ButterflyManifesto

The Butterfly Manifesto (DRAFT)

This is a first draft. Comments are welcome (isn't there a comment feature in MoinMoin?). If you agree or want to contribute, please undersign.

Introduction

We all know Cocoon has some problems. I am not referring here to bugs, poor documentation or a steep learning curve for beginners. I am talking about structural deficiencies that make it hard to test it, refactor it, deploy it and manage it.

The most distressing fact, for us developers, is that it's hard to **change it**. We want to have real blocks for the next major release, but it seems quite improbable that we'll get there with the current Avalon-based architecture. Meanwhile, Avalon continues to be plagued by community problems. This was quite clearly argumented by Stefano on the On building on stone thread.

While Avaloners were fighting and flaming, a new breed of lightweight containers based on a form of Inversion of Control called Dependency Injection saw the light. Among them: PicoContainer, HiveMind and Spring.

What is Butterfly

Butterfly is an experiment aiming to implement a (simplified) Cocoon clone but based on Spring instead of Avalon (hence the name Butterfly: what do cocoons turn into in spring?). **This is not a fork of Cocoon**. This is a sandbox where new ideas can be tested without fear of breaking some eggs. If something valid emerges, we can discuss if it can be backported to Cocoon or not. In any case, we won't be reimplementing the whole of Cocoon. Even though Stefano suggested doing our own container, we would like to avoid doing more work than necessary and reuse something existing, if possible. Our initial choice is to use Spring, as it:

- has a healthy community
- is very well documented
- is rapidly growing in acceptance and usage
- is being used in real world applications
- aims to do one thing and do it well and does not attempt to be everything for everyone
- integrates well with other useful technologies (Hibernate, JDO, OJB, EJB, Struts, JMS, AOP ...) instead of trying to replace them
- fits perfectly in J2EE environments, where we expect Cocoon to be often deployed.

Build or buy?

Even if the initial implementation will use Spring, some valid concerns have been raised on the developers' mailing list. Basing our core on a framework that is being developed elsewhere carries a certain amount of risk. If the Spring community evaporates, goes down in flames like the Avalon one or takes a road that is incompatible with our needs, we risk finding ourselves in the same situation we are now. Moreover, Spring is not an ASF project.

On the other hand, consider:

1. The benefits that Spring can give.
2. The fact that Spring's core is relatively small and thus maintaining a fork wouldn't be unthinkable.
3. The fact that Spring's design promotes a design whereby components do not have dependencies upon the framework and thus could be moved to a different container, as long as it supports Type 2 and Type 3 IoC, at least in principle.
4. At least one other Apache project (JetSpeed 2) now uses Spring, and Apache Geronimo will include support for Spring.
5. The Spring community is presently large and growing, so its viability looks good.

Design principles

1. Simplicity
2. Avoid dependencies
3. Testability
4. Use the standards
5. Courage

Practices

Practices embody the above-mentioned design principles in a set of practical guidelines.

Avoid unnecessary dependencies (and excessive coupling)

Following the Spring mantra, components should not depend upon the framework, for instance by implementing interfaces or extending classes provided by the framework, unless this provides real value and is hidden in implementation classes.

This allows for greater portability and testability of components in isolation.

Importing framework packages in components' source code is a sign of dependency. Doing it in glue code is OK.

Remove any possible dependency on Avalon. If you want to reuse some of its functionalities, copy classes over and refactor them as necessary (see org.apache.butterfly.source package).

Limit reuse by implementation inheritance

Case in point: most of Cocoon's sitemap components inherit from AbstractLog'***Enabled. This must go away: logging is an aspect, not a more general concept. In other words, a FileGenerator is not an AbstractLog***'Enabled.

Another symptom of this is very deep inheritance hierarchies. We should prefer delegation or composition over inheritance whenever we want to reuse.

Use standard JDK packages whenever possible

Use JAXP and TrAX for processing XML.

We can debate whether to use JDK logging or Log4J (or Commons Logging) instead. Since Spring uses the latter and everybody says JDK logging sucks, we should use it too, probably. If necessary, provide a thin wrapper around third-party APIs for adaptation (see org.apache.butterfly.xml.Parser).

Don't be dragged down by backward compatibility

We are exploring new ground, so we should be courageous and not be too limited by considerations of backward compatibility.

JDK 1.4 is a minimum requirement.

Strive for 100% unit test coverage

Code that isn't exercised by tests is not there.

Avoid checked exceptions

Checked exceptions do more harm than good, so it would be wise to avoid them altogether. Design instead a hierarchy of runtime exceptions and, whenever you need to catch an exception thrown by a third party library, decide whether you can handle it locally. If you cannot, wrap it in a runtime exception and throw the latter.

Niclas Hedhman: If everyone follows this rule, then we don't know when there are exceptions to catch, no situations you try to recover from, and we will have runtime unstable systems. Sorry, I don't buy this argument for a second. Programmers need to do there job, not ignore it for the 'time being' as lazy developers will definately do otherwise. Perhaps you can get away with this in a request/response system like Cocoon, but a general recommendation that nothing should throw checked Exception is foolish.

Ugo: I invite everyone to read what others have to say on the subject, then make up your mind:

- http://www.mindview.net/Etc/Discussions/CheckedExceptions
- http://www.artima.com/intv/jdom3.html
- http://www.artima.com/intv/handcuffs.html
- http://www.theserverside.com/articles/article.tss?l=RodJohnsonInterview (read inside the PDF of the sample chapter)
- http://www.artima.com/intv/solid.html

After this, take a careful look at Cocoon's code base and count how many exceptions are swallowed or rethrown as unrelated kind of exceptions (with the ensuing mile-long stacktraces). It's not that checked exceptions are bad in theory. It's that in practice very few developers deal with them sensibly.

Vadim: I'd much prefer to first deal sensibly with what we got, instead of making blanket statements "checked exceptions are bad". As a counterexample, take a look at TransformerImpl, lines 3394 and 3418.

- http://cvs.apache.org/viewcvs.cgi/xml-xalan/java/src/org/apache/xalan/transformer/TransformerImpl.java?annotate=1.158

Code

The Subversion repository for Butterfly is here.

At the moment we have implemented ridiculously simplified versions of:

- FileGenerator
- TraxTransformer
- XMLSerializer
- NonCaching_'_'ProcessingPipeline

plus some parts of the Source package from Excalibur. This can at least allow us to compose a simple pipeline. Most of the features of the above mentioned components have intentionally been dropped, since our focus is presently on testing how sitemap components can be managed by a Spring bean factory.

You can run this code via the respective unit tests in the `src/test` directory.

As of Aug 4th 2004, there's an Ant build file that deploys a web application and the same simplified instance of Jetty that is used by Cocoon is available to run it. So, after having checked out the source, build it with `ant`, run it with `./butterfly.sh servlet` or `butterfly.bat servlet` and surf to http://localhost:8888/.

Javascript beans

The version of Spring included in Butterfly is a preview version with support for implementing beans in Javascript, using Mozilla Rhino. Hopefully this change will make it into Spring 1.2.

With this we could have, for instance, container-managed flowscripts who get their dependencies injected to them instead of having to look them up. Real IoC in your flowscripts! We could also use Spring's AOP to apply aspects to them, so that you might, for instance, transparently apply declarative authorization checks to your flowscripts or wrap transactions around them.

References

1. On building on stone
2. Serviceable considered harmful
3. Checked exceptions considered harmful
4. Spring+JMX == Real Blocks?

First version drafted by UgoCei on July 20th 2004.

This sounds interesting. Please ask for help on the Spring lists, or contact me (rod.johnson at interface21.com) if you need any help. For example, from a quick look at the list discussions, pooling came up: Spring can provide pooling transparently. – Rod Johnson