JavaBeansVsConfiguration

(this page is part of the wiki materials for ApacheAvalon; avalon main page in the wiki is AvalonProjectPages)

JavaBeans Vs. Configuration

One of the most common ways that developers work with JavaBeans is to use them as an interface for configuring a component. It makes sense because of all the tooling that comes for free with the java.beans package. Nevertheless there are a number of problems with the JavaBean approach, so they fall short on a number of points.

Avalon Framework has a Configuration and Parameters object that allows us to provide rich configuration options without a lot of overhead. Because there is a standard interface, all users of the Configuration and Parameters objects know how to use them directly. They are really strong on a number of points--mainly because they are focused on one purpose, storing settings for components.

JavaBeans

JavaBeans were Java's first stab at reusable components. The concept was pretty cool, but very muched tied to GUI programming. A JavaBean would expose a standard set of properties that were externally editable, and the client of the JavaBean would change the properties to make the bean perform the way that they wanted. Part of the process also allowed for *introspection*--the process of discovering the beans properties, events, and more by examining the exposed methods.

As a result, we can look at the JavaBeans setter/getter idiom as a form of managing configuration settings. Autodiscovery of properties allows the bean writer to add and remove configuration settings from a bean without destroying software that is already written. As an additional benefit, you have strongly typed properties for each configuration point. From the bean or component writer, there is alot of power in that process--but at an equally high price. Reading and assigning properties using introspection is anywhere between 9x and 15x slower than directly setting the properties.

To escape the necessity for introspection, bean writers tend to agree on a standard set of properties (like the javax.sql.DataSource specification) and then use introspection for vendor specific properties. The advantage of this approach is that if a bean does not support a property, it won't cease to work. Also, the client can directly call a number of methods that they *know* will exist--no matter what vendor supplies the beans.

Configuration

Avalon's Configuration and Parameters (from this point referred collectively as Configuration) are a very simple tool. Its power lies in its simplicity and its focus. A component can have strongly typed configuration elements, as long as they are one of Java's primitive types or a java.lang.String. In 99% of all cases, that is all a user needs.

Because of the Configuration process, components can easily read the settings and containers can easily manage the settings. More importantly, neither side of the equation (user or provider) needs to use introspection to ever set all the properties for a component. This reduces the *need*
for complex tooling, and increases productivity because we are not concerned with writing complex glue logic or wondering how the magick is performed. When we look at the code for the component and the code for the container, we know exactly what each is doing.

We can easily boil Avalon Configuration into a namespace for configuration points. The Configuration object (as opposed to the Parameters object) allows us to find settings in a heirarchical namespace. The Parameters object allows us to find settings in a flat namespace. Nevertheless, we are using names in either case to find what we want.

Pros and Cons

What is really better?

Pros for JavaBeans

- JavaBeans allow you to grab settings using method calls, and not working with a lot of strings.

```
That reduces the amount of temporary objects created.
```

- JavaBeans allow for rich hierarchical configurations as well as a simple flat set of properties.
- JavaBeans have a rich set of tools built into the language and provided by third parties.
- The MetaData for the properties is built into the bean class's bytecode.

```
That makes validation a matter of checking method signatures.
```

```
Nothing special has to be done.
```

Cons for JavaBeans

- Container side programming is made much more difficult because of introspection and autodiscovery of vendor specific properties.
- When tooling goes wrong, it goes very wrong.

```
If you don't catch all the potential runtime exceptions, you're application will crash.
```

- Introspection, while powerful, comes at a very high price (9x to 15x slower than simply calling a method).
- Hierarchical configuration is fairly difficult--you need more beans to capture the sub elements.
- Settings can be changed at any time, so settings that affect the runtime use of the bean require special concurrency management for those settings.

```
That may be even more difficult if a group of settings need to be made at one time.
```

Pros for Configuration

- Configuration gives you strong typing and a simple interface.
- Tooling is not necessary for the majority of cases.
- You get a heirarchical configuration space for free.
- Configuration happens at a defined time in the component's lifecycle, so settings won't have the potential to change at unpredictable times.

Cons for Configuration

- It is generic, so validation and verification is more difficult.

```
It requires external MetaData to perform its functions.
```

- It requires the use of temporary objects (Strings) to get and set properties.

```
That creates more work for the garbage collector.
```

- It is generic, so something that would take one call in the JavaBean approach might take two or three calls for Configuration.

What Should We Do About It?

Should we make all Configuration based on a namespace? Should we look at the use of JavaBeans as dumb configuration objects? The latter approach is used by Tomcat and other projects. How do we make it *easy* for a component writer to configure their components, and for the container writer to verify the configuration is valid?