Pattern based user interface generation

# How user interfaces are built

There are several techniques and several tools to build user interfaces. Some are more intuitive and easy to learn while others are more flexible but harder to learn and thus more time consuming.

In this chapter I’ll be focusing on some of the most used techniques for building user interfaces. I’ll try to explain what are the main advantages and disadvantages related to each technique while using some examples to justify them.

Probably the most used technique is also the oldest one, manually coding interfaces. It’s hard and time consuming but it’s usually preferred by most experienced developers because it’s more flexible and if they’re good at what they’re doing the final code can be very good and maintainable.

The second technique we’ll see in this chapter is code generation through WYSIWYG [[1]](#footnote-1) tools. There are many tools of this kind that support the most popular languages and frameworks for developing user interfaces. They’re very used mainly by novice developers and designers. The final code isn’t always the best but if you’re using a robust tool there’s little chance of finding bugs in it.

The third and last technique I’ll talk about isn’t the most popular in the industry but there’s a lot of work surrounding it in the academic world. Model driven development is widely used for the bottom layers in software development but is not that popular for the presentation layer. Although this technique is not as widely used as the previous ones it brings many advantages such as platform independence.

Manually coding user interfaces

Before there were more advanced tools user interfaces were coded manually, like everything else. Nowadays although we have these tools, most developers still think this is the best way because it gives them more control over their work.

This is a very time consuming technique because humans have to do most of the work but in the end it really depends on what language or framework you’re working on. Most popular and modern programming languages give developers access to frameworks for building GUI’s[[2]](#footnote-2) like GTK+, Swing or Windows Forms. These examples are for the desktop side. On the web side everything is (X)HTML, CSS and JavaScript but there are a lot of frameworks to abstract from these languages like JSF[[3]](#footnote-3), Struts or ASP.net.

Most desktop GUI frameworks use the same language for views and the other software layers. This means that a lot of code has to be written In order to get things done. Frameworks like GTK+, Swing and Windows forms are very hard to use without help from more advanced tools.

Let’s take a look at simple Swing example that shows a basic login window.

*public static void main(String[] args) {*

*SwingManualTest sm = new SwingManualTest();*

*sm.showLoginWindow();*

*}*

*private void showLoginWindow(){*

*Container c = getContentPane();*

*c.setLayout(new GridLayout(3, 2));*

*c.add(new JLabel("Username:"));*

*c.add(new JTextField());*

*c.add(new JLabel("Password:"));*

*c.add(new JTextField());*

*c.add(new JButton("Login"));*

*c.add(new JButton("Cancel"));*

*setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);*

*pack();*

*setVisible(true);*

*}*

It’s plain Java so every control is an object. For some OOP[[4]](#footnote-4) enthusiasts this is a good thing but it’s incomprehensible for designers and even for most developers this is very hard and thus very time consuming.

Fortunately, on the web side things are simpler. Most frameworks use HTML with some specific extensions to specify the views. This offer developers a more declarative paradigm which makes a lot more sense when building interfaces. This approach also produces a lot less code which makes maintenance a lot easier.

Let’s take a look at an example similar to the previous one but this time using JSF.

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:h="http://java.sun.com/jsf/html">

<h:head>

<title>Login</title>

</h:head>

<h:body>

<h:form>

<h:outputLabel value="Username:" />

<h:inputText />

<h:outputLabel value="Password:" />

<h:inputSecret />

<h:button value="Login" />

<h:button value="Cancel" />

</h:form>

</h:body>

</html>

This is very different than the first example. It’s not just more intuitive for developers, it’s a little bit more understandable for designers too because it’s based on HTML.

Recently some new frameworks for desktop GUI’s, which resemble the web ones that were referenced earlier, have been developed. One good example is the WPF[[5]](#footnote-5) framework. It uses the XAML[[6]](#footnote-6) language to specify views. It’s a markup language based on XML[[7]](#footnote-7) and, thus, more like HTML. Let’s take a look at the login window coded for WPF.

<Window x:Class="WpfApplication1.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Login Window" >

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<TextBlock Text="Username:" Grid.Column="0" Grid.Row="0" />

<TextBox Grid.Column="1" Grid.Row="0" Width="150" />

<TextBlock Text="Password:" Grid.Column="0" Grid.Row="1" />

<TextBox Grid.Column="1" Grid.Row="1" Width="150" />

<Button Grid.Column="0" Grid.Row="2" Width="150">Login</Button>

<Button Grid.Column="1" Grid.Row="2" Width="150">Cancel</Button>

</Grid>

</Window>

Even though this language is a lot more verbose than HTML and other markup languages it’s a very good alternative for building desktop GUI’s, especially if you’re going to write all the code manually.

The conclusion of this section is that manually coding user interfaces isn’t always a good idea depending on the technology you’re using. The first frameworks that were presented use programming languages to specify the views. That doesn’t look like a very good approach because it’s not intuitive for the developer and incomprehensible for designers. On the other hand the later solutions use specific languages for specifying views which are more intuitive and easy to write but they oblige developers to learn these new languages.

Code generation through WYSIWYG tools

The concept of WYSIWYG is used in a variety of situations. From text processing to building user interfaces. One of the most recognized tools of this kind is Microsoft Word for text processing. What tools of this kind attempt to do is offer the user an interface that shows exactly the final result of what they’re doing.

In software development the most popular WYSIWYG environments are the ones provided by Java IDE’s like *Netbeans* to build Swing interfaces or Microsoft Visual Studio that provides WYSIWYG tools for a variety of frameworks like Windows Forms, WPF or ASP.net. Let’s take a closer look to *Netbeans.*

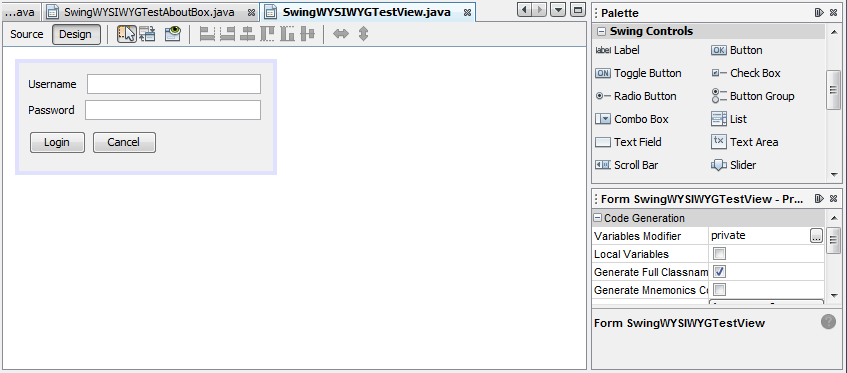


Ilustração Netbeans Swing WYSIWYG tool

Like you would expect from this tool, it has a canvas where the final GUI appears and a side menu from where you can drag controls and drop them into the canvas. It’s very simple and intuitive and thus very attractive for novices. The problem with this kind of tools is maintainability. It’s very easy and quick to build something, if it’s not very advanced, but it´s a real challenge when there’s a need to change the layout. Making a manual change in the generated code is not an option, mainly because is too complex but also because it’s often blocked by the IDE itself. The other tools are very similar so there is no need to give further examples.

In conclusion, WYSIWYG tools are good for novices but don’t suppress all the needs of the software industry where maintainability is a very important issue. There’s also the problem of portability, the produced code is platform specific. Other important issue is reusability. GUI’s frameworks usually offer some way to reuse components in different contexts. This is can be easily achieved while manually coding everything but it’s a lot harder with a higher level of abstraction.

Model driven development of user interfaces

Model driven development defining characteristic is that software development’s primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages (Quote The Pragmatics of Model-Driven Development).

Models are easier to maintain than the code itself and, most important, they’re platform independent. This means that the same model can be used to generate code that runs on a desktop environment, a web environment or even a mobile environment. This makes a lot of sense for user interfaces because modern applications are becoming more and more ubiquitous and it’s highly complex and time consuming to build a GUI for every supported platform.

UML is the industry standard for software modeling but, unfortunately, is not fit to model user interfaces. With this in mind, the software engineering community has developed some new modeling languages in the past few years to overcome this problem. The most relevant are probably UMLi[[8]](#footnote-8), an extension to UML and CTT[[9]](#footnote-9) which aims task modeling.

UMLi provides an alternative diagram notation for describing abstract interaction objects (Quote User Interface Modeling in UMLi).

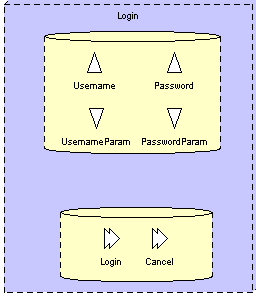


Ilustração 2 Login window modeled in UMLi

With this notation you can specify inputs, outputs and actions in a way that classic UML notation doesn’t support. Tasks can also be specified in UMLi, but without any extension to UML. Tasks can be modeled using Use Cases and Activity Diagrams which are part of the UML specification.

Task modeling has become one of the cornerstones of model-based user interface design (Quote IDEALXML: AN INTERACTION DESIGN TOOL). (Falar em CTT)

# Patterns in software engineering

# How to specify user interface patterns

1. What you see is what you get. [↑](#footnote-ref-1)
2. Graphical user interface. [↑](#footnote-ref-2)
3. Java server faces. [↑](#footnote-ref-3)
4. Object oriented programming. [↑](#footnote-ref-4)
5. Windows Presentation Foundation. [↑](#footnote-ref-5)
6. Extensible Application Markup Language. [↑](#footnote-ref-6)
7. Extensible Markup Language. [↑](#footnote-ref-7)
8. Unified Modeling Language for Interactive Applications [↑](#footnote-ref-8)
9. Concur Task Trees [↑](#footnote-ref-9)