Data Access Developer's Guide

uDig

October 12, 2004

Refractions RESEARCH INC.

Submitted To: Program Manager
GeoConnections
Victoria, BC, Canada

Submitted By: Jody Garnett
Refractions Research Inc.
Suite 400 – 1207 Douglas Street
Victoria, BC V8W 2E7
E-mail: jgarnett@refractions.net
Phone: (250) 383-3022
Fax: (250) 383-2140

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

The User-friendly Desktop Internet GIS (uDig) Platform is an extensible framework for GIS application development. The uDig Platform is based on the Eclipse Rich Client Platform (RCP).

The Eclipse RCP is structured around the concept of plug-ins: structured bundles of code that contribute functionality to the system. Plug-ins are integrated with the existing system through well-defined extension points.

This document describes the access of spatial information from the perspective of a developer implementing an extension point defined by the uDig Platform.

uDig makes use of the feature model provided by the Geotools toolkit. In addition to these formal extension points, the Geotools data access model is permitted.

2 uDIG PLATFORM ARCHITECTURE

The Rich Client Platform is structured around the concept of plug-ins. Plug-ins contribute to the system only through extending existing plug-ins. Plug-ins may define their own extension points allowing others in turn to contribute behavior.

The architecture allows for interaction without direct dependence, similar to the Java event model. The difference between this extension point system and the traditional publish/subscribe model of events is two fold:

1. A third party, known as Platform, plays the part of a Mediator: a series of `plugin.xml` files are read, in which each plug-in defines its own extension points and those it extends.
2. At runtime a Plug-in can ask the Platform for a list of Plug-ins "listening" on a given extension point.

To bootstrap this process the RCP provides a set of core plug-ins to start things off. The uDig Application extends an Application's extension point.

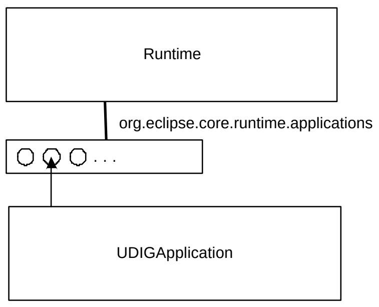


Figure 1: uDig Application extends Runtime

This design allows for incredible scalability, and a fast startup. Plug-ins are not loaded until required, and may be unloaded when not in use.

2.1 uDig Data Access Tiers

The uDig Application is divided into tiers based on the level of data abstraction.

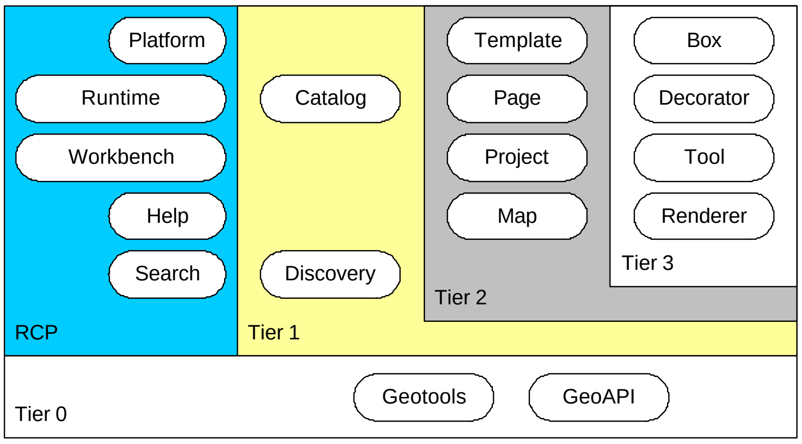


Figure 2: uDig Data Access Tiers

A uDig plug-in communicates with spatial information via the interfaces defined for its tier. When extending the uDig application you should keep this in mind – or you may end up doing more work than you need to.

Tier 0: Toolkit Access	The Toolkits used by uDig provide their own plug-in architecture (which is beyond the scope of this document).
Tier 1: Catalog	Files, Services and Transactions
Tier 2: Project	FeatureTypes
Tier 3: Rendering	Layers and Features

In each case, moving up to a higher tier provides you with a more specific data abstraction, and more work is done for you behind the scenes.

3 TIER ZERO - STANDARDS AND EXISTING TOOLKITS

uDig currently is influenced by the following toolkits:

- Geotools – provides data sources, rendering and other services
- Deegree – reference implementation of Catalog Services version 2.0
- GeoAPI – defines OGC compliant interfaces

Between these projects we have several interfaces for data access and discovery:

- Discovery – (pending) from Catalog Services version 2.0
- Catalog – from Catalog Services version 1.0
- DataStore – data sources via a FeatureType model
- Repository – used for cross-data-source operations
- Grid Coverage Exchange – access to Raster formats

We have chosen to use these interfaces, where possible, when advertising the capabilities of the UDIG framework. This has been done for two reasons:

1. GeoAPI interfaces are often standards-based; we can assume that existing GIS developers will be familiar with the abstractions and terminology therein
2. Allows for client code reuse with other toolkit-based projects

Over the course of this project we will be simplifying several of these interfaces.

3.1 Discovery

Catalog Services version 2.0 defines a revised Catalog model with an explicit Discovery interface.

Intended implementation:

```
interface Discovery extends Catalog{
  RecordType describeRecord(RecordTypeRequest request)
  Domain domain( DomainRequest request )
  Present present( PresentRequest request )
  QueryResponse query( QueryRequest request )
}
interface QueryResponse {
 String getResultSetID()
 ResultType getResultType()
 Collection<Record> retrievedData()
 int getCursorPosition()
 int getHits();
}
```

The specification currently defines retrievedData as a String (that would require parsing). In the proposed API above, Record would be similar to CatalogEntry in the next session.

3.2 Catalog

GeoAPI defines a Catalog API based on the Catalog version 1.0 document. This API consists of the following interfaces:

```
interface Catalog {
  void add( CatalogEntry );
  void iterator();
  QueryResult query( QueryDefinition query );
  void remove( CatalogEntry );
}
interface CatalogEntry {
  String getDataName();
  Map(<String>,<Metadata>) metadata();
  Object getResource();
}
```

MetadataEntity is a placeholder for Metadata Information. Please see Appendix C for an overview of applicable Metadata information.

3.3 DataStore Read

DataStore is the Geotools API for data source access. It is used to access both file and Service information. Wide ranges of data sources are supported from simple Shapefile access through to Database and Web Feature Services.

DataStore operates on an OGC Feature Model notable in its use of XPATH queries to access nested attribute type information. Creation of everything from FeatureTypes to Filter Expression is controlled by the use of Factories.

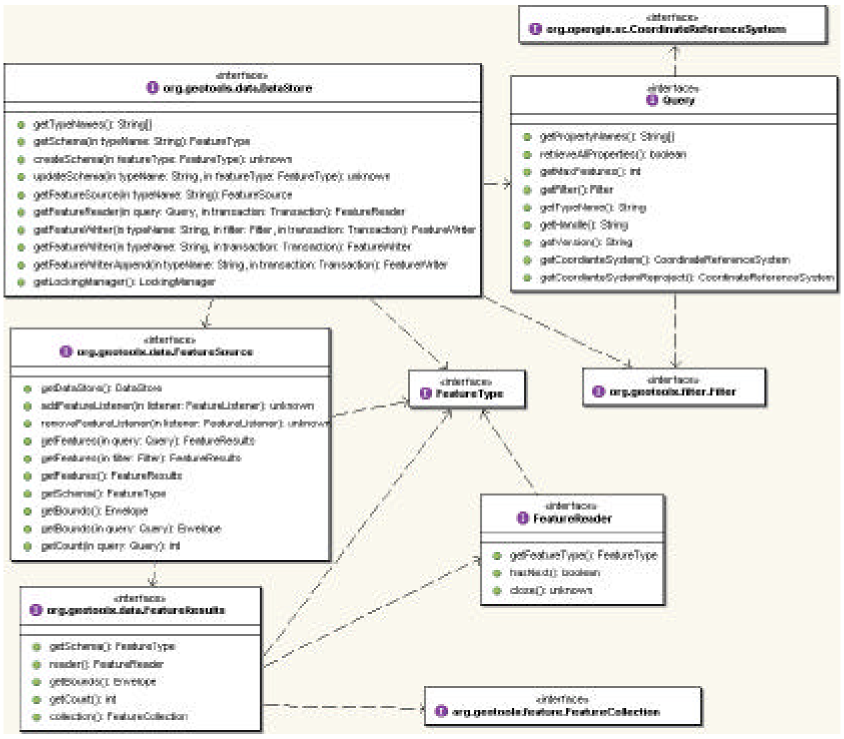

«interface»
org.opengis.sc.CoordinateReferenceSystem
«interface»
org.geotools.data.DataStore
getTypeNames(): String[]
getSchema(in typeName: String): FeatureType
createSchema(in featureType: FeatureType): unknown
updateSchema(in typeName: String, in featureType: FeatureType): unknown
getFeatureSource(in typeName: String): FeatureSource
getFeatureReader(in query: Query, in transaction: Transaction): FeatureReader
getFeatureWriter(in typeName: String, in filter: Filter, in transaction: Transaction): FeatureWriter
getFeatureWriter(in typeName: String, in transaction: Transaction): FeatureWriter
getFeatureWriterAppend(in typeName: String, in transaction: Transaction): FeatureWriter
getLockingManager(): LockingManager
«interface»
Query
getPropertyNames(): String[]
retrieveAllProperties(): boolean
getMaxFeatures(): int
getFilter(): Filter
getTypeName(): String
getHandle(): String
getVersion(): String
getCoordinateSystem(): CoordinateReferenceSystem
getCoordinateSystemReproject(): CoordinateReferenceSystem
«interface»
org.geotools.data.FeatureSource
getDataStore(): DataStore
addFeatureListener(in listener: FeatureListener): unknown
removeFeatureListener(in listener: FeatureListener): unknown
getFeatures(in query: Query): FeatureResults
getFeatures(in filter: Filter): FeatureResults
getFeatures(): FeatureResults
getSchema(): FeatureType
getBounds(): Envelope
getBounds(in query: Query): Envelope
getCount(in query: Query): int
«interface»
FeatureType
«interface»
org.geotools.filter.Filter
«interface»
FeatureReader
getFeatureType(): FeatureType
hasNext(): boolean
close(): unknown
«interface»
org.geotools.data.FeatureResults
getSchema(): FeatureType
reader(): FeatureReader
getBounds(): Envelope
getCount(): int
collection(): FeatureCollection
«interface»
org.geotools.feature.FeatureCollection

Figure 3: DataStore Read Access

As the above diagram illustrates, there are two APIs for access to spatial information:

- **FeatureReader**: an iterator of Features (sequential access)
- **FeatureSource**: provides getBounds, getCount and getFeatures operations. For most DataStore implementations these are optimized to make use of the native functionality of the underlying service.

In addition to the API above many DataStores implement Catalog to provide access to Metadata based FeatureType queries. In this case the CatalogEntry.getResource() returns the FeatureSource associated with the FeatureType.

3.4 DataStore Write

DataStore offers a comprehensive writing API with support for Transactions and optimized updates.

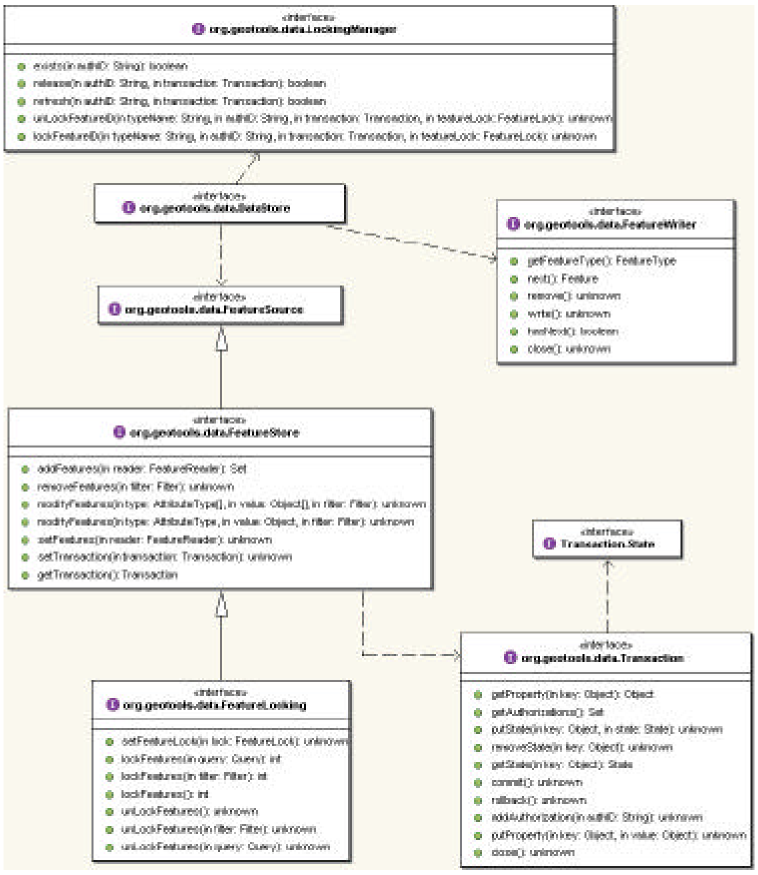

«interface»
org.geotools.data.LockingManager
exists(in authID: String): boolean
release(in authID: String, in transaction: Transaction): boolean
refresh(in authID: String, in transaction: Transaction): boolean
unLockFeatureID(in typeName: String, in authID: String, in transaction: Transaction, in featureLock: FeatureLock): unknown
lockFeatureID(in typeName: String, in authID: String, in transaction: Transaction, in featureLock: FeatureLock): unknown
«interface»
org.geotools.data.DataStore
«interface»
org.geotools.data.FeatureWriter
getFeatureType(): FeatureType
next(): Feature
remove(): unknown
write(): unknown
hasNext(): boolean
close(): unknown
«interface»
org.geotools.data.FeatureSource
«interface»
org.geotools.data.FeatureStore
addFeatures(in reader: FeatureReader): Set
removeFeatures(in filter: Filter): unknown
modifyFeatures(in type: AttributeType[], in value: Object[], in filter: Filter): unknown
modifyFeatures(in type: AttributeType, in value: Object, in filter: Filter): unknown
setFeatures(in reader: FeatureReader): unknown
setTransaction(in transaction: Transaction): unknown
getTransaction(): Transaction
«interface»
Transaction.State
«interface»
org.geotools.data.Transaction
getProperty(in key: Object): Object
getAuthorizations(): Set
putState(in key: Object, in state: State): unknown
removeState(in key: Object): unknown
getState(in key: Object): State
commit(): unknown
rollback(): unknown
addAuthorization(in authID: String): unknown
putProperty(in key: Object, in value: Object): unknown
close(): unknown
«interface»
org.geotools.data.FeatureLocking
setFeatureLock(in lock: FeatureLock): unknown
lockFeatures(in query: Query): int
lockFeatures(in filter: Filter): int
lockFeatures(): int
unLockFeatures(): unknown
unLockFeatures(in filter: Filter): unknown
unLockFeatures(in query: Query): unknown

Figure 4: DataStore Write Access

This time there are four classes to consider:

- **Transaction:** offers workflow control and rollback. Can be used with several DataStores at once.
- **FeatureWriter:** an iterator of Features supporting modification.
- **FeatureStore:** provides add, update and remove as high level requests that are optimized for most DataStores.
- **FeatureLock:** offers locking for the duration of a Transaction and WFS Style Long Term Transaction support.

3.5 DataStore Creation

As promised, even DataStore creation is controlled by use of a Factory.

```
interface DataStoreFactorySpi {
   boolean canProcess( Map params );
   DataStore createDataStore( Map params );
   DataStore createNewDataStore( Map params );
   String getDescription();
   Param[] getParametersInfo();
   ParameterGroupDescriptor getParameterGroup();
}
```

The map used to create a DataStore is described by:

- A legacy getParametersInfo() method
- ParameterValueGroup[1] adapted for general use by GeoAPI

There are several difficulties with this approach:

- A Map is not optimal – for drag and drop support we would like to use a File or URL directly
- The ParameterValueGroup is difficult to understand
- ISO 19119 is designed to describe Services

The plug-in system FactorySPI is used to “discover” the appropriate DataStoreFactory based on a Map of parameters. In practice many DataStoreFactory implementations require a “magic” key be entered in Map to support this functionality.

It is likely that DataStore will be simplified over the coming weeks to take these factors into consideration.

3.6 Repository

Offers cross DataStore operations, and FeatureSource access by namespace URI and type name irrespective of DataStore.

```
Interface Repository {
  Map getDataStores();
  SortedMap getFeatureSources();
  Set getPrefixes();
  boolean lockExists(String lockID);
  boolean lockRefresh(String lockID, Transaction transaction)
  boolean lockRelease(String lockID, Transaction transaction)
  FeatureSource source(URI namespace, String typeName)
}
```

[1] OpenGIS® Spatial Referencing by Coordinates (Topic 2), http://www.opengis.org/docs/03-073r1.zip

It is an interesting consequence of Locks being held across DataStores that a central Repository is required to unlock them safely.

3.7 Grid Coverage Exchange

Grid Coverage Exchange[2] provides access to Grid Coverage information. This has been assembled as part of the uDig project, and has been donated to the GeoAPI project.

```
interface GridCoverageExchange {
  void dispose();
  Format[] getFormats()
  GridCoverageReader getReader(Object source)
  GridCoverageWriter getWriter(Object destination, Format format)
}
interface Format {
  String getDescription()
  String getDocURL()
  String getName()
  ParameterValueGroup getReadParameters()
  String getVendor()
  String getVersion()
  ParameterValueGroup getWriteParameters()
}
interface GridCoverageReader {
  String getCurrentSubname()
  Format getFormat()
  GridCoverage read(GeneralParameterValue[] parameters)
}
interface GridCoverageWriter {
  Object getDestination()
  Format getFormat()
  void write(GridCoverage coverage, GeneralParameterValue[] parameters)
}
```

The one difficulty with the above API is that one gets no indication what the source parameter is when acquiring a GridCoverageReader. Once again Catalog has stepped in to the rescue. Many GCE implementations are using the Catalog API to provide client code with an avenue to discover a source parameter (based on a remote Web Map Server, or a local file system).

The complete API also provides facilities for GridCoverageReader and GridCoverageWriter to be used with streamed content. This has been omitted for brevity.

[2] Grid Coverages Implementation Specification, http://www.opengis.org/docs/01-004.pdf

4 TIER ONE – FILES, SERVICES AND TRANSACTIONS

Tier One represents the first level of the uDig Platform. There is one service provided over and above the raw functionality defined by the toolkits: the prevention of File and Service duplication.

4.1 Direct Toolkit Access (Forbidden)

The usual method of accessing information from a Shapefile is as follows:

```
// Define Connection parameters for the shapefile
Map params = new HashMap();
Map.put( “url”, shapeURL );

// Connect!
DataStore store FactoryFinder.createDataStore( params );
String typeName = store.getTypeNames()[0];
FeatureSource source = store.getFeatureSource(typeName);
FeatureType type = source.getSchema();

Expr expr = Exprs.attrb( “name” ).eq( “Victoria” );
FeatureResults results = source.getFeatures( expr.filter( type ));

FeatureReader reader = results.reader();
try {
  while( reader.hasNext() ){
    System.out.println( reader.next() );
  }
}
finally {
  reader.close();
}
```

To adapt this code for use in uDig we **must not**:

- make use of FactoryFinder; nor
- construct our own DataStore by hand.

4.2 Catalog Access

Many DataStores would really like to be singletons (they may maintain a cache of Database Connections, for example). It is up to us, the developers, to prevent more than once instance being created for the same data source.

Normally, when you use Geotools you are on your own; but this time you have Catalog to help.

```
Map params = new HashMap();
Map.put( “url”, shapeURL );

DataStore store CatalogPlugin.findDataStore( params );
```

In addition there are several helper methods:

```
Object CatalogPlugin.open( File file )
Object CatalogPlugin.open( URL url)
Object CatalogPlugin.connect( ParameterValueGroup params)
```

4.3 Toolkit Access

The Catalog plug-in makes use of the traditional Geotools/GeoAPI data access and discovery interfaces to allow for code reuse.

The Registry uses the IAdaptable facilities provided by Eclipse to allow the support of an arbitrary interface.

```
Registry registry = CatalogPlugin().getDefault().getRegistry();

registry.getAdapater( org.geoapi.catalog.Catalog.class );
registry.getAdapater( org.geoapi.catalog.Discovery.class );
registry.getAdapater( org.geotools.data.Repository.class );
```

There is a large body of code willing to operate against these APIs. In particular, a complete set of validation tests operates against org.geotools.data.Repository and is directly reusable.

4.4 Catalog Extensions

Catalog has one responsibility for client code:

- Catalog prevents duplicate DataStore instances.

4.4.1 Extension net.refractions.udig.catalog.content

The content extension point allows the plug-ins to contribute kinds of content that the platform catalog understands. There are two forms of contribution: *file association* and *mime association*.

A *file association* represents a file format and naming conventions. This information is used during File import and File Drag-and-Drop operations.

A *mime association* represents a data stream format and mime type. This information is used during URL import and URL Drag-and-Drop operations.

This extension point requires an implementation of net.refractions.udig.catalog.IFileAssociation or net.refractions.udig.catalog.IMimeAssociation.

This extension point is used internally to register existing Geotools capabilities with the GIS Platform. You can use this extension point to provide additional file associations to existing Geotools DataStores, or provide your own DataStore implementation directly to the uDig Application, forgoing the Geotools FactoryFinder.

To register your new data:

```
Registry registry = CatalogPlugin().getDefault().getRegistry();
registery.add( new FeatureSourceRegistryEntry( server ) );
registery.add( new GCRegistryEntry( gce ) );
```

If required, you may make your own instance of RegistryEntry:

```
interface RegistryEntry extends CatalogEntry {
  void addRegistryListener( RegistryListener listener )
  void removeRegistryListener( RegistryListener listener )
}
```

RegistryEntry operates as a CatalogEntry with event notification.

It is expected that getResource() is a FeatureSource, or GridCoverage when used with File or URL association.

4.4.2 Extension net.refractions.udig.catalog.service

The service extension point allows plug-ins to contribute kinds of servers that the platform catalog understands. There are two contributions: *data source*, and *open web service.*

A *data source* represents a service captured by the Geotools DataStore API such as a Database. The extension point offers persistence for connection parameters at the project level. Authorization is maintained on a per user basis and will not be shared between projects.

An *open web service* represents a service with a GetCapabilities document. The extension point offers persistence for connection parameters at the project level. Authorization is maintained on a per user basis and will not be shared between projects. The framework operates as a true registry allowing an Open Web Service to have its GetCapability cached.

The extension point requires an implementation of net.refractions.udig.catalog.IDataSource, net.refractions.udig.catalog.IOpenWebService.

This extension point is used internally to register existing Geotools capabilities with the GIS Platform.

To actually register your new service:

```
Registry registry = CatalogPlugin().getDefault().getRegistry();
registery.add( new DataStoreRegistryEntry( server ) );
registery.add( new GCERegistryEntry( gce ) );
registery.add( new WMSRegistryEntry( wms ) );
```

If required, you may make your own instance of RegistryEntry:

```
interface RegistryEntry extends CatalogEntry {
  void addRegistryListener( RegistryListener listener )
  void removeRegistryListener( RegistryListener listener )
}
```

RegistryEntry operates as a CatalogEntry with event notification.

It is expected that getResource() is a Catalog when used with a DataStore, Grid Coverage Exchange, or Service.

Note: It is common for Open Web Services such as WMS and WFS to extend both net.refractions.udig.catalog.service and net.refractions.udig.catalog.content while providing implementations of IopenWebService and ImimeAssociation.

4.5 Discovery Extension

Discovery is used to connect Catalog and Discovery Services to the RCP Search System.

4.5.1 Extension net.refractions.udig.catalog.discovery

The discovery extension point allows plug-ins to contribute kinds of catalogs that the platform catalog understands. There is a single contribution: *discovery*.

A *discovery* service represents a lookup service capable of being searched. Parameter information and Authorization information are maintained on a per user basis.

The extension point requires an implementation of net.refractions.udig.catalog.IDiscovery.

This extension point is used internally to register the existing Catalog with the Discovery system (as used by Search).

4.6 *Catalog User Interface Extensions*

This is the first really interesting extension point; one where you actually get to affect what appears on the screen.

Many of the extension points support the use of the IOp interface to define user interface needs. This represents a suitable compromise that can be maintained both by uDig and Toolkit providers.

```
interface IOp {
  InternationalString getName();
  InternationalString getDescription ();
  Map params( Object resource );
}
```

The Map params is initially a set of "defaults" that may be used by the user to configure the activity. The Map keys are of type InternationalString that are suitable for display.

When this contribution is used to populate the context menu of a FeatureType in the Catalog UI, a small wizard will be activated with the following workflow:

1. Based on resource selected, the appropriate IOps will be presented to the user. The user selects the desired operation.
2. If the Map params is not null, a Dialog will be created and the user can choose to modify the values. The resource is provided allowing the IOp to supply custom defaults.
3. The appropriate op method will be called.
4. If an Exception is thrown it will be sent to the Log.
   - You may wish to provide your own feedback via the issues list
   (ie. when performing data validation)

4.6.1 Extension net.refractions.udig.catalog.ui.service

Supports access of services, including the creation of new spatial information.

```
interface IDataStoreOp extends Iop {
  void op( DataStore service ) throws Exception;
}
interface IWMSOp extends IOp {
  void op( WebMapServer wms ) throws Exception;
}
interface IGCE extends IOp {
  void op( GridCoverageExchange gce ) throws Exception;
}
```

4.6.2 Extension net.refractions.udig.catalog.ui.featureType

A slightly easier to use version of service, it operates at the level of a FeatureSource. Client code is responsible for doing any Transaction stuff they want.

The extension point requires an implementation of net.refractions.udig.catalog.ui.IFeatureTypeOp.

IFeatureTypeOp receives several objects during execution of its op method:

```
interface IFeatureTypeOp extends IOp{
  FeatureType op( FeatureType schema ) throws Exception;
}
interface IFeatureSourceOp extends IFeatureTypeOp {
  void op( FeatureSource source ) throws Exception;
}
interface IFeatureStoreOp extends IFeatureTypeOp {
  void op( FeatureStore source ) throws Exception;
}
interface ILayerOp extends IOp {
  void op( WMSLayer layer ) throws Exception;
}
interface IGridCoverageOp extends IOp {
  void op( GridCoverage grid ) throws Exception;
}
```

Note: When working with a FeatureStore actual data modification is expected; the FeatureStore you receive will be using a Framework supplied and managed Transaction. If your op completes without exception, the Transaction will be committed.

5 TIER TWO – FEATURETYPES

Tier Two defines the bulk of the uDig Platform; at this level several new constructs are available:

- Project: provides persistence of activity based information
- Map: provides visualization services and a viewport
- Page: physical output
- Layer: provides the combination of spatial information with an appearance
- Selection: filters selected by the user for “special” activity

In addition, several concepts are now managed by the user interface:

- Transaction: transaction control is now maintained by the Map
- Area of Interest: a viewport is defined by the Map indicating what is visible

5.1 Project Extension Points

5.1.1 Extension net.refractions.udig.project.persistence

Defines a persistence mechanism for plug-ins that wish to have information stored with the project. Map and Page should use this extension point, but have been optimized due to frequent use.

5.1.2 Extension net.refractions.udig.project.templates

Templates are used to create Pages to be sent to the printer. They are aware of which paper sizes they are capable of working with, and must be accompanied with a name, description and an optional preview graphic, to display to the user what its outline looks like. The supported paper sizes are declared in the extension point schema, allowing only the desired templates to be loaded.

Most templates, when created, will be given access to a Map, which can be used to create other decorators, such as a legend or scalebar. No modifications should be performed on the map when a template is created. The only instance where a template should create a Page with no map, is when the given map is null. This indicates that a blank Page is to be created.

Each extension shall extend the net.refractions.udig.project.AbstractTemplate class.

```
public abstract class AbstractTemplate {
      public abstract String getName();
      public abstract String getDescription();
      public abstract Image getPreview();
      public abstract Page create(Map map);
}
```

5.1.3 Extension net.refractions.udig.project.layerOp

A layer operation extension is an extension that performs an operation on one or more layers in a map. The layer operation extension point will allow programmatic access to a layer. Layer operations would be listed under the layer menu and in context menus in the layer view when a layer is right-clicked. In order to provide a scalable solution, the layer operation extension point will require an extension to declare a filter. The filter will allow the menu managers to determine whether the layer operation is interested in the layer. If it is, then the operation will be added to the menu. Otherwise, the operation will be left out of the menu.

The following interface defines the layer object that a LayerOp obtains from uDig:

```
public interface Layer {
    public StyleMemento getStyle();
    public MetadataEntity getMetadata();
    public int getZOrder();
    public boolean isVisible();
    public String getName();
    public Query getQuery();
    public FeatureStore getFeatureStore();
    public FeatureSource getFeatureSource();
}
```

A LayerOp must implement the following interface:

```
public interface LayerOp extends IOp{
    public void op( Layer layer );
}
```

6 TIER THREE – LAYERS AND FEATURES

Tier Three defines the very narrow scope of rendering spatial information. This tier starts to manage Printers, JAI, Image and Graphics 2D constructs for client code.

6.1 Extension Points

6.1.1 Extension net.refractions.udig.project.ui.decorator

A Decorator adds meaning to a map or page. The Decorator interface that a decorator extension must extend consists of a draw() method and a setToolkit() method. The toolkit that is passed in provides context for the decorator. For example, a scalebar requires a viewport model and the ability to calculate the extent of a map; the toolkit provides access to that data. The toolkit that the decorator receives is a read-only toolkit. A decorator is not permitted to change uDig model information. It can only display information.

```
public interface Decorator {
    public void draw(ViewportGraphics graphics);
    public void setToolkit(Toolkit toolkit);
}
```

The toolkit class has the following interface:

```
public class Toolkit {
    public ViewportModel getViewportModel();
    public ContextModel getContextModel();
    public MapDisplay getDisplay();
    public Map getMap();
    public Project getProject();
}
```

A simple example of a decorator extension follows:

```
public class ImageDecorator implements Decorator {
    RenderedImage image;
    private Toolkit toolkit;
    public void draw( ViewportGraphics graphics ) {
      graphics.drawImage(image, 0,0 );
    }
    public void setToolkit( Toolkit toolkit ) {
      this.toolkit=toolkit;
    }
}
```

6.1.2 Extension net.refractions.udig.project.ui.render

A Renderer interprets spatial data and represents the data in a visual manner. In uDig there are different types of renderers that can render different types of data. For example, a feature renderer can render feature data. A WMS renderer can communicate with and render images from a web map server.

The API requires that a Renderer extension must create a RenderMetrics class, which can provide metrics about how fast a renderer can provide its service; and a RenderMetricsFactory class, which can determine if a data source can be rendered by the renderer and can create RenderMetrics that provide metrics with regard to a particular data source. Furthermore, a renderer extension must provide an implementation of the Renderer Interface, normally by extending the abstract superclass that handles threading and event notification for the renderer.

```
public abstract class AbstractRenderer{
    protected abstract void stopRendering();
    protected abstract void draw(IProgressMonitor monitor);
    public abstract void render(Graphics2D graphics);
    public void setToolkit(RenderToolkit tools);
}
```

For the sake of simplicity and security a toolkit object is provided that the renderer can use. The toolkit object is a facade for the renderers into the uDig system. It provides access to the current map, project, viewport and display. It also provides support methods such as pixelToWorld(), which calculates the world location corresponding to a pixel in the display. Lastly, the renderer toolkit provides access to the layer to be rendered and the service object that will provide the spatial data.

```
public class RenderToolkit extends Toolkit {
    public Layer getLayer();
    public Service getService();
}
```

6.1.3 Extension net.refractions.udig.project.ui.tool

The tool extension point allows third-party developers to develop new tools for uDig and is one of the most used points of extension. There are three different type of tools and two ways of grouping tools. The three types of tools are as follows:

- **Action Tool** - A single fire tool that performs a single action and is not modal; a button that sets the viewport so it frames the current selection is an example of an action tool. Action tools must implement the ActionTool interface.

- **Modal Tool -** A tool that has on and off modes. When a modal tool is "on" it waits for user input and reacts on it. An example of a modal tool is the zoom tool. Modal tools must implement the ModalTool interface and are recommended to implement the AbstractModalTool class.
- **Background Tool -** A tool that is always active in the background. A typical background tool would be limited to providing user feedback. An example is the cursor position tool that displays the current mouse location in world coordinates. Background tools must implement the Tool interface and are recommended to implement the AbstractTool class.

To address the need to provide locations for large numbers of tools, developers can add tools to views. In addition, a standard tool view is defined and new tools are added to the tool viewer by default. It is recommended that if a large number of tools are being added, then a new view should be created to hold the tool set.

The tool interfaces are as follows:

```
public interface Tool {
    public void setToolkit(RenderManager rmanager);
    public void fill(Composite parent);
    public void dispose();
}
public interface ActionTool extends Tool {
    public void run();
}
public interface ModalTool extends Tool {
    public void setEnabled(boolean active);
}
```

The setToolkit() method is called by uDig in order to provide a tool with a toolkit object that it can use as a facade for accessing uDig, uDig command factories and send command objects to uDig.

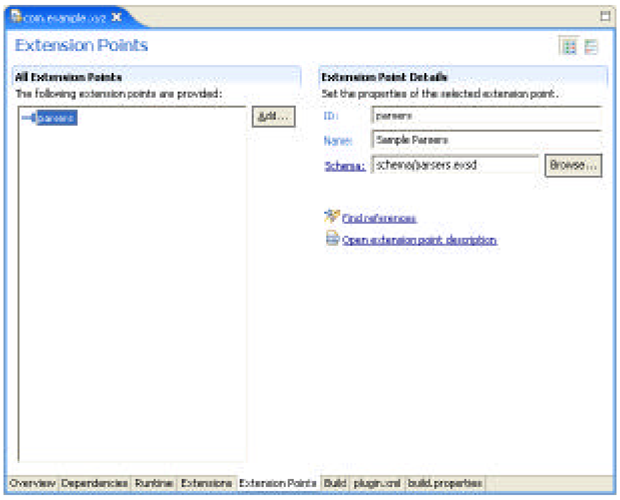
```
public class ToolsToolkit extends Toolkit{
    public DrawCommandFactory getDrawFactory();
    public EditCommandFactory getEditFactory();
    public NavigationCommandFactory getNavigationFactory();
    public SelectionCommandFactory getSelectionFactory();
    public void sendDraw(DrawCommand command);
    public void sendEdit(EditCommand command);
    public void sendNavigation(NavCommand command);
    public void sendSelection(Command command);
}
```

APPENDIX A – ONLINE EXTENSION POINT DESCRIPTION

This documentation has been exported from the current release of uDig 0.4. This documentation is generated and is available as part of your Eclipse development environment.

To view this Extension Point documentation in your Eclipse development environment:

1. Navigate to the Extension points page of your Plug-in manifest editor.



All Extension Points

The following extension points are provided:

parsers

Extension Point Details

Set the properties of the selected extension point.

ID: parsers

Name: Sample Parsers

Schema: schema/parsers.exsd

Figure 5: Extension Point Description

2. Use the "Add..." button to add one of the uDig extension points described in this document.

3. Select the extension point you just added.

4. Click "Open extension point description" to open a document similar to those found in this guide.

APPENDIX B – DATASTORE CONNECTION PARAMETERS

Shapefile Options

Option	Description	Default
url	URL of a shapefile	
memory mapped buffer	Enables/disables memory mapped IO	true for file urls, false for http urls

Web Feature Server Options

Option	Capability
GET_CAPABILITIES_URL	URL to a WFS GetCapabilities document. (This is an explicit url you can use in a browser)
SERVER_URL	A URL to a WFS Service (An incomplete URL that requires a "capability" to be pre-pended)
Option	**Request Method**
USE_POST	Use "Post" to requesting content, allows complex queries
USE_GET	Use "Get" to requesting content, can be cached by proxies
Option	**Authentication**
USERNAME	User name for HTTP authentication
PASSWORD	Password for HTTP authentication

APPENDIX C – METADATA

The MetadataEntity referenced by the Catalog and Discovery APIs is a placeholder for ISO 19119 or ISO 19115 Metadata information. This information is used during the data discovery/search process.

ISO/DIS 19115 Geographic Information[3]

The following table lists Metadata information available through the Discovery API according to XPath expression.

Table 1: Geographic Information Metadata

	Geographic Metadata
Title	Identification.citation / Citation.title
Reference Date	Identification.citation / Citation / Date.date Identification.citation / Citation / dateType
Responsible party	Identification.pointOfContact / ResponsibleParty
Geographic location	DataIdentification.geographicBox DataIdentification.geogrphicIdentifier
Dataset language	DataIdentification.lauguage
Dataset character set	DataIdentification.characterSet
Dataset topic category	DataIdentification.topicCategory
Spatial resolution	DataIdentification.spatialResolution / Resolution.equivalentScale DataIdentification.spatialResolution / Resolution.distance
Abstract	Identification.abstract
Distribution Format	Distribution / Format.name Distribution / Format.version
Extent information	DataIdentification.extent / Extent
Spatial representation	DataIdentification.spatialRepresentationType
Reference system	ReferenceSystem
Lineage	DataQuality / Lineage.statement
On-line resource	Distribution / DigitalTransferOption.onLine / OnlineResource

Table 2: Metadata Information

	Metadata Elements
file identifier	fileIdentifier
standard name	metadataStandardName
standard version	metadataStandardVersion
language	language
Metadata character set	characterSet
Metadata point of contact	contact / ResponsibleParty)
date stamp	dateStamp

[3] Topic 11 – Metadata, http://www.opengeospatial.org/docs/01-111.pdf

ISO 19119 Service Metadata Service[4]

In addition to the queryable properties defined for Geographic Information above, ISO 19119 defines the following useful properties for describing services.

	Service Metadata
Service Type	IdentificationInfo/ServiceIdentification/serviceType
Service Type Version	IdentificationInfo/ServiceIdentification/serviceTypeVersion
OperatesOn	IdentificationInfo/ServiceIdentification/operatesOn/DataIdentification/ citation/Citation/identifier
Operation	IdentificationInfo/ServiceIdentification/ containsOperation/ OperationMetadata/operationName
DCP	IdentificationInfo/ServiceIdentification/containsOperation/ OperationMetadata/operationName.DCP/*

Note: DCP information stands for "Distributed Computing Platform"; these entry points describe the information available in an Open Web Services GetCapabilities Document. Given the appropriate ISO 19119 DCP information our client should not need to request a GetCapabilities Document to interact with a service.

[4] Application Profile for CSW 2.0, https://portal.opengeospatial.org/files/?artifact_id=6495