 2009).

11 Appendix

11.1 Swing Code Template

Figure 11.1 shows the code template used by the Swing code generator.

```
import net.miginfocom.swing.MigLayout;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

$imports

public class $filename extends JFrame
{
	public $filename()
	{
		JPanel p = createGUI();

		this.setContentPane(p);
		this.pack();
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);
		this.setLocationRelativeTo(null);
		this.setVisible(true);
	}

	/**
	 * Creates the GUI controls and adds the layout code
	 */
	private JPanel createGUI()
	{
		$guicode
	}

	/**
	 * Standard static void main that creates the JFrame
	 *
	 * @param args
	 */
	public static void main(String args[])
	{
		SwingUtilities.invokeLater(new Runnable()
		{
			@Override
			public void run()
			{
				JFrame mainFrame = new $filename();
				mainFrame.setVisible(true);
			}
		});
	}
}
```

Figure 11.1: Code template for Swing code generator.

11.2 User Questionnaire

	Questions
1	I would use this system again
2	The system was difficult to use
3	The stages were easy to navigate
4	I was not in control of the system
5	The toolbox was useful and informative
6	The toolbox got in the way and didn't provide enough information
7	Sketchi produces comparable results to good manual coding
8	I feel Sketchi is harder to use than manually coding
9	The system was fast
10	The system was slow and unresponsive
11	Mistakes were easy to rectify
12	Mistakes I made were not easy to fix and I wasn't told about them
13	I feel Sketchi is easier to use than other GUI builders
14	I think that drag and drop is more intuitive than drawing on a screen
15	I could easily tweak the layout when I wanted to
16	I didn't have enough options when customizing the GUIs
17	Sketchi classified my sketch correctly
18	I could not produce the GUI I wanted
19	The code generated was clean and readable
20	The code generated was unusable in an application

Table 11.1: Likert scale questionnaire give to test users.

	Strongly Agree	Agree	Neither	Disagree	Strongly Disagree
Positive	5	4	3	2	1
Negative	1	2	3	4	5

Table 11.2: Likert scale ratings.