XML or Annotations in Spring

Annotations have their use, but they are not the one silver bullet to kill XML configuration. I recommend mixing the two!

For instance, if using Spring, it is entirely intuitive to use XML for the dependency injection portion of your application. This gets the code's dependencies away from the code which will be using it, by contrast, using some sort of annotation in the code that needs the dependencies makes the code aware of this automatic configuration.

However, instead of using XML for transactional management, marking a method as transactional with an annotation makes perfect sense, since this is information a programmer would probably wish to know. But that an interface is going to be injected as a SubtypeY instead of a SubtypeX should not be included in the class, because if now you wish to inject SubtypeX, you have to change your code, whereas you had an interface contract before anyways, so with XML, you would just need to change the XML mappings and it is fairly quick and painless to do so.

I haven't used JPA annotations, so I don't know how good they are, but I would argue that leaving the mapping of beans to the database in XML is also good, as the object shouldn't care where its information came from, it should just care what it can do with its information. But if you like JPA (I don't have any expirience with it), by all means, go for it.

In general: If an annotation provides functionality and acts as a comment in and of itself, and doesn't tie the code down to some specific process in order to function normally without this annotation, then go for annotations. For example, a transactional method marked as being transactional does not kill its operating logic, and serves as a good code-level comment as well. Otherwise, this information is probably best expressed as XML, because although it will eventually affect how the code operates, it won't change the main functionality of the code, and hence doesn't belong in the source files.

There is a wider issue here, that of externalised vs inlined meta-data. If your object model is only ever going to persisted in one way, then inlined meta-data (i.e. annotations) are more compact and readable.

If, however, your object model was reused in different applications in such a way that each application wanted to persist the model in different ways, then externalising the meta-data (i.e. XML descriptors) becomes more appropriate.

Neither one is better, and so both are supported, although annotations are more fashionable. As a result, new hair-on-fire frameworks like JPA tend to put more emphasis on them. More mature APIs like native Hibernate offer both, because it's known that neither one is enough.

I always think about annotations as some kind of indicator of what a class is capable of, or how it interacts with others.

Spring XML configuration on the other hand to me is just that, configuration

For instance, information about the ip and port of a proxy, is definetly going into an XML file, it is the runtime configuration.

Using @Autowire, @Element to indicate the framework what to do with the class is good use of annotations.

Putting the URL into the @Webservice annotation is bad style.

But this is just my oppinion. The line between interaction and configuration is not allways clear

I've been using Spring for a few years now and the amount of XML that was required was definitely getting tedious. Between the new XML schemas and annotation support in Spring 2.5 I usually do these things:

1. Using "component-scan" to autoload classes which use @Repository, @Service or @Component. I usually give every bean a name and then wire them together using @Resource. I find that this plumbing doesn't change very often so annotations make sense.
2. Using the "aop" namespace for all AOP. This really works great. I still use it for transactions too because putting @Transactional all over the place is kind of a drag. You can create named pointcuts for methods on any service or repository and very quickly apply the advice.
3. I use LocalContainerEntityManagerFactoryBean along with HibernateJpaVendorAdapter to configure Hibernate. This lets Hibernate easily auto-discover @Entity classes on the classpath. Then I create a named SessionFactory bean using "factory-bean" and "factory-method" referring to the LCEMFB.
4. I think that visibility is a big win with an XML based approach. I find that the XML isn't really that bad, given the various tools out there for navigating XML documents (i.e. Visual Studio + ReSharper's File Structure window).
5. You can certainly take a mixed approach, but that seems dangerous to me if only because, potentially, it would make it difficult for new developers on a project to figure out where different objects are configured or mapped.
6. I don't know; in the end XML Hell doesn't seem all that bad to me.

It depends on what everything you want to configure, because there are some options that cannot be configured with anotations. If we see it from the side of annotations:

* plus: annotations are less talky
* minus: annotations are less visible

It's up to you what is more important...

In general I would recommend to choose one way and use it all over some closed part of product...

(with some exceptions: eg if you choose XML based configurations, it's ok to use @Autowire annotation. It's mixing, but this one helps both readability and maintainability)

I might be wrong, but I thought Annotations (as in Java's @Tag and C#'s [Attribute]) were a compile-time option, and XML was a run-time option. That to me says the are not equivalent and have different pros and cons.

I also think a mix is the best thing, but it also depends on the type of configuration parameters. I'm working on a Seam project which also uses Spring and I usually deploy it to different development and test servers. So I have split:

* Server specific configuration (Like absolute paths to resources on server): Spring XML file
* Injecting beans as members of other beans (or reusing a Spring XML defined value in many beans): Annotations

The key difference is that you don't have to recompile the code for all changing server-specific configurations, just edit the xml file. There's also the advantage that some configuration changes can be done by team members who don't understand all the code involved.

An important part in using an annotation-only approach is that the concept of a "bean name" more or less goes away (becomes insignificant).

The "bean names" in Spring form an additional level of abstraction over the implementing classes. With XML beans are defined and referenced relative to their bean name. With annotations they are referenced by their class/interface. (Although the bean name exists, you do not need to know it)

I strongly believe that getting rid of superfluous abstractions simplifies systems and improves productivity. For *large* projects I think the gains by getting rid of XML can be substantial.

In the scope of DI container, I consider annotation based DI is abusing the use of Java annotation. By saying that, I don't recommend to use it widely in your project. If your project does really needs the power of DI container, I would recommend to use Spring IoC with Xml based configuration option.

If it is just for a sake of Unit-test, developers should apply Dependency Inject pattern in their coding and take advantages from mocking tools such as EasyMock or JMock to circumvent dependencies.

You should try to avoid using DI container in its wrong context.

This is the classic 'Configuration versus Convention' question. Personal taste dictates the answer in most cases. However, personally I prefer Configuration (i.e. XML based) over Convention. IMO IDE's are sufficiently robust enough to overcome some of the XML hell people often associate w/ the building and maintaining an XML based approach. In the end, I find the benefits of Configuration (such as building utilities to build, maintain and deploy the XML config file) outweighs Convention in the long run.

I use both. Mostly XML, but when I have a bunch of beans that inherit from a common class and have common properties, I use annotations for those, in the superclass, so I don't have to set the same properties for each bean. Because I'm a bit of a control freak, I use @Resource(name="referredBean") instead of just autowiring stuff (and save myself a lot of trouble if I ever need another bean of the same class as the original referredBean).

**Advantages of the annotation:**

1) All the information is in a single file (no need to open two files to configure a given behavior)   
2) When the class changes, no need to modify the xml file

**Advantages of xml file:**

1) Clear separation between the POJO and its behavior   
2) When you do not know which POJO is responsible for the behavior, it is easier to find that POJO (searching in a subset of files rather than all the source code)