Developing With Apache Avalon

Apache Avalon Project

Berin Loritsch

Developing With Apache Avalon

Apache Avalon Project
by Mr. Berin Loritsch

Developer's Guide version 1.0 for Avalon Framework published 2001

Revision History:
30 Oct 2001: Revision 1.3
19 Oct 2001: Revision 1.2
23 Jul 2001: Revision 1.1
15 Jun 2001: Revision 1.0

This developer's guide is dedicated to the three people who's vision started the Avalon project: Federico Barbieri, Stefano Mazzocchi, and Pierpaolo Fumagalli. Their concept for the Avalon project has stood the test of time.

Introduction and Overview

A brief history of Avalon and overview of the design principles used to create it.

In the beginning was Apache JServ. Stefano Mazzocchi and others helping develop Apache JServ realized that several patterns used in that project were generic enough to create a Server Framework. On Wednesday January 27, 1999 (roughly a month after release 1.0b of JServ) Stefano put together a proposal to start a project called the Java Apache Server Framework. It was to be the basis for all Java server code at Apache. The idea was to provide a framework to put together components and reuse code across a number of projects.

Stefano Mazzocchi, Federico Barbieri, and Pierpaolo Fumagalli created the initial version. Later in 2000, Berin Loritsch and Peter Donald joined the project. By that time, Pierpaolo and Stefano had moved on to other projects and Java Apache Server Framework started to use the name Avalon. Those five developers are the main people responsible for the current design and concepts used by the framework. The current version is very similar to the version that was released in June 2000. In fact, the major difference is the reorganization of the packages, and splitting the project into subprojects. The same design patterns and interfaces exist today.

What is Avalon?

Avalon is a parent project for five sub-projects: Framework, Excalibur, LogKit, Phoenix, and Cornerstone. Most people think of the Framework when they hear the name Avalon, but it is more than that. Avalon began as the Java Apache Server

Framework that had the framework, utilities, components, and a server's kernel implementation all in one project.

Since all the pieces of Avalon are of different maturity levels, and have different release cycles, we have decided to break Avalon into the smaller projects mentioned above. That move also enables new developers to understand and learn Avalon in distinct chunks—something that was almost impossible before.

Framework

Avalon Framework is the basis for all the other projects under the Avalon umbrella. It defines the interfaces, contracts, and default implementations for Avalon. The Framework has the most work put into it, and consequently is the most mature project.

Excalibur

Avalon Excalibur is a collection of server side Components that you can use in your own projects. It includes pooling implementations, database connection management, and Component management implementations among others.

LogKit

Avalon LogKit is a high speed logging toolkit used by Framework, Excalibur, Cornerstone, and Phoenix. It is modeled on the same principles as the JDK 1.4 Logging package but is compatible with JDK 1.2+.

Phoenix

Avalon Phoenix is the server kernel that manages the deployment and execution of Services (implemented as server components called Blocks).

Cornerstone

Avalon Cornerstone is a collection of Blocks or services that you can deploy in the Phoenix environment. The Blocks include socket management and job scheduling among others.

Scratchpad

Scratchpad is not really an official project, but it is the staging area for Components that are not ready for inclusion in Excalibur yet. They are of varying quality, and their APIs are not guaranteed to remain consistent until they are promoted into Excalibur.

Focus for this Overview

We are focusing on Avalon Framework in this overview, but we will cover enough of Avalon Excalibur and Avalon LogKit to get you started. We will use a hypothetical business server

to demonstrate how to practically use Avalon. It is beyond the scope of this overview to define a full-blown methodology, or to cover every aspect of all the sub projects.

We decided to focus on Avalon Framework because it is the basis for all of the other projects. If you can comprehend the framework, you can comprehend any Avalon based system. You will also become familiar with some of the programming idioms common in Avalon. Another reason for focusing on the framework and touching on the Avalon Excalibur and Avalon LogKit projects is that they are officially released and supported.

What Can Avalon Be Used For?

I have been asked on a couple of occasions to identify what Avalon is good for, and what it is not good for. Avalon's focus is server side programming and easing the maintainability and design of server focused projects. Avalon can be described as a framework that includes implementations of the framework.

While Avalon is focused on server side solutions, many people have found it to be useful for regular applications. The concepts used in Framework, Excalibur, and LogKit are general enough to be used for any project. The two projects that are more squarely focused on the server are Cornerstone and Phoenix.

Framework
1. A supporting or enclosing structure.
2. A basic system or arrangement as of ideas.
(see N401617 below)

The word *framework* is broad in application. Frameworks that focus on a single industry like medical systems or communications are called vertical market frameworks. The reason being that the same framework will not work well in other industries. Frameworks that are generic enough to be used across multiple industries are known as horizontal market frameworks. Avalon is a horizontal market framework. You would be able to build vertical market frameworks using Avalon's Framework.

The most compelling example of a vertical market framework built with Avalon is the publishing framework Apache Cocoon. Apache Cocoon version 2 is built using Avalon's Framework, Excalibur, and LogKit projects. It makes use of the interfaces and contracts in the Framework to reduce the time it takes for a developer to learn how Cocoon works. It also leverages the data source management and component management code in Excalibur so that it does not have to reinvent the wheel. Lastly, it uses the LogKit to handle all the logging in the publishing framework.

Once you understand the principles behind Avalon Framework, you will be able to

N401617) Webster's II New Riverside Dictionary

comprehend any system built on Avalon. Once you can comprehend the system, you will be able to catch bugs more quickly that are due to the misuse of the framework.

There is no Magic Formula

It is important to state that trying to use any tool as a magic formula for success is begging for trouble. Avalon is no exception to this rule. Because Avalon's Framework was designed to work for server solutions, it would not be a good idea to use it for building a Graphical User Interface (GUI). Java already has a framework for building a GUI called Swing.

While you need to consider if Avalon is right for your project, you can still learn from the principles and design that went into it. The question you need to ask yourself is, "Where is this project going to be used?" If the answer is that it will be run in a server environment, then Avalon is a good choice whether you are creating a Java Servlet, or creating a special purpose server. If the answer is it will be run on a client's machine with no interaction with a server, than chances are that Avalon might not be a good fit. Even then, the Component model is very flexible and can help manage complexity in a large application.

Principles and Patterns

All of Avalon is built with specific design principles. The two most important patterns are *Inversion of Control* and *Separation of Concerns*. Component Oriented Programming, Aspect Oriented Programming, and Service Oriented Programming also influence Avalon. Volumes could be written about each of the programming principles, however they are design mindsets.

Inversion of Control

Inversion of Control (IOC) is the concept that a Component is always externally managed. This phrase was originally coined by Brian Foote in one of his papers (see N40164C below) . Everything a Component needs in the way of Contexts, Configurations, and Loggers is given to the Component. In fact, every stage in the life of a Component is controlled by the code that created that Component. When you use this pattern, you implement a secure method of Component interaction in your system.

Warning:

IOC is not equivalent to security! IOC provides a mechanism whereby you can implement a scalable security model. In order for a system to be truly secured, each Component must be secure, no Component can modify the contents of objects that are passed to them, and every interaction has to be with known entities. Security is a major topic, and IOC is a tool in the programmer's arsenal to achieve that goal.

Separation of Concerns

The idea that you should view your problem space from different concern areas resulted in the Separation of Concerns (SOC) pattern (see N401665 below) . An example would be viewing a web server from different viewpoints of the same problem space. A web server must be secure, stable, manageable, configurable, and comply with the HTTP specifications. Each of those attributes is a separate concern area. Some of these concerns are related to other concerns such as security and stability (if a server is not stable it can't be secure).

The Separation of Concerns pattern in turn led to Aspect Oriented Programming (AOP) (see N401672 below) . Researchers discovered that many concerns couldn't be addressed at class or even method granularity. Those concerns are called aspects. Examples of aspects include managing the lifecycle of objects, logging, handling exceptions and cleaning up resources. With the absence of a stable AOP implementation, the Avalon team chose to implement Aspects or concerns by providing small interfaces that a Component implements.

Component Oriented Programming

Component Oriented Programming (COP) is the idea of breaking a system down into components, or facilities within a system. Each facility has a work interface and contracts surrounding that interface. This approach allows easy replacement of Component instances without affecting code in other parts of the systems. The major distinction between Object Oriented Programming (OOP) and COP is the level of integration. The complexity of a COP system is more easily managed due to fewer interdependencies among classes, promoting the level of code reuse.

One of the chief benefits of COP is the ability to modify portions of your project's code without breaking the entire system. Another benefit is the ability to have multiple implementations of the Component that you can select at runtime.

Service Oriented Programming

Service Oriented Programming (SOP) is the idea of breaking a system down into services provided by the system.

Service
1. Work or duties performed for others.
2. A facility offering repair or maintenance.
3. A facility providing the public with a utility.
(see N4016AE below)

N401665) http://www.research.ibm.com/hyperspace/MDSOC.htm
N401672) http://www.aspectj.org
N4016AE) Webster's II New Riverside Dictionary

Avalon's Phoenix identifies a service as the interface and contracts for a facility that Phoenix will provide. The implementation of the service is called a Block. It is important to realize that a server is made up of multiple services. To take the example of a Mail server, there are the protocol handling services, the authentication and authorization services, the administration service, and the core mail handling service.

Avalon's Cornerstone provides a number of low-level services that you can leverage for your own systems. The services provided are connection management, socket management, principal/role management, and scheduling. We touch on services here because it is relevant to the process of decomposing our hypothetical system down into the different facilities.

Decomposing a System

Just how do you decide what makes up a Component? The key is defining the facilities that your solution needs to operate efficiently.

We will use a hypothetical business server to demonstrate how to identify services and Components. After we define some services that are used in the system, we will take one of those services and define the different components needed by the service. My goal is to pass on some concepts that will help you define your system in manageable pieces.

System Analysis—Identifying Components

While it is beyond the scope of this presentation to provide a full-blown methodology, I do want to provide some pointers. We will start with the implementation oriented definition of Components and Services, and then provide a practical definition.

Component

A Component is the combination of a work interface, and the implementation of that interface. Its use provides a looser coupling between objects, allowing the implementation to change independently of its clients.

Service

A Service is a group of one or more Components that provide a complete solution. Examples of a Service are protocol handlers, job schedulers, and authentication and authorization services.

While these definitions provide a starting place, they don't give the whole picture. In order to decompose a system (defined as a group of facilities that comprise a project) into the necessary parts, I advocate a top-down approach. That way you will avoid being bogged down in details before you know what the different facilities are.

Determining the Scope of Your Project

You always have to start out with a general idea of what your project is supposed to accomplish. In the commercial world, the initial statement of work accomplishes this. In the open source world, this is usually accomplished by an idea or brainstorming session. I can't stress enough the importance of having a high level view of the project.

Obviously, a large project will be comprised of many different services, and a small project will only have one or two. If you start to feel a bit overwhelmed, just remind yourself that a large project is really an umbrella for a bunch of smaller projects. Eventually, you will get to the point where you will be able to comprehend the big picture.

Statement of Work: Business Server

The Business Server is a hypothetical project. For the purpose of our discussion, its function is to handle sales orders, automatically bill customers, and manage the inventory control. Sales orders have to be processed as they come in, using some kind of transaction system. The server automatically bills the customers 30 days after the sales order is filled. The inventory is managed by both the server and by the current inventory counted at the factory or warehouse. The business server will be a distributed system, and each server will communicate with others via a messaging service.

Finding the Services

We will use the Business Server Project to discover the services. Considering the overly broad statement of work, we can immediately begin to see some services defined in the description of the project. The list of services will be split into explicit ones (services that can immediately be derived from the statement of work) and implicit ones (services that are discovered due to similar work or as supporting the explicit services). Please note that the implementing company will develop not all of the services-some will be purchased as commercial solutions. In those cases, we will probably put a wrapper so that we still have a specific way of interacting with the commercial product. The implementing company will build the majority of the services.

Explicit Services

We can quickly derive a number of services from the statement of work. Our work is not done after this initial analysis, because the definition of some services requires the existence of other services.

Transaction Processing Service

The statement of work specifies that "Sales orders have to be processed as they come in". This means we need to have a mechanism of receiving sales requests and automatically process them. This is similar to the way web servers work. They receive a request for a resource, process it, and return a result (e.g. the HTML page). This is known as Transaction Processing.

To be fair, there are different types of transactions. The generic transaction service will most likely have to be broken down into something more specific like a "Sales Order Processor". The approach has to do with how generic you make your service. There is a balance between usability and reusability. The more generic a service is, the more reusable it is. Usually it is also more difficult to comprehend.

Scheduling Service

There are a couple of instances where an event must be scheduled for a specified amount of time after a transaction. In addition, the inventory control processes need to kick off supply orders on a periodic basis. Because the statement of work states "server automatically bills the customers 30 days after the sales order is filled" we need a scheduling service. The good news is that Avalon Cornerstone provides one for us so we don't have to create our own.

Messaging Service

The statement of work specifies that "each server will communicate via a messaging service" in our distributed system. Let's face it, sometimes customers want a specific product or method they want to use. The messaging service is a prime example of using another company's product. Most likely, we would use Java Messaging Service (JMS) to interface with the Messaging Service. Since JMS is a standard, it is unlikely that the interface will change any time soon.

In practical experience, a well-designed message oriented system will scale better than object oriented systems (like EJB). One reason for better scalability is that messaging tends to have lower concurrent overhead memory. Another reason for this is that it is easier to spread the load of message processing across all servers instead of concentrating all the processing in a small cluster of servers (or even just one server).

Inventory Control Service

While this is not a classic server piece in textbooks, it is a requirement of this system. The

inventory control service routinely monitors the records for what the factory or warehouse has in stock, and triggers events when stock starts running out.

Implied Services

Using experience with past systems, and further breaking down other services will yield a number of services that the system needs that wasn't specified. Due to space limitations, we will avoid doing a full decomposition.

Authentication and Authorization Service

The authentication and authorization service is not necessarily specified in the statement of work—but all business systems must take security seriously. That means all clients of the system must be authenticated, and every action of the user must be authorized.

Workflow Automation Service

Workflow automation is a hot development area in enterprise systems. If you don't use a third party workflow management server, you will have to invent your own. Workflow automation is generally the act of using a software system to route tasks through a Company's business process. For more information, view the Workflow Management Council's web page at http://www.wfmc.org.

Document Repository Service

This definition of a "document repository" is very loosely defined as the current state of information in a task. In other words, when the company receives a purchase order, our system needs to store and recall the purchase order information. The same goes for billing and any other process in the system from inventory to new customer requests.

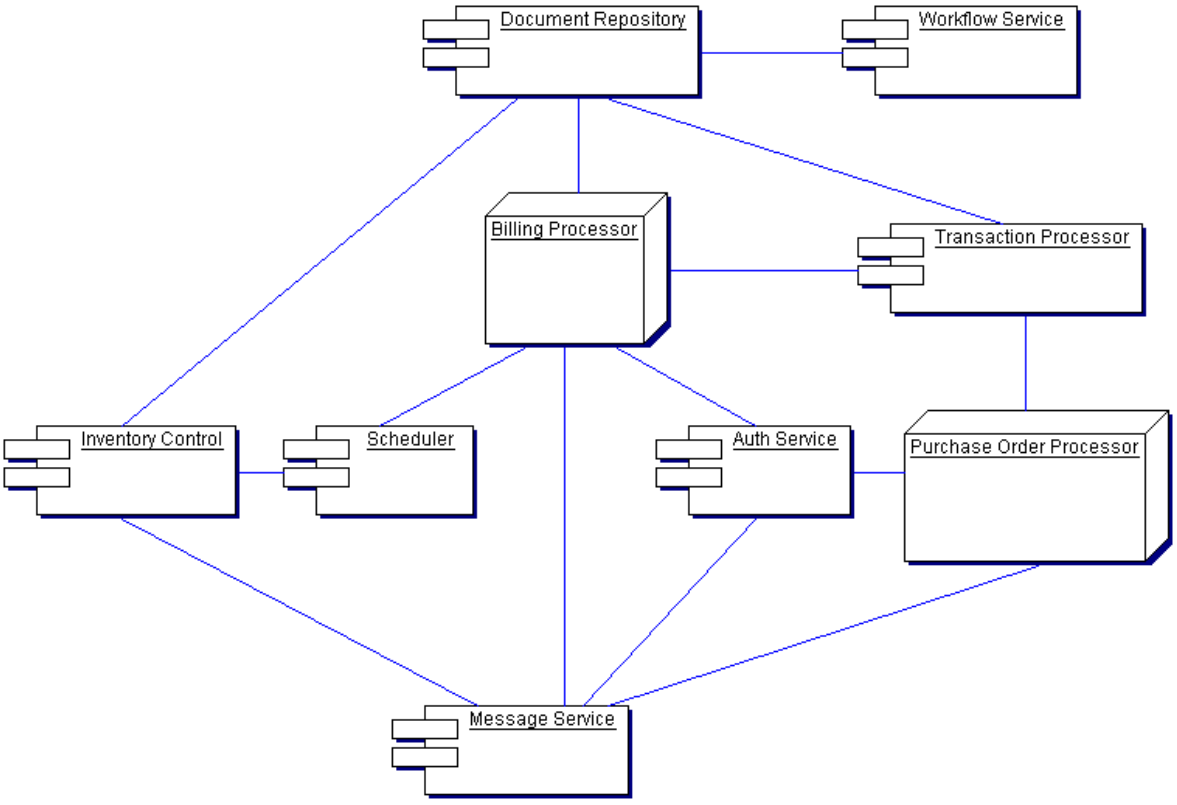
Summary

I hope that the examples of services for the Business Server project will help you discover more. You will find that as you go from higher levels of abstraction down to lower levels, you will find more types of services required like Connection Management to handle requests on open ports. Some of the services we defined will be implemented by third party systems such as the Messaging Service and the Workflow Management Service. It is in your best interest to use a standard interface for these services so that you can change vendors later. Some services are actually multiple services acting as one larger service. Some are already available within Avalon Excalibur or Avalon Cornerstone.

One thing to keep in mind while discovering the services in a system is that a service should be a high level sub-system. This will help you define components using teams of analysts. Because we have already identified the main services, you can have more than one person (or team) decompose each of the services in parallel. The boundaries are well defined, so there is little chance for overlap. If you decide to do the parallel analysis, you should come back

together to identify common Components so that you can reuse as much code as possible.

UML Diagram for the Business Server



• Berin Loritsch, 2001

Finding Components

We will use the Document Repository Service mentioned already for the process of identifying the proper Components. For the sake of our conversation, we will now state the requirements of the Document Repository Service. The repository will use a database for persistent storage, identify and authorize clients, and cache documents in memory.

Practical Definition of Components

When we talk about components, you have to think in terms of "What facilities does my service need to operate?" Avalon was conceived with the concept of *casting* your system. The developer of the system would come up with a list of responsibilities for the Component known as its *role*.

What is a Role?
The concept of roles comes from the theater. A play, musical, or movie will have a certain number of roles that actors play. Although there never seems to be a shortage of actors, there are a finite number of roles. Its *script* defines the function or action of a role. Just like the theatrical version, the script determines how you interact with the Component. Think of the different roles in your system, and you will have your *cast* of Components so to speak.

A role is the contract for a type of component. For example, our Document Repository Service needs to interact with a database. Avalon Excalibur defines a Component that satisfies the role "Data Source". Excalibur includes two different Components that satisfy that role, depending on the setting our Service will be living in; however, they both satisfy the same contracts. The majority of Avalon based systems will only use one active Component for each role. The script is the work interface: the interface with which other components interact.

There are specific contracts that you must define and keep in mind when you specify interfaces for your Components. The contracts specify what users of the Component must provide, and what the Component produces. Sometimes you must include usage semantics in the contract. An example is the difference between a temporary storage Component and a permanent storage Component. When the interface and contract are defined, you can work on your implementation.

What is a good candidate for a Component?

We have already identified four possibilities for Components within our Document Repository Service: DataSourceComponent (from Excalibur), Cache, Repository, and Guardian. You should look for roles with a high likelihood of multiple implementations that need to inter-operate seamlessly.

Using that example, you will discover several instances where you need replaceable facilities. Most of the time, you will only be using one implementation of the facility, but you need the ability to upgrade it independently of the rest of the system. Other times, you will need alternate implementations due to environmental issues. For example, the "Data Source" that Excalibur defined will usually handle all the JDBC Connection pooling itself-but sometimes you want to take advantage of that facility built into a Java 2 Enterprise Edition (J2EE) server. Excalibur solves this by having a "Data Source" that directly pools and manages the JDBC connections, and one that uses Java's Naming and Directory Interface (JNDI) to get the specified connection.

What is not a good Component?

People who are used to JavaBeans tend to implement everything as a JavaBean. This means everything from data modeling to transaction processing. If you used this approach with Components, you might end up with an overly complex system. Think of Components as

modeling a service or facility, and not data. You could have a Component that pulls data from another resource, but the data should remain distinct as data. An example of this philosophy in Avalon Excalibur is the fact that the Connection is not a Component.

Another example could be the Guardian Component we specified earlier. It could be argued that the logic involved in the Guardian is so specific to the Document Repository Service that it could not be used again for a completely different service. While there are ways of managing the complexity, and ways of making it flexible-sometimes the extra work is not worth it. You have to weigh your decisions in such cases carefully. If the logic performed in a potential Component is going to be applied consistently then it might make sense to keep it a Component. There is room to have multiple instances of a Component in a system, and they would be selected at run time. If the logic for a potential Component is specific to only one other Component, it might be worth it to absorb the logic into the other Component. Using the example of the Guardian Component and the Repository Component, we could argue that our Guardian is so specific to the Repository, that it is not implemented as a Component.

Decomposing the Document Repository Service

We will list the Components that we are going to implement with a description of their roles, the rationale, and their origination (if the component already exists).

DocumentRepository

The DocumentRepository is the parent Component of the whole service. In Avalon, services are implemented as Blocks, which are a specific kind of Component. The Block must have a work interface that extends the Service marker interface. The Block interface also extends Avalon's Component interface. Please note that Block and Service are interfaces that are part of Avalon Phoenix. In the end, a Service is still technically just a specific type of Component.

The DocumentRepository is our method of getting Document objects from persistent storage. It interacts with the other Components in the service to provide security, functionality, and speed. This particular DocumentRepository will connect to a database and employ the logic to build the Document objects internally.

DataSourceComponent

The DataSourceComponent is supplied by Avalon Excalibur. It is our method of retrieving valid JDBC Connection objects for our use.

Cache

The Cache is a short-term memory-based storage facility. The DocumentRepository will use it to store Document objects referenced by a hash algorithm. In order to promote the reusability of the Cache Component, the stored object must implement a Cacheable interface.

Guardian

The Guardian Component is used to manage permissions based on the Principal. The Guardian will load its permission sets from a database. The Guardian will use the standard Java security model to enforce access to the specific Document objects.

Summary

At this point, you should have an idea of what makes a good Component. The examples describe all the Components that will be in the Document Repository Service, with a brief summary of what they will do. A quick glance through the list supports the approach of only implementing facilities as Components—not data. At this point, you should be able to determine what components your services need to operate.

Framework and Foundations

We will describe Avalon's contracts and interfaces so we have a foundation to actually build our Components.

Avalon Framework is the central piece to the entire Avalon project. If you understand the contracts and constructs defined in the framework, you can understand anything that uses it. Remember the principles and patterns we have already discussed so far. In this section, we will expound on how the Role concept works practically, the lifecycle of Components, and how the interfaces work.

Defining the Component's Role

In Avalon, all Components have a role. The reason is that you retrieve your Components by role. At this stage, the only concern area we are using is the signature of the role. If you recall in the second section, we defined a Component as "the combination of a work interface and the implementation of the interface". That work interface is your role.

Creating the Role's Interface

Below you will find an example interface, followed by some best practices along with their reasoning.

```
package org.apache.bizserver.docs;
```

```
public interface DocumentRepository extends Component
{
    String ROLE = "org.apache.bizserver.docs.DocumentRepository";

    Document getDocument(Principal requestor, int refId);
}
```

Best Practices

- Include a String called "ROLE" that has the role's official name. That name is the same as the fully qualified name for the work interface. This helps later on when we need to get an instance of the Component later.
- Do extend the Component interface if possible. This makes it easier on you when it is time to release your Component. If you are not in control of the work interface, then you do not have this option. It is not the end of the world, as you can recast the instance to `Component` when it is time to release it.
- Do one thing and do it well. A Component should have the simplest interface possible, When your work interface extends several other interfaces, you muddy the contract for this Component. An old American acronym helps define this pattern: Keep It Simple, Stupid (KISS). It's not hard to outsmart yourself—I've done it a number of times myself.
- Only specify the methods you need. The client should have no knowledge of implementation details, and too many alternative methods only introduce unneeded complexity. In other words pick an approach and stick with it.
- Don't let your Role's interface extend any lifecycle or lifestyle interfaces. By implementing any of those classes of interfaces, you are tying an implementation to the specification. This is a bad pattern and this will only lead to debugging and implementation problems later.

Choosing the Role's Name

In Avalon, every Role has a name. It is how you get references to other Components in the system. The Avalon team has outlined some idioms to follow for the naming of your role.

Naming Idioms

- The fully qualified name of the work interface is usually the role name. The exceptions are listed after this general rule. Using this example, our theoretical Component's name would be "org.apache.bizserver.docs.DocumentRepository". This is the name that would be included in your interface's "ROLE" property.
- If we obtain the reference to this Component through a Component Selector, we usually take the role name derived from the first rule and append the word "Selector" to the end.

The result of this naming rule would be "org.apache.bizserver.docs.DocumentRepositorySelector". You can use the shorthand `DocumentRepository.ROLE + "Selector"`.

- If we have multiple Components that implement the same work interface, but are used for different purposes, we have separate roles. A Role is the Component's purpose in the system. Each role name will start with the original role name, but the purpose name of the role will be appended with a `/${purpose}`. By example we could have the following purposes for our DocumentRepository: PurchaseOrder and Bill. Our two roles would be expressed as `DocumentRepository.ROLE + "/PurchaseOrder"` and `DocuementRepository.ROLE + "/Bill"`, respectively.

Overview of Framework Interfaces

The entire Avalon Framework can be divided into seven main categories (as is the API): Activity, Component, Configuration, Context, Logger, Parameters, Thread, and Miscellany. Each of those categories (except Miscellany) represents a unique concern area. It is common for a Component to implement several interfaces to identify all the concern areas that the Component is worried about. This will allow the Component's container to manage each Component in a consistent manner.

Lifecycle for Avalon Interfaces

When a framework implements several interfaces to separate the concerns of the Component, there is potential for confusion over the order of method calls. Avalon Framework realizes this, and so we developed the contract for lifecycle ordering of events. If your Component does not implement the associated Interface, then simply skip to the next event that will be called. Because there is a correct way to create and prepare Components, you can set up your Components as you receive events.

The Lifecycle of a Component is split into three phases: Initialization, Active Service, and Destruction. Because these phases are sequential, we will discuss the events in order. In addition, the act of Construction and Finalization is implicit due to the Java language, so they will be skipped. The steps will list the method name, and the required interface. Within each phase, there will be a number of stages identified by method names. Those stages are executed if your Component extends the associated interface specified in parenthesis.

Initialization

This list of stages occurs in this specific order, and occurs only once during the life of the Component.

1. `enableLogging()`[`LogEnabled`]
2. `contextualize()`[`Contextualizable`]

3. `compose()` [`Composable`]
4. `configure()` [`Configurable`]
5. `parameterize()` [`Parameterizable`]
6. `initialize()` [`Initializable`]
7. `start()` [`Startable`]

Active Service
This list of stages occurs in this specific order, but may occur multiple times during the life of the Component. Please note that should you choose to not implement the Suspendable interface, it is up to your Component to ensure proper functionality while executing any of the Re* stages.

1. `suspend()` [`Suspendable`]
2. `recontextualize()` [`Recontextualizable`]
3. `recompose()` [`Recomposable`]
4. `reconfigure()` [`Reconfigurable`]
5. `resume()` [`Suspendable`]

Destruction
This list of stages occurs in the order specified, and occurs only once during the life of the Component.

1. `stop()` [`Startable`]
2. `dispose()` [`Disposable`]

Avalon Framework Contracts

In this section, we will cover all the sections alphabetically with the exception of the most important concern area: Component.

When I use the word "container" or "contains" when describing Components, I have a very specific meaning. I am referring to child Components that the parent Component has instantiated and controls. I am not referring to Components obtained through a ComponentManager or ComponentSelector. Furthermore, some Avalon stages received by a container must be propagated to all of its children implementing the appropriate interface. The specific interfaces in question are Initializable, Startable, Suspendable, and Disposable. The reasoning for this contract is that these particular interfaces have specific execution contracts.

Component

This is the core of Avalon Framework. Any interface defined in this concern area will throw ComponentException.

Component

Every Avalon Component *must* implement the Component interface. The Component Manager and Component Selector only handle Components. There are no methods associated with this interface. It is only used as a marker interface.

Any Component must use default no parameter constructors. All configurations are done with the `Configurable` or `Parameterizable` interfaces.

Composable

A Component that uses other Components needs to implement this interface. The interface has only one method `compose()` with a `ComponentManager` passed in as the only parameter.

The contract surrounding this interface is that the `compose()` is called once and only once during the lifetime of this Component.

This interface along with any other interface that has methods specified uses the Inversion of Control pattern. It is called by the Component's container, and only the Components that this Component needs should be present in the `ComponentManager`.

Recomposable

On rare occasions, a Component will need a new `ComponentManager` with new Component role mappings. For those occasions, implement the recomposable interface. It has a separate method from Composable called `recompose()`.

The contract surrounding the interface states that the `recompose()` method can be called any number of times, but never before the Component is fully initialized. When this method is called, the Component must update itself in a safe and consistent manner. Usually this means all processing that the Component is performing must stop before the update and resume after the update.

Activity

This group of interfaces refers to contracts for the life cycle of the Component. If there is an error during any method call with this group of interfaces, then you can throw a generic Exception.

Disposable

The `Disposable` interface is used by any Component that wants a structured way of

knowing it is no longer needed. Once a Component is disposed of, it can no longer be used. In fact, it should be awaiting garbage collection. The interface only has one method `dispose()` that has no parameters.

The contract surrounding this interface is that the `dispose()` method is called once and the method is the last one called during the life of the Component. Further implications include that the Component will no longer be used, and all resources held by this Component must be released.

Initializable

The `Initializable` interface is used by any Component that needs to create Components or perform initializations that take information from other initialization steps. The interface only has one method `initialize()` that has no parameters.

The contract surrounding this interface is that the `initialize()` method is called once and the method is the last one called during the initialization sequence. Further implications include that the Component is now live, and it can be used by other Components in the system.

Startable

The `Startable` interface is used by any Component that is constantly running for the duration of its life. The interface defines two methods: `start()` and `stop()`. Neither method has any parameters.

The contract surrounding this interface is that the `start()` method is called once after the Component is fully initialized, and the `stop()` method is called once before the Component is disposed of. Neither method will be called more than once, and `start()` will always be called before `stop()`. Implications of using this interface require that the `start()` and `stop()` methods be conducted safely (unlike the `Thread.stop()` method) and not render the system unstable.

Suspendable

The `Suspendable` interface is used by any Component that is running for the duration of its life that permits itself to be suspended. While it is most commonly used in conjunction with the `Startable` interface, it is not required to do so. The interface defines two methods: `suspend()` and `resume()`. Neither method has any parameters.

The contract surrounding this interface is that `suspend()` and `resume()` may be called any number of times, but never before the Component is initialized and started or after the Component is stopped and disposed. Calls to `suspend()` when the system is already suspended should have no effect as well as calls to `resume()` when the system is already running.

Configuration

This group of interfaces describes the concern area of configuration. If there are any problems like required `Configuration` elements that are missing, then you may throw a `ConfigurationException`.

Configurable

Components that modify their exact behavior based on configurations must implement this interface to obtain an instance of the `Configuration` object. There is one method associated with this interface: `configure()` with a `Configuration` object as the only parameter.

The contract surrounding this interface is that the `configure()` method is called once during the life of the Component. The `Configuration` object passed in *must not be null*.

Configuration

The `Configuration` object is a representation of a tree of configuration elements that have attributes. In a way, you can view the configuration object as an overly simplified DOM. There are too many methods to cover in this document, so please review the JavaDocs. You can get the `Configuration` object's value as a `String`, `int`, `long`, `float`, or `boolean`—all with default values. You can do the same for attribute values. You may also get child `Configuration` objects.

There is a contract that says that if a `Configuration` object has a value that it should not have any children, and the corollary is also true—if there are any children, there should be no value.

You will notice that you may not get parent `Configuration` objects. This is by design. To reduce the complexity of the `Configuration` system, containers will most likely pass child configuration objects to child Components. The child Components should not have any access to parent configuration values. This approach might provide a little inconvenience, but the Avalon team opted for security by design in every instance where there was a tradeoff.

Reconfigurable

Components that implement this interface behave very similar to `Recomposable` Components. It's only method is named `reconfigure()`. This design decision is used to minimize the learning curve of the Re* interfaces. `Reconfigurable` is to `Configurable` as `Recomposable` is to `Composable`.

Context

The concept of the `Context` in Avalon arose from the need to provide a mechanism to pass simple objects from a container to a Component. The exact protocol and binding names are purposely left undefined to provide the greatest flexibility to developers. The contracts

surrounding the use of the `Context` object are left for you to define in your system, however the mechanism is the same.

Context

The `Context` interface defines only the method `get()`. It has an `Object` for a parameter, and it returns an object based on that key. The `Context` is populated by the container, and passed to the child Component who only has access to *read* the `Context`.

There is no set contract with the `Context` other than it should always be *read-only* by the child Component. If you extend Avalon's `Context`, please respect that contract. It is part of the Inversion of Control pattern as well as security by design. In addition, it is a bad idea to pass a reference to the container in the Context for the same reason that the Context should be *read-only* .

Contextualizable

A Component that wishes to receive the container's `Context` will implement this interface. It has one method named `contextualize()` with the parameter being the container's `Context` object.

The contract surrounding this interface is that the `contextualize()` method is called once during the life of a Component, after `LogEnabled` but before any other initialization method.

Recontextualizable

Components that implement this interface behave very similar to `Recomposable` Components. It's only method is named `recontextualize()`. This design decision is used to minimize the learning curve of the Re* interfaces. `Recontextualizable` is to `Contextualizable` as `Recomposable` is to `Composable`.

Resolvable

The Resolvable interface is used to mark objects that need to be resolved in some particular context. An example might be an object that is shared by multiple `Context` objects, and modifies its behavior based on a particular `Context`. The `resolve()` method is called by the `Context` before the object is returned.

Logger

Every system needs the ability to log events. Avalon uses its LogKit project internally. While LogKit does have ways of accessing a Logger instance statically, the Framework wishes to use the Inversion of Control pattern.

LogEnabled

Every Component that needs a Logger instance implements this interface. The interface has

one method named `enableLogging()` and passes Avalon Framework's `Logger` instance to the Component.

The contract surrounding this method is that it is called only once during the Component's lifecycle before any other initialization step.

Logger

The `Logger` interface is used to abstract away the differences in logging libraries. It provides only a client API. Avalon Framework provides three wrapper classes that implement this interface: `LogKitLogger` for LogKit, `Log4jLogger` for Log4J, and `Jdk14Logger` for JDK 1.4 logging.

Parameters

Avalon realizes that the Configuration object hierarchy can be heavy in many circumstances. Therefore, we came up with a `Parameters` object that captures the convenience of `Configuration` objects with a simple name and value pair.

Parameterizable

Any Component that wants to use `Parameters` instead of `Configuration` objects will implement this interface. `Parameterizable` has one method named `parameterize()` with the parameter being the `Parameters` object.

The contract is that this is called once during the lifecycle of the Component. Usually this interface is used in lieu of the `Configurable` interface, however if both are used, the `parameterize()` method is called after the `configure()` method.

Parameters

The `Parameters` object provides a mechanism to obtain a value based on a `String` name. There are convenience methods that allow you to use defaults if the value does not exist, as well as obtain the value in any of the same formats that are in the `Configurable` interface.

While there are similarities between the `Parameters` object and the `java.util.Property` object, there are some important semantic differences. First, `Parameters` are *read-only* . Second, `Parameters` are easily derived from `Configuration` objects. Lastly, the `Parameters` object is derived from XML fragments that look like this:

```
<parameter name="param-name" value="param-value"/>
```

Thread

The thread marker interfaces are used to signal to the container essential semantic information regarding the Component use. They mark a component implementation in regards to thread safety. It is a best practice to delay implementing these interfaces until the final Component implementation class. This avoids complications when an implementation is marked `ThreadSafe`, but a component that extends that implementation is not. The interfaces defined in this package comprise part of what I call the *LifeStyle* interfaces. There is one more *LifeStyle* interface that is part of the Excalibur package—so it is an extension to this core set—`Poolable` that is defined in Excalibur's pool implementations.

SingleThreaded

The contract with `SingleThreaded` Components is that the interface or the implementation precludes this Component being accessed by several threads simultaneously. Each thread needs its own instance of the Component. Alternatively, you may use Component pooling instead of creating a new instance for every request for the Component. In order to use pooling, you will need to implement Avalon Excalibur's `Poolable` interface instead of this one.

ThreadSafe

The contract with `ThreadSafe` Components is that both their interface and their implementation function correctly no matter how many threads access the Component simultaneously. While this is generally a lofty design goal, sometimes it is simply not possible due to the technologies you are using. A Component that implements this interface will generally only have one instance available in the system, and other Components will use that one instance.

Miscellany

The classes and interfaces in the root package for Avalon Framework incorporates Cascading Exceptions, and a couple of generic utilities. However, one class deserves mention beyond the others.

Version

Java™ versioning techniques are entries in the manifest file in a jar. The problem is, when the jar is unpacked you lose the versioning information, and the versioning is in an easily modified text file. When you couple this with a higher learning curve, detecting Component or Interface versions is difficult.

The Avalon team came up with the Version object to allow you to have easily determined versions, and to compare versions. You may implement the `Version` object in your Components and your tests for the proper Component or minimum version level will be much easier.

Implementing the Dream

We will show how you can use Avalon Framework and Avalon Excalibur to realize your services. We will show just how easy Avalon is to use.

After your analysis is complete, you need to create the Components and Services that make up your system. Avalon would be of little use if it only described some idioms for you to use. Even then, the use of those idioms and patterns would still help in understanding the overall system. Avalon Excalibur provides some useful Components and utilities that you can incorporate into your own system that will make your life much easier. For our demonstration, we will go through the process of defining a Component that retrieves a document instance from a repository. If you recall our discussion about the theoretical Business Server, we identified this Component as a Service. In practical situations, a Service is a Component that has a larger scope.

Implementing the Component

At this point, we define how to implement our Component. We will go through the process of implementing the DocumentRepository Component previously mentioned. The first things we need to figure out are the concern areas for our Component. Then we have to figure out how our Component will be created and managed.

Choosing the Concern Areas

We have already defined the Role and the Interface for our DocumentRepository Component in the last chapter, we are ready to create the implementation. Because the interface for the DocumentRepository only defines one method, we have an opportunity to create a thread-safe Component. This is the most desired type of component because it allows for the least amount of resource utilization. In order for our implementation to be thread-safe, we do need to be careful about how we implement the Component. Since all of our documents are stored in a database, and we desire to use an external Guardian Component, we will need access to other Components. As responsible developers, we will want to log messages that will help us debug our component, and track down what is going on internally. The beauty of the Avalon Framework is that you only implement the interfaces you need, and ignore the ones you don't. This is where Separation of Concerns pays off. As you find you need a new concern area addressed, you merely implement the associated interface, and incorporate the new functionality. To the client of your Component, there is no difference.

Since it is a design goal to be thread-safe, we already know that we need to implement the ThreadSafe interface. The DocumentRepository interface only has one method, so the use of the Component's work interface is compatible with that requirement. Furthermore, we know that a Component will not be used before it is fully initialized, nor will it be used once it is destroyed.

There are a couple of implicit interfaces that we need to implement to accomplish the design. We want our solution to be as secure as possible and explicitly track whether the Component is fully initialized or not. To accomplish this goal, we will implement the Initializable and Disposable interfaces. Since specific information about our environment may change, or may need to be customized, we need to make our DocumentRepository Configurable. Our Component makes use of other Components, and the method that Avalon provides to get instances of the required Component is by using a ComponentManager. We will need to implement the Composable interface to get an instance of the ComponentManager.

Because the DocumentRepository accesses the documents in the database, we need to make a decision. Do we want to take advantage of the Avalon Excalibur DataSourceComponent, or do we want to implement our own Connection management code. For the sake of this paper, we will use the DataSourceComponent.

At this point, our skeleton class looks like this:

```
public class DatabaseDocumentRepository
extends AbstractLogEnabled
implements DocumentRepository , Configurable, Composable, Initializable,
           Disposable, Component, ThreadSafe
{
    private boolean initialized = false;
    private boolean disposed = false;
    private ComponentManager manager = null;
    private String dbResource = null;
```

```
    /**
     * Constructor.  All Components need a public no argument constructor
     * to be a legal Component.
     */
    public DatabaseDocumentRepository() {}

    /**
     * Configuration.  Notice that I check to see if the Component has
     * already been configured?  This is done to enforce the policy of
     * only calling Configure once.
     */
    public final void configure(Configuration conf)
        throws ConfigurationException
    {
        if (initialized || disposed)
        {
            throw new IllegalStateException ("Illegal call");
        }

        if (null == this.dbResource)
        {
            this.dbResource = conf.getChild("dbpool").getValue();
            getLogger().debug("Using database pool: " + this.dbResource);
            // Notice the getLogger()?  This is from AbstractLogEnabled
            // which I extend for just about all my components.
        }
    }

    /**
     * Composition.  Notice that I check to see if the Component has
     * already been initialized or disposed?  This is done to enforce
     * the policy of proper lifecycle management.
     */
    public final void compose(ComponentManager cmanager)
        throws ComponentException
    {
        if (initialized || disposed)
        {
            throw new IllegalStateException ("Illegal call");
        }

        if (null == this.manager)
        {
            this.manager = cmanager;
        }
    }

    public final void initialize()
        throws Exception
    {
        if (null == this.manager)
        {
            throw new IllegalStateException("Not Composed");
        }

        if (null == this.dbResource)
        {
            throw new IllegalStateException("Not Configured");
```

```
        }

        if (disposed)
        {
            throw new IllegalStateException("Already disposed");
        }

        this.initialized = true;
    }

    public final void dispose()
    {
        this.disposed = true;
        this.manager = null;
        this.dbresource = null;
    }

    public final Document getDocument(Principal requestor, int refId)
    {
        if (!initialized || disposed)
        {
            throw new IllegalStateException("Illegal call");
        }

        // TODO: FILL IN LOGIC
    }
}
```

You will notice some constructs in the above code. When you are designing with security in mind, you should explicitly enforce every contract on your Component. Security is only as strong as the weakest link. You should only use a Component when you are certain it is fully initialized, and never use it when it is disposed of. I placed the logic that you would need in this skeleton class because that way you can adopt the same practices in classes that you write.

Instantiating and Managing Components

In order for you to understand how the Container/Component relationship works, we will first discuss the manual method of managing Components. Next, we will discuss how Avalon's Excalibur Component infrastructure hides the complexity from you. You will still find times when you would rather manage components yourself. Most of the time the power and flexibility of Excalibur is just what you need.

The Manual Method

All of Avalon's Components are created somewhere. The code that creates the Component is that Component's Container. The Container is responsible for managing the Component's lifecycle from construction through destruction. A Container can be the static "main" method called from a command line, or it can be another Component. Remember the Inversion of Control pattern when you design your Containers. Information and method calls should only

flow from the Container to the Component.

Warning: Subversion of Control

Subversion of Control is the anti-pattern to Inversion of Control. Subversion of control is done when you pass a reference to a Component's Container to the Component. It is also done when you have a Component manage it's own lifecycle. Code that operates in this manner should be considered defective. The interactions that happen when you confuse the Container/Component relationship make the system harder to debug and security harder to audit.

In order to manage the child Components, you need to keep a reference to them for their entire lifetime. Before the Container or any other Component can use the child Component, it must go through the initialization phase of its lifecycle. For our DocumentRepository, the code will look something like the following:

```
class ContainerComponent implements Component, Initializable, Disposable
{
    DocumentRepository docs = new DatabaseDocumentRepository();
    GuardianComponent guard = new DocumentGuardianComponent();
    DefaultComponentManager manager = new DefaultComponentManager();

    public void initialize()
        throws Exception
    {
        Logger docLogger = new LogKitLogger( Hierarchy.defaultHierarchy()
                                             .getLoggerFor( "document" ) );

        this.docs.enableLogging( docLogger.childLogger( "repository" ) );
        this.guard.enableLogging( docLogger.childLogger( "security" ) );

        DefaultConfiguration pool = new DefaultConfiguration("dbpool");
        pool.setValue("main-pool");
        DefaultConfiguration conf = new DefaultConfiguration("");
        conf.addChild(pool);

        this.manager.addComponent( DocumentRepository.ROLE, this.docs );
        this.manager.addComponent( GuardianComponent.ROLE, this.guard );
        this.docs.compose( this.manager );
        this.guard.compose( this.manager );

        this.docs.configure(conf);

        this.guard.initialize();
        this.docs.initialize();
    }

    public void dispose()
    {
        this.docs.dispose();
        this.guard.dispose();
    }
}
```

For the sake of brevity, I removed all the explicit checking from the above code. You can see that manually creating and managing Components is very detailed. If you forget to do one step in the life of a Component, you will see bugs. This also requires intimate knowledge of the Components you are instantiating. An alternate approach would be to add a couple methods to the above `ContainerComponent` that handles the initialization of the components dynamically.

Automated Autonomy

Developer's are naturally lazy, so they would spend the time to write a specialized ComponentManager that became the Container for all of their Components in the system. That way they would not have to be bothered with intimately knowing the interfaces of all the Components in a system. That can be a daunting task. The Avalon developers have created just such a beast. Avalon Excalibur's Component architecture includes a ComponentManager that is controlled by configuration files written in XML.

There is a tradeoff when you relinquish the responsibility of managing a Component to Excalibur's ComponentManager. You relinquish the fine control over what Components are included in the ComponentManager. However, if you have a large system, you will find that manual control is a daunting task. In that case, it is better for the stability of the system for one entity to centrally manage all the Components in a system.

Since there are varying levels of integration you want to achieve with Excalibur's Component Architecture, we will start with the lowest level. Excalibur has a group of ComponentHandler objects that act as individual Containers for one type of Component. They manage the complete life of your Component. Let me introduce the concept of lifestyle interfaces. A lifestyle interface describes how the system treats a Component. Since the lifestyle of a component has impact on the running of a system, we need to discuss the implications of the current lifestyle interfaces:

- org.apache.avalon.framework.thread.SingleThreaded
    - Not thread-safe or reusable.
    - When no lifestyle interface is supplied, this is assumed.
    - A brand new instance is created every time the Component is requested.
    - Creation and initialization is delayed until you request the Component.
- org.apache.avalon.framework.thread.Threadsafe
    - Component is fully reentrant, and complies with all principles of thread safety.
    - One instance is created and shared with all Composables that request it.
    - Creation and initialization is done when ComponentHandler is created.

- org.apache.avalon.excalibur.pool.Poolable
    - Not thread-safe, but is fully reusable.
    - A pool of instances is created and the free instances are returned to Composables that request it.
    - Creation and initialization is done when ComponentHandler is created.

The ComponentHandler interface is very simple to deal with. You initialize the Constructor with the Java class, the Configuration object, the ComponentManager, a Context object, and a RoleManager. If you know that your Component will not need any of the aforementioned items, you can pass a null in its place. After that, when you need a reference to the Component, you call the "get" method. After you are done with it, you call the "put" method and pass the Component back to the ComponentHandler. The following code will make it easier to understand.

```
class ContainerComponent implements Component, Initializable, Disposable
{
    ComponentHandler docs = null;
    ComponentHandler guard = null;
    DefaultComponentManager manager = new DefaultComponentManager();

    public void initialize()
        throws Exception
    {
        DefaultConfiguration pool = new DefaultConfiguration("dbpool");
        pool.setValue("main-pool");
        DefaultConfiguration conf = new DefaultConfiguration("");
        conf.addChild(pool);
        this.docs.configure(conf);

        this.docs = ComponentHandler.getComponentHandler(
                                        DatabaseDocumentRepository.class,
                                        conf, this.manager, null, null);
        this.guard = ComponentHandler.getComponentHandler(
                                        DocumentGuardianComponent.class,
                                        null, this.manager, null, null);

        Logger docLogger = new LogKitLogger( Hierarchy.defaultHierarchy()
                                             .getLoggerFor( "document" ) );

        this.docs.enableLogging( docLogger.childLogger( "repository" ) );
        this.guard.enableLogging( docLogger.childLogger( "security" ) );

        this.manager.addComponent(DocumentRepository.ROLE, this.docs);
        this.manager.addComponent(GuardianComponent.ROLE, this.guard);

        this.guard.initialize();
        this.docs.initialize();
    }

    public void dispose()
```

```
    {
        this.docs.dispose();
        this.guard.dispose();
    }
}
```

At this point, we only saved ourselves a few lines of code. We still manually created our Configuration object, we still had to set the Logger, and we still had to initialize and dispose of the ComponentHandler objects. What we did at this point is simply protect ourselves from changing interfaces. You may find it better for your code to use this approach. Excalibur went further though. Most complex systems have configuration files, and they allow an administrator to alter vital Configuration information. Excalibur can read a configuration file in the following format, and build the Components in a system from it.

```
<my-system>
  <component
role="org.apache.avalon.excalibur.datasource.DataSourceComponentSelector"
class="org.apache.avalon.excalibur.component.ExcaliburComponentSelector">
     <component-instance name="documents"
       class="org.apache.avalon.excalibur.datasource.JdbcDataSource">
         <pool-controller min="5" max="10"/>
         <auto-commit>false</auto-commit>
         <driver>org.mysql.MySqlDriver</driver>
         <dburl>jdbc:mysql:localhost/mydb</dburl>
         <user>test</user>
         <password>test</password>
      </component-instance>
      <component-instance name="security"
        class="org.apache.avalon.excalibur.datasource.JdbcDataSource">
         <pool-controller min="5" max="10"/>
         <auto-commit>false</auto-commit>
         <driver>org.mysql.MySqlDriver</driver>
         <dburl>jdbc:mysql:localhost/myotherdb</dburl>
         <user>test</user>
         <password>test</password>
      </component-instance>
  </component>
  <component
    role="org.apache.bizserver.docs.DocumentRepository"
    class="org.apache.bizserver.docs.DatabaseDocumentRepository">
      <dbpool>documents</dbpool>
  </component>
  <component
    role="org.apache.bizserver.docs.GuardianComponent"
    class="org.apache.bizserver.docs.DocumentGuardianComponent">
      <dbpool>security</dbpool>
      <policy file="/home/system/document.policy"/>
  </component>
</my-system>
```

The root element can be anything you want. You will notice that we now have several

Components defined. We have our familiar DocumentRepository class and GuardianComponent class, as well as a couple of Excalibur DataSourceComponent classes. In addition, now we have some specific configuration information for our Guardian Component. In order to read that information into your system, Avalon Framework provides some conveniences for you:

```
DefaultConfigurationBuilder builder = new DefaultConfigurationBuilder();
Configuration systemConf = builder.buildFromFile("/path/to/file.xconf");
```

This does simplify all the code we had for hand-building the Configuration element earlier, and it limits the amount of information we need to explicitly know right away. We will take one last look at our Container class and see if we really have some savings. Keep in mind that we have five components specified (a ComponentSelector counts as a Component), and configurations for each of them.

```
class ContainerComponent implements Component, Initializable, Disposable {
    ExcaliburComponentManager manager = new ExcaliburComponentManager();

    public void initialize()
        throws Exception
    {
        DefaultConfigurationBuilder builder = new
DefaultConfigurationBuilder();
        Configuration sysConfig =
builder.buildFromFile("./conf/system.xconf");

        this.manager.setLogger(  Hierarchy.getDefaultHierarchy()
                                        .getLoggerFor("document") );
        this.manager.contextualize( new DefaultContext() );
        this.manager.configure( sysConfig );
        this.manager.initialize();
    }

    public void dispose()
    {
        this.manager.dispose();
    }
}
```

Isn't this amazing? We have more than twice the number Components initialized and ready for use with less than half the code (six lines of code instead of thirteen lines of code). There is the drawback of the Configuration file looking somewhat crazy, but it minimizes the amount of code you have to write.

There is a lot of activity happening under the hood of the ExcaliburComponentManager. For each "component" element in the configuration file, Excalibur creates a ComponentHandler for each class entry and maps it to the role entry. The "component" element and all it's child

elements are used for the Configuration of the Component. When the Component is an ExcaliburComponentSelector, the Excalibur reads each "component-instance" element and performs the same type of operation as before-this time mapping to the hint entry.

Making the Configuration Pretty

We can manage the configuration file's appearance with the use of aliases. Excalibur uses a RoleManager to provide aliases for the configuration system. A RoleManager can either be a dedicated class that you create, or you can use the DefaultRoleManager and pass in a Configuration object. If I use the DefaultRoleManager, I will hide the role configuration file inside the jar with the rest of the system. This is because the role configuration file is only going to be altered by developers. Below is the interface for the RoleManager:

```
interface RoleManager
{
    String getRoleForName( String shorthandName );
    String getDefaultClassNameForRole( String role );
    String getDefaultClassNameForHint( String hint, String shorthand );
}
```

Let's take a look at how Excalibur uses the RoleManager in our scheme. First, Excalibur will cycle through all the elements that are direct children of the root element. This includes all "component" elements like before, but this time when Excalibur doesn't recognize an element name, it asks the RoleManager which role we should use for this Component. If the RoleManager returns null, the element and all it's child elements are ignored. Next, Excalibur derives the class name from the role name. The last method is to dynamically map a class name to a ComponentSelector's child type.

Excalibur provides a default implementation of the RoleManager that is configured with an XML configuration file. The markup is very simple, and it hides all the extra information you don't want your administrator to see.

```
<role-list>
  <role
name="org.apache.avalon.excalibur.datasource.DataSourceComponentSelector"
    shorthand="datasources"
default-class="org.apache.avalon.excalibur.component.ExcaliburComponentSelector">
     <hint shorthand="jdbc"
class="org.apache.avalon.excalibur.datasource.JdbcDataSourceComponent"/>
     <hint shorthand="j2ee"
class="org.apache.avalon.excalibur.datasource.J2eeDataSourceComponent"/>
  </role>
  <role
    name="org.apache.bizserver.docs.DocumentRepository"
    shorthand="repository"
    default-class="org.apache.bizserver.docs.DatabaseDocumentRepository"/>
  <role
```

```
    name="org.apache.bizserver.docs.GuardianComponent"
    shorthand="guardian"
    default-class="org.apache.bizserver.docs.DocumentGuardianComponent"/>
</role-list>
```

In order to use the RoleManager, you do need to alter the "initialize" method of our Container class. You are using the configuration builder to build a Configuration tree from this file. Please remember, if you are going to use a RoleManager, you must call the "setRoleManager" method *before* the "configure" method. To demonstrate how you would retrieve this XML file from the class loader, I will demonstrate the technique below:

```
DefaultConfigurationBuilder builder = new DefaultConfigurationBuilder();
Configuration sysConfig = builder.buildFromFile("./conf/system.xconf");
Configuration roleConfig = builder.build(
            this.getClass().getClassLoader()
.getResourceAsStream("/org/apache/bizserver/docs/document.roles"));

DefaultRoleManager roles = new DefaultRoleManager();
roles.enableLogging(Hierarchy.getDefaultHierarchy().getLoggerFor("document.roles"));
roles.configure(roleConfig);

this.manager.setLogger( Hierarchy.getDefaultHierarchy()
                        .getLoggerFor("document") );
this.manager.contextualize( new DefaultContext() );
this.manager.setRoleManager( roles );
this.manager.configure( sysConfig );
this.manager.initialize();
```

Since we added six more lines of code, we need to see what it bought us. Our final configuration file can be written like this:

```
<my-system>
  <datasources>
     <jdbc hint="documents">
         <pool-controller min="5" max="10"/>
         <auto-commit>false</auto-commit>
         <dburl>jdbc:mysql:localhost/mydb</dburl>
         <user>test</user>
         <password>test</password>
      </jdbc>
      <jdbc hint="security">
         <pool-controller min="5" max="10"/>
         <auto-commit>false</auto-commit>
         <dburl>jdbc:mysql:localhost/myotherdb</dburl>
         <user>test</user>
         <password>test</password>
      </jdbc>
  </datasources>
  <repository>
      <dbpool>documents</dbpool>
```

```
  </repository>
  <guardian>
      <dbpool>security</dbpool>
      <policy file="/home/system/document.policy"/>
  </guardian>
</my-system>
```

As you can see, this is much more readable than how we started. Now we can add any number of components to our system, and we won't have to write any more code to support them.

Using the Component

Now that we have created our Components, we will want to use them. You access Components the same way regardless of how they were instantiated or managed. You must implement the Composable interface to get a reference to the ComponentManager. The ComponentManager holds all the references to the Components you need. For the sake of our discussion, we will assume that the ComponentManager given to us is configured in the same manner as the final Configuration file in the last section. This means that we have a Repository, a Guardian, and two DataSources.

Rules for Using the Component Management Infrastructure

The Component management infrastructure requires that you release any Component for which you have obtained a reference. The reason for this restriction is so that the Component's resources can be properly managed. A ComponentManager is designed for cases when you have multiple types of Components with distinct roles. A ComponentSelector is designed for cases when you have multiple types of Components with the same role. Another unique aspect of the ComponentSelector is that it is a Component by design. This enables us to get a ComponentSelector from a ComponentManager.

There are two valid approaches for handling references to external Components. You can obtain your references during initialization, and release them during disposal. You may also encapsulate the Component handling in a try/catch/finally block. Each has its advantages and disadvantages.

Initialization and Disposal Approach

```
class MyClass implements Component, Composable, Disposable
{
    ComponentManager manager;
    Guardian myGuard;

    /**
     * Obtain a reference to a guard and keep the reference to
     * the ComponentManager.
```

```
     */
    public void compose(ComponentManager manager)
        throws ComponentException
    {
        if (this.manager == null)
        {
            this.manager = manager;
            myGuard = (Guardian) this.manager.lookup(Guardian.ROLE);
        }
    }

    /**
     * This is the method that uses the Guardian.
     */
    public void myMethod()
        throws SecurityException
    {
        this.myGuard.checkPermission(new BasicPermission("test"));
    }

    /**
     * Get rid of our references
     */
    public void dispose()
    {
        this.manager.release(this.myGuard);
        this.myGuard = null;
        this.manager = null;
    }
}
```

As you can see by the sample code, this is easy to follow. The object gets a reference to a Guardian Component when it first receives the ComponentManager. If you could be guaranteed that the Guardian Component was ThreadSafe, then this is all that is necessary. Unfortunately, you cannot guarantee this for the long term. To properly manage resources, we must release the Component when we are done with it. That's why we kept a reference to the ComponentManager.

The main disadvantage of this approach comes into play when you are dealing with pooled Components. The reference of the Component is kept for the life of this object. It might not be a problem if the object had a short life span, but if it was a Component managed by the Excalibur component management architecture, its life span is as long as the Component whose reference it has. What this means is that we are essentially turning the Component's pool into a Factory.

The main advantage of this approach is that the code is very clear on how a Component is obtained and released. You don't have to have any understanding of exception handling.

One other nuance is that you are tying the existence of the Guardian to the ability to initialize this object. Once an Exception is thrown during the initialization phase of an object, you must assume that the object is not valid. Sometimes you want to fail if a required Component

does not exist so this is not a problem. You do need to be aware of this implication when you are designing your Components though.

Exception Handling Approach

```
class MyClass implements Composable, Disposable
{
    ComponentManager manager;

    /**
     * Obtain a reference to a guard and keep the reference to
     * the ComponentManager.
     */
    public void compose(ComponentManager manager)
        throws ComponentException
    {
        if (this.manager == null)
        {
            this.manager = manager;
        }
    }

    /**
     * This is the method that gets the Guardian.
     */
    public void myMethod()
        throws SecurityException
    {
        Guardian myGuard = null;
        try
        {
            myGuard = (Guardian) this.manager.lookup(Guardian.ROLE);
            this.criticalSection(myGuard);
        }
        catch (ComponentException ce)
        {
            throw new SecurityException(ce.getMessage());
        }
        catch (SecurityException se)
        {
            throw se;
        }
        finally
        {
            if (myGuard != null)
            {
                this.manager.release(myGuard);
            }
        }
    }

    /**
     * Perform critical part of code.
     */
    public void criticalSection(Guardian myGuard)
```

```
        throws SecurityException
    {
        myGuard.checkPermission(new BasicPermission("test"));
    }
}
```

As you can see, the code is a bit more complex. In order to understand it, you have to understand Exception handling. This is not necessarily a problem, because most Java developers know how to handle them. You don't have to worry so much about the Component life style with this approach, because we are releasing it as soon as we no longer need it.

The main disadvantage of this approach is the added complexity of the exception handling code. In order to minimize the complexity and make the code more maintainable, we extracted the working code into another method. Keep in mind that we can get the reference to as many Components as we possibly want inside the try block.

The main advantage of this approach is that you are managing your Component references more efficiently. Again, there is no real difference if you are using ThreadSafe Components, but it makes a real difference when you have pooled Components. There is a slight overhead dealing with getting a new reference every time you use a Component, but the likelihood of being forced to create a new instance of the Component is minimized.

Just like the Initialization and Disposal Approach, you have to understand a subtle nuance. The Exception Handling Approach does not fail on initialization if the Component is missing from the manager. As mentioned before, this is not entirely bad. Many times, you want an object to exist, but it is not a failure if a desired Component is missing.

Getting Components from a ComponentSelector

For most operations, you will only need the ComponentManager. Since we decided that we needed multiple instances of the DataSourceComponent, we need to know how to get the instance we want. ComponentSelectors are a little trickier than ComponentManagers because we are dealing with hints to get the reference we need. A Component has a specific Role, and this contract is well documented. However, sometimes we need to select one of many Components for a Role. A ComponentSelector uses an arbitrary object for the hint. Most of the time, the object is a String, although you might want to use a Locale object to get a proper internationalization Component.

In our system we have set up, we chose to use Strings to select the correct instance of the DataSourceComponent. We even gave ourselves a Configuration element that references the exact string we need to get the right Component. This is a good practice to follow, as it makes it easier on administrators of a system. It is easier for an administrator to see a reference to another Component than it is for them to remember magic values for the configuration.

Conceptually, getting a Component from a ComponentSelector is no different than getting a Component from a ComponentManager. You just have one more step. Remember that a ComponentSelector is a Component. The ComponentManager will be set up to return the ComponentSelector when you lookup its role. You then need to select the component from the selector. To demonstrate, I will extend the code from the Exception Handling Approach discussed previously.

```
public void myMethod()
    throws Exception
{
    ComponentSelector dbSelector = null;
    DataSourceComponent datasource = null;
    try
    {
        dbSelector = (ComponentSelector)
                this.manager.lookup(DataSourceComponent.ROLE + "Selector");
        datasource = (DataSourceComponent)
                dbSelector.select(this.useDb);

        this.process(datasource.getConnection());
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (datasource != null)
        {
            dbSelector.release(datasource);
        }

        if (dbSelector != null)
        {
            this.manager.release(dbSelector);
        }
    }
}
```

As you can see, we got the reference to the ComponentSelector using the Role specified for the Component. We followed the Role naming guidelines outlined in a previous chapter by adding the "Selector" suffix to the Role name. It is also perfectly acceptable to use a static interface for all the Role names in your system to minimize the number of String concatenation in your code.

Next, we obtained the reference to the DataSourceComponent from the ComponentSelector. Our sample code assumed that we had already pulled the required information from the Configuration object and placed it in a class variable named "useDb".

Excalibur's Utilities

This last section is included to give you an idea of the types of Components and utilities that are included with Apache Avalon Excalibur. These utilities are robust, and fully usable in production systems. We do have an unofficial staging project called "Scratchpad" where we iron out implementation details for potential new utilities. Scratchpad utilities are of varying quality, and their use is not guaranteed to remain the same—although you may have good experience with them.

Command Line Interface (CLI)

The CLI utilities are used by a number of projects including Avalon Phoenix and Apache Cocoon to process command line arguments. It provides facilities to print help responses, and to process options by either a short name or a long name.

Collection Utilities

The collection utilities provide some enhancements to the Java™ Collections API. Among them is the ability to find the intersections between two lists and a `PriorityQueue` that is an enhancement to `Stack` to allow the priority of objects override the simple first in/last out `Stack` implementation.

Component Management

We already discussed the use of this in the previous section. This is Excalibur's most complex beast, but it provides a lot of functionality in just a few classes. It will make one distinction more than simple `SingleThreaded` or `ThreadSafe` for managing a component type: `Poolable`. If a Component implements Excalibur's `Poolable` interface instead of the `SingleThreaded` interface, it will maintain a pool of Components and reuse instances. Most of the time this works great. For those last remaining times where a Component cannot be reused, use the `SingleThreaded` interface.

LogKit Management

The Avalon development team realized that many people wanted a simple mechanism to build complex Logging target heirarchies. In the same spirit as the `RoleManager` the team developed a `LogKitManager` that can be given to the Excalibur Component Management system meantioned above. Based on the "logger" attribute it will give the proper `Logger` object to the different Components.

Thread Utilities

The *concurrent* package contains several classes to assist in multithreaded programming: `Lock` (a mutex implementation), `DjikstraSemaphore`, `ConditionalEvent`, and `ThreadBarrier`.

Datasources

This is modeled after the `javax.sql.DataSource` class, but simplified. There are two implementations of the `DataSourceComponent`: one that pools JDBC connections explicitly, and one that uses a J2EE application server's `javax.sql.DataSource` class.

Input/Output (IO) Utilities

The IO utilties provide a number of `FileFilter`s and other `File` and IO specific utilities.

Pool Implementations

The Pool implementations provide a `Pool` for every occasion. You have an implementation that is blazingly fast, but only usable in one thread—which should be ok for implementing a FlyWeight pattern. You also have `DefaultPool`, which does not manage the number of objects in its pool. `SoftResourceManagingPool` decommissions objects that exceed a threshold when they are returned. Lastly, `HardResourceManagingPool` throws an exception when you have reached the maximum number of objects. The last three pools are all `ThreadSafe`.

Property Utilities

The property utilities are used in conjunction with Context objects. They give you the ability to expand "variables" in your `Resolvable` object. It works like this: `"${resource}"` will look for a Context value named `"resource"` and substitute its value for the symbol.

Conclusion

Avalon has come of age, and it is ready for you. The arguments presented in this section can help convince you and others that using a mature framework is better than creating your own.

Maybe you are already convinced, but need some help convincing your colleagues that Avalon is right for you. Maybe you need some convincing yourself. Either way, this chapter will help wrap everything up, and provide you with some convincing arguments. We all need to fight Fear, Uncertainty, and Doubt (FUD) with the Open Source model. For arguments on the validity of Open Source, I will direct you to Eric S. Raymond's excellent treatises on the subject (see N404729 below) . Regardless of your opinions on his politics, the papers he wrote and compiled into the book *The Cathedral and the Bazaar* will give you the information you need to be convinced about the open source model as a whole.

Avalon Works

The bottom line is that Avalon accomplishes the goal it was originally designed to fulfill. Avalon does not introduce new concepts and ideas, but rather uses and formalizes several concepts that have stood the test of time. The newest concept that influenced the design of Avalon is the Separation of Concerns pattern introduced sometime around 1995. Even then, Separation of Concerns is a formalization of

N404729) http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

System Analysis techniques.

Avalon's user base is measured in the hundreds. Several projects like Apache Cocoon, Apache JAMES, and Jesktop are all built on Avalon. Developers for those projects are users of Avalon Framework. Because of the number of users Avalon has, it is very well tested.

Designed by the Best Minds

The authors of Avalon recognize that we are not the sole experts on server side programming. We use concepts and ideas from other people's research. We respond to feedback from our users. Avalon is not just designed by the five developers mentioned in the introduction—the people who came up with the concepts of Inversion of Control, Separation of Concerns, and Component Oriented Programming designed it.

The beauty of Open Source projects is that the result is an amalgamation of the best ideas and the best code. Avalon has gone through periods of testing ideas and rejecting them because there was a better solution. You can take the knowledge gained by the Avalon team and use it in your own systems. You can take the predefined components in Excalibur and use them in your own projects—they have been tested to work under heavy load without errors.

Compatible License

The *Apache Software License* (ASL) is compatible with just about every other license known. The biggest known exceptions are the *GNU Public License* (GPL) and the *Lesser GNU Public License* (LGPL). The important thing is that the ASL is friendly to corporate development, and does not force you to release your source code if you don't want to. It is the same license used for the Apache Software Foundation's venerable HTTP server.

Pooled Research

Most of Avalon's users contribute back to the project in some way. This spreads the cost of developing, debugging, and documenting the framework across several users. It also means that Avalon's code has gone through a more extensive peer review than would ever be possible in one company. In addition, users of Avalon support Avalon. While it is true open source projects do not typically have a help desk or telephone support line, we do have a mailing list. Many times your questions can be answered in less time on the list than it would take on some support lines.

Simplified Analysis and Design

Developing on Avalon helps the developer to get into a mindset. That mindset focuses the efforts on how to discover Components and Services. Since many of the details regarding the life of the Components and Services in the system are already analyzed and designed, the developer only has to choose which ones they need.

It is important to state that Avalon development does not replace traditional Object Oriented

Analysis and Design, but enhances it. You are still using the same techniques you did before, only now you have a tool set you can use to achieve your design faster.

Avalon is Ready

Avalon Framework, Avalon Excalibur, and Avalon LogKit are ready for you to use now. They are mature and only getting better. While Avalon Phoenix and Avalon Cornerstone are under heavy development, anything you write for them will be usable with only minor modifications in the future.

Revision History

Revision 1.3 (30 Oct 2001)

- UPDATED — Added some clarifications to the best practices based on discussions with Gerhard Froehlich (g-froehlich@gmx.de) (BL)
- UPDATED — Reformatted code examples so they are more readable. (BL)
- UPDATED — Applied fixes from Marcus Crafter (crafterm@fztig938.bank.dresdner.net) and Ovidiu Predescu (ovidiu@cup.hp.com). (BL)

Revision 1.2 (19 Oct 2001)

- ADDED — Added support for printing to PDF. Now everone can enjoy a nicely printed document. (BL)
- REMOVED — Removed mention of Testlet as it is a dead project now. (BL)

Revision 1.1 (23 Jul 2001)

- FIXED — Fix bugs in the example code and config files, which I encountered while trying to get it to compile. Submitted by: jeff@socialchange.net.au (jeff) (PD)
- CHANGED — Applied documentation patches for incorrect code, spelling mistakes, etc. (BL)
- CHANGED — Updated sample configuration file to reflect the new "driver" configuration element. (BL)
- CHANGED — Corrected errors in sample code, as well as clarified thread safety contracts. (BL)

Revision 1.0 (15 Jun 2001)

- ADDED—Initial Port from MS Word. (BL)

About the Authors

Mr. Berin Loritsch

Affiliations

- [ASF] **Release Manager** *Apache Software Foundation/Apache Avalon*
- [IPMS] **Programmer/Analyst** *Information Planning & Management Services, Inc.*
- [TTG] **Web Developer** *The Technologies Group, Inc.*

Bio

Berin has helped define and document the Avalon projects since 2000. He has been involved in Apache Avalon and Apache Cocoon. He is the author of the current thread-safe pool implementations as well as the DataSourceComponent. Berin and Giacomo Pati were the architects of Excalibur's Component Management infrastructure.

Outside of the public view of the Apache Software Foundation, Berin has developed workflow based web applications as well as data manipulation services. He has nine years of experience developing database backed applications, and eight years experience with technical writing. Berin has only been developing Java since 1999, but his background in other Object Oriented Languages and architectures like C++ and CORBA helped him get a jump start.