FeatureCollection for Java 5

Motivation:	To come up with a better api for FeatureCollection
Contact:	Justin Deoliveira
Tracker:	http://jira.codehaus.org/browse/GEOT-1922 Additional tasks needed

This page is being placed under GeoTools 2.5 to reflect what actually happened; further clean up of FeatureCollection is of course possible and this proposal may be used as a reference point.

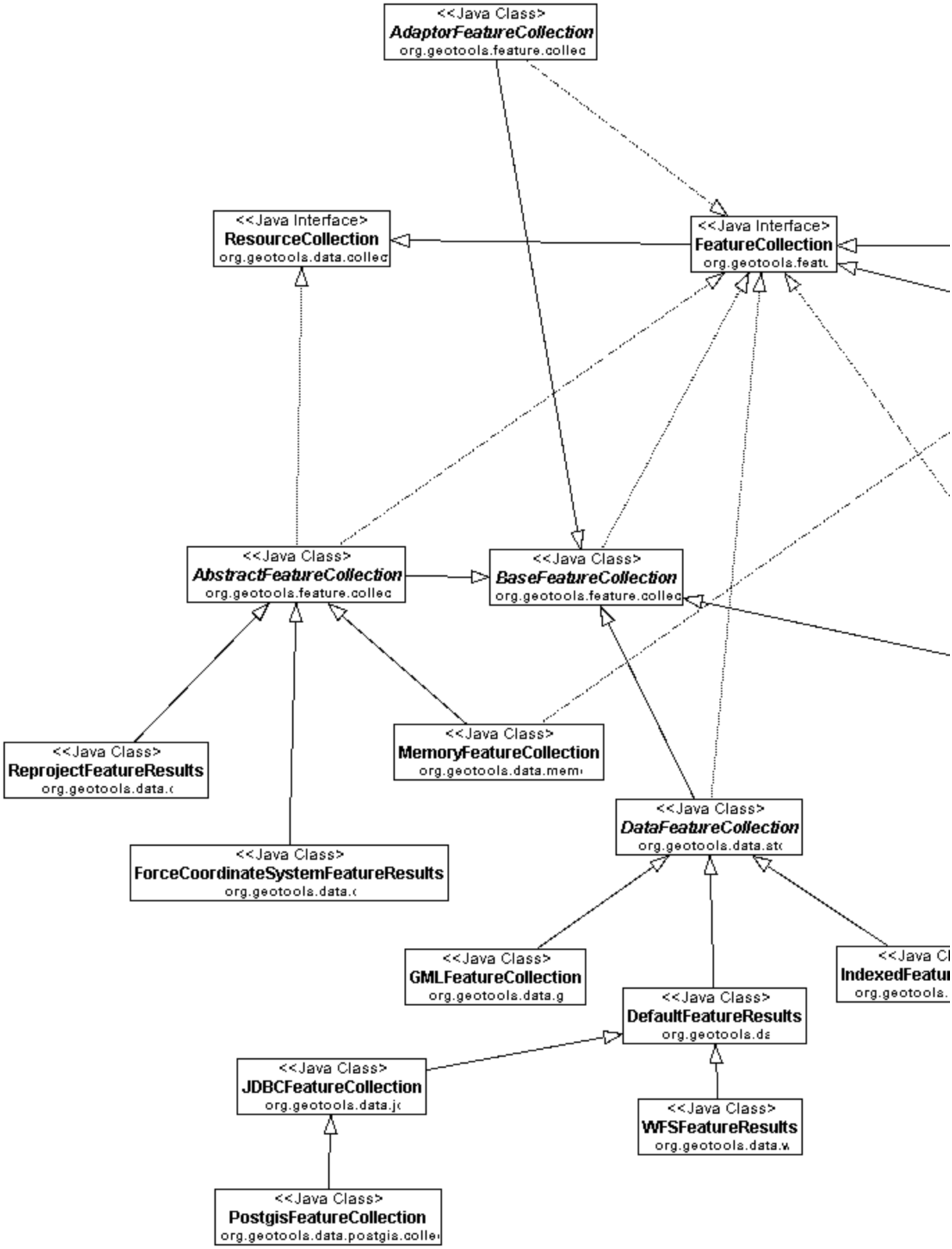
Children:

Description

Over the past two years, the FeatureCollection interface has been a constant source of problems. The major ones being:

- It implements both Collection and Feature, making it too complex and hard to implement correctly
- The Collection api is biased to an in memory data model, not suitable for spatial data formats

People have tried to implement an abstract base implementation of feature collection, but this has more or less been a failed attempt and now we are left with three different hierarchies of feature collections. The following is the class diagram:


<<Java Class>>
AdaptorFeatureCollection
org.geotools.feature.collec
<<Java Interface>>
ResourceCollection
org.geotools.data.collec
<<Java Interface>>
FeatureCollection
org.geotools.featu
<<Java Class>>
AbstractFeatureCollection
org.geotools.feature.collec
<<Java Class>>
BaseFeatureCollection
org.geotools.feature.collec
<<Java Class>>
ReprojectFeatureResults
org.geotools.data.(
<<Java Class>>
MemoryFeatureCollection
org.geotools.data.mem
<<Java Class>>
ForceCoordinateSystemFeatureResults
org.geotools.data.(
<<Java Class>>
DataFeatureCollection
org.geotools.data.st(
<<Java Class>>
GMLFeatureCollection
org.geotools.data.g
<<Java Class>>
DefaultFeatureResults
org.geotools.da
<<Java Class>>
JDBCFeatureCollection
org.geotools.data.j(
<<Java Class>>
WFSFeatureResults
org.geotools.data.w
<<Java Class>>
PostgisFeatureCollection
org.geotools.data.postgis.colle

Not very appealing. To add insult to injury it gets worse. FeatureCollection was come up with to replace FeatureReader and FeatureWriter. The motivation was the following:

- The Collection api is already a well known one to java developers
- Extended flexibility over reader / writer api, things like sorting, reprojection, etc..

However the reality of the situation is that because FeatureCollection is such a mess it has not been a good replacement and we have two data access apis: Feature Reader/Writer and FeatureCollection.

This is a proposal to fix all this. The primary goal of which is to come up with a simple as possible api in which people can actually implement.

Status

This proposal was never completed and voted on; the work required to
prevent FeatureCollection being used by Java 5 "for each" syntax was completed as documented by the tasks section.

Proposal

The FeatureCollection interface can be made much simpler by removing the restriction that a FeatureCollection be a Feature. This eliminates over half of the methods one must implement. The only downside of this is that it brings us away from the idea of a GML FeatureCollection. So users that actually wish to model a FeatureCollection as it is in GML are at a loss. However there is hope for such users as it they can simply just model it as a regular Feature, which has a Collection of features as an argument. Association vs Inheritance.

The second thing that doesn't quite fit is the fact that a FeatureCollection implements java Collection. Some of hte methods from the java Collection interface do make sense such as:

- `iterator`
- `add`
- `size`

But many do not. While a case can probably me made for all the methods on the Collection interface the reality is that implementors just do not implement them, either leaving them as stubs, or throwing UnsupportedOperationException.

Furthermore, the methods that do make sense do not allow one to throw IOException. A FeatureCollection is meant to be something very close to the actual format of the data, so it has to do I/O. Not being able to throw IOException in these methods means that implementors have to wrap all code in try catch logic and then just throw some runtime exception, which users probably are not handling since its unchecked.

So we remove the restriction that a FeatureCollection be a java Collection and add just that api that we need. The following is what we come up with:

FeatureCollection

```
interface FeatureCollection {

    /** The type of the features in the collection. */
    SimpleFeatureType getMemberType() throws IOException;

    /** The bounds of the feature collection */
    ReferencedEnvelope getBounds() throws IOException;

    /** The number of features in the feature collection */
    int size() throws IOException;

    /** An iterator over the contents of the feature collection. */
    FeatureIterator iterator() throws IOException;

    /** Accepts a visitor over the contents of the feature collection. */
    void accept( FeatureVisitor visitor, ProgressListener listener );

    /** Adds a feature to the feature collection. */
    void add( SimpleFeature feature ) throws IOException;

    /** Removes a feature from the feature collection */
    void remove( SimpleFeature feature ) throws IOException;

    /** Clears the contents of the feature collection.
    void clear() throws IOException;

    /** Disposes of the feature collection */
    void dispose() throws IOException;
}
```

The above is a minimal implementation. In the following sections are things we may wish to add to the interface and are open to feedback and discussion.

 See the Backward Compatibility section for additional notes on maintaining backwards compatibility.

Filtering

The ability to filter a feature collection on the fly:

```
  FeatureCollection subCollection( Filter filter );
```

Sorting

The ability to sort a feature collection on the fly:

```
  FeatureCollection sort( SortBy sortBy );
```

Write Access

Using an iterator from a FeatureCollection is a alternative to using a FeatureReader. And there are indeed many implementations which back a FeatureReader directly onto a FeatureCollection iterator, and vice versa. However there is no such alternative to a FeatureWriter.

While the FeatureCollection interface provides some api for modification via `add`, `remove`, and `clear` its incomplete. What would be nice is the ability to have an iterator that can be used for writing.

```
  FeatureIterator writer();
```

This would also involve adding some methods to the FeatureIterator interface, namely `remove` and `write`.

```
  interface FeatureIterator {

     /** remove the current feature from underlying storage */
     void remove() throws IOException;

     /** persist any changes to the current feature to underlying storage */
     void write() throws IOException;
  }
```

So what would this look like to client code. Well it would look very much like using a FeatureWriter:

```
FeatureCollection features = ...;
FeatureIterator writer = features.writer();

while( writer.hasNext() ) {
  SimpleFeature feature = writer.next();

  //set some attributes
  feature.setAttribute( 0, ... );
  feature.setAttribute( 1, ... );

  //write the feature
  feature.write();
}
```

What about inserting new features? The same FeatureIterator api can be used but another method on FeatureCollection is necessary:

```
  FeatureIterator appender();
```

Or alternatively a flag to the `writer` method:

```
  FeatureIterator writer( int mode );
```

Regardless, inserting new features uses the same FeatureIterator api:

```
FeatureCollection features = ...;

FeatureIterator writer = features.appender();

//OR

FeatureIterator writer = features.writer( FeatureCollection.INSERT );

while( writer.hasNext() ) {
  SimpleFeature feature = writer.next();

  //set some attributes
  feature.setAttribute( 0, ... );
  feature.setAttribute( 1, ... );

  //insert the feature
  feature.write();
}
```

Random Access

Ability to index a feature collection, as an extension:

```
interface RandomAccessFeatureCollection extends FeatureCollection {

  /** gets a feature from the feature collection by index */
  SimpleFeature get( int index ) throws IOException;
}
```

contains()

The proposed FeatureCollection interface above does not include the `contains` method. A case could be made for this method providing a quick way to do the lookup of a feature. An implementation could load the feature from storage by fid and then do the equals comparison.

```
boolean contains( SimpleFeature feature );
```

Backward Compatibility

Client API

As with any change, we would like to minimize the amount of client api breakage. For the most part we can follow a standard deprecate remove cycle. However there no explicit way of deprecating the interfaces that it extends, so this will be a breakage. However impact on client code should be minimized because:

1. We maintain the signatures of the methods from Collection that apply
2. There is not much client code that actually uses FeatureCollection as a Feature

As an experiment in my eclipse ide, I modified the FeatureCollection removing the extension of Collection and Feature. Compilation resulted

errors due to the absense of the following methods:

- `size()`
- `add()`
- `clear()`
- `iterator()`
- `remove()`

All of which are to be included in the new FeatureCollection interface. There were some additional compile errors:

- `contains()`
  A handful of references to this method, but only from test cases. Including this method in the new FeatureCollection is currently under dis cussion.
- `toArray()`
  In the validation module, in a number of test cases.

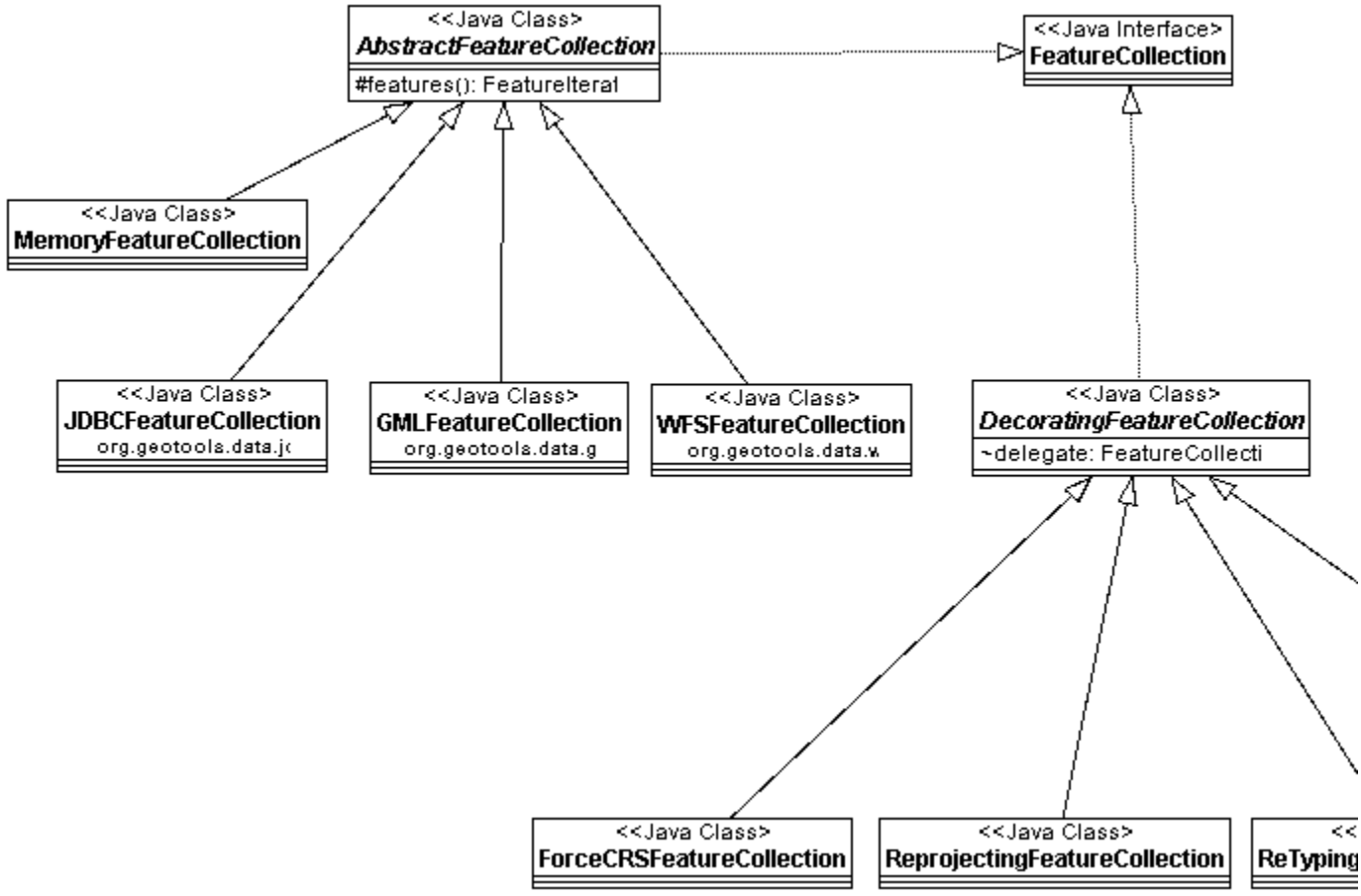
There is a single case of a FeatureCollection being treated as a Collection in the render module. In this case it is being used to render straight POJO's, objects that do not implement feature. Although this was more or less a failed experiment, some code still lurks.

There is a single case of a FeatureCollection being treated as a Feature in the wfs module. It is in a single test case which does not actually appear to do anything with the resulting type, just calls the method.

So the bottom line is that impact on client code is negligible. And any api we do end up breaking we can cleanly with a deprecation cycle.

Internal API

Part of this proposal is cleaning up the internal api. As stated above the internal api for feature collections is a mess. Ideally, we would like to transform the class diagram shown above into this:



Unfortunately getting to the above picture is where most breakages occur. All of the implementations of FeatureCollection in geotools will have to be modified to fit into this picture. Also, this is where most breakages will occur in uDig and GeoServer.

So the bottom line here is that if you have implemented the FeatureCollection interface... you are screwed. However, the simpler FeatureCollection interface allows to actually come up with a decent AbstractFeatureCollection class which all the implementations can extend:

AbstractFeatureCollection

```
abstract class AbstractFeatureCollection implements FeatureCollection {

  /** return the number of features in the collection */
  abstract int size();

  /** return the bounds of the collection */
  abstract ReferencedEnvelope getBounds();

  /** create an iterator over the features in the collection */
  protected abstract FeatureIteartor createFeatureIterator() throws IOException;
}
```

Subclasses have up to 3 responsibilities:

1. Creating the iterator
2. Performing an optimized query for the number of features
3. Performing an optimized query for the spatial extent

All the rest of the operations are performed in terms of iterators. Even for `size()` and `getBounds()`, default implementation might even be implemented by the base class which fall back on creating an iterator and calculating manually.

Tasks

GeoTools 2.5 - the follow tasks are needed right now to prevent users tripping over the Java for/each syntax (see: GEOT-1922)

- FeatureCollection extends ResourceCollection does not extends java.util.Collection
- Deprecate unused java.util.Collection methods left in ResourceCollection; as measued by not being implemented
- If FeatureCollection is the only implementation of ResoruceCollection merge the two interfaces
- Stop FeatureCollection extending SimpleFeature; cleaning up any utility classes like FeatureState