Homework 4
Danny Wyatt
CSE503, Spring 2004

1 Information Hiding

1A Poor Information Hiding

1. in `JukeXTrackStore.java`, in the inner class `JukeXTrackStore.JukeXTrackLoader`, lines 730 and following:

```
730 JukeXTrackStore trackStore =
                    (JukeXTrackStore)TrackStoreFactory.getTrackStore();
731 JukeXTrack currTrack = null;
732
733 while ( rs.next() )
734 {
735    lastID = currID;
736    currID = new Long( rs.getLong( 1 ) );
737
738    // Check if we've changed track
739    if (!lastID.equals(currID))
740    {
741        currTrack =
           (JukeXTrack)trackStore.getCachedTrack(currID.longValue());
742        if ( currTrack == null)
743        {
744            // Make a new Track object
745            try {
746                currTrack = new JukeXTrack(currID.longValue(),
747                                           new URL(rs.getString(2)),
748                                           new Date(rs.getLong(3)));
749            } catch ( MalformedURLException mue ) { }
750            // Then cache it so this doesn't happen again
751            trackStore.cacheTrack( currTrack );
752        }
```

   At line 730, the implementation assumes that the specific `TrackStore` implementation will be an instance of `JukeXTrackStore` and casts it as such. This assumption is put into use when the package-private methods `getCachedTrack()` and `cacheTrack()` are used at lines 741 and 751, respectively. Any caching behavior of `TrackStore` is meant to be entirely hidden from clients, but these method calls seek to expose that behavior.

   This is not only poor information hiding, but it specifically violates the intent of the Factory pattern by making assumptions about the specific class of the object (the Concrete Product) it returns and not using that object only through its abstract interface (the Abstract Product).

   If the JukeX software is extended to have more than just the `jukex.sqlimpl` package implementing the interfaces of the `jukex` package, and the `TrackStoreFactory` is configured to return a different implementation of `TrackStore`, this class will have to be changed to ensure that it uses only the public interface of `TrackStore` and makes no assumption about the caching behavior of the implementation.

2. In `ControlPlaybackManager.java`, lines 63 and following:

```
63     /**
64      * List all of the registered ControlPlaybacks
65      *
```

```
     * @return A Set of the ControlPlayback names
     */
    public Set listAllControlPlaybacks()
    {
        return controlPlaybacks.keySet();
    }
```

The `ControlPlaybackManager` class keeps track of all `ControlPlayback` objects in the application. It stores them in a `HashMap`, keyed by their names. The method `listAllControlPlaybacks` is supposed to return a list of those names. It actually returns the `keySet` of the backing map directly. The documentation for `keySet()` explains that the set it returns

> is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from this map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations.

(from `http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html#keySet()`)

Thus any caller could change or remove `ControlPlayback` objects from the map by modifying the set returned. This is particularly bad given that `ControlPlaybackManager` specifically provides the methods `registerControlPlayback()` and `unregisterControlPlayback()` for tracking when `ControlPlayback` objects are added or removed. If callers can remove them directly via the `keySet` then the unregister method can be bypassed.

With this poor information hiding, future modifications cannot rely on `ControlPlayback` objects being unregistered only through the public interface—even though the interface design suggests exactly that.

1B Good Information Hiding

The class `JukeXAttribute` has good information hiding.

It has 3 instance variables and they are all private (lines 31-33):

```
31     private String _name = null;
32     private long _id = -1;
33     private int _type = 0;
```

They are all available only through accessor methods, and they are either native types (`long` and `int`) or immutable (`String`) so passing them to callers does not allow them to be changed.

Its two `getAttributeValue()` methods that do return more complex types are Factory Methods for creating `JukeXAttributeValue` objects.

Additionally, the return type of these methods is the interface `AttributeValue` and the method names correctly give no hint about the dynamic type of the actual implementing class that will be returned.

1C Coupling and Cohesion

- The class `JukeXTrackStore.JukeXTrackLoader` exists as an implementation of the `BatchTrackLoader` interface. An implementer of this interface should build an internal list of track ID's and then retrieve all of the corresponding `Track` objects at once. `JukeXTrackLoader` does that and only that. As such, it has high cohesion.

  However, as shown above, its relies on casting an object it handles to that object's dynamic type and then calling non-public methods of that object. This is a high degree of coupling. Oddly enough, since it is an inner class of exactly the dynamic type that it hopes to rely on, this would be forgivable: inner classes cannot exist outside an instance of their enclosing class. However, `JukeXTrackLoader` does not access its enclosing `JukeXTrackStore` instance via the ways available to it as an inner class. It instead goes through the Factory, which not only makes its existence as an inner class unnecessary but couples it to another class (which is only incidentally its enclosing class) in a bad way.

- The current implementation of `ControlPlaybackManager` is not much more than a wrapper for a hashtable. As such, its cohesion is high since it does exactly one thing and nothing more. Its coupling is low since it interacts with no classes other than its own backing hashtable instance and all (current) classes that interact with it do so only through its public methods. That said, it is mostly a trivial class (in its current implementation), so its good design is not that remarkable.

- `JukeXAttribute` has moderately high cohesion. It stores a name and a type, and returns them on request. It can create `AttributeValue` objects that are then be associated with it. It is essentially a class for representing the name and type of an attribute, and it does that well. However, I explain below why I qualify its cohesion as "moderately" high.

  The coupling of `JukeXAttribute` with `JukeXAttributeValue`, `JukeXTrack`, and `JukeXTrackStore` is high, as well as confusing.

  As I understand it, JukeX manages attributes as follows:

  There is a global attribute namespace managed by the central `TrackStore` repository. `TrackStore` can return `Attribute` objects. `Attribute` objects only store their name and type (string or integer). An `Attribute` can have many values. These values are represented in `AttributeValue` objects. The pairing of an `Attribute` with an `AttributeValue` is managed at a `Track` object. Thus, `Attribute` objects are keys used to retrieve their associated `AttributeValue` objects from `Track` objects. These keys must themselves be retrieved (by their names) from either `TrackStore` or the `Track`.

  Overall, `JukeXAttribute` is just a key in the key-value system for storing and retrieving attributes. It is useless outside of this system, so its coupling is very high. But as a key, it does what it needs to do: store its name and type. It can additionally create new `AttributeValues`. This is the only piece of its behavior that would decrease its cohesion, since that behavior could perhaps more properly be put into `TrackStore` or `Track`. This is why I qualify its high cohesion as "moderate".

2 Design Rationale

2A All Logging Concern Appearances

In class `JukeXAttributeValue`:

lines 36-38, logging Singleton is retrieved:

```
private static final Category log =
               Category.getInstance(JukeXAttributeValue.class.getName());
private static final boolean logDebugEnabled = log.isDebugEnabled();
private static final boolean logInfoEnabled = log.isInfoEnabled();
```

lines 105-108, error logging:

```
if ( !newEntryId.next() )
{
  log.error("Something awful happened.
             An INSERT somehow failed to appear in the database");
}
```

lines 182-185, error logging:

```
catch ( Exception e )
{
  log.error("Encountered an exception attempting
             to change an AttributeValue string value");
}
```

In class `JukeXPlaylist`:

static initialization block, lines 53-55, logging Singleton is retrieved:

```
private static final Category log = Category.getInstance(JukeXPlaylist.class.getName());
private static final boolean logDebugEnabled = log.isDebugEnabled();
private static final boolean logInfoEnabled = log.isInfoEnabled();
```

method `getNextTrack()`, lines 129-132, informational logging for development:

```
} else {
    if (logDebugEnabled) log.debug("I'm spent, delegating...");
    retVal = delegateGetNextTrack();
}
```

method `peekTracks()`, line 158, informational logging for development:

```
if (logDebugEnabled) log.debug("Peeking ahead for " + count + "tracks, remainder " + rem);
```

method `readTrackListing()`, lines 200-205, error logging (both serious and semi-serious):

```
catch ( SQLException se )
{
    log.error( "Failed due to an exception reading a Track listing into a playlist" , se );
} catch (Exception e) {
    log.warn("Encountered exception while reading track listing: ", e);
}
```

method `persist()`, lines 250-254, error logging:

```
catch ( SQLException se )
{
    try { conn.rollback(); } catch ( SQLException ignore ) { }
    log.error( "Encountered an error persisting a playlist" , se );
}
```

In class `JukeXTrack`:

static initialization block, lines 46-48, logging Singleton is retrieved:

```
private static final Category log = Category.getInstance(JukeXTrack.class.getName());
private static final boolean logDebugEnabled = log.isDebugEnabled();
private static final boolean logInfoEnabled = log.isInfoEnabled();
```

method `addAttributeValue()`, lines 119-131, error logging:

```
   else
   {
     log.error("JukeXTrack encountered an attribute
                with an unknown type ["+attribute.getType()+"]");
   }

   addEnumeration.executeUpdate();
   _attributes.put( attribute , value );
   addEnumeration.close();
}
catch ( Exception e )
{
  log.error("JukeXTrack encountered an exception whilst
             attempting to add an AttributeValue", e);
}
```

method `clearAttribute()`, lines 152-155, error logging:

```
catch ( Exception e )
{
    log.error("Exception encountered attempting to clear attribute values",e);
}
```

method `readAttributesFromDB()`, lines 210-213, error logging:

```
catch (Exception e)
{
    log.error("Encountered an exception whilst
              reading attributes from the database" , e );
}
```

method `setUpdatedDate()`, lines 254, informational logging incorrectly performed as error logging:

```
log.error( "Updating track "+_id+" date to: " + newdate + " ["+newdate.getTime()+"]" );
```

method `setUpdatedDate()`, lines 267-270, error logging:

```
catch ( SQLException se )
{
    log.error( "Exception whilst changing modified date on track with id="+this._id , se );
}
```

method `getAttributeValue(Attribute)`, lines 300-301, semi-serious error logging:

```
log.warn("No values for attribute "+attribute.getName());
return null;
```

method `getAttributeValue(String)`, lines 317-318, semi-serious error logging:

```
log.warn("Cannot find attribute name " + attributename);
return null;
```

In class `JukeXTrackStore`:

static initialization block, lines 53-55, logging Singleton is retrieved:

```
private static final Category log = Category.getInstance(JukeXTrackStore.class.getName());
private static final boolean logDebugEnabled = log.isDebugEnabled();
private static final boolean logInfoEnabled = log.isInfoEnabled();
```

method `getTrackCount()`, lines 129-131, error logging:

```
} catch ( Exception e ) {
  log.error("An exception was encountered whilst
            trying to count the number of tracks", e );
}
```

method `getTrack(URL)`, lines 165-168, error logging:

```
catch ( Exception e )
{
  log.error("An exception was encountered whilst
            trying to retrieve a track with the URL ["+url+"]", e);
}
```

method `getTrack(long)`, lines 225-228, error logging:

```
catch ( Exception e )
{
  log.error("An exception was encountered whilst
            trying to retrieve a track with id: "+id, e);
}
```

method `getTracks()`, lines 265-269, semi-serious error logging:

```
else
{
  log.warn("Could not retrieve all tracks specified in a getTracks() call.  Track "+
  currID+" could not be found");
  resultList.add( y , null );
}
```

method `storeTrack()`, lines 308-311, severe error logging:

```
if ( !id.next() )
{
    log.fatal("Something went really badly wrong whilst trying to store a track."+
    "  Could not fetch the LAST_INSERT_ID().");
}
```

method `getAttribute()`, lines 349-352, error logging:

```
catch (SQLException se)
{
    log.error( "Encountered an SQL error attempting to retrieve an attribute" , se );
}
```

method `getAttributes()`, lines 386-389, error logging:

```
catch ( SQLException se )
{
  log.error("Encountered an exception whilst fetching attributes from the database", se);
}
```

method `createAttribute()`, lines 424-428, error logging:

```
if ( getAttribute( name ) != null )
{
    log.error( "Skipping duplicate addition of attribute ["+name+"]" );
    return getAttribute( name );
}
```

method `getTrackIds()`, lines 498-501, error logging:

```
catch ( SQLException se )
{
    log.error("Encountered an exception whilst
              getting track ids from the database" , se );
}
```

method `getPlaylist()`, lines 537-540, error logging:

```
catch ( SQLException se )
{
    log.error("Encountered an exception whilst
              getting a playlist from the database" , se );
}
```

method codecreatePlaylist(), lines 581-584, error logging:

```
catch ( SQLException se )
{
log.error( "Failed due to an Exception whilst creating a playlist" , se );
}
```

method `loadPlaylists()`, lines 602, informational logging:

```
if (logDebugEnabled) log.debug( "Loading playlists from database..." );
```

In class `JukeXTrackStore.JukeXTrackLoader`:

method`getTracks()`, lines 769-772, semi-serious error logging:

```
catch ( SQLException se )
{
    log.warn( "Batch getter encountered an exception whilst retrieving tracks" , se );
}
```

Methodology I located these appearances using the FEAT plugin for Eclipse (more on FEAT in Section 5) I first used Eclipse's search function to find all uses of the class `org.apache.log4j.Category` in the package `com.neoworks.jukex.sqlimpl`. After finding `log4j` in the 4 classes listed above, I added each of their Singleton instances of `Category` to a FEAT concern. I then used FEAT's "fan-in → referenced by" function to find all uses of this `Category` instance in the 4 classes.

2B Modularizing Logging in Standard Java

It is not possible to modularize the logging concern in standard Java without significant refactoring of the code—amounting to almost a total redesign. My answer to this question assumes a modularizing *that does not cut across the code*. There is a different modularization possible where cross-cutting remains that I describe in Section 2C.

The primary organization of the system is around the playback control of music that is arranged in playlists of tracks with attributes, and the ability to persist such arrangements. This is an intuitive organization for a jukebox system. Logging is quite a secondary (much lower than secondary, tertiary or below, even) concern in this organization.

If one wanted to modularize logging so that it did not cut across this organization it would require re-organizing the classes so that they all extended some "logger" class and used the methods they inherited from that logger class to do all of their logging. This would require breaking apart the playlist/track/attribute inheritance hierarchy. It would yield an organization whose primary concern was logging, with the secondary concern of being a jukebox system. Clearly, this is not desirable.

2C Modularizing with AspectJ

It is not possible to perfectly modularize the logging concern with AspectJ. One can come close, but no closer than is possible using techniques available in standard Java (different from the above technique).

By using an external package to handle all logging (as opposed to putting printing or file writing statements directly into each class that logs something) the developers have already modularized the logging concern out to the best extent possible in standard Java. They can change the type of object returned by `Category.getInstance(JukeXTrackStore.class.getName())` to be any class that extends `Category` and then they can handle the logging methods (`debug()`, `error()`, etc.) however they wish.

Of course, there are details that muddy this and make the practice harder than the theory. By opting for efficiency and using the `isDebugEnabled()` and `isInfoEnabled()` methods to decide whether to call `debug()` and `info()` (though `info()` is never actually called), they have tied their logging concern to `log4j`'s fixed priority levels. Additionally, by using the specific `debug()`, `error()`, etc. methods they have further increased their ties to the fixed priorities than they would had they used the more generic `log(Object msg, int priority)` methods.

If they wanted to modularize out the logging with AspectJ they would need to define the following pointcuts:

```
calls(Category Category.getInstance(Class))
calls(boolean Category.isDebugEnabled())
calls(boolean Category.isInfoEnabled())
calls(void Category.debug(Object))
calls(void Category.info(Object))
calls(void Category.warn(Object))
calls(void Category.error(Object))
calls(void Category.fatal(Object))
```

They would then need to attach advice to each of these pointcuts that handled the specific logging category, flag, or Singleton getter in a new way. This would be equivalent to simply adding a new subclass of `Category` and configuring `log4j` to return that class when `Category.getInstance(JukeXTrackStore.class.getName())` is called. As such, all of the messiness outlined in the third paragraph above would ensue in the AspectJ modularization as well.

Finally, the changes just described—whether achieved through a new `Category` or via AspectJ—would only change the way they handle currently present logging statements. There is no way to add new logging statements—particularly those that access local variables—arbitrarily throughout as they like to do with informational statements. For example, `JukeXTrackStore.getTracks()` line 267 (shown above) uses a variable, `currId`, that is only in scope within a `for` loop. To my knowledge, there is no way to get at that value via AspectJ.

3 Cross-cutting Concerns

3A Identifying a Concern

In looking through the `jukex.query` package, I noticed that it has database-accessing code scattered through more than one class. This scattering of database code across the package is the symptom of the concern. A deeper examination of the symptom shows a pattern of "cut-and-paste" fragments that mark appearances of the concern. At each appearance the code must:

- Get the Singleton connection pool manager
- Get a connection from the connection pool manager
- Create a SQL statement object
- Populate the statement
- Execute the statement

That these steps are repeated identically at many spots throughout the package reveals a cross-cutting concern.

3B Occurrences of the Concern

The concern spreads across three files in the package: `Query.java`, `JukeXExpression.java`, and `AttributeValueResultSet.java`. Below, I show all occurrences of the concern from the first two files. Since the `AttributeValueResultSet` class is essentially a wrapper for a `javax.sql.RowSet` object, I would have to include the entire file.

In class `JukeXExpression.Relop`, in the method `getSql()`, lines 203 and following:

```
conn = PoolManager.getInstance().getConnection(JukeXTrackStore.DB_NAME);
StringBuffer sql= new StringBuffer().append(
                     "SELECT AttributeEnum.id, Attribute.name, Attribute.type,
                     AttributeEnum.value FROM Attribute, AttributeEnum
                     WHERE Attribute.id = AttributeEnum.attributeid AND " );
//TODO: Numeric value change op
if ( literal.val instanceof String )
{
    sql.append( "( Attribute.name=" ).append(
                JukeXExpression.escapeString( variable.val ) ).append(
                " AND AttributeEnum.value ")/*LIKE " )*/.append(operator).append(
                JukeXExpression.escapeString( (String) literal.val ) ).append( " )" );
}
else
{
    sql.append( "( Attribute.name=" ).append(
                JukeXExpression.escapeString( variable.val ) ).append(
                " AND AttributeEnum.value" ).append(operator).append(
                JukeXExpression.escapeString( (String) literal.val ) ).append( " )" );
}
ResultSet rs = conn.createStatement().executeQuery( sql.toString() );

if ( rs.next() )
{
    possValues.append( rs.getLong(1) );
    while ( rs.next() )
    {
        possValues.append(',').append( rs.getLong(1) );
    }
}

[...]

buffer.append( "( bind_" ).append( variable.val ).append( ".attributeenumid " );
buffer.append("IN (").append( possValues ).append(") ");
buffer.append( ')' );
}
else if ( attr.getType() == Attribute.TYPE_INT )
{
  int intval = ((Integer)literal.val).intValue();

  buffer.append( "( bind_" ).append( variable.val ).append( ".numericvalue" );
  buffer.append( operator ).append( intval ).append( ')' );
}
```

In class `Query`, in the method `getTracks()`, lines 114 and following:

```
114  try
115  {
116    conn = _poolmanager.getConnection( JukeXTrackStore.DB_NAME );
117    Statement state = conn.createStatement();
118
119    System.out.println( this.getSQL() );
120
121    ResultSet rs = state.executeQuery( this.getSQL() );
122
123    while ( rs.next() )
124    {
125       trackids.add( new Long( rs.getLong( 1 ) ) );
126    }
127
128    return trackstore.getTracks( trackids );
129  }
```

In class Query, in the method getAttributeValues(), lines 167 and following:

```
167  try
168  {
169     conn = PoolManager.getInstance().getConnection( JukeXTrackStore.DB_NAME );
170     Statement state = conn.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE ,
                                              ResultSet.CONCUR_READ_ONLY );
171     //ResultSet rs = state.executeQuery( this.getSQL() );
172
173     CachedRowSet cs = new CachedRowSet();
174     cs.populate( state.executeQuery( this.getSQL() ) );
175
176     retval = new AttributeValueResultSet( cs , this.selectAttributes );
177
178     state.close();
179  }
```

The reason they could not modularize this concern is that the primary organization of the `query` package is around parsing and representing queries—it mirrors the `rdc` package—not executing them on a specific SQL back end.

Furthermore, the way they have modeled attribute values as having two possible types has given rise to a classic "impedance mismatch" between their system's object-oriented in-memory model and the storage of it in a relational system. String and integer valued `AttributeValue` objects are effectively subclasses of `AttributeValue` since an `AttributeValue` object can appear wherever that type is needed regardless of its internal "subtype". This polymorphism is not expressible in relational algebra that is in first-normal form. Thus, they end up storing integer values in one database table and string values in another despite the fact that these values will become instances of the same class. Lines 241-152 of `JukeXExpression.java` (quoted above) show exactly the consequence of this mismatch: they must query the database differently depending on the internal type of the attribute value.

3C Modularization of the Concern with AspectJ

It is not possible to modularize this concern with AspectJ. As shown above, all of the database-accessing code is intertwined with other code throughout several methods. To even begin to abstract the feature out one would need cutpoints for hijacking many different method calls to the classes `ConnectionPoolManager`, `ConnectionPool`, `Connection`, `Statement`, and `ResultSet`. This would be as involved and brittle as the attempt modularize the logging concern from Section 2.

The best attempt would take control of all the `getSQL()` methods in each subclass of `Expression`, as well as the `getTracks()` and `getAttributeValues()` methods in `Query`. This would effectively amount to separating data persistence from internal representation (something they should have done in the first place) via an aspect.

That said, even if all of the database accesses were pulled out into a separate aspect, that would not magically solve the mismatch between object-oriented and relational representations. Some objects will need more involved interactions with the database than others, and these interactions will twine throughout any persistence code they require.

4 Design Patterns

4A Singleton

The class `JukeXTrackStore` is a Singleton. Observe lines 62, 74 and following. (The code snippet is unedited, including the misspelling in line 90.)

```
 62     private static TrackStore _instance = null;   // Singleton instance

[...]

 74     /**
 75      * Get an instance of the TrackStore
 76      *
 77      * @return A TrackStore instance
 78      */
 79     public synchronized static TrackStore getInstance()
 80     {
 81         if ( _instance == null )
 82         {
 83             _instance = new JukeXTrackStore();
 84             ((JukeXTrackStore)_instance).initialise();
 85         }
 86         return _instance;
 87     }
 88
 89     /**
 90      * Private consrtuctor
 91      */
 92     private JukeXTrackStore()
 93     {
 94         _poolmanager = PoolManager.getInstance();
 95
 96         // TODO: Read the playlists in from the database on startup
 97         _playlists = new HashMap();
 98         _attributes = new HashMap();
 99         _tracksByURL = new HashMap();
100         _tracksByID = new HashMap();
101     }
```

This is a canonical Java Singleton implementation: a private, static reference to the single instance (line 62, as the comment explains), a private constructor (lines 92-101), and a public `getInstance()` method (lines 79-87) that returns a reference to the single instance, creating that instance if it has not yet been created (lines 81-85).

4B Strategy

The entire `jukex.tracksource.filter` package is an example of the Strategy pattern. The Abstract Strategy is the interface `TrackFilter`. The algorithm left to be implemented by the Concrete Strategies is the `match()` method. The classes `AttributeEqualityTrackFilter`, `AttributeRegexTrackFilter`, and `AttributeStartsWithTrackFilter` are the Concrete Strategies.

The Context that makes use of the Strategy is the `FilterPipelineElement` class. The "Context Interface" method that does the actual applying of the Strategy is the `applyFilters()` method:

```
160     private boolean applyFilters(Track t)
161     {
162         boolean retVal = false;
163         if ( filters != null )
164         {
165             Iterator i = filters.iterator();
166             TrackFilter f = null;
167
168             while (i.hasNext())
169             {
170                 f = (TrackFilter)i.next();
171                 retVal |= f.match(t);
172             }
173         }
174         return retVal;
175     }
```

Line 171 is where it dispatches the matching operation to a Concrete Strategy (possibly one of a collection of Concrete Strategies), which has been configured at runtime via a previous call to the method `addFilter()`.

5 Tools

I used the Eclipse IDE for browsing and searching the code. Eclipse's functions for finding all references to a class or method, or all read/write access to an instance variable were very helpful for navigating the code. I also used to connect to the SourceForge CVS server to download the code, and to generate the JavaDocs.

I also used two plugins for Eclipse:

- Metrics (`http://metrics.sourceforge.net/`)
- FEAT (`http://www.cs.ubc.ca/labs/spl/projects/feat/`)

Metrics provides several object-oriented metrics for rating (among other things) coupling and cohesion. It also has a dependency graph view that will highlight cliques—groups of modules that all reference each other and are thus highly coupled.

FEAT provides a browser for cross-cutting concerns. The user creates a concern and then adds classes, fields, and methods to the concern. All references to the members of the concern can then be easily found.