

# GeoGit DataStore

This is an RnD page as we explore design ideas for a GeoGit DataStore.

- [GeoGitDataStore](#)
  - [GeoGitFacade](#)
  - [Director or Index BLOB](#)
- [DataStore Delegate Design](#)
  - [Pure DataStore Wrapper](#)
  - [JDBCDataStore Wrapper](#)
    - [JDBCDataStore Interface](#)
    - [SQLDataStore Abstract Class](#)
  - [RevisionSupport Strategy Object](#)
  - [getJDBCDataStore Wrapper](#)

Reference:

- <http://geoserver.org/display/GEOS/GeoGit+approach>
- <https://github.com/opegeo/GeoGit>

This page is backed by an implementation on a [github branch](#) "markles":

- <http://www.github.com/markles/geotools>

[Walter Deane](#) is doing the work; with backup from [Mark Leslie](#) and [Jody Garnett](#).

## Open Questions

- Schema Definition (as it changes over time)
- "Directory" Access is a work in progress (it needs to cover more than just a list of Name; as the names published change over time)
  - Ability to Browse / Explore GeoGit contents prior to checkout
  - We may need a separate data model to track the ResourceIDs
- Having a lot of trouble wrapping JDBCDataStore cleanly and efficiently

## GeoGitDataStore

The design of the GeoGit DataStore is pretty straight forward:

- It extends ContentDatastore
- It maintains an internal "delegate" DataStore which is used as the "workspace" to hold the current checkout
- It maintains a GeoGit class and uses it to access the BLOB structures maintained in BerklyDB (or other Key Value database). To make common tasks easier a GeoGitFacade class with utility methods for performing common blobbing tasks (such as access an index BLOB describing the contents).

## GeoGitFacade

This section will be filled in with notes on the GeoGitFacade class; currently the design is straight forward.

```
package org.geotools.jdbc.versioning.geogit;
```

```

public class GeoGitFacade {
    public GeoGitFacade(GeoGit ggit) {
    }
    protected ObjectId insertAndAdd(Feature
f) throws Exception {    }
    public Feature getFeature(String id){
    }
    public Collection<FeatureType>
getFeatureTypes(){    }
    public Collection<FeatureType>
getFeatureTypeForFeature(String featureId,
String revisionId){ }
    protected ObjectId insert(Feature f)
throws Exception { }
    protected void insertAndAdd(Feature...
features) throws Exception { }
    protected void close() throws Exception
{ }
    protected void insert(Feature...
features) throws Exception { }
    protected boolean deleteAndAdd(Feature
f) throws Exception { }
    protected boolean delete(Feature f)
throws Exception { }
    public String toString() { }
    public SimpleFeatureCollection
getLog(String fromVersion, String toVersion,
FeatureType featureType, Filter filter,
Integer maxRows) throws Exception{
    }
}

```

```

    private SimpleFeature
feature(SimpleFeatureType type, String id,
Object... values)          throws
ParseException {          }
    private void processFilter(Filter
filter, FeatureType featureType, LogOp
logOp) throws UnsupportedOperationException
{          }
    public void commit() throws Exception
{          }
}

```

## Director or Index BLOB

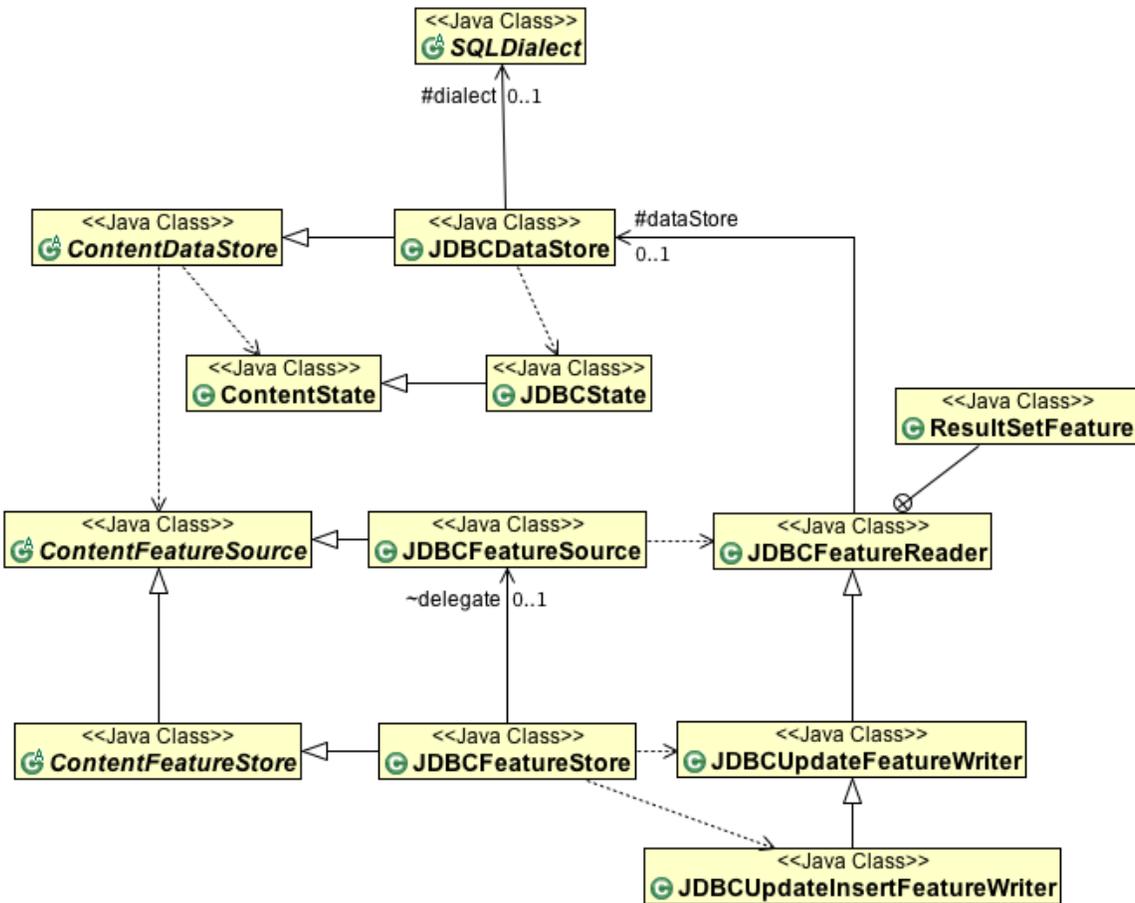
GeoGit contents is pretty unstructured (as expected for a pile of BLOBS). We should look into supporting:

- A spatial index? This is a spatial

## DataStore Delegate Design

Walter got a bit of technical direction from [Justin Deoliveira](#) / [Gabriel Roldán](#) have expressed a strong preference for a "wrapper" approach. Rather than subclassing JDBCDataStore.

- The original "PostGIS-versioned" datastore was implemented as a direct wrapper around PostGISDataStore offering an example of how this can be performed.
- The JDBCDataStore classes are marked final specifically to prevent subclassing; instead a strategy object SQLDialect is used to configure JDBCDataStore teaching it the abilities of each database. This handles all the responsibilities of SQL encoding; and mapping attributes to attributes and feature id.



The above shows *JDBCDataStore* and the "support classes" that actually make up the implementation. These classes are considered part of the JDBCDataStore implementation and are carefully tested to work together. Central to this design is marking the classes final to prevent subclasses from locking down the API contract (allowing JDBCDataStore and support classes to be fixed and improved over time without the burden of subclasses to lug about).

✓ The central challenge here is how to reuse the above work; as GeoGitDataStore would like to also make SQL queries and take part in the generation of featureIds; splice in ResourceIds and so on.

[Justin Deoliveira](#) has asked that we work in a fork of GeoTools in order to explore options (and then report back on what needs to be changed when merging in the work).

## Pure DataStore Wrapper

The goal here would be to define a pure DataStore wrapper that would encode revision+featureid information into the attributes provided by the delegate DataStore (which would be treated as a simple storage of "row sets").

So for this approach we would have *GeoGitFeatureWriter* (similar to FilteringFeatureWriter):

```

public GeoGitFeatureWriter(
    GeoGitFeatureSource featureSource,
  
```

```

FeatureWriter writer ){
    featureSource = featureSource;
    writer = writer;
}
protected GeoGitFacade getGit(){
    return featureSource.getGeoGit();
}
public void write() throws IOException {
    if (writer == null) {
        throw new
IOException("FeatureWriter has been
closed");
    }
    GeoGitFacade ggit = getGit();

    if (current == null || row == null)
{
        throw new IOException("No
feature available to write");
    }
    SimpleFeature rowUpdate =
ggit.revisionEncode( current );

    currentRow.setValues(
rowUpdate.getValues() ); // copy values into
the current row
    writer.write(); // writer
responsible for writing out row

    currentFeature = null;
    rowFeature = null;
}

```

```
}  
public SimpleFeature next()  
    if (writer == null) {  
        throw new  
IllegalStateException("Writer has already  
been closed");  
    }  
    nextRow = writer.next();  
    nextFeature =
```

```
getGit().revisionEncode( row );  
  
return nextFeature;
```

Pros:

- Able to be used on any DataStore (would recommend property DataStore as it supports multi geometry)

Cons:

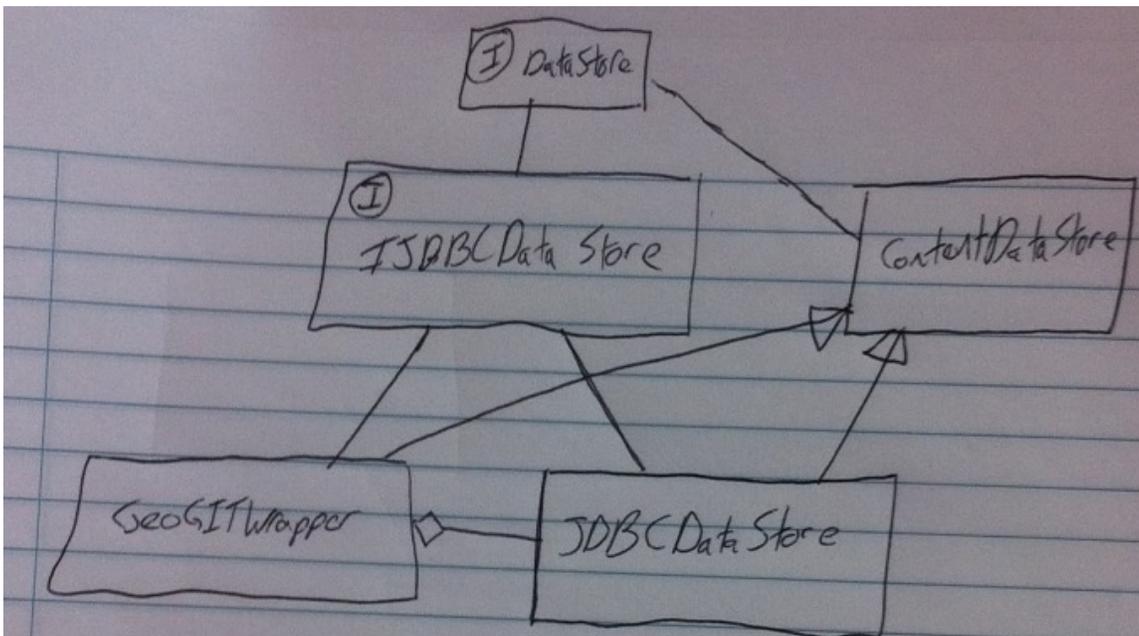
- Lack of tight integration with SQL Generation may result in inefficient code
- Need to write delegating implementations for ContentDataStore implementation (and all support classes)

## JDBCDataStore Wrapper

Following the example of postgis-versioned a wrapper DataStore is defined to add versioning against a JDBCDataStore.

### IJDBCDataStore Interface

Definition of a IJDBCDataStore which is implemented by JDBCDataStore and GeoGitWrappingDataStore. This allows classes that previously expected direct access to work with either.



This has currently been implemented and all jdbc-ng plugins test. Test for dependent modules also pass as well as a few tests that have been written using a GeoGitWrapperDataStore.

Walters Notes:

- The rationale behind adding the interface was so that the GeoGitWrappingDataStore or a separate implementation or direct versioning could still use the existing class structure of JDBC classes (e.g. SqlDialect).

- The only other option would have been to reimplement the entire module for **\*gt-jdbc\***. This would have resulted in a lot of duplicated classes with only package name changes or with only a redeclaration of a field type. This would have caused a lot more maintaining effort in the long run and was counter to DRY principles.
- Our intent was to develop a solution that allowed the API to maintain final implementation of existing classes without blowing out the classes.

Setup:

```
public final class JDBCDataStore extends
ContentDataStore implements IJDBCDataStore {
    ....
}
public final class GeoGitDataStore extends
ContentDataStore implements IJDBCDataStore {
    IJDBCDataStore delegate;
    GeoGit geogit;
    GeoGitDataStore( GeoGit geogit,
IJDBCDataStore dataStore ){
    }
}
```

Follow through:

```
public class JDBCSimpleFeatureWriter {
    JDBCSimpleFeatureWriter(
IJDBCDataStore, Query query ){
    }
    IJDBCDataStore getDataStore(){
        return dataStore;
    }
}
```

In the above example the type narrowing where a JDBCSimpleFeatureWriter getDataStore() returns a JDBCDataStore is causing the use of IJDBCDataStore to spread throughout all JDBCDataStore implementations

classes; and the various implementations for PostGIS, DB2, Oracle and so forth.

**i** This approach has been more complex than originally thought. The implementation of an interface was not difficult, however, a lot of the JDBCDataStore methods had to be made public rather than private due to the number of package class calls that were performed. This exposes a much larger public API for JDBCDataStore than the community would probably like.

Though many classes had to be changed most of the changes were trivial. The main issues encountered stemmed from the type narrowing of overridden methods. Superclasses that were returning ContentDataStore instead of DataStore were narrowed to JDBCDataStore. These calls broke as we moved to interface returns. A few changes were required up the tree to allow the type narrowing with interfaces instead.

Changes were also required to introduce IJDBCFeatureStore and IJDBCFeatureSource. These interfaces were needed to provide similar wrapping functionality at low levels.

Code Example:

```
versionedjdbcupdatewriter is a different
class with this implementation:
    public void write() throws IOException {
        try {
            GeoGITFacade ggit =
((VersionedJDBCFeatureSource<SimpleFeatureType, SimpleFeature>)
this.featureSource).getGeoGIT();
                //figure out what the fid is
                PrimaryKey key =
dataStore.getPrimaryKey(featureType);
                String fid =
dataStore.encodeFID(key, rs);

                Id filter =
dataStore.getFilterFactory()

                .id(Collections.singleton(dataStore.getFilterFactory())
```

```

        .featureId(fid)));

        //figure out which attributes
changed
        List<AttributeDescriptor>
changed = new
ArrayList<AttributeDescriptor>();
        List<Object> values = new
ArrayList<Object>();

        for (AttributeDescriptor att :
featureType.getAttributeDescriptors()) {
            if
(last.isDirty(att.getLocalName())) {
                changed.add(att);

        values.add(last.getAttribute(att.getLocalName()));
            }
        }

        // do the write
        datastore.update(featureType,
changed, values, filter,
st.getConnection());
        ggit.insertAndAdd(last);
        ggit.commit();
        // issue notification
        ContentEntry entry =
featureSource.getEntry();
        ContentState state =

```

```
entry.getState( this.tx );
    if( state.hasListener() ){
        state.fireFeatureUpdated(
featureSource, last, lastBounds );
    }
} catch (Exception e) {
    throw (IOException) new
```

```
IOException().initCause(e);
    }
}
```

Pros:

- It is implemented and all existing tests pass!

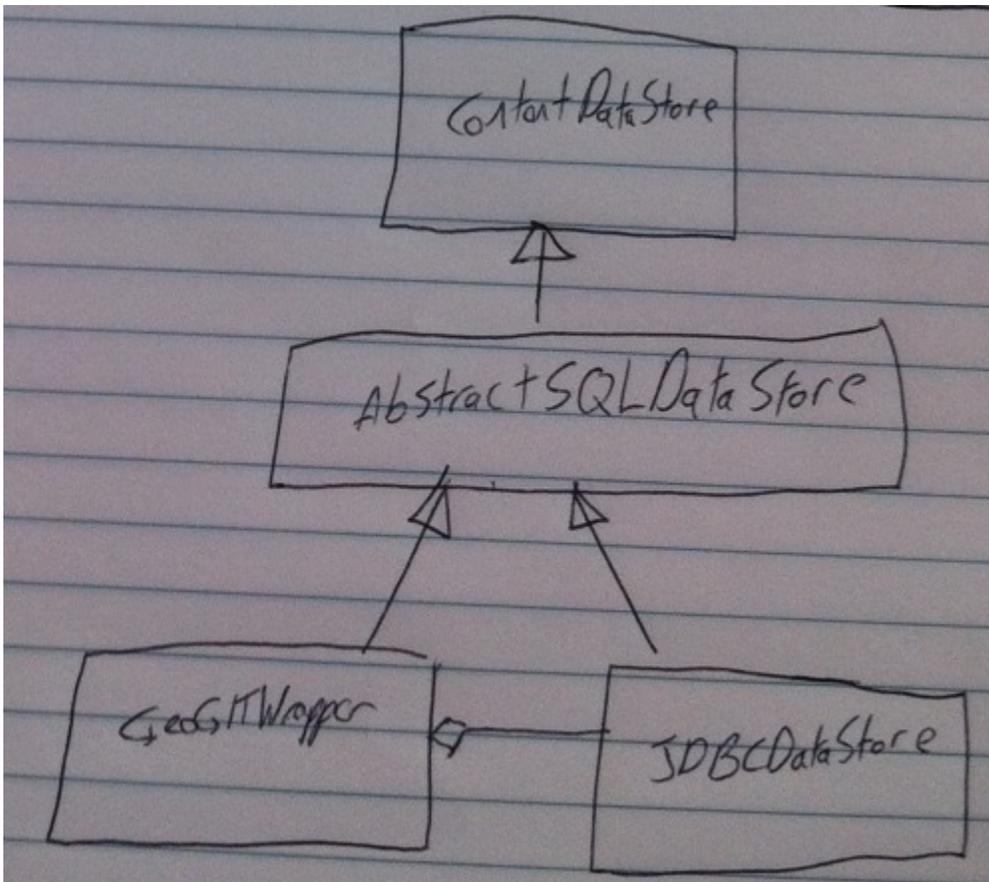
Cons:

- Making more of JDBCDataStore method public
- Introduction of IJDBCDataStore interfaces gives the API more surface area to maintain
- The IJDBCDataStore ends up being a grab bag of methods at different levels of abstraction (basically a side effect of an implementation issue rather than a clear concept)

### SQLDataStore Abstract Class

This is a follow up Idea; which should result in a cleaner implementation of the above approach without introducing an interface (is a bit scary and runs counter to the design goals of JDBCDataStore).

Rather than introduce an Interface; we could also pull up the concept into an abstract class (suggested name "SQLDataStore"). This has the advantage of still extending ContentDataStore; and the name matches up with **\*SQL Dialect\***; giving it a nice clear set of responsibilities.



Direction: Jody has suggested this idea in response to IJDBCDataStore; as we really don't want an interface (abstract class offers greater stability).

Code Sample:

```
public abstract class SqlDataStore extends
ContentDataStore {
    ....
}
public final class JDBCDataStore extends
SqlDataStore {
    ....
}
public final class GeoGitDataStore extends
SqlDataStore {
    SqlDataStore delegate;
    GeoGit geogit;
    GeoGitDataStore( GeoGit geogit,
SqlDataStore dataStore ){
    }
}
```

Follow through:

```

public class JDBCFeatureSource extends
SqlFeatureSource {
    ...
    public SqlDataStore getDataStore()
    {
        return (SqlDataStore)
super.getDataStore();
    }
    ...
}

```

In this scenario, JDBCFeatureStore and JDBCFeatureSource would also undergo a similar abstraction parenting with Version-aware wrapper classes.

Classes like VersionedJDBCInsertFeatureWriter et al can be merely subclassed as they are not marked final.

**i** The difficulty that has arisen with type narrowing and interface/class usage has made the Interface solution feel overly complex.

The solution also exposes more of the interface than the community will probably feel comfortable with. At this point, with the DataStoreWrapper approach it makes sense to restart using an AbstractSQLDataStore that will have a similar purpose to the Interface (however) an interface representing the public API might still be useful for exposing to external modules. Type narrowing will be easier to manage and issues of protected properties on objects that were hidden from the wrapping class would also be easier to overcome (The versioning datastore was in a separate package so a few of the properties of the wrapped datastore were unreachable without altering access).

Pros:

- Clear migration from existing IJDBCDataStore implementation
- Clear definition of SQLDataStore as an abstract class
- Allows JDBCDataStore support classes to maintain their package visibility relationship with SQLDataStore and JDBCDataStore
- Concepts pulled up into SQLDataStore would be those common to both GeoGitDataStore and JDBCDataStore
- Can use package visibility to avoid exposing too much to subclasses

Cons:

- Large number of classes introduced
- Still gives more surface area to JDBCDataStore concepts (at least it is still controlled and locked down to GeoGIT and JDBCDataStore)

## RevisionSupport Strategy Object

Another design alternative suggested by [Jody Garnett](#) was to extend JDBCDataStore from the inside with a strategy object to sort to how revision information is handled. This could be defined as an **RevisionSupport**; with a default implementation that encodes revision information into a 'revision' attribute which can be combined with the normal FeatureID.

If an implementation of SQLDialect supported RevisionSupport it would be used directly; allowing a datastore implementation such as Oracle to directly use native functionality.

This design is only a suggestion; it probably provided too much internal access to JDBCDataStore classes beyond what SQLDialect already provides?

In discussion with ~aaime this is the approach he would take with the following clarifications:

- modify JDBCDataStore with a small extension point to leverage native joins
- version accomplished with minimal changes
  - amend the way primary keys are handled
  - mostly primary key value generation
  - ability to add indexes (clean addition useful elsewhere)
- do the rest with simple wrapping / delegation
- original design subclassed mostly to alter primary key handling; the rest was about querying and interpreting the attributes in a different manner, altering the queries and so on

We would add a second strategy object (called RevisionSupport) and use it wherever we use sqlDialect:

```
class JDBCDataStore {
    SQLDialect sqlDialect;
    RevisionSupport revisionSupport;
}
```

Code Example VersionedJDBCUpdateWriter:

```
public void write() throws IOException {
    try {
        GeoGITFacade ggit =
            ((VersionedJDBCFeatureSource<SimpleFeatureType, SimpleFeature>)
             this.featureSource).getGeoGIT();

        //figure out what the fid is
```

```

        PrimaryKey key =
dataStore.getPrimaryKey(featureType);
        String fid =
dataStore.encodeFID(key, rs);

        Id filter =
dataStore.getFilterFactory()

.id(Collections.singleton(dataStore.getFilterFactory()

        .featureId(fid)));

        //figure out which attributes
changed

        List<AttributeDescriptor>
changed = new
ArrayList<AttributeDescriptor>();
        List<Object> values = new
ArrayList<Object>();

        for (AttributeDescriptor att :
featureType.getAttributeDescriptors()) {
            if
(last.isDirty(att.getLocalName())) {
                changed.add(att);

        values.add(last.getAttribute(att.getLocalName()));
            }
        }

```

```
        // do the write
        datastore.update( featureType,
changed, values, filter, st.getConnection()
);

        // notice there is no change
here for revision support

        // issue notification
        ContentEntry entry =
featureSource.getEntry();
        ContentState state =
entry.getState( this.tx );
        if( state.hasListener() ){
            state.fireFeatureUpdated(
featureSource, last, lastBounds );
        }
    } catch (Exception e) {
        throw (IOException) new
```

```
IOException().initCause(e);
    }
}
```

So most of the JDBCSupport classes are unchanged; the core JDBCDataStore would make use of both SQLDialect and RevisionSupport to do its job:

```
protected void update(SimpleFeatureType
featureType, AttributeDescriptor[]
attributes,
    Object[] values, Filter filter,
Connection cx) throws IOException,
SQLException {
    if ((attributes == null) ||
(attributes.length == 0)) {
        LOGGER.warning("Update called
with no attributes, doing nothing.");
        return;
    }
    Versioned row =
revisionSupport.update( featureType,
attribute, values, filter );

    if ( dialect instanceof
PreparedStatementSQLDialect ) {
        try {
            PreparedStatement ps =
updateSQLPS( row.getFeatureType(),
row.attributes, row.values, row.filter, cx);
            try {
```

```
((PreparedStatementSQLDialect)dialect).onUpdate(ps, cx, featureType);
        ps.execute();
    }
    finally {
        closeSafe( ps );
    }
}
catch (SQLException e) {
    throw new RuntimeException(
e );
```

```
}  
}  
.....
```

Pros:

- Keeps the relationships internal to JDBCDataStore and controlled by a strategy object
- Similar to the design of LockingManager; allows internal access to SQLDialect and participation in the generation of select statements etc.

Cons:

- Nobody has been interested or understood this suggestion

## getJDBCDataStore Wrapper

One of the smallest changes would be to copy the design of java.sql Wrapper:

```
interface Wrapper {  
    boolean isWrapperFor(Class<?>)  
    T unwrap(Class<T>)  
}
```

This could be use to quickly allow classes that expect access to their JDBCDataStore to unwrap the delegate as in the following example:

```

JDBCDataStore getJDBCDataStore(){
    if( dataStore instanceof JDBCDataStore){
        return (JDBCDataStore) dataStore;
    } else if (dataStore instanceof
Wrapper){
        Wrapper wrapper = (Wrapper)
dataStore;
        if( wrapper.isWrapperFor(
JDBCDataStore.class ){
            return wrapper.unwrap(
JDBCDataStore.class );
        }
    }
    return null; // JDBCDataStore not
available!
}

```

Any code (such as a reader) that needed access to JDBCDataStore could now do so with a single line of code:

```

if( count > getJDBCDataStore().fetchSize){
    break;
}

```

Pros:

- Allows JDBCDataStore to continue its package visibility contracts with JDBCDataStore support classes.

Cons:

- Updating references to getDataStore() in JDBCDataStore support classes.
- Not sure if it solves the problem