

Supporting Hacks and Versions

Supporting Hacks and Versions

★ Architecture Astronauts Alert -If you have a low BS tolerance tune out now 😊

If you want a hint at how geotools plans to support the next big thing - that is easy we write it!

It is the next small thing that has me worried - a lot of the services we use are changing on us, and they are changing versions. How are we to write classes that support both cool shiny new stuff (tm) and last years fashions?

And is there a difference between hackers and the spec writers? Does the tail wag the dog?

FooBar 1.0

To start with lets take a simple example:

```
interface FooBar {
    boolean isGood( Geometry geom );
}
```

And we can assume the usual suspects: FooBarFactory and FooBarFactoryFinder.

Let's say some time passes and a new version 2 comes out that has support for morals, morality is given a range of 0-1 (we are computer scientists after all).

We would like to add the following:

```
interface FooBar {
    boolean isGood( Geometry geom );

    /** @since version 2. */
    double getMorality();

    /** @since version 2. */
    void setMorality( double morality );
}
```

But if we do that we lose something...

Using Javadocs

If we make the above change we are now dependent on programmers to read the javadocs, and we all know how well that goes. I tend to use geotools as a binary and hope for the best, especially given the source code releases.

Let's look for something better.

Using Interfaces

What I would really like to preserve is Strong Typing so the compiler (not me) can worry about the different versions.

```
interface FooBar implements FooBarVersion1,
FooBarVersion2 {
    boolean isLabelShield();
}
interface FooBarVersion1 {
    boolean isGood( Geometry geom );
}
interface FooBarVersion2 {
    double getMorality();
    void setMorality( double morality );
}
```

This approach makes everything explicit, and allows for fun little non standard geotools hacks (like LabelShield above).

We have a number of benefits, code that is really limited to Version 1 can say so with FooBarVersion1 interfaces. Code that does not care can track the FooBar baseline. Seems like a winner to me...

Using Factory

So what is my hesitation? Usability let's look at how this plays out, and see what we can do to make it worthwhile...

We can make explicit factories for each version:

```
interface FooBarVersion1Factory {
    FooBarVersion1 create(); // default
implementation
}
interface FooBarVersion2Factory
    FooBarVersion2 create( double morality );
// default implementation w/ custom morals
}
```

Our default use of Factory should still work:

```
interface FooBarFactory extends
    FooBarVersion1Factory, FooBarVersion2Factory
{
    FooBar create(); // default
implementation
    FooBar create( double morality ); //
default implementation w/ custom morals
}
```

Is it Good?

So we should be able to answer the question - is this a good design idea?

Lets see how we do?

```
FooBar hacker = FooBarFactoryFinder.create(
0.9 ); // hackers need a bit of flexibility
FooBarVersion1 greenshag =
FooBarVersion1Finder.create(); // They were
going to get it right the first time
FooBarVersion2 black =
FooBarVersion2Finder.create( 0.98 ); // They
are pretty sure they got it right
```

Seriously, this can work. It will let me support Filter 1.0 and Filter 1.1, SLD 1.0 and SLD 1.1 and by extension we can make strongly typed interfaces for WFS1.0 and WFS1.1 clients.

I want to balance:

- end user frustration, making a Filter and not having it work would be annoying
- developer frustration, having a great idea (by definition non standard) and not getting your patch in would be annoying

By extension our GeoTools interfaces (and implementations) will need to step back from the standards and be bigger than them. For examples of how to do this look at the WebMapServer implementation that Richard Gould put together.

Can J2SE 1.5 annotations work?

The *annotation processing tool* bundled with J2SE 1.5 may be an alternative worth to explore. First, let's note that Java 2 Standard Edition faced a similar problem with JDBC. Some drivers are JDBC 2 compliant, while some other drivers are JDBC 3 compliant. Java uses the same set of interfaces for all versions; the JDBC versions seem to be handled through documentation.

I would like to avoid the extra complexity of multi-set of interfaces for each Filter versions. But this is true that a "javadoc only" solution is not type-safe. However, we can get something stronger than javadoc (while similar in principle) with J2SE 1.5 annotations (reminder: annotations are erased in the J2SE 1.4 profile of GeoAPI). I propose to define a `@Version` annotation and uses it in every methods declared in Filter interfaces. So our source directory would contain a single set of interfaces with compile time information about versions.

The next step would be to write a processor for the annotation processing tool (`apt`). Our processor would ensure that only Filter 1.0 methods are invoked in code that wants to be 1.0 compatible. There is different ways to achieve this task. The easiest way may be to generate a new set of interfaces which contains only the 1.0 filter methods. This is similar to the multi-set of interfaces described above, except that those interfaces would be automatically generated using the `@Version` information. The `apt` tool is designed exactly for this kind of task.

A better long-term approach may be to keep only a single set of interfaces, and write a `javac` processor that print warnings at compile time when a Filter 1.1 method is used in a 1.0 compatible code. I have not explored this path, but I believe that this is among the scope of annotations. Anyway, using annotations allows both approaches (automatically generated 1.0 interfaces for now, compile-time warning later). An open question is whatever or not

the automatically generated 1.0 interfaces should be included in the standard GeoAPI releases. My feeling is that they should not (in order to propose the compile-time warning approach as the "official" one at some later stage), but we can provide them as non-official extension.

The annotation processing tool is described there: <http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html>

What it may looks like

For API developers, an interface declaration looks like:

```
@XmlSchema( specification=OGC_02059,  
version="1.0" )  
void foo();
```

This annotation appears in the javadoc, which means that this approach already meet the first proposal (deal with versions in documentation, see *Using javadocs* above). In addition, a set of filter 1.0 or filter 1.1 interfaces can be automatically derived, which means that the second proposal (see *Using interfaces* above) can be meet as well. Those interfaces are generated by the API developers with the `apt` tool (Geotools users don't need to care; it should be part of the Maven build process). The `apt` tool compiles source code in `.class` files exactly like `javac`. It accepts exactly the same command line options, together with some additional ones. The 1.0 interfaces are automatically created and compiled from the 1.1 interfaces at compile time.

The last approach (compile-time warnings) need more investigation. In any case, the consequence on interface definitions is identical in all case. Developers just need to provide some annotation like the `@XmlSchema` annotation above, and the door is open for all usages (documentation, interfaces or compile-time warnings).

For Geotools users, the consequences depend on what we choose to do with annotations. It may be documentation only, in which case the user is responsible for ensuring that he doesn't use a filter 1.1 method in a code targeted for filter 1.0 schema. This documentation approach may be completed later by compiler warnings.

The second solution (*Using interfaces*) is the most disruptive one. Users need to import the interfaces from the right package. Except for the `import` statements, we don't expect any special action needed in users' code.