OverviewOfAvalon

Overview Of Avalon

We'll start with a summary of the Avalon Framework, its terms, concepts and components. A more thorough review can be found in the Developing With Avalon white paper.

So what is Avalon? Perhaps the easiest way to answer that is by describing the problems Avalon can solve.

Problems and Solutions

Coupling: In any reasonably complex application, coupling between components or modules becomes an issue. Changes in one section can cause a ripple effect through the application as a whole. The result is often a monolithic monstrosity.

Avalon offers a consistent and elegant solution. Each component in an application plays a "role" which is defined by the "service" it offers. Components come in all shapes and sizes – most everything can be defined as a component at one layer or another. For example, the "persistence" component may be comprised of datasources, caching, and transaction components. The component is written as an interface and it's methods define the "contract" that component has with any other component which might use it. Components within an Avalon system only interact with one another via these contracts.

Once we know what services and components our application is made of we need an actual concrete implementation which will fulfill that component's "role." In fact, in some cases we may have several different implementations of a particular "service" which are needed for different contexts within the over all application. In order to sort out what implementation fulfills what role, Avalon introduces the idea of a "RoleManager." The RoleManager, usually configured via an XML file, keeps track of these relationships.

The use of roles solves another problem: implementation lock . Since components are only tied to one another via their roles, you are free to change, upgrade, swap, migrate, or in general transform your component implementation without incurring application instability. As long as the new implementation can fulfill the contract defined by the component role, you can replace existing implementations with newer ones. For example, one could easily write adaptors for third-party modules, be they commercial or open source, to be used as components in your Avalon-based project.

You might be wondering right now, "Well, that's all well and good, but how do I get a hold of these components in my application?" In other words, you still need a way to consistently instantiate, configure, use and release these components. You need lifecycle management .

Every object has a lifecycle of some sort, though you may not be used to thinking about it in that way. Consider a simple JavaBean. It's instantiated usually via its constructor and some fields are set (or unset) via getters and setters. Methods are then called and re-called before the object is finally discarded for the garbage collector to take care of. Another example of an object lifecycle are the ejbActivate and ejbRemove methods of an Enterprise Session Bean.

The Avalon Framework provides a consistent set of lifecycle methods for all components. These include:

1. Log Enabled 2. Contextualizable 3. Serviceable 4. Configurable 5. Parameterizable 6. Initializable 7. Startable 8. Suspendable 9. Recontextualizable
2. Recomposable
3. Reconfigurable
4. Reparameterizable
5. Stoppable
6. Disposable

If that list doesn't mean much to you yet, don't worry about it. The important thing to know right now is that components have lifecycles which are distinct from the "services" or contract they offer.

A "container" is a special component which handles the lifecycles of the components it hosts. Only the container has this responsibility of lifecycle management. The Avalon Framework provides a number of utilities which ease this function. Specifically a "ServiceManager" is used by the container to properly handle component lifecycle methods. Between the ServiceManager and the RoleManager, the container can find roles and bring them to life.

Other components get access to the ServiceManager via the "serviceable" lifecycle listed above, allowing components to interact with one another. When one component requests another component via it's role name, the ServiceManager and container insure that the referenced component has gone through the appropriate lifecycle methods before the calling component gets it's grubby little hands on it. Additionally, the role design allows the calling component to interact with the referenced component only via the defined methods of the role (i.e.- the interface).

This is the crux of the "inversion of control" ideas expressed in the Avalon framework. Components only interact via well defined contracts and calling (or parent) components are responsible for the handling of referenced (or child) components. Following this design increases the stability of your code since each component behaves in well defined ways and lifecycle management is strictly controlled.

Moreover, by using properly defined roles one gains "Separation of Concerns." Each role or component is only concerned with fulfilling its contract and nothing more. It allows components (and developers) to focus on one thing at a time.

The results is that each component is very modular and consequently easier to reuse in other projects. So you only need to write the perfect security layer once. Well defined roles are even easier to export between projects, giving a consistency to all your applications.

Avalon provides all the interfaces, utilities, and several implementations that make these design ideas work – a complete framework. You'll also find other features and benefits which I haven't elaborated here, like integrated logging and configuration management and instrumentation support.

Hopefully, I've whet your appetite. If you're still a little confused and would like to better understand these concepts, you can look at the Framework documentation itself or the Developing With Avalon whitepaper. However, the next section shows some concrete examples of the ideas discussed here. The best way to really understand Avalon is to see it in action. Avalon: Parts and Pieces

What's Next

Now that we have a little idea of what problems Avalon is trying to solve, why don't we look at how Avalon solves them in it's various projects and components. See AvalonPartsAndPieces

[Back to AvalonForBeginners]