# AvalonFiveArchitectureProposal

{{http://jakarta.apache.org/avalon/images/header.gif}} _(this page is part of the wiki materials for [ApacheAvalon]; avalon main page in the wiki is [AvalonProjectPages])_ h1. Avalon 5 Architecture We have learned much from our experiences with Avalon up to this point. We learned the importance of the foundational principles (IOC, SOC), and the overall architecture of component based design. We also learned the limitations of our Framework 4.x generation container designs and the fact that our contracts are too generic. For the future we want to be able to support an architecture that is powerful, flexible, but sufficiently specified so that we can have multiple implementations of containers. To achieve that end we need proper component verification, and modular container architecture. h2. Container Contracts We have learned a great deal from all of our container implementations, and have learned what could be made better and what could be lived without. Most importantly we need to document what a container is required to provide, beyond the lifecycle objects and lifestyle management. We need to describe the contracts for what is inside the lifecycle objects. h2. Meta Information Due to the experience with the Phoenix and Merlin containers, we learned the value of using Meta information to describe and validate our components. For this document, we don't want to make any distinction between Meta Information and [MetaData]. We only want to say that it is necessary to properly describe our components to the container. h2. Lifestyle We have learned that while we can only think of a handful of implementations for lifestyle, we do not want to limit ourselves to only those implementations. Our experience with ECM and Fortress brought to light the implementation choices of "singleton" components, pooled components, one-off components, and thread local components. The flexibility of just these four lifestyles allow for a wide range of deployment options. However, there are other lifestyles that will more than likely be necessary in different usage scenarios (like SEDA or request/response based environments). What we need to do is describe the particular environment that the component is designed to work within, and let the container decide what the proper lifestyle implementation should be. We need to look more into this, but so far the two major proposals on the table for this is to either infer it based on component _scope_, or based on lifestyle _properties_ (e.g. reusable, sharable, etc.). We need to figure out what is the most scalable solution. h2. Lifecycle We have several lifecycle interfaces and lifecycle objects that are required for Framework 4.x compliance. Some of those like Component and Composable are deprecated. Others like Context have come under recent fire due to insufficient description of what goes inside. There is even the idea of a unified namespace being thrown around like JNDI--but without its weight. Lastly, we are looking at extensible lifecycles as introduced by the \{ { \{ Merlin/Fortress \} } \} unified proposal and Phoenix's next-generation interceptor concepts. We need an extensible lifecycle so that we can easily adapt to new needs in a compatible manner. h3. Unified Namespace I think this idea has some serious merit. The Context concept is so generic it is really hard to nail down. The JNDI concept has a scalable namespace environment, but a heavy implementation. The [ServiceManager] concept is fairly well defined (in community concept if not in a document), but its namespace environment is fairly restricted. By "unified namespace" we do not mean that all components share the same namespace, we mean that we can get all information we need for a component from the same namespace. That will allow us to provide a namespace mechanism that binds configuration information, system artifacts, and services.  It should also allow the component itself to "rebind" or "write" information back to the namespace so that the container has the option of storing it for later use with the component. An example would be a component that writes its configuration back to the namespace so that it can either "self-heal" (i.e. upgrade its configuration elements to the latest non-deprecated format) or add new entries (i.e. for things like Apache Axis that needs to manage the SOAP message handlers that may have been added or removed during runtime). An example follows. {noformat} /** {{{ * LogEnabled carried forward. {noformat} {noformat} */ {noformat} public interface [LogEnabled] \{ {noformat} enableLogging(Logger logger); {noformat} \} /** {noformat} * LookUpEnabled, the merging of config/context/ {noformat} {noformat} * and component lookup. {noformat} {noformat} */ {noformat} public interface [LookupEnabled] \{ {noformat} enableLookup (ResourceManager manager); {noformat} \} /** {noformat} * ResourceManager maps the content of A4 lifecycle {noformat} {noformat} * artifacts like Context, Configuration, Parameters, {noformat} {noformat} * and ServiceManager to a unified namespace. {noformat} {noformat} */ {noformat} public interface [ResourceManager] \{ {noformat} Object lookup( String URN ) {noformat} {noformat} throws ResourceException; {noformat} {noformat} boolean isBound( String URN ); {noformat} {noformat} void rebind( String URN, Object value ) {noformat} {noformat} throws ResourceException; {noformat} \} }}} One strategy is to follow the Sun "Locator" pattern, which abstracts away the JNDI system and manages cached entries. The advantage is that the design pattern is documented elsewhere, we do not have to back it with JNDI for [J2ME] systems, and we have the option to back it with JNDI for [J2SE] or [J2EE] environments. The disadvantage is that the concept of "Locator" is biased for _finding_ things, not _managing_ things. Another strategy is to provide a simpler variant of JNDI much like JDOM did for the DOM package. The advantage here is that we have contextual purity, and we have the same advantages with JNDI as the "Locator" pattern. The disadvantage is that we are working with a proprietary interface, and not a pattern that has been documented elsewhere. Whatever solution we choose, we need to work together to describe its interface and contracts. We need to look at what things are *required* to be bound, and what things can be made optional. h3. Extensible Lifecycle Phoenix is the first container to experiment with extensible lifecycle. Later, Fortress and Merlin cooperated to develop a simple variant. The underlying motivation is that we want to be able to support user-defined lifecycle that is application specific in a manner that is not container dependent. In order to make a cross-container extensible lifecycle system work when lifecycle artifacts are required, we need to standardize *how* we provide those lifecycle artifacts. I propose that we establish an interface that exposes all the Avalon specific artifacts to the lifecycle management mechanism. The lifecycle management mechanism can in turn work with the standard lifecycle artifacts to take care of application specific lifecycle events. One method of accomplishing this is using a utility that compiles the actual logic to satisfy each lifecycle stage into a "handler" class. The "handler" class takes care of the creation, destruction, and access/release cycles of a component. The Lifecycle Artifacts interface describes to the [LifecycleManager] how to get the information to pass in to the Configurable interface for example. The Lifecycle Artifacts interface is implemented differently for each container. One container may use the [JavaBean] approach, storing unique values for each component. Another container may use that as an alias to itself and supply a common set of artifacts to all components. The [LifecycleManager] interface allows us to change the implementation details of how we create the managers and provide a standard way for the container to perform all of the functions with the manager. This allows us to experiment with the relatively simple Merlin/Fortress approach or the more complicated Phoenix/interceptor approach. It also allows us to plug in the logic once we have figured out how to make it work. I would like the actual creation/assembly of the [LifecycleArtifacts] type object and [LifecycleManager] type object to be done by the container itself. The container will use standard meta information and assembly information to perform the logic. There are a couple variations we can look at, such as one manager per component type or one for all components. I prefer a type based approach as it allows some containers to customize the manager for each type (so that no `instanceof` or casting needs to be done on initialization).