AspectsAndHandlers

WARNING: This is a big document. However, I have all the important stuff in the first page. The rest is basically just where I go on about some interesting consequences of the first page.

Background

All containers start out small and simple, but even though we try to Keep It Simple, it is a fact of life that complex things are, well, complex. Therefore the little container tends to grow like a pet alligator into a scary beast before you know it.

This proposal is an attempt to remedy that by allowing container features to be pluggable "aspects". The proposal is limited to the actual aspect architecture - the actual implementation of the aspects, what aspects we should have, and so on are out of scope. The proposal also deliberately avoids talking about any configuration format or metadata format. It is intended that everything at this level should be doable programatically -
if one wants to make the configuration driven by an XML configuration or any other data source, this is possible but not required.

Aspects

The basic idea is to split the container into three principal parts:

- The Handlers, which handle component instances.
- The Aspects, which provide container functionality across multiple Handlers.
- The Kernel, which routes events and connections between the Handlers and Aspects, and manages both.

The idea is then to have all functionality either implemented as an Aspect or a Handler. There are few contracts between the Kernel and the Aspects or Handlers, but the number of contracts between the Aspects and Handlers and between different Aspects may be numerous - but they are also optional.

For example, Avalon components will be handled by an AvalonHandler, which will get its logger from a LoggingAspect. The Handler will also get a Service Manager (that it can give to the component instance) from a ServiceManagerAspect.

Other Aspects are:

- Dependency validation
- Remoting
- Automatic downloading of components from a repository
- JMX Management
- ...

Handlers are:

- PicoHandler
- AvalonHandler
- POJOHandler

and so on...

Contracts Between Kernel and Aspects

The Kernel provides a Kernel interface to all Aspects.

```
interface Kernel {
    void addAspect (String name, Aspect aspect);
    Aspect getAspect (String name);
    void removeAspect (String name);
    String[] getAspects ();

    void addHandler (String name, Handler handler);
    void removeHandler (String name);
    Handler getHandler (String name);
    String[] getHandlers ();

    void   registerEventListener (Class eventInterface, Object listener);
    void   unregisterEventListener (Class eventInterface, Object listener);
    Iterator getEventListeners (Class eventInterface);
}
```

The kernel does not provide a Kernel interface to the Handlers. The handlers have to access the kernel via an Aspect if they want to.

Events

Events are defined via interfaces. For example:

```
interface HandlerEvents {
    public void handlerAdded(String name, Handler handler);
    public void handlerRemoved(String name, Handler handler);
}
```

A class wishing to listen for those two events will implement the interface and then call:

```
Kernel.registerEventListener( HandlerEvents.class, this );
```

The class will then have the appropriate methods called. A class wishing to produce events will get an Iterator over all listeners of a certain type:

```
Iterator iter = Kernel.getEventListeners( HandlerEvents.class );
```

The class can then send events by:

```
while (iter.hasNext ()) {
    ((HandlerEvents) iter.next()).handlerAdded("myHandler", handler);
}
```

The Aspect Interface

```
public interface Aspect {
    public void initAspect( Kernel kernel );
    public void disposeAspect();
}
```

Upon having initAspect called, the Aspect should apply itself to all existing Handlers and other Aspects where possible, and then wait for handlerAdded / aspectAdded events to apply itself to handlers and aspects that are added after the aspect itself.

Contracts Between Aspects

Apects can be applied to other aspects. For example, a Logging aspect can supply loggers to other aspects. The question then is, what if you have two Aspects, say SecurityAspect and LoggingAspect, and one of them have a dependency on the other - in this case, the SecurityAspect is dependent on the LoggingAspect for logging.

So if you add:

```
Kernel kernel = new DefaultKernel ();
kernel.addAspect ("logging", new LoggingAspect ());
kernel.addAspect ("security", new SecurityAspect ());
```

you are fine, since the SecurityAspect, upon being added, will receive a logger from the LoggingAspect. But what happens if you do:

```
kernel.addAspect ("security", new SecurityAspect ());
kernel.addAspect ("logging", new LoggingAspect ());
```

Since the SecurityAspect cannot wait for the LoggingAspect to be added, we have a problem.

The solution is to make all dependencies among Aspects optional. The SecurityAspect will have to default to a NullLogger (preferable) or ConsoleLogger (if you must). Then the SecurityAspect can start up, and then switch to a logger provided by the LoggingAspect. If a logger is required, the SecurityAspect is expected to deny all security checks until a logger is provided, or do something similar.

Contracts Between Aspects and Handlers

The Aspects work on interfaces exposed by the Handlers. That is, for each aspect, the handler can decide to support it or not. For example, in order to support a (hypothetical) DependencyValidationAspect, the handler must implement an interface that allows the aspect to determine what dependencies this handler has, and what it provides.

Just like other Aspects, all dependencies between Handlers and Aspects must be optional. If, for example, a LoggingAspect isn't available, then the Handler will have to default to a NullLogger.

Aspects can also communicate via Events and look each other up via the Kernel interface for aspect-to-aspect interaction defined by some suitable interface.

Aspects are selectively applied to Handlers. It is expected that a configuration file will list the Handlers that an Aspect must be applied to, the ones it may not be applied to, and the ones it should apply to if possible. For example, you may want to disable security checks for some handlers, and throw exceptions if the aspect could not be applied to some other handlers.

Handler

```
interface Handler {
    public Object get () throws Exception;
    public void release (Object o);
}
```

Handlers can't access each other. An Aspect is required for that.

Consequences

So what consequences does this arhcitecture lead to?

Avalon Micro, Standard, Enterprise

We have talked about different levels of containers, but this has always stalled since all functionality had to be implemented directly in the container. It therefore made little sense to offer a "lightweight" container, since such an offering would in effect mean a completely separate container that wasn't much smaller than the full-blown one.

However, with Aspects we can do this. Avalon Micro, for example comes with no aspects. Standard comes with a LoggingAspect and an AvalonComponentAspect. Enterprise has DependencyValidationAspect and SecurityAspect, and so on.

This means that Aspects and Handlers can target a set of Aspects.

Logging Implementation

Logging is an Aspect defined by one Aspect and a Handler interface:

```
interface LoggingHandler {
    public void setLogger (Logger logger);
}
```

The Aspect supplies a logger to all Handlers and Aspects that implement the interface.

Proxying and Interceptors

Proxying and Interceptors (in general) would be implemented via a Handler interface:

```
interface Proxying {
    public void addProxyProvider (ProxyProvider provider);
}

interface ProxyProvider {
    /**
     * Wraps objectToProxy in a proxy. Since objectToProxy may already
     * have passed through another ProxyProvider, we give you the
     * originalObject, which is the un-proxied instance (useful for
     * figuring out the original class of it etc.)
     */
    public Object proxy (Object objectToProxy, Object originalObject);
}
```

Every Handler that supports the creation of a proxy wrapper around the component instance implements the Proxying interface. The Aspect then registers itself as a ProxyProvider for each of those Handlers.

When the Handler is to produce a new component instance, it allows every registered ProxyProvider to wrap the instance in a proxy facade.