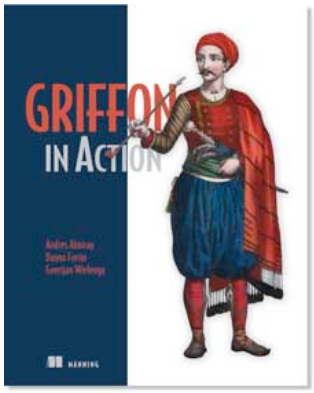
MANNING PUBLICATIONS

Green Paper From Griffon in Action



EARLY ACCESS EDITION

Welcome to the Griffon Revolution

Andres Almiray, Danno Ferrin, and Geertjan Wielenga

MEAP Release: March 2009
Softbound print: December 2009 (est.) | 375 pages
ISBN: 97819351822381

*This article is based on chapter 1 from **Griffon in Action** by Andres Almiray, Danno Ferrin, and Geertjan Wielenga.*

Welcome to a revolution in how regular desktop applications are designed, developed and maintained. There are many steps to be covered to get an application out of the door. Many hurdles and obstacles lurking in your path as well, waiting their turn to make you slip that important deadline or drive you to frustration in no time.

Griffon is a revolutionary solution that can make your job easier, while bringing back the fun to doing your job and then some.

1.1 What is Griffon?

The short answer would be: a framework for developing desktop applications for the JVM with high productivity gains. No, we didn't make a mistake; we do mean high productivity gains. While developing desktop applications on the Java platform is attainable, the trick is to neutralize the common pain points you usually face when embarking in such a task. Those pain points are:

1. Essence versus ceremony
2. UI definition complexity
3. Application structure
4. Application life cycle management
5. Build management

Let's review each one of these paint points carefully.

1.1.1 Essence vs. ceremony

The Java platform is a great place to develop applications, as witnessed by the myriad of libraries, frameworks and enterprise solutions that rely on it. It is also a wonderful host to several programming languages, of which the Java programming language is the first and the most widely used so far. Unfortunately the language is showing its age, since introduced in 1995.

The Java programming language was a refreshing change when it was first introduced. Developers around the world were able to pick up the language and "jump ship", so to speak, in a matter of weeks. The language's syntax and features were similar to what developers were used to programming with, while at the same time Java included new and desired features baked right into the language, like threading concerns and the notion that the

network should be a first class citizen. In other words everybody marveled at it. That is perhaps why there were few complaints about the amount of code it took to perform a simple task, like printing a sequence of characters to the console. Listing 1.1 is a very descriptive example of this. In fact it is the often-used HelloWorld example:

Listing 1.1 The typical HelloWorld example in Java

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!")                                    1
  }
}
```

Cueball in code and text

1

Don't get us wrong, this example is much better than what was previously available, but there is still room for improvement. Imagine for a moment that you know little about the Java language (but if that is really the case, don't worry we'll explain what is going on with that snippet of code). The task that we like to accomplish is pretty much described by (#1), but as you can see there are a few additional things you must type to please the compiler.

Every piece of code you write on the Java language must be tied to a class, as it is an object oriented language that uses classes to define what object can do[1], so we must define a HelloWorld class. A class may define a method with a special signature (the main method) that will be used as the entry point of your program, so we define it as well. Inside that method we place the code that actually fulfills the task. How would you explain a person new to the Java language that she must know all this things (and a few more like access modifiers and static vs. instance class members) just to print a message to the console? All of this just to please the compiler in order to make a simple example. Wouldn't it be easier if the compiler accepted something like the following?

Listing 1.2 A simplified HelloWorld example

```
println "Hello World!"
```

If listing 1.2 makes more sense than listing 1.1 it is because the essence of the task has been made explicit, there are neither additional keywords nor syntax constructs that distract you from understanding what the code does. This is exactly what we mean by *essence vs. ceremony*, a term Neal Ford[2] makes sure to mention quite regularly.

Java is a good language to develop applications, but it requires a steeper learning curve as opposed to other languages that are able to run on the JVM, languages that provide the same behavior in many respects, but in a more expressive manner.

1.1.2 UI definition complexity

Strongly related to the previous point we found that defining an UI can be overly complex and verbose. The Java Standard Library, which comes bundled with the Java language when you download the JDK (Java Development Kit) or JRE (Java Runtime Environment), delivers two windowing toolkits: the AWT (Abstract Windowing Toolkit) and Swing. Of which Swing is the one with the most wide spread use, as it is highly configurable and extensible. But those benefits come with a price, you have to write a lot of code to get a decently looking UI to work, you have to deal with many collaborators and helpers in order to react to events and provide feedback, on top of it you have to please the compiler again. Let's review the following example of a basic application that copies the value of a text widget into another when you press a button, in straight Java

Listing 1.3 Basic Java Swing application

[1] as opposed to Javascript that uses prototypes to accomplish the same feat
[2] You can read Neal's thoughts at http://memeagora.blogspot.com/

```
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.SwingUtilities;

public class JavaFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable(){
      public void run() {
        JFrame frame = buildUI();
        frame.setVisible(true);
      }
    });
  }

  private static JFrame buildUI() {
    JFrame frame = new JFrame("JavaFrame");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(
      new GridLayout(3,1)
    );
    final JTextField input = new JTextField(20);
    final JTextField output = new JTextField(20);
    output.setEditable(false);
    JButton button = new JButton("Click me!");
    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
        output.setText(input.getText());
      }
    });
    frame.getContentPane().add(input);
    frame.getContentPane().add(button);
    frame.getContentPane().add(output);
    frame.pack();
    return frame;
  }
}
```

There have been several attempts on simplifying how UIs are created in Java, to the point of externalizing them in a different format, most of the times the chosen format is XML. For some strange reason Java developers have a very strong love/hate relationship with XML since the early days. Whenever a configuration challenge appears or the need to externalize an aspect of an application arises XML is the choice that comes to mind 99% of the time. The problem with XML is that it is so easy to hurt yourself with it[3], once you go down the path of declarative UI programming with XML you will eventually find yourself in a world of pain.

In listing 1.3 you can appreciate that the widgets are instantiated and configured with additional properties, they are also added to a parent container according to the rules of a predefined layout. Behavior is wired in a convenient way, well as convenient as it is defining an anonymous inner class as the event handler on the button, which is a common pattern in Swing applications[4]. If you were to define the UI in XML format you would need to perform at least the following tasks:

- read the XML definition by means of SAX, DOM or your own parser.
- translate the declarative definitions into widgets instances.
- wire the behavior, perhaps by following a particular configuration path or a predefined convention.

Of course you'll have to deal with exceptions, classpath configuration and additional setup, in other words make sure you have your favorite debugger close by, we can guarantee you'll be reaching for it before the job is done.

[3] hey, it has these sharp, pointy things < > or didn't you notice? stay way from sharp edges!

[4] anonymous inner classes, along with the other 3 types of inner classes, belong to the advanced concepts that people new to Java tend to stay away from, go figure.

Now compare listing 1.3 to listing 1.4, which removes most of the cruft and verbosity while retaining the same essence

Listing 1.4 A simplified Swing application

```
import groovy.swing.SwingBuilder
import static javax.swing.JFrame.EXIT_ON_CLOSE

new SwingBuilder().edt {
  frame(title: "GroovyFrame", pack: true, visible: true,
        defaultCloseOperation: EXIT_ON_CLOSE) {
    gridLayout(cols: 1, rows: 3)
    textField(id: "input", columns: 20)
    button("Click me!", actionPerformed: {
      output.text = input.text
    })
    textField(id: "output", columns: 20, editable: false)
  }
}
```

It may be hard to believe at first but both listings provide the same behavior, the advantages of listing 1.4 should be apparent to the naked eye: we have shaved more than half of lines of code, the relationship between widgets and container is more explicit (they all are contained in a block that is defined inside the container), the event handler's code has been reduced to its minimal essence, and some values (like rows: and columns:) now have a sensible meaning.

1.1.3 Application structure

Every application requires some structure, otherwise it would be a complete maintenance nightmare to say the least. Applications were being built since the Java platform was born and for a while everybody followed their best criteria to build them. That is until Struts[5] came into the web application development scene. Almost of all of a sudden people realized that a predefined structure which everybody agreed upon was a good idea, not only was it possible to recognize the function of a particular component by its place and naming convention, but also you could switch from one Struts application to another and reap the benefits of knowing the basic structure, you could be productive from the get go.

As web application development gained terrain over desktop application development, more and more frameworks were created, to the point where you have a plethora of options to choose from, all of them with pros and cons. Glancing back to the desktop arena there are not so many options, we hardly had the explosion of framework development triggered by Struts. Regardless, many still longed for a desktop version of the Struts framework to appear.

1.1.4 Application life cycle management

Tied to a particular structure, an application should be able to manage all aspects of its life cycle, what to do to bootstrap itself, initialize its components (perhaps by type, layer or responsibility), allocate resources, bind all event handlers, just to name a few common tasks.

It would be very easy to get lost in the details, it should be more convenient to let a framework resolve those matters for you, if only a framework like Struts appeared, in the sense that made people think on a good set of options for developing desktop applications and agree on them.

1.1.5 Build management

Finally, there is the matter of building the application in a reliable way, while taking care of proper dependency management, version control and infrastructure upgrades. You should also keep an eye on IDE integration and reporting tools, even perhaps continuous integration environments if you're into agile development.

Again the Java platform offers a good number of tools for project management, dependency management, IDE integration and so on, the problem is choosing the ones that solve a particular problem in a good way while also

[5] http://struts.apache.org/

being able to integrate with other tools, oh and yes, being extensible enough. The question is not if the tool is extensible or not, but how soon would you require to extend the tool, sooner than later you will face that decision.

Now that we have explained what are the usual pain points let's discuss what Griffon brings to the table.

1.1.6 The Griffon alternative

First and foremost the application must have a well defined kernel as structure, something that ties all components given their responsibilities and their relationships with one another, for which Griffon developers decided to pick the popular Model-View-Controller pattern (or MVC for short) as the basis. This is the framework's shaping element.

The next step is finding the proper balance between configuration and expected conditions or conventions, we're sure you agree you would like to spend more time in pushing code to production than figuring out the proper configuration flags and properties to get a particular piece of the application working. Along with common components and tools this would be the framework's constituent element.

Lastly there is the matter of the amount of verbosity that the Java language imposes, we would like you to be productive without needing to learn a new language altogether nor leaving all your Java knowledge behind, this is where the Groovy language comes in, as a binding agent between the shaping and constituent elements of the framework.

Let's see how each of these options combine together to offer you a better experience while developing a desktop application.

1.2 The convention over configuration paradigm

Configuration is one of the key aspects of any framework, finding the correct amount of configuration is a tricky task. It is well known that you can't please everybody so compromises must be made and boundaries should be defined. To give you and idea let's revisit Struts, a popular web application development framework that came with a high price for quite a while: over-configuration. Every single property of every single component had to be defined on a configuration file, for a small application that perhaps wasn't that bad, but for full range enterprise applications that meant a lot of work, and let's not go into what kind of nightmare it was maintaining such an application.

Imagine for a moment you have a bookstore's domain model, you will most likely encounter Book, Author and Publisher domain objects. Following the MVC design pattern you will surely have a controller for each domain object, say BookController, AuthorController and PublisherController. You will also need at least a view for each domain object, if you're application provides CRUD like operations on your domain it is very likely you'll see 3 different views for each one, perhaps something like BookListView (lists all books), BookShowView (shows one book at a time), and BookEditView (lets you update a book's properties. Setting up this basic application already required a lot of configuration on the Struts config file. Now imagine having a domain comprised of dozens of domain objects, not a happy picture.

Did you notice what we just did? we assumed that the controllers had a particular suffix, perhaps to easily identify them when browsing the application's source code, the views also followed a naming convention. If we take the naming convention a step further we may organize those components by responsibility in well defined file folders, like models, views and controllers. Suddenly you realize that if each component follows these simple rules the previously required heavy configuration is no longer needed, bootstrapping code should be able to figure out how each component must be assembled given these conventions.

That is precisely the power of the convention over configuration paradigm[6], a developer should be required to configure a particular aspect of a component or a set of components only when that configuration deviates from the standard. This means that if you stick to a well-known set of conventions you will be able to create an application faster, as you spend less time configuring it, it is even possible that relationships between components can be managed in this way as well.

[6] http://en.wikipedia.org/wiki/Convention_over_configuration

By leveraging the convention over configuration paradigm Griffon is able to figure out the responsibilities of a component just inspecting its name, place on the directory structure and perhaps some of its properties. Say goodbye to long and painful XML configuration files, search no more for obscure configuration flags, all the configurable information you need to get your application off the ground is located close to the place where it is needed.

Griffon also manages an application's build cycle, by providing a rich set of command line scripts and bindings to the popular Ant project.

Being able to follow a predefined convention requires a rich environment where definitions can be created at the most convenient moment, even if that means at the last possible moment, at runtime. This calls for an environment that accepts a dynamic a highly adaptable solution, something that the Java language cannot provide, but another language can, that language is Groovy.

1.3 Summary

This chapter introduced you to the core concepts behind the Griffon framework and what you can expect from developing an application with it.

The Java platform is a great place to develop desktop applications but it is not without its fair share of traps and obstacles. Griffon avoids those traps and sorts the obstacles by standing on the shoulders of giants, the Grails framework and its community, the Groovy language, well-known design patterns and the convention over configuration paradigm. Together they bring synergy and high productivity gains to desktop development.