

MapContext Refactor

Motivation:	MapLayer is doing too much
Contact:	full name
Tracker:	GEOT-3136 GEOT-3150 GEOT-3565
Tagline:	the map is back

This page represents the **current** plan; for discussion please check the tracker link above.

- [Summary](#)
- [Status](#)
 - [Tasks](#)
 - [Documentation Changes](#)
- [History and Motivation](#)
- [Proposal](#)
 - [MapLayer to Layer](#)
 - [Planning for DirectLayer](#)
 - [DefaultMapContext to MapContent](#)
 - [Discussion and Ideas](#)

Children:

Summary

MapContext and MapLayer have really grown over time - and in some cases have diverged from their actual use.

The focus is on cleaning up MapLayer:

- MapLayer replaced by Layer class
- Specific Layer subclasses for different kinds of content; this is an open ended set allowing additional kinds of layers to be added over time for TileServers, Google Maps and so forth
- DefaultMapLayer re-factored to use an internal Layer delegate (so existing code will not be broken; and importantly will not be duplicated)

Over the course of the proposal MapContext and DefaultMapContext were also re-factored:

- Turn MapContext into a class allowing DefaultmapContext to be deprecated (same solution as Query)
- Super class called MapContent introduced
 - MapContent uses layers() method to provide direct access to the layer list
 - MapContent improved "viewport" methods
- Responsibilities for maintaining "area of interest" will be isolated into a separate MapViewport class
 - MapContent can use a Mapviewport internally to maintain screen and where to draw information
 - MapViewport can be used directly when drawing the same MapContent into several tiles

This proposal does not break any existing API; it provides a safe migration path forward (and like the Query proposal) makes use of classes directly for a simplified experience.

Status

This proposal is ready for voting; initial commit targeted for Sunday June 12th.

Voting has not started yet:

- +1 [Andrea Aime](#)
- +1 [Ben Caradoc-Davies](#)
- +0 [Christian Mueller](#)
- +1 [Ian Turton](#)
- +0 [Justin Deoliveira](#)
- +1 [Jody Garnett](#) +1
- +1 [Michael Bedward](#)
- +0 [Simone Giannecchini](#)

Tasks

	no progress	✓	done	✗	impeded	⚠	lack mandate /funds/time	?	volunteer needed
--	-------------	---	------	---	---------	---	--------------------------	---	------------------

Initial refactoring:

1. ✓ Layer abstract class
 2. ✓ Split apart DefaultMapLayer into specific implementations
 - ✓ FeatureLayer
 - ✓ GridCoverageLayer
 - ✓ WMSLayer
 - ✓ GridReaderLayer
 3. ✓ Introduce DirectLayer subclass
 - ✓ MessageDirectLayer - example implementation
 4. ✓ Refactor DefaultMapLayer to use a delegate Layer
 - ✓ implement methods making use of internal Layer
 - ✓ implement equals and hashCode against internal Layer; making it a pure wrapper
 5. ✓ Refactor DefaultMapContext
 - ✓ Refactor renderer methods into MapContent and MapViewport
 - ✓ Remove Collection<Object> support
 - ✓ Change MapContext into a class extending MapContent
- ✓ <http://jira.codehaus.org/browse/GEOT-3136>

Related activities:

- ⚠ Update internals of GTRenderer implementations to take advantage of new classes
 - ✓ Waiting on screen map patch
 - ✓ <https://jira.codehaus.org/browse/GEOT-3150> (duplicated)
 - <http://jira.codehaus.org/browse/GEOT-3565> (patch from mbedward)
 - Change GTRender interface to expect MapContent (and MapViewport?)
 - Update implementation to use layers() list; it is a CopyOnWriteArray so it is safe
 - Delegate to DirectLayer implementation
- ⚠ Update gt-swing code to take advantage of new classes

Documentation Changes

- Target for sphinx documentation

History and Motivation

The MapContext (and MapLayer) classes are the last on [Jody Garnett's](#) top-to-bottom QA run for GeoTools 3.0. I am not quite sure of the history and motivation behind these classes; I suspect they were created by James in his early work on the SLD specification, or perhaps Martin in his working a "Lite" Java renderer. Oh wait it says Cameron Shorter (gasp!)

Here is where we are at today:

```
public interface MapLayer {
    getFeatureSource()
    getSource()
    getStyle()
    setStyle(Style)
    getTitle()
    setTitle(String)
    isVisible()
    setVisible(boolean)
    isSelected()
    setSelected(boolean)
    getQuery()
    setQuery(Query)
    getBounds()
    addMapLayerListener(MapLayerListener)
    removeMapLayerListener(MapLayerListener)
}
```

There is **one** implementation with an impressive array of constructors:

```

public class DefaultMapLayer implements
MapLayer {
    DefaultMapLayer(FeatureSource, Style,
String)
    DefaultMapLayer(CollectionSource, Style,
String)
    DefaultMapLayer(FeatureSource, Style)
    DefaultMapLayer(FeatureCollection, Style,
String)
    DefaultMapLayer(Collection, Style,
String)
    DefaultMapLayer(FeatureCollection, Style)
    DefaultMapLayer(Collection, Style)
    DefaultMapLayer(GridCoverage, Style)

DefaultMapLayer(AbstractGridCoverage2DReader
, Style, String, GeneralParameterValue[])

DefaultMapLayer(AbstractGridCoverage2DReader
, Style, String)

DefaultMapLayer(AbstractGridCoverage2DReader
, Style)
    DefaultMapLayer(GridCoverage, Style,
String)
    ...
}

```

Over time they have been hooked up to more and different kinds of data with some truly amazing twists and turns in order to:

- Render a FeatureCollection: This is obviously the start (ie what MapLayer) is responsible for rendering. These days it is **always** wrapped up in a DataUtilities.collection(featureCollection) call so that a FeatureSource is available.
- Render a FeatureSource: This is what we want to work with today; the FeatureSource api allows the renderer to perform queries against the data; and better yet look at the capabilities of the feature source in order to perform these efficiently; decimate the geometry and other assort tricks
- Render a Grid Coverage: formally GridCoverages extend Feature; so you should just be able to render one correct? Well for GeoTools you package up the GridCoverage in a Feature with a magic FeatureType which it knows to un wrap it out of the Feature and then draw it as a GridCoverage.
- Render a Grid Coverage: You thought that was fun? Well after looking at the fun low memory use provided by FeatureSource GeoTools 2.3 had a go at making use of a GridCoverageReader. Just pop a **read** into your feature and proceed like normal (can that be considered normal?)
- Web Map Server Layer: a couple of attempts have been tried; the current one ends up as a GridCoveargeReader as above and is working quite well
- Collection: a collection can also be used to render "Plain old Java Objects"; all that is needed expression based access provided by a PropertyAccessor implementation - currently supports use of Java Beans

Notes:

- My original idea was to follow the progress of the "Open Web Service" Context specification; it is formally defining different layers.
- The OSGeo Foundation **standards** list is coming up with OWS Context conventions for more then just OGC services (WMS-C, Google Maps etc...)

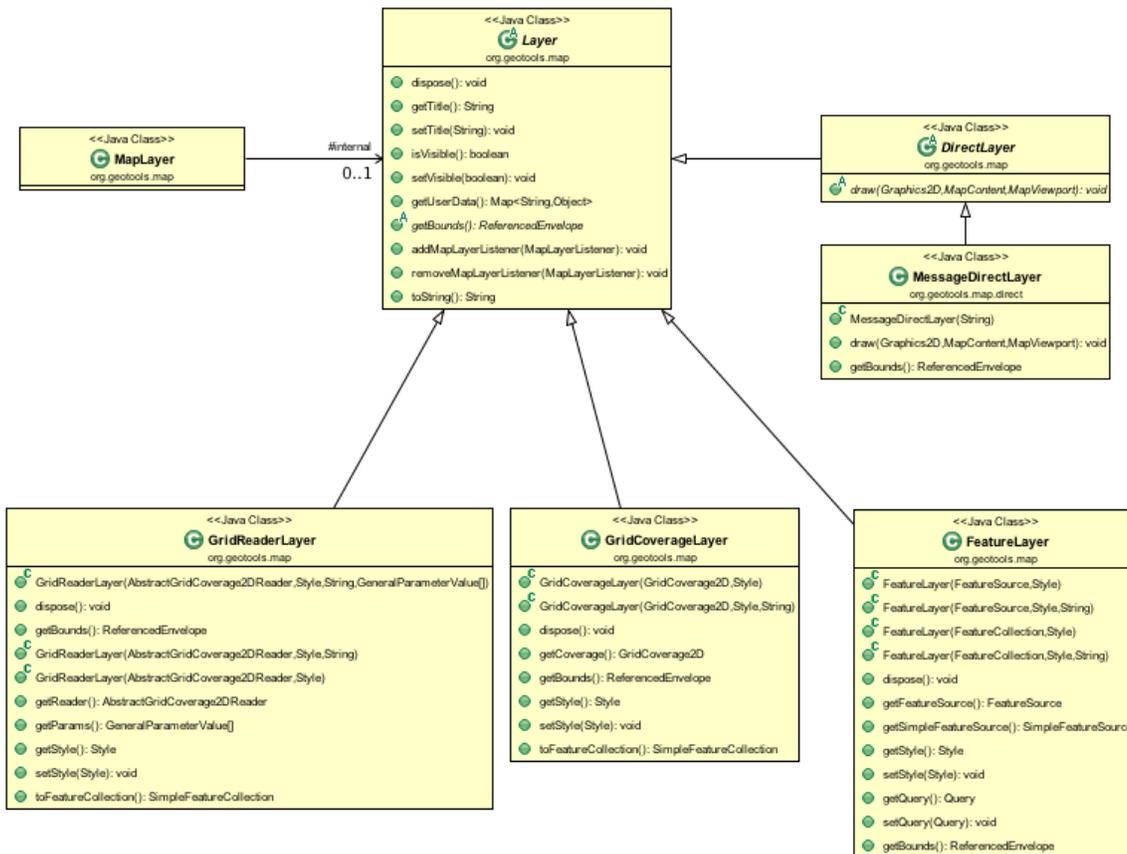
Proposal

So the proposal is pretty straight forward; a one two punch.

MapLayer to Layer

Here is a plan to tame map layer; while not breaking any existing client code:

1. Introduce "Layer" as an abstract super class
 2. Create subclasses for each kind of content: GridCoverageLayer, FeatureSourceLayer, WMSLayer, etc... and Unit test the heck out of them
 3. Set up DefaultMapLayer to have a delegate Layer; have each constructor create the appropriate Layer instance to act as the delegate
- Change MapLayer over to a class; and make it a pure data object with equals and hashCode based on the internal Layer



Note the class MapLayer gained the following method:

```

@Deprecated
public interface MapLayer() {
    ..
    Layer toLayer();
}
  
```

Notes:

- We are deliberately using an abstract super class rather than an interface for Layer because life is short and we don't want to break people over time.
- In general each layer should have data + style at the end of the day
- These are all **concrete** layers represented as specific combinations of data+style
- New code examples will use the concrete classes directly; however DefaultMapLayer constructors will still function

Planning for DirectLayer

1. We can now introduce the concept of "DirectLayer" that is a layer that is responsible for its own paint method
 - This can be used for WMS
 - Map decorations such as scalebars or grids

2. Other concrete Layers

- TileSet layers (provided by WMS-C or WMTS etc...)

DefaultMapContext to MapContent

Not a lot of thought needed on this one; the Interface/Class has grown over time and can be cut back:

- remove layer management methods; replace with direct access to a layers list
- remove addLayer variations as Layer instances should make this explicit now
- remove "ogc context" information such as title, keywords, abstract and replace with a "user data" map
- Set up MapContent <|-- MapContext <|-- DefaultMapContext
- Isolate "viewport code" into a MapViewport class



```
setKeywords(String[]): void
getTitle(): String
setTitle(String): void
addPropertyChangeListener(PropertyChangeListener): void
removePropertyChangeListener(PropertyChangeListener): void
```

I would like to move to an abstract class here; I can make DefaultMapContext extend it so that there is no code breakage.

Discussion and Ideas

Andrea had a number of ideas, but I could not keep track of them all:

- remove Collection<Object> as it is a pain to support and not used
- map context taking a referenced envelope is a pain for users (as it is hard to make referenced envelope of the correct aspect ratio)
 - the idea of "location+scale" - andrea did not like this because scale is hard to define. The idea of using "zoom" (ie ratio of pixel to map unit) may however be okay
 - mbedward would like to see this viewport model stuff separate from the map context
- there is a handover between streaming renderer and grid coverage renderer, perhaps this could be better represented as a DirectLayer
- Imagine we need a bit of a review of how a MapLayer is connected with some code that is capable of rendering it. We may be able to handle this by producing a List<DirectLayer> where some of the entries **are** a specific renderer "Wrapped around" the original Layer
- Taking these kind of steps should allow the rendering system to be better able to be multithreaded
- Some of the methods are specific to the visual JMapPane such as isSelected; once again this could be handled as a wrapper specific to the application in question

Discussion of listeners:

- Implementation uses CopyOnWriteArray for threadsafety; and is aggressive about setting up listeners only if client code starts listening
- Looked into using WeakReferences to hold listeners ... but it would clean up the kind of listeners we want to keep (say a listener that writes log information; or sends email) but not clean up the kind of listeners we want to throw away (listeners on user interface / featureSource etc...)
- Explored use of List<Reference<XXXListener>> so that list could contain Reference and WeakReference on a case by case basis - added complexity not worth it

Use of dispose()

- We all don't like the use of dispose()
- We need dispose to clean up the listeners (see above)
- It should remove references to featureSource, grid coverage reader etc .. but not close them
- will use finalize to scream out a warning should dispose() not be called