JAVA FRAMEWORKS AND COMPONENTS

ACCELERATE YOUR WEB APPLICATION DEVELOPMENT

MICHAEL NASH
JGlobal Ltd.
Freeport, Bahamas

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typefaces Berkeley Oldstyle 10.75/13.5 pt. and Franklin Gothic *System* $\LaTeX\,2_\varepsilon$ [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Nash, Michael, 1964-

Java frameworks and components : accelerate your Web application development / Michael Nash.

p. cm.

Includes bibliographical references and index.

ISBN 0-521-52059-2 (pb.)

1. Java (Computer program language) 2. Application software – Development. I. Title.

QA76.73.J38N355 2003
005.13′3 – dc21 2002041684

ISBN 0 521 52059 2 paperback

Contents

CHAPTER 1

Components and Application Frameworks

1.1 INTRODUCTION

Welcome, I would like to introduce myself, and discuss the explorations that I would like to take you on in this book. I am a software developer, specifically, an application developer. I build software that people use to get their jobs done – practical, everyday software that is being used.

As part of building applications, in some cases I have also had to build some of the tools I needed including components and an application framework. My programming language of choice for the last several years has been Java. My applications have been targeted for Internet/Intranet uses.

In my exploration, I will examine the nature of software development. It is a complex and demanding process, and developers can use all the tools that are available to them to make it easier and more efficient. We also explore how and why developers resist using those very tools, and how you can make the right decisions in your development projects. It has been a long road, and my hope is to impart some knowledge of its many potholes to you.

I think that I have spent a fair portion of my career in the field resisting the "jargonification" of software development. I am convinced that development can be discussed in plain English, as has been done in this book. It is, however, important to understand the basic terminology used to describe frameworks and components, so we will do this first.

We start by examining what is meant by "web applications," "components," and "application frameworks." In a jargon-ridden industry, it is always important to be sure that a common meaning exists for terms. I will define components and frameworks in terms of what they do, so it will be apparent how each is applied to practical application development in the real world. We also discuss the users of these tools, further defining them and see where the components and application frameworks fit into the environment of the standard Java APIs (application programming interface), JavaBeans, and Enterprise JavaBeans (EJBs).

Then we look at components in more detail, and talk about the advantages of component-based development. We also discuss why these advantages make components such a significant part of the future of Web-application development. We examine exactly what the components and frameworks do, and what you can expect from them.

Then we take a brief look at a number of actual examples of frameworks, comparing their similarities and differences, their strengths and weaknesses, and the design patterns they use. We discuss how open source fits into the framework scenario, and why frameworks are a natural fit for this development model.

Finally, we take a look at the future of components and frameworks, and the new technologies they are incorporating. We conclude with a few actual case studies, putting some frameworks through their paces to create a simple Web application.

1.2 What Are They?

What is a Web application? What is a component? What is an application framework? What are they *not*? These are important questions and deserve carefully considered answers.

To-*may*-to, to-*mah*-to. Like so many things in the software industry, ask nine developers for a definition of components, and you will get eighteen opinions. All of them will have some similarities though – and we find the commonality in them.

1.2.1 Web Applications

This is perhaps the easiest of the three terms we aim to define: almost everyone agrees that a Web application is a piece of interactive software that runs on the

Internet or on a corporate Intranet. A Web application is really a specialized case of a traditional client/server application – in this case, the client is a Web browser or some other Internet-enabled device, the “universal client.” So a Web application, simply put, is an application in which the users access business logic via their browser.

The server in a Web application is typically an enhanced Web server: for Java Web applications, this almost always means a server that implements Sun’s Java Servlet API, usually in conjunction with the JavaServer Pages (JSP) API. At the top end of the scale, the “server” in fact might be a cluster of systems, each implementing a sophisticated full J2EE (Java 2 Enterprise Edition) application server.

Some examples of Web-application servers are as follows:

- **Tomcat**: an open-source project hosted by the Apache Foundation, Tomcat is the reference implementation for the Servlet API and a popular server for development.
- **Jetty**: another open-source Web server project hosted by Mort Bay Consulting, Jetty is well known for its excellent performance and other features.
- **BEA** WebLogic: a powerful commercial application server, it implements the entire J2EE standard, including EJBs.
- **iPlanet**: a flexible and highly configurable server developed by the Sun/Netscape alliance, iPlanet is available in versions with and without full J2EE capabilities.
- **IBM’s** Websphere: also available both with and without EJB capabilities, the Websphere server is an enhanced and commercialized extension of the popular Apache HTTP server.

These are just a few of the more well-known choices. One of the great advantages of Java Web applications is the the fact that a wide range of companies support the standards.

The term “Web application” also has a specific technical meaning within the context of J2EE and its standards: it refers to a means of “packaging” an entire application, complete with code, configuration, HTML (Hyper Text Mark-up Language) and JSP pages into a single archive file – a “.war” file, specifically (Web Application aRchive). This file format is used by almost all current J2EE servers as one of the possible deployment mechanisms for Web applications. The idea is that you can simply place the .war file containing such an application into the appropriate directory, and the server will have everything it needs to fully configure and deploy the application. For the most part, this is true; however, sometimes special circumstances require post-deployment configuration. (And in some cases, “special circumstances” is a term for “mistakes.” It is always best to strive to comply with the standards for deployment, making the extra steps at least optional, if not avoiding them altogether.)

Refer to the glossary for detailed definitions of any other terms that are unfamiliar to you.

1.2.2 Components

A “component,” according to common usage, is a piece or part. The mind’s eye conjures up some kind of mechanical assembly, perhaps consisting of a number of parts, connected and related in some way, but comprising a single complete unit. A software component is similar: it is a collection of parts; in this case these are the methods and objects, which provide some specific functionality. Just like its mechanical counterpart, a component can be simple or complex, and it can work by itself or work only in conjunction with some larger unit.

We can think of components as the software equivalent of Lego (TM), the popular toy with interlocking blocks that can be used to easily assemble many different interesting things. As with Lego, there are different kinds of blocks that have different uses – some are larger, some are smaller, some are more generalized, such as the plain rectangular piece, and some are very specific, such as the block with wheels on it. As with Lego, the most interesting part of these blocks is not so much the individual pieces, but the small interlocking protrusions that let them be easily connected. Components that are designed to work together should have the same kind of generalized way of connecting them – most are perhaps not quite as easy to connect as in Lego, but the principle exists.

To define a component more formally, we can say that it is a “unit of functionality with a contractually specified interface.” Note the “interface” part here – the specification of a component always provides a concrete definition of the way we interact with the component. This interface may be divided into two parts: the functional interface, which we use from our application when we want the component to perform its functions, and the configuration interface, which is used to change settings and set properties of the component that modify its behavior in some way.

A component is specifically designed with reuse in mind. Unlike a specific once-off application, the component is intended to be configurable for use in many different completed applications, and should be flexible enough to make this easy, but at the same time it should present a well-defined means to the developer to make the connections to the rest of the application.

1.2.2.1 Separation of Interface and Implementation

One of the key attributes of a component is its separation of *interface* and *implementation*. The internals of the component can change, but the interface of the component with the outside world remains constant. This separation means that the actual implementation of the component is free to be improved, while the improved component remains compatible with the original.

Some components take this a step further, and have a "pluggable" implementation. This is where the actual code that does the work is determined dynamically, perhaps via a configuration setting. An example of this is a user authentication component that can use either an LDAP (Lightweight Directory Access Protocol) data source or a relational database to look up users. The LDAP lookup is one implementation, and the database version is another.

An example of this is explored in detail when we discuss design patterns in Chapter 8.

1.2.2.2 Inversion of Control – Don't call us, We'll call you

A feature of some (but certainly not all) components is that they themselves do not directly access their environment. Instead, they use a design pattern sometimes called "inversion of control," where the container is responsible for "handing" the component everything it needs to perform its tasks. For example, in the case of a component that calculates a customer total and writes to the database, the container would need to pass a database connection to it, as opposed to the component trying to access some external service to "request" a connection. This guarantees the isolation of the component and its independence from the external supporting environment – as long as it is handed the correct inputs, it functions correctly.

This pattern also allows the container or framework to have full control over the sequence of processing, as it always initiates the actions of the component. This lays a strong foundation for work-flow processing, and allows the flow of control to be easily changed without the components themselves requiring modification. Components written to this pattern use a *passive* structure – they act only when requested by the calling object. The calling object must also be carefully constrained: action is *only* initiated by the framework/container, and never by any other objects.

Inversion of control is also a firm foundation for a secure system. It does not guarantee security, but it is a good start. If control can be exercised from only one point, it eliminates the opportunity for component operation to be hijacked at some point in the hierarchy, and makes for only a single point that must be actually secured.

Not all component models adhere to this theory, and some experts argue that it is not necessarily the right idea, but it is a frequently seen pattern for components. Design patterns, as we will see later, are an important topic when discussing both components and frameworks.

Inversion of control is also important when designing a system to achieve scalability by distributing its components among multiple servers – it guarantees that the component does not care *where* it executes, as long as it is handed all the inputs and contexts it requires.

1.2.2.3 Component Execution Environment

Components are designed to operate in a specific environment – for example, EJB (Enterprise Java Beans) components are expected to be deployed in a suitable EJB server. The deployment environment often defines the choice of available components – in other words, if you know you are working in an EJB environment for deployment, then EJB-based components are of course a good choice.

Another term for a component's execution environment is a *container*. Simply put, a container takes components through their life cycles, managing their creation, initialization, operation, and finalization.

It is seldom quite that simple, however, because many components operate in multiple environments. A few frameworks even offer component collections that can scale from a simple JSP/Servlet environment on a single server to a distributed environment.

This is where component standardization comes to a developer's aid: if components are created according to specific standards, then they can be used in more than one environment. It means that the total pool of components available in any given environment gets larger, giving the developer more prebuilt components to choose from.

JavaBeans, for example, are components that operate in many different runtime environments.

1.2.2.4 Components and Objects

Although most components are composed of objects, there are fundamental differences between the two structures. A component can be differentiated from an object in that its "encapsulation" is guaranteed: there are no exposed implementation dependencies. An object might only be used within a single application, but a component has been designed with reuse in mind and cannot assume much about the environment in which it will be used. An object typically defines a much smaller portion of a problem space: an email message, for example. A component operates at a somewhat higher level, for example, for sending and receiving email.

A component typically contains a mechanism for configuring its operation in addition to its basic methods for performing its functions. This provides a means for a component to be adjusted to a range of different preferences, whereas an individual object does not usually provide this range of configurability. Sometimes this mechanism is accessed via a graphical configuration tool – as is often the case with JavaBeans. In this way, the configuration of the component and its current state can be both serialized and restored later for use in the finished application.

A component is often composed of a number of objects, which are designed to work together to provide specific functionality. Some component standards (such as EJB) also provide a recommended means to package their components for deployment (e.g., EJBs in a .jar file, containing the objects themselves and a deployment description file).

OBJECTS COMPARED TO COMPONENTS

OBJECTS	COMPONENTS
MANY INTERDEPENDENCIES	FEW INTERDEPENDENCIES
COMPOSED INTO COMPONENTS	COMPOSED INTO SERVICES
INTERFACE AND IMPLEMENTATION TOGETHER	SEPARATE INTERFACE AND IMPLEMENTATION

1.2.2.5 Component-based Development

Component-based development (sometimes abbreviated "CBD") is a term that describes the process of creating applications from existing components. This is distinct from the process of creating components themselves. Component-based development involves more than just deployment. It begins with the process of analysis and design with components in mind, and continues through an assembly-like development phase, to deployment.

CBD should be differentiated from the process of creating components themselves. This is referred to as component development and is a lower level process than CBD, because it creates the units of functionality, which are later assembled into one application.

One-Off Development versus Component-Based Development

One-Off Development	Component-Based Development
Timeframe hard to predict	Predictable development time
Longer time to completion	Shorter overall time
Infrastructure problems probable	Infrastructure exists, problems less likely
Harder to make flexible	Very flexible
Can't test until created, must test as a unit	Pre-tested building blocks
Integration ability must be created	Built for integration

Examining the process will give us a better understanding of components, and the frameworks within which they operate.

Component-based development begins with analysis and design, just as any other development process. With component-based development, however, the roles of component creator and component consumer are more clearly separated. Often two different companies, or at least two different teams, take on these two roles. When creating a system with component-based development, it is assumed that we are not going to start from scratch – we work with the intent of assembling existing components into an application. Similar in many ways to the roles in J2EE development/deployment, which we will discuss later, there is a distinction between the component creator (or supplier), the component assembler, and often even the component "manager," who administers the operation of the components after assembly and deployment.

Part of the design process involves the determination of the set of components that we have to choose from. Sometimes this is mandated (a company has a specific component library or framework established as a standard), in which case this part of the process is very short indeed. If there is no preordained component library to choose from, however, the same techniques that we discuss later for selecting an application framework can be applied successfully here. Reuse and acquisition, as opposed to development from scratch, are emphasized at this stage.

Once the component library is selected, matching the existing components to the problem at hand is the next step of the design sequence. Traditional software engineering practices do not fit this design process very well, and new approaches and design patterns are appropriate. Later, we discuss in detail design methodologies that are applicable to both component-based and framework-based developments.

The actual "development" phase of a component-based project should primarily be a process of configuration and assembly, with perhaps some coding for the custom business logic of the application at hand. The goal is to maximize reuse and long-term maintainability, and one of the factors that affects this is the amount of custom code that must be created.

Finally, there is the testing and deployment phase, where the finished assembly of components is verified and the component deployment environment is set up. Often, this involves deploying the appropriate support environment for the component library that was chosen: for example, a J2EE server and a related framework. One of the advantages of using the same set of components for subsequent development is that the environment is already in place – only the new components need to be deployed.

So, component-based development primarily consists of the *aggregation* and *interconnection* of components, as well as the configuration of the components themselves. It provides a powerful paradigm for rapid development of reliable, high-quality applications.

1.2.2.5.1 Types of Components

- **Client-side components:** Client-side often describes visual components intended to function on the client – in a Web application, they usually function in the user's browser. Client-side components are often user-interface elements – in other words, components with a visual representation, such as data entry fields, calendar components, folders, and other such elements. In the past these have been relatively low-level components, which are often used for building data entry screens. In the Web-application projects, visual components can be more complex, often used in conjunction with specific server-side code to form a complete application component, such as an email component, portal components, and so forth.

An Applet is a good example of a client-side component. Not all Applets can be considered components in the strictest sense, because an Applet could comprise an entire application without any ability for reuse in other applications. Often though, Applets are ideal client-side components, operating within the context of a Web application to provide a level of interactivity or visual display that is difficult to achieve with plain HTML.

- **Visual Components:** Visual components can overlap with client-side components: most client-side components are also visual because the best separation of responsibilities is often to have the visual rendering of an application happen on the client system. Many integrated development environments (IDEs), particularly those that support Applet development, provide visual components such as data entry fields, drop-down selection boxes, image frames, and so forth.

  Visual components for a Web application, though, are often generated by means of HTML, XHTML, or JavaScript provided by the server. A text box on an HTML form is a visual component, even though it is created as required by the browser in response to HTML from the server-side application.

  Visual components tend to be fairly "low-level," less complex elements than general client-side components.

- **Server-side components:** Server-side, or nonvisual, components are differentiated from client components usually by the lack of a specific user interface. They may be combined with a user interface (UI) to provide a more complete building block for the application, but this need not be the case.

  Some examples of server-side components are components that provide FTP services, database reporting, XML transformation, and practically any other application service.

1.2.2.6 JavaBeans

The JavaBean specification is an example of a component standard, and has given rise to many component libraries of different types that conform to this standard of access to component functionality. Originally thought of as visual components, JavaBeans can be much more.

First introduced in 1996 during the first JavaOne conference, the Bean API (Application Programming Interface) was initially quite specific: it called for a JavaBean to be a reusable software component that was capable of being manipulated visually in a development/builder tool. JavaBeans have come a long way since then, and have expanded even further in the latest release of the Java development kit (JDK) version 1.4.

A bean is defined as an active component that can be manipulated from within an application builder that possesses certain properties and conforms to certain design patterns. This includes being associated with a BeanInfo object that provides definitions of the bean and its properties, methods, and events. Beans were a departure from other component technologies in that they provided the means for the actual component, not a visual representation, to be manipulated from within a development tool. This is possible for the most part because of the serialization capabilities inherent in beans – they can be "serialized" to and from persistent storage as required, allowing the current state of a bean and its configuration to be stored at any point in time. When a bean is read back from its serialized state, the *core reflection* and *introspection* capabilities of Java allow the bean container to examine the bean to determine its classes, interfaces, methods, and method parameters. The container can then use this information to invoke methods, as required, on the bean itself. This is where method signatures come into play: by following specific patterns when coding the bean, the developer can ensure that its methods conform to the expected signatures and can be invoked at run time by a container that had no previous direct knowledge of the bean.

One of the better known method signatures used in JavaBeans is the set/get pattern for setting and retrieving properties of the bean: a method called "setName" with one parameter is assumed to be the one that is used for setting the property called "name." Correspondingly, the "getName" method is assumed to be the means to retrieve the name later. These patterns are recognized by the core reflection and introspection API, and should be adhered to for best portability of the finished bean.

JavaBeans also utilize a special descriptive class called "BeanInfo." This class can be entirely or partly generated, or can be implemented manually by the bean developer. It describes the bean itself, and is used as the standard "view" of the bean by builder/developer tools so that they can manipulate the bean. This separates the interface and the component implementation, a key element in component design.

An example of a component "container," or execution environment is the simple "bean box" utility provided by Sun as an illustration of JavaBean operation. It provides the necessary services to the JavaBean to configure and run the bean, although not much more, because it is designed as a simple utility and demonstration. Other component containers, such as the Phoenix subproject of the Apache Avalon project, provide much more sophisticated services to their components, such as logging, scheduling, connection pooling, and so forth.

JavaBeans are used extensively within application frameworks. For example, the Struts framework makes use of the bean pattern to allow the developer to associate a bean with every form, and to easily perform validations and data transfers between the bean and the page. Even where JavaBeans are not used directly, the get/set and other patterns that are associated with beans are still accepted as a standard. Coding in accordance with such standards not only ensures that the code is consistently readable, but also helps to enforce proper object-oriented encapsulation. Variables are kept private, and get/set methods are used to access them, hiding the internal operation of the bean from the code that uses it. This allows the internal operation to change without affecting the use of the bean, allowing improvements to be made without any reintegration with the rest of the system.

1.2.3 Application Frameworks

1.2.3.1 What Do They Do?

What are application frameworks? Why are they important to you? What do they provide that can help you get your job done?

A framework's primary purpose is to aid and ease your application development process. It should allow you to develop the application quickly and easily and should result in a superior finished application.

It is important that you see the real benefit to determine whether mounting the learning curve of a framework is worthwhile.

Frameworks, in brief, provide you with a powerful *tool box*. The tools in this box help in many different areas of application development. They provide essential design patterns and structure to your application development project, and also provide the backbone and container for the components you create for your application to operate within. In Chapter 2 we will explore the kinds of services and tools you will commonly find in frameworks.

Frameworks are valuable at all stages of development, from design to deployment and beyond, perhaps more so in ongoing maintenance. They usually apply to almost all stages of the life cycle of an application.

1.2.3.2 Application Framework Characteristics

In the process of creating Web applications, a number of basic tasks are encountered repeatedly. As in most cases of programming, where the code that is needed more than once is broken into methods or procedures, these repeated tasks can be generalized and the code reused across multiple projects. This process of generalization and reuse leads to the creation of most frameworks.

Initially generalization sounds pretty easy when you say it fast: you observe what is being done over and over, and break it into a reusable element. It is not quite that simple. The resulting elements that are broken out must form a cohesive, understandable whole. It is this design process that makes the framework itself so useful later – the basic principles of how everything is supposed to hang together have already been worked out, allowing the developer using the framework to concentrate on the specifics of the application at hand.

In its simplest form, the framework has been in existence for a long while: any body of code, such as a library, that reduces development time on future projects by being reusable, constitutes a framework of sorts. Libraries have previously tended to be specific to one area of the development process, though for example, a graphics library, a floating-point library, and so forth. Web-application development frameworks are more applicable to the entire development process of the finished application, and are often made up of a number of different functional areas, each serving one aspect of the Web-application building process.

Many progressive companies invest in frameworks specifically – either by developing one of their own, or by purchasing or otherwise adopting an existing framework. From their point of view, it is an investment for the future. Such an investment reduces their development time overall, and improves the quality of the finished product. Their developers are able to focus on the unique business requirements of their projects, rather than on infrastructure design and development. Maintainability is also substantially improved, so the money saved in adopting frameworks increases over time.

A framework also provides a well-defined extension mechanism, allowing new capabilities and services to be added without loss of structure due to personal coding styles of individual developers and design decisions. A framework must be inherently extensible, and new services should have the ability to be easily added as required. A well-defined extension mechanism prevents the framework from disintegrating into many different styles.

A framework must be as simple as possible, but no simpler. In other words, unnecessary complexity should not get in the way, and yet there must be enough capability provided that the framework has real and measurable benefit. The framework's API should be consistent across modules within the framework to make it easier to use each module once you have learned the overall style of the API.

A good framework should also have complete documentation – something that unfortunately is fairly rare today. “Complete” documentation means that all functions and parts of the framework are documented. It does not necessarily mean volumes of reference and learning material, as extensive documentation does not necessarily mean better documentation. On the contrary, the *right* documentation is much more essential than having a large tome in which you cannot find what you need. Diagrams and possibly UML serve to provide a good guide to large and complex projects, such as frameworks, and should be used as required.

A framework should be widely applicable – that is, it should have the ability to be used in many different kinds of development scenarios, and its functionality called from many different places, including EJBs, Servlets, regular Java classes, and others. This requires giving careful thought to design, because some techniques (such as thread-safety) must be taken into account. Although a framework hides much complexity from its user, it should still allow all of its functionality to be accessed. This means multiple “levels” of access, so that lower level features (such as database access or file system interaction) as well as higher level functionality (such as UI independence or logic components) can be used directly.

A framework should approach the problem domain from a generic point of view – the more involved a framework is in providing business logic, the less widely applicable it is. Many applications require logging, access to a user interface, configuration, and other common services. Fewer applications require, for example, mortgage rate calculations. If a framework does provide more business-logic level services, they should at least be optional or replaceable, otherwise its applicability might be sharply limited. A framework should also provide its functionality in modular fashion. This avoids the unfortunate fact that the more a framework does often means the less it is reusable. Every project does not need every service – some applications require database access, others do not. Developers will probably resist using a framework that provides substantial database-access features if their application does not need it – even if many other features of the framework would be valuable. If the database-access functionality is optional, however, then there is no need to carry any excess baggage.

There is a trade-off at work when building reusable components. On the positive side, time is saved by reusing the component. This must take into account the number of times it can be reused – for example, in how many projects do you think you will be able to reuse this component? Also, the time to develop the component as a once-off can be subtracted from the total time for each of those projects. On the negative side, the additional time it takes