uDig Requirements Document

June 17, 2004

Submitted To: Program Manager
GeoConnections
Victoria, BC, Canada

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

TABLE OF CONTENTS

1 INTRODUCTION 5

2 PROJECT OVERVIEW.. 6

3 PROJECT REQUIREMENTS 7

3.1.1 Functional Requirements... 7
3.1.2 Non-Functional Requirements 8

4 TERMINOLOGY AND DEFINITIONS... 9

4.1.1 Eclipse Framework Terminology 9
4.2 TERMINOLOGY.. 10
4.3 COMMON ACRONYMS... 10
4.4 MISCELLANEOUS TERMS 11

5 UDIG MODEL AND ARCHITECTURAL OVERVIEW 12

5.1 SYSTEM DATA MODELS.. 13
5.1.1 User Models:.. 13
5.1.2 Architectural Models:... 13
5.2 SYSTEM MODULE OVERVIEW 14
5.3 USER INTERFACE 16
5.3.1 Components.. 18
5.4 CORE.. 19
5.4.1 Components: 20
5.5 DATA DISCOVERY AND ACCESS 21
5.5.1 Components.. 22
5.6 RENDERING 23
5.6.1 Components.. 24
5.7 PRINTING.. 25
5.7.1 Components.. 26
5.8 UDIG EXTENSION FRAMEWORK.. 27

6 APPLICATION REQUIREMENTS... 28

7 UI MODULE REQUIREMENTS... 29

7.1 LAYERS (MAP VIEW).. 30
7.2 VIEWPORT (MAP EDITOR)... 31
7.3 EDIT TOOLBAR (VIEWPORT TOOLBAR).. 32
7.4 SELECTION TOOLBAR (VIEWPORT TOOLBAR)... 32
7.5 SELECTION VIEW (DEPENDS ON VIEWPORT VIEW) 33
7.6 FEATURE VIEW (DEPENDS ON VIEWPORT VIEW)... 34
7.7 QUERY WIZARD... 35
7.8 PRINTING WIZARD 35
7.9 STYLE WIZARD/EDITOR.. 36
7.10 CATALOG VIEW.. 37

8 CORE MODULE REQUIREMENTS... 38

8.1 LAYER MANAGER 39
8.2 CONTEXT MODEL 40
8.3 SELECTION MANAGER 41
8.4 SELECTION MODEL 42
8.5 STYLE MANAGER 43

9 DATA DISCOVERY AND ACCESS MODULE REQUIREMENTS 44

9.1 LOCAL CATALOG 45
9.2 WEB FEATURE CLIENT 46
9.3 WEB MAP CLIENT 47
9.4 DATABASE ACCESS 48
9.5 RASTER FILE FORMATS 49
9.6 SHAPEFILE FORMAT 50

10 RENDERING MODULE REQUIREMENTS 51

10.1 RENDERING MANAGER 52
10.2 VIEWPORT MODEL 53
10.3 RENDERER 54
10.4 RENDERSTACK 55
10.5 FEATURERENDERER 56
10.6 WMCRENDERER 57
10.7 GRIDCOVERAGERENDERER 58

11 PRINTING MODULE REQUIREMENTS 59

11.1 LAYOUT 60
11.2 PAGE 61
11.3 PRINTINGCONTEXT MODEL 62
11.4 PRINTINGENGINE 63

12 UDIG EXTENSION MODULE REQUIREMENTS 64

TABLE OF FIGURES

Figure 1: General Architectural Overview 14
Figure 2: uDig UI Mockup 16
Figure 3: Core Module Interface API 19
Figure 4: Data Discovery and Access Module Interfaces 21
Figure 5: Rendering Module Interfaces 23
Figure 6: Printing Module Interfaces 25
Figure 7: Layer View Component 30
Figure 8: Viewport Component 31
Figure 9: Selection View Component 33
Figure 10: Feature View Component 34
Figure 11: Style Editor Component 36
Figure 12: Catalog View Component 37
Figure 13: Model Interaction 38
Figure 14: Catalog Interaction Diagram 44
Figure 15: Rendering Interaction Diagram 51
Figure 16: Printing Interaction Diagram 59
Figure 17: Extension Interaction Diagram 64

1 INTRODUCTION

This document outlines the requirements for the User Friendly Desktop Internet GIS (uDig). The purpose of this document is to act as a guide for uDig developers. This document is not a design document. Its focus is on identifying the requirements of each module so that they will satisfy the overall project requirements when integrated.

This document is organized as follows:

- **Section 2, Project Overview:** provides an overview of the uDig project itself.
- **Section 3, Project Requirements:** discusses the requirements of the complete project.
- **Section 4, Terminology and Definitions:** a terminology and acronyms reference. Within the terminology section a brief explanation of the terminology conflicts between GIS terminology and Eclipse platform terminology is provided. (Remember: Eclipse is one of the key technologies being used to develop uDig.)
- **Section 5, uDig Model and Architectural Overview:** is a high-level look at the architecture of the project. This section is intended to provide some context for the reader for the later sections.
- **Sections 6-12, Application Requirements: a** list of the requirements of the different modules and sub-modules of the project.

The last sections will be used as a reference for the uDig developers in order to ensure that all the requirements of the individual modules are met, and in satisfying the module requirements the project requirements will be met.

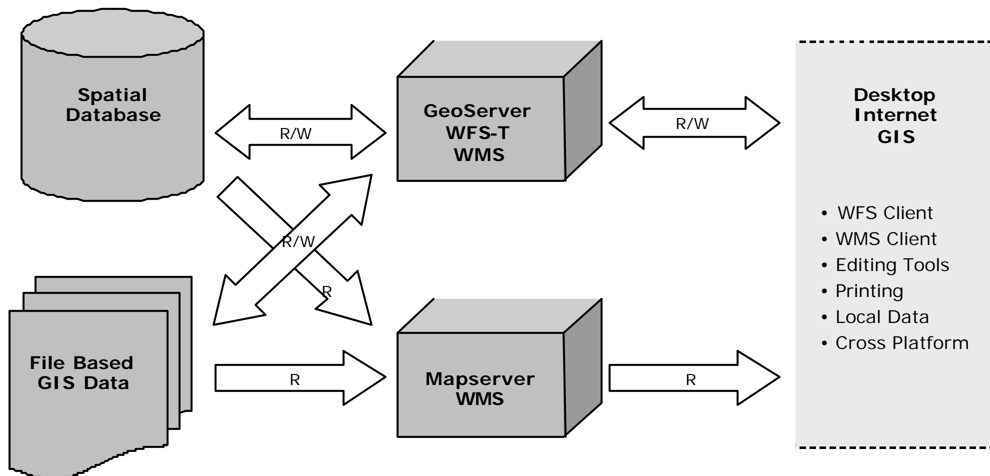
There are two types of requirements that are examined in this document:

- **Functional Requirements**, which constrain and describe the feature set of a module
- **Non-Functional Requirements**, which capture acceptance testing considerations, such as performance and usability

2 PROJECT OVERVIEW

The User Friendly Desktop Internet GIS for OpenGIS Spatial Data Infrastructures project (*uDig*) will create an open source desktop GIS application, to make viewing, editing, and printing data from Open Web Services (OWS) and local data sources simple for ordinary computer users.

Open source components are a critical part of the uDig vision, because they allow organizations to deploy infrastructure widely, in a distributed fashion, without incurring multiple licensing fees. Open source components are also the most tractable for fast support of new OpenGIS interoperability standards.



Interactive desktop access is the missing application in OpenGIS standards-based spatial infrastructures

There are already many different pieces of open source software that implement OpenGIS server standards: Mapserver implements WMS, GeoServer implements WMS and WFS-T, PostGIS implements SFSQL, DeeGree implements WMS and WFS, and so on. However, there is not a single piece of desktop software capable of binding information from all these servers together into a unified desktop view.

uDig is the open source application which will bring OWS data sources to the desktop, and integrate them with local data sources for standard business processes – data viewing, data editing, and data printing.

3 PROJECT REQUIREMENTS

3.1.1 Functional Requirements

The *uDig* application will have the following capabilities:

- **WFS client read/write support**, to allow direct editing of data exposed via transactional Web Feature Servers (WFS-T).
- **WMS support**, to allow viewing of background data published via WMS.
- **Styled Layer Descriptor (SLD) support**, to allow the client-directed dynamic re-styling of WMS layers.
- **Web Registry Server support**, for quick location of available GIS information.
- **Printing support**, to allow users to create standard and large format cartography from their desktops using GIS data sources.
- **Standard GIS file format support**, to allow users to directly open, overlay, and edit local Shape and GeoTIFF files with online data.
- **Coordinate projection support**, to transparently integrate remote layers in the client application where necessary.
- **Database access support**, to allow users to directly open, overlay and edit data stored in PostGIS, OracleSpatial, ArcSDE, and MySQL.
- **Cross-platform support**, using Java as an implementation language, and providing one-click setup files for Windows, OS/X and Linux.
- **Multi-lingual design**, allowing easy internationalization of the interface, with French and English translations of the interface completed initially.
- **Customizability and modularity**, to allow third party developers to add new capabilities, or strip out existing capabilities as necessary when integrating the application with existing enterprise infrastructures.

3.1.2 Non-Functional Requirements

The following constraints are emphasized:

- **Well-Rounded Framework,** built on standard and best-of-breed libraries to offer a sustainable, competitive advantage to uDig developers.
  - **Plug-in Deployment Model,** with versioning and plug-in management to ease the cost of deployment, upgrading and installation.
  - **Integration/Extension,** maintain common appearance, workflow, framework and persistence mechanisms between built-in editing and third-party modules.
  - **Logs,** make use of logging standards and libraries.
- **Open Development Process,** capture developer interest and third party contributions.
- Marketing
  - **Release Management,** stable and development releases.
  - Product Development and Branding, continued use of JUMP branding.
- Licensing Model and Business Model
  - **Application License Model,** open-source license to allow distribution and extension without incurring multiple licensing fees, commercial support allows for a business model.
  - **Extension License Model**, open-source Framework API allows GPL or Commercial extension.
- **Usability**, use industry standard user-interface constructs and terminology to reduce training time.
  - **Configuration and Preferences,** make use of sensible defaults, use context where possible.
  - **Installation,** allow installation with sensible defaults and little user input.
  - **Professional Appearance,** integrate with existing installation base.
  - **Quick Response,** provide immediate feedback.
- Performance
  - **Data Access Performance**, ESRI Shapefile access is a significant measure of application performance and must be more than competitive.
  - **Operative Performance,** application must be sufficiently responsive so that an operator can maintain concentration.
- **Security,** considered where applicable: database passwords will not be stored with project file; the OWS infrastructure lacks a strong security model.

4 TERMINOLOGY AND DEFINITIONS

This document defines and refers to the following user-interface constructs:

Project	Consists of Maps and Pages
Map	Project entry defining content the user is interested in.
Viewport	Area of Interest (AOI) and projection (CRS) used to render a Map
Page	Project entry defining content and layout used to print a page
Catalog	List of Data Sources known to local user
Data Source	Catalog entry that defines data connection information for remote servers, local files or temporary results.

4.1.1 Eclipse Framework Terminology

The Eclipse Framework provides a series of high-level concepts that we will leverage to provide a consistent look and feel to the uDig application. We may choose to extend these assumptions as the needs of our project dictate.

The Eclipse framework makes use of several metaphors:

Workbench	A single application window is assumed
View	A panel in the application window. There are a number of predefined areas in the Application window where a view may be located.
Editor	A special view associated with a resource. A resource is anything that holds state
Perspective	A grouping of views assembled for a specific purpose
Main Menu	Menu associated with the main application window
Main Toolbar	Toolbar associated with the main application window
View Menu	Available as a black triangle in the title bar of the view
View Toolbar	Additional icons that are available in the view title bar

At a low level, the Eclipse Framework makes use of JFace and SWT to provide:

Viewer	A consistent form of View creation
Actions	Define shared behaviors between menus, toolbars and buttons
Registries	Which hold images and fonts
Dialogs	Which provide simple interactions with the user
Wizards	Which provide complex interactions with the user

4.2 Terminology

In specifying requirements, particularly non-functional requirements, we have used the following Terminology.

Throughput	Number of bits/features/transactions per unit of time
Performance	Time per bit/feature/transaction (inverse of throughput)
Latency	Wait time for a response
Capacity	Number of entities the system can support in a given configuration at a fixed level of performance
Scalability	Ability to increase capacity by adding hardware
Reliability	Length of time the system can run without failing
Response Time	Total perceived time it takes to process a request

Many modules will have several performance requirements that will need to be balanced.

4.3 Common Acronyms

The following acronyms are used in describing subsystem requirements.

API	Application Programmer Interface
GML	Geography Markup Language
MVC	Model-View-Controller
OGC	Open GIS Consortium
OWS	Open Web Service specification
SLD	Style Layer Descriptors
SMS	Style Management Service
WMS	Web Map Server
WMC	Web Map Client
WRS	Web Registry Service
WFS	Web Feature Server
WFC	Web Feature Client

4.4 Miscellaneous Terms

Area of Interest	The area of a feature set that is being focused on at a given time. For example, if the data set is the set of roads and rivers in BC but a viewport is viewing Vancouver Island then the *Area of Interest* would be a box around Vancouver Island.
Decorator	A symbol on a map for display, not data, purposes. A North arrow is an example of a decorator.
Layer Reference	Typically is used to mean a reference to a Catalog entry. However in the context where *Layer Reference* is used the Catalog entry reference represents a Layer in a map.
Plug-in	Used synonymously with extension. A plug-in is a portable collection of code that can be "plugged" into uDig to provide additional functionality.
RenderedImage	A Java interface. One of the main image interfaces.

5 uDig Model and Architectural Overview

This section is intended to provide a high-level overview of the uDig core architecture in order to allow readers who are less acquainted with uDig's design to be able to follow the module requirements sections. There are no justifications for the architecture; that is not the intent of this section.

The architecture of uDig is intended to consist of a number of well-defined modules that cooperate to satisfy the project requirements. The core and most important parts of the system are the model abstractions, see Section 5.1. It is important to note the difference between the "User" models and the "Architectural" models.

This section is divided into several sections; most sections simply introduce a module and the module's primary components.

5.1 System Data Models

An important aspect to be aware of during this section is the relationship between the data models as the user views them and how the models are represented programmatically. For example, at an operational level there is a *Context Model*, which is a *Data Model,* combined with some of the state information, such as layer visibility, from a *Map*.

5.1.1 User Models:

- **Project,** the highest level of abstraction. A *Project* contains a user's work for a particular job. A *Project* consists of maps and pages.
- **Map,** similar to an ArcView "view". A *Map* encapsulates a view of the *Project* data and how the data should be displayed on a screen for viewing and editing. A map usually has *Viewports* defined for visualizing the data set.
- **Viewport,** an area of the data that is displayed in a display window.
- **Page**, project entry defining content and layout used to print a page. A *Page* is closely related to a *Map* except a *Map* models how the data is displayed on the screen for editing and viewing and a *Page* is used to print maps.
- **Catalog,** list of Data Sources known to local user

5.1.2 Architectural Models:

- **Context Model,** the primary realization of a map. When a *Map* is opened a *Context Model* is instantiated. If many maps are open then there will be many *Context Models*. A *Context Model* contains the layer information of the map but not the information about the decorators and the viewport data.
- **Viewport Model,** encapsulates the Area of Interest and CRS of the data to be displayed as well as how and what decorators are displayed. The *Viewport Model* is derived from the *Map*.
- **SelectionModel,** encapsulates queries and references that permit identification of all the selected features.
- **PrintContext,** encapsulates all the data required to print a map. The PrintContext is instantiated from the data in *Page*
- **Catalog,** the same artifact as the *Catalog* discussed in Section 5.1.1, User Models.

5.2 System Module Overview

This section presents the different modules in the uDig system and discusses the role of each module. Many of the modules will leverage existing technology in order to satisfy their requirements.

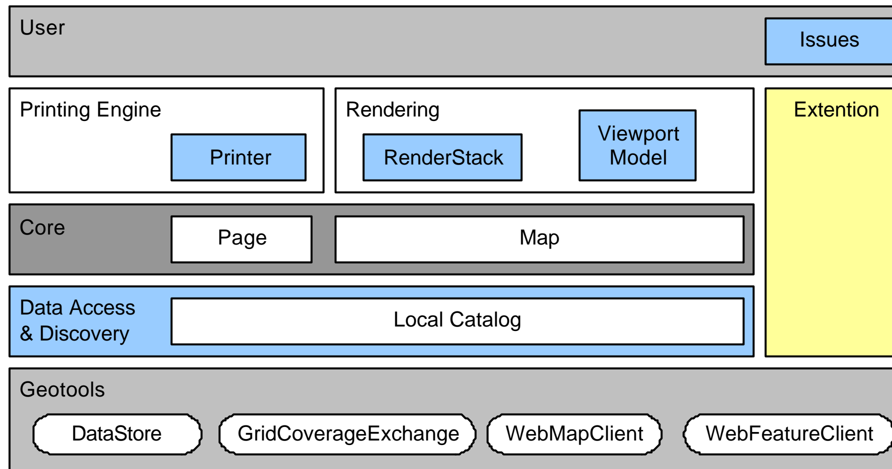


Figure 1: General Architectural Overview

The diagram above illustrates the basic modules of the uDig system. The User Interface, Core and Data Discovery and Access modules are the most critical modules in the system.

Data Discovery and Access is responsible for fetching and compiling service metadata and connection information into a catalog; this is performed by the Catalog. The Data Access creates objects for reading and writing to the services. Examples of a service may be a local file or a remote server. Section 5.5 discusses Data Access in more detail.

The User Interface module is a separate module because it provides a workflow for the user and is crucial to the use of the application.

At the User Interface level concepts like *Project*, *Map* and *Page* are of primary concern. The user interface is based on the existing Eclipse platform so the look of all the views (see Section 5.8, uDig Extension Framework) will be consistent even if a third party developer creates an interface in an extension (see Section 5.8, uDig Extension Framework). Section 5.3 discusses the User interface in more detail.

The Core module contains the classes that Extensions, Rendering and User interfaces operate through. The *Context Model*, the *Layer Manager* (which is the controller for *Context Model*) and the *Selection Model* and *Manager* all are parts of the Core module. See Section 5.4 for more on the Core module.

The *Printing* and *Rendering* modules are essentially plug-ins that are packaged with uDig and are required to satisfy the uDig project requirements.

The *Rendering* module's user interface component is the *Viewport* and its primary responsibility is rendering user defined maps. Its second crucial responsibility is to provide visual editing and selection interaction. See Section 5.6.

The *Printing* module has functionality for printing maps. Section 5.7 discusses the components of printing, such as the *Page*.

Section 5.8 discusses the goals and intent of the Extension Framework which is intended to allow third party developers to extend the functionality of uDig in a simple and effective manner.

5.3 User Interface

The uDig user interface uses the Eclipse framework to provide a consistent and pleasant user interface. Eclipse allows the user to move panels into different docking areas and allows panels to be resized to provide a flexible user interface that can be fully defined by the user; Figure 2 shows one likely configuration.

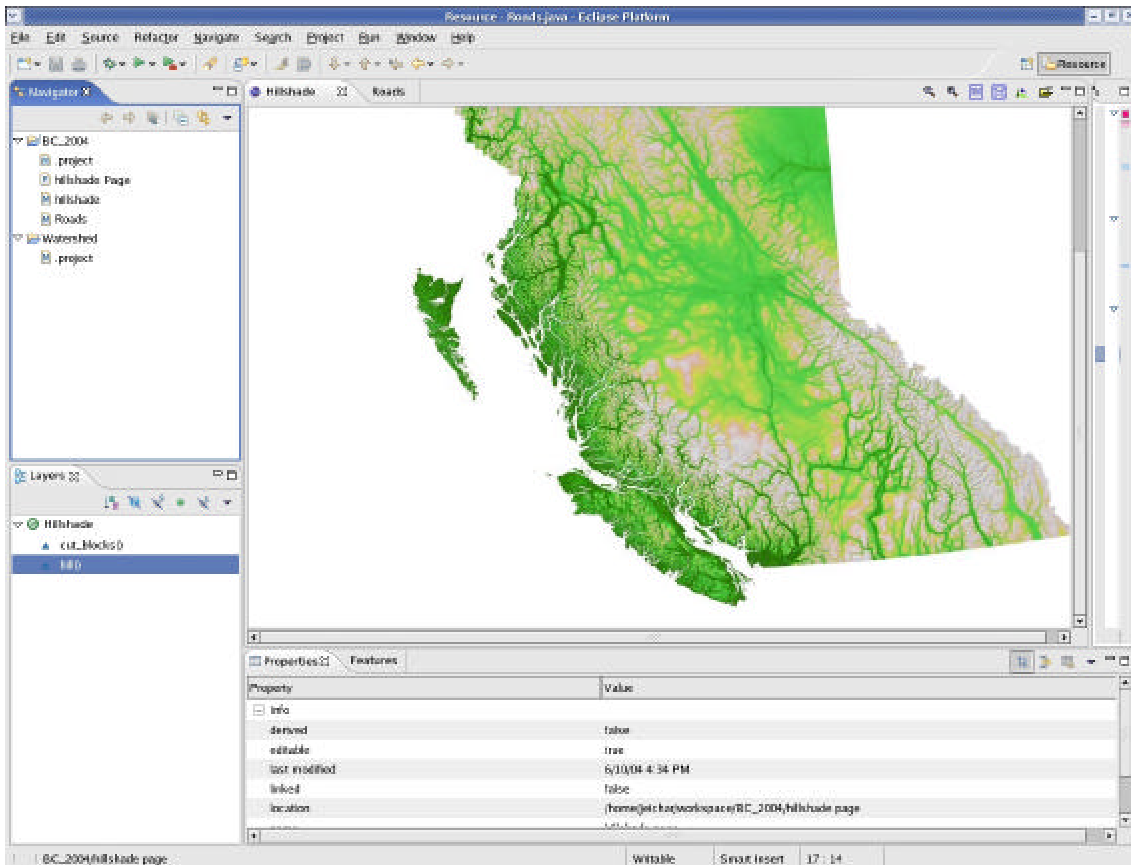


Figure 2: uDig UI Mockup

The image above is based on a "screenshot" of Eclipse, the framework used by uDig. The top-left panel is the project navigator. In this example there are two projects the user can operate on. Each project has entries that belong to them; the entries can be either *Maps* or *Pages*.

Below the Navigator is the Layers view. See Figure 7 for a close-up view of the Layers view. The Layers view lists the layers in a particular map. The "main" panel, the *Viewport*, is showing the Hillshade *Map*. The *Layers* view is linked to the *Viewport* so the *Layers* view panel shows the layers of the Hillshade *Map*. Each layer has symbols associated that indicate the state of the layer, ie visible or editable (the final application will have different and more logical symbolism). The "main" panel is the *Viewport*, which can be used to edit the GIS data. There can be multiple viewports open; notice the Roads tab above the *Viewport*. At the

bottom of the screenshot is a Properties panel. This is not a defined uDig panel but could be an example of a plug-in's user interface.

5.3.1 Components

This section lists the User Interface components required by the uDig project. See Section 7 for the User Interface Requirements:

- **Layer View**, The Layer View of the Map is a list of the layers displayed in rows similar to most GIS software.
- **Viewport,** The *Viewport* provides a visualization of the current *Map*.
- **Edit Toolbar,** Allows editing of a Feature selected by the *Viewport*.
- **Selection Toolbar,** A toolbar containing tools that can be used to interactively select features in the *Viewport*.
- **Selection View,** Permits viewing and editing feature selections.
- **Feature View,** Provides Display and Editing of the currently indicated Feature. Any actions requiring spatial input (like geometry editing) have been gathered into the Edit Toolbar.
- **Query Wizard,** Allows the definition of selection information as a series of constraints.
- **Printing Wizard,** The printing wizard component is a set of questions, like all wizards, that allows the user to create a Page (see Section 5.1.1).
- **Style Wizard/Editor,** Styles can be created and customized using the style wizard and the style editor.
- **Catalog View,** The Catalog View is a user interface that provides access to the Local Catalog. The Catalog View allows the user to define and modify the Data Sources known locally as well as provide functionality for adding the Data Sources to Projects.

5.4 Core

The Core module is central to the operation of uDig and contains the classes that Extensions, Rendering and User interfaces operate through. The *Context Model*, the *Layer Manager* (which is the controller for *Context Model*) and the *Selection Model* and *Manager* all are parts of the Core module.

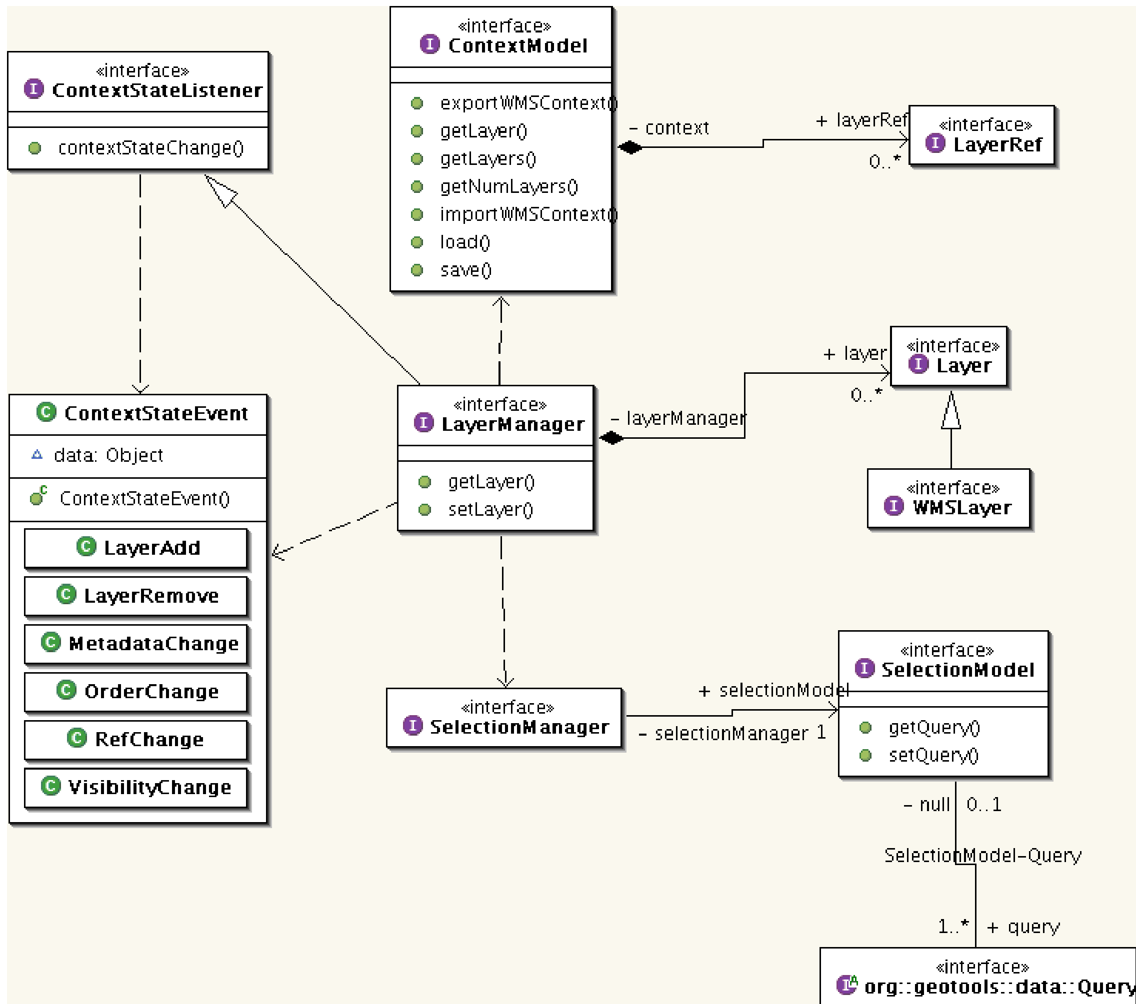


Figure 3: Core Module Interface API

This figure shows the components of the Core module as well as the relationships between the core components. The next subsection discusses the different components' purposes.

5.4.1 Components:

- **ContextModel,** the primary realization of a *Map* (see Section 5.1.1). *Context Model* sends events to its Listeners when its internal state changes.
- **LayerRef,** represents a layer in the Map. In essence a *LayerRef* is a reference to an entry in the *Local Catalog*.
- **LayerManager,** is perhaps the most central component of the entire system.
    - Controls the *ContextModel*
    - Creates *Layer* objects that realize the *LayerRefs* in the *ContextModel.*
    - By controlling access to the *Layer* objects, *LayerManager* also controls access to the actual data.
- **Layer,** has knowledge of what the layer data is and how the data is stored and can be accessed. *Layers* are created and managed by the *LayerManager* and used by other components to obtain data access objects such as *Data Stores* and *Data Sources*.
- **ContextStateEvent,** encapsulates context state changes. When the *Context Model* changes ContextStateEvents are sent to all of the model's Listeners.
- **SelectionModel,** encapsulates queries and references that permits identification of all the selected features.
- **Selection Manager,** the controller for the *SelectionModel*. Restricts write access to the *SelectionModel*.

5.5 Data Discovery and Access

Data Discovery and Access is responsible for fetching and compiling service metadata and connection information into a catalog; *LocalCatalog* performs this. The Data Access creates objects for reading and writing to the services. Examples of a service may be a local file or a remote server.

Data Discovery and Access is complicated in a distributed environment. The existing data discovery specifications are based on the OGC Catalog Specification. The Web Registry Service is an example of the OGC Catalog Service implemented as an Open Web Service specification (OWS).

In order to capitalize on these developments we will need a strong object model following the OGC conventions.

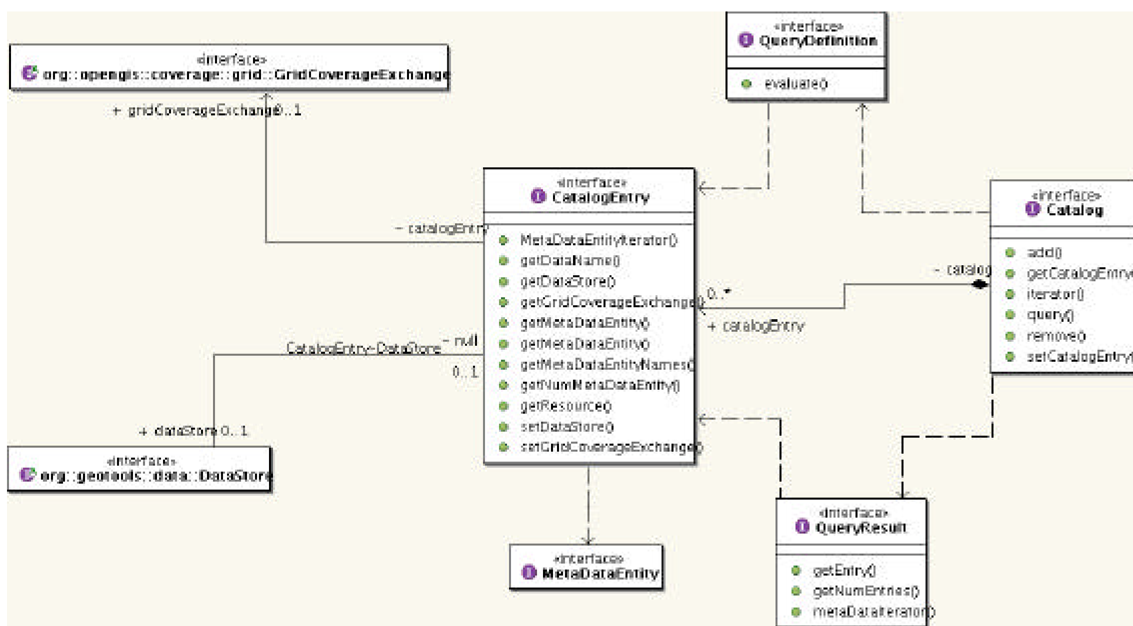


Figure 4: Data Discovery and Access Module Interfaces

Figure 4 shows the components and relationships of the Data Discovery and Access Module. Subsection 5.5.1 discusses the components of the Data Discovery and Access Module.

5.5.1 Components

- **Catalog,** consists of entries with service metadata and the connection information for the associated service.
- **CatalogEntry,** associates Metadata with service connection information. For example, if a WMS was the service referenced by an entry, the entry would also have the name of the WMS and, if available, a reference to a WFS where the features being rendered could be obtained. The entry may also list the styles and features rendered by the WMS.
- **MetaDataEntity,** A metadata of a service. There may be many pieces and types of metadata for each service.
- **DataStore,** one of the three major access components. Provides Read/Write access to Feature data. The *DataSource* has the same level of abstraction but only allows read access to Feature data.
- **DataSource,** provides read access to Feature data. *DataSource* is not shown in Figure 4 but it has the same relationships as *DataStore*.
- **GridCoverageExchange,** provides access to grid coverages. Additionally, *GridCoverageExchange* has functionality for exchanging between persistent grid coverage formats. GridCoverageExchanges are used to communicate with WMS services because WMS requests are satisfied with raster images, which are a form of grid coverage.

5.6 Rendering

The Rendering module uses the uDig Extension Framework. The *Rendering* module's user interface component is the *Viewport* and its primary responsibility is rendering user defined maps. The second crucial responsibility is to provide visual editing and selection interaction.

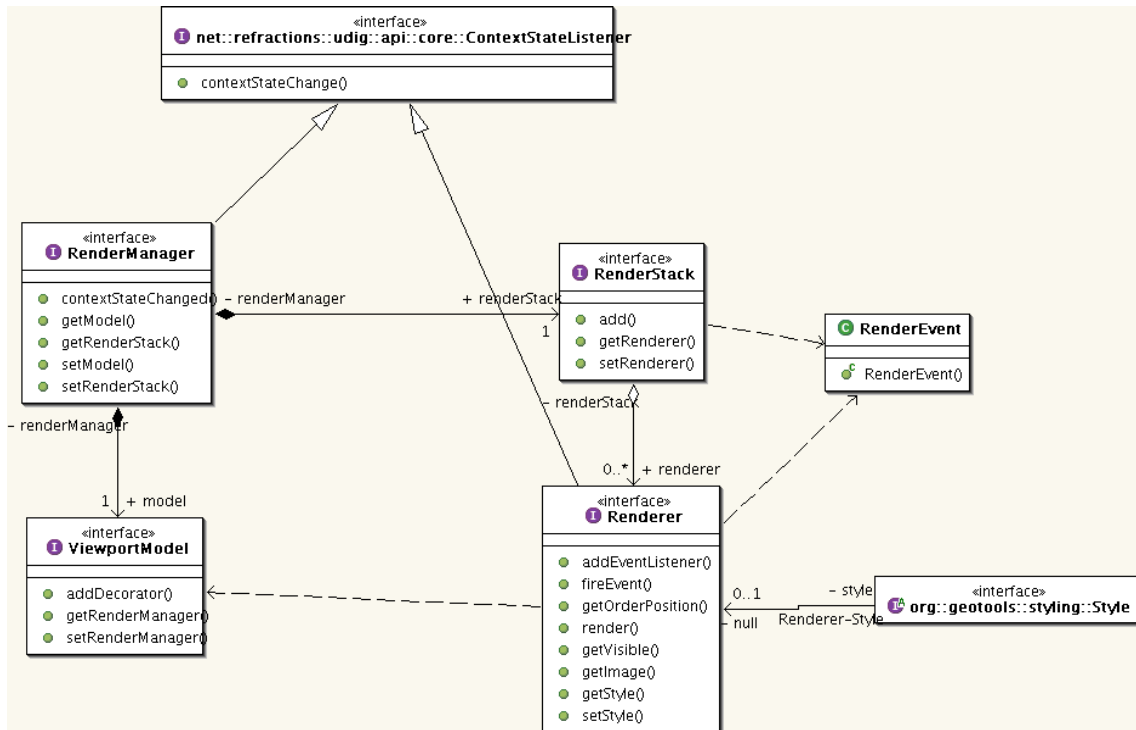


Figure 5: Rendering Module Interfaces

Figure 5 shows the major non-UI components of the Rendering module. Additionally, there are important subclasses to the Renderer class that are not shown in the diagram but are discussed in the following section.

The Rendering Technologies Report contains additional information about printing and uDig.

5.6.1 Components

- **RenderManager,** the "brains" of the rendering module. Responsibilities of the *RenderManager* include acting as the control for the *ViewportModel* and creator of Renderer instances for each map, decorator and interaction layers. The *RenderManager* also acts on ContextStateEvents when the event requires changes in the rendering module.
- **ViewportModel,** encapsulates the *viewport* state. This includes the map's area of interest, the local CRS and the decorators that displayed are in the *viewport*.
- **RenderStack,** acts as a container for the list of *Renderers*. In addition, the *RenderStack* builds Java Advanced Imaging (JAI) chains that combine the output from the *Renderers* for display on the screen.
- **Renderer,** creates visual output. Some *Renderers* render feature and raster GIS data but other types may render dynamic information based on computations. For example, a *Scalebar Renderer* would output visual data but the image would be based on dynamic computational input, not stored information. *Renderers* render data to a Graphics2D (a Java object), specified by one of its peers, or to an internal image that can be obtained upon request.
- **FeatureRenderer,** a subclass of *Renderer* that renders features from a FeatureSource.
- **WMCRenderer,** subclass of *Renderer* that renders images obtained from WMSs. A *WMCRenderer* has the knowledge to create and send WMS requests.
- **RasterRenderer,** subclass of *Renderer* that renders Raster images.
- **RenderEvent,** occurs when the state of a *Renderer* occurs. For example, a *RenderEvent* is raised when a *Renderer* has data ready for display.
- **Style,** a definition of how data should be rendered. A style is a set of filters and a set of associated symbolizers. Filters are used to identify a set of data and symbolizers define how to represent the set.

5.7 Printing

Similar to the Rendering module, the Printing module is an extension that is critical for satisfying the uDig project requirements. The Printing module has the functionality required for printing user-defined maps.

The crucial difference between Rendering and Printing is that we cannot use RenderImage and JAI to combine the output of Renderers. Instead, the Graphics2D provided by the Java2D Printing API must be used.

The size and resolution of the Java2D Printing Graphics2D provides an interesting challenge for some of the renderers. In particular, WMCRenderer needs to issue several GETMAP requests and tile together the result at the resolution of the Printer.

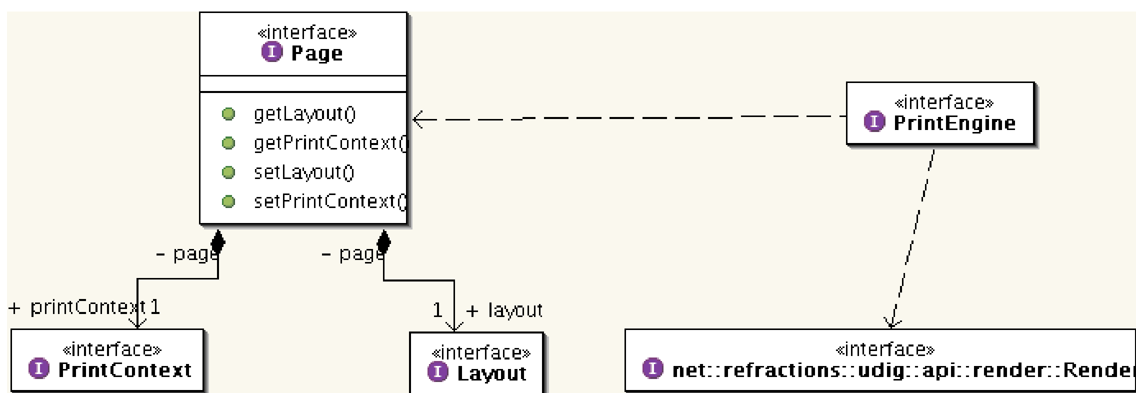


Figure 6: Printing Module Interfaces

The Print Module is a conceptually simple module. A Page represents a page to be printed. PrintContext is the model. A Layout dictates how the model is organized on a page and a PrintEngine prints using renderers from the Rendering module.

The Printing Technologies Report contains additional information about printing and uDig.

5.7.1 Components

- **Page,** represents a page to be printed. A *Page* consists of a *Layout* and a *PrintContext*.
- **PrintContext,** is the data model. The *PrintContext* combines the model information from *ContextModel*(Section 5.4), the *SelectionModel* (Section 5.4) and the *ViewportModel* (Section 5.6). The layer references, map decorators, area of interest, CRS and current selection are all included in the PrintContext.
- **Layout,** defines how a *PrintContext* will be displayed on a page.
- **PrintEngine,** contains the functionality for printing a page.

5.8 uDig Extension Framework

The ability to extend the uDig project is one of its central measures of success. A strong extension Framework is a key provision facilitating an open source community. By providing an extension framework, the initial outlay to join the community is reduced, and the administrative overhead associated with contribution is reduced.

A plug-in (extension) is developed by accessing and operating with the components exposed to the plug-in developers. The components are referred to sometimes as hooks and sometimes as extension points. The primary extension points available to developers are the following:

- LayerManager
- ContextModel
- SelectionManager
- SelectionModel
- RenderManager
- Renderer
- StyleManager
- Style

To aid the development of Third Party Plug-Ins, a uDig Extension Framework will be provided, allowing the development of Plug-Ins through sub classing. For example, a plug-in developer could extend a data operation class that would contain the functionality for obtaining data and the plug-in developer would only have to provide the operation functionality.

6 APPLICATION REQUIREMENTS

When considering the capabilities of the uDig Application we have drawn from a number of sources:

- The Project Requirements
- The operation of popular GIS desktop applications.
- A series of Use-Cases capturing intended workflow

Each of the following sections lists the requirements for the components in a particular module. Section 7 lists the requirements for the UI Module, for example.

The non-functional requirements, followed by the functional requirements are listed for each component. The non-functional requirements are listed first because often they are simpler and therefore more comprehensible to non-developers.

This section is intended as a guide for the uDig developers to ensure that all the uDig requirements are met. Non-developers may have a difficult time understanding this section.

7 UI Module Requirements

The UI module encompasses all of the application that the users sees and interacts with. The following sections list the requirements of the UI components and have a visual example each. Many of the descriptions mention relationships with ArcView because it is one of the best-known GIS clients.

Section 5.3 provided a high-level discussion of the UI Module that explains how it fits with the rest of the system.

7.1 Layers (Map View)

The *Layer View* of the *Map* is a list of the layers displayed in rows much like ArcView's Table of Contents, sometimes referred to as the Legend View. Each row in the *Layer View* contains icons and other decorators to provide visual clues about the state of the layer. Each row (layer) can be interacted with by modifying the layer's configuration. For example, a right click might open a menu for advanced options and dragging the layer might change the order of the layers.

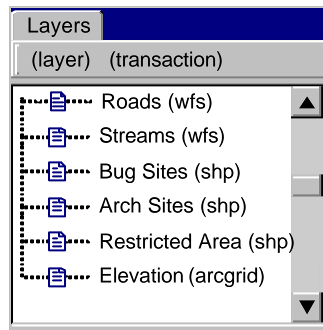


Figure 7: Layer View Component

Figure 7 provides an example of what a *Layer View* component might look like. It would be one of a number of UI components displayed. Figure 2 showed an example of a complete user interface.

Non-Functional Requirements:

- **Industry Standard**, appearance and behavior is similar to other popular GIS clients;
- **Legend Glyph,** accurately displays the glyphs associated with each layer. Based on the SLD where possible.

Functional Requirements:

- **Display Layers,** communicate layers status with respect to visibility, editability, and availability. Any errors, conditions or issues associated with the layer must also be indicated;
- **Toggle Visibility,** allows a user to set which layers are visible;
- **Toggle Editability,** allows a user to set which layers can be edited;
- **Reorder Layers,** permits the user to change the order in which the layers are displayed;
- **Add/Remove Layers,** provides access to a wizard for adding layers and permits the user to delete layers

7.2 Viewport (Map Editor)

The *Viewport* provides a visualization of the current *Map*. If more than one *Viewport* is used, each may have a particular Area of Interest (AOI) and projection (Coordinate Reference System).

The *Viewport* closely interacts with many parts of the user experience; this interaction is governed by a series of actions provided by other subsystems. This is the major focus of many third-party extensions and the plug-in framework provided must be well thought out.

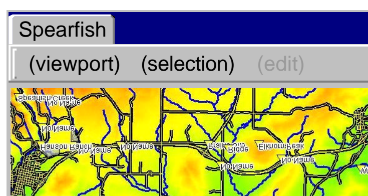


Figure 8: Viewport Component

Figure 8 is an example of a *Viewport*. A *viewport* being used during operation would likely be much larger. However, this example shows that a *viewport* displays GIS data. In addition, the *viewport* is an interface for feature and map editing. NOTE: In the context of the last statement, “map” is intended to mean the representation of the GIS data; not the *Map* data model.

Non-Functional Requirements:

- **Standard Workflow,** appearance and behavior is similar to other GIS clients.
- **Editing Performance,** immediate editing feedback for handle manipulation.
- **Selection Performance,** immediate feedback of selection changes and visualization.

Functional Requirements:

- **Visualization,** displays the map specified by the *Viewport*.
- **Zoom,** allows the user to choose the scale of the map.
- **Pan,** allows the user to interactively move the area displayed.
- **Selection,** integrates with *Selection Toolbar*, allowing the user to select feature information and display the current selection.
- **Transaction,** allows the user to control the current transaction.
- **Feature Edit,** integration with the *Edit Toolbar* and *Feature View* to allow modification of a *Feature* indicated by a series of handles for each vertex.
- **Tooltips,** shows a short summary when the mouse pointer pauses over a feature.
- **Context Menus,** context sensitive pop-up menus determined by *Viewport*, *Layers* and current *Feature*.

7.3 Edit Toolbar (Viewport Toolbar)

Allows editing of a *Feature* selected in the *Viewport*. The *Feature* may have more than one *Geometry* component.

Functional Requirements:

- **Vertex Selection,** selects a *feature's* vertex.
- **Move Tool,** moves the selected vertex.
- **Draw Rectangle,** draws a rectangle *feature.*
- **Draw Line,** draws a line *feature.*
- **Draw Point,** draws a point *feature.*
- **Draw Polygon,** draws a Polygon *feature.*
- **Insert Vertex,** inserts a vertex in a line string.
- **Join Features,** joins two *features* into one.
- **Delete Feature,** deletes a *feature.*
- **Snapping,** enable/disable inter-layer snapping for editing.

7.4 Selection Toolbar (Viewport Toolbar)

The *Selection Toolbar* contains tools that can be used to interactively select features in the *Viewport*.

Functional Requirements:

- **Rubber Band,** the rubber band tool selects *features* within the rectangular area indicated by a click and drag “stretching box”.
- **Select Feature,** a single *feature* can be selected by clicking on a *feature*.
- **Multiple Selections,** *features* can be added to or removed from a *Selection* by holding shift or control and clicking on *features*.
- **Magic Polygon,** the “magic polygon” is a polygon created by clicking in the *viewport*. All *features* that are fully or partly within the polygon are selected.

7.5 Selection View (Depends on Viewport View)

Permits viewing and editing of feature selections. Any actions requiring spatial input have been gathered into the *Selection Toolbar* of the current *Viewport*.

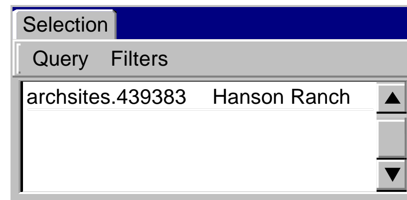


Figure 9: Selection View Component

In the example above the selected features list is displayed, there is only one selected feature. The Query menu (not shown) allows the user to view the selection queries and filters.

Non-Functional Requirements:

- **Extension Point,** common extension point for plug-ins operating against the current selection.

Functional Requirements:

- **Tabular Summary,** provides a tabular summary of the current selection.
- **Query,** access to the *Query Wiz*ard to define selection via constraints.

7.6 Feature View (Depends on Viewport View)

Provides display and editing of the currently indicated *Feature*. Any actions requiring spatial input (like geometry editing) have been gathered into the *Edit Toolbar*.

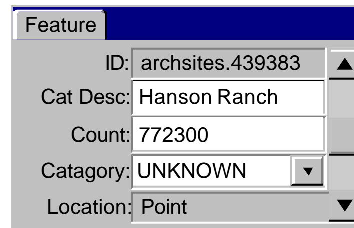


Figure 10: Feature View Component

As Figure 10 shows, a *Feature View Editor* has editable fields (white) and non-editable fields. The editor can be used to directly edit or examine the feature attributes.

Functional Requirements:

- **Attribute Editing,** displays and allows modification of attribute information
- **Geometry Indication**, indicate the current geometry attribute to be edited by the *Edit Toolbar*

7.7 Query Wizard

Allows the definition of selection information as a series of constraints.

Non-Functional Requirements:

- **Prototype Queries,** made by selecting a feature and selecting attributes to match or not match. (See mail filter creation on mail clients such as Thunderbird.)
- **SQL-style Queries,** defined in a language similar to SQL.

Functional Requirements:

- **Add/Remove Selections,** selection queries can add or remove a group of features to/from the currently selected feature set.
- **History,** a history of past selection queries is maintained for future use.

7.8 Printing Wizard

The printing wizard component is a set of questions, like all wizards, that allows the user to create a Page (see Section 5.1.1). Normally the user will be able to simply accept the defaults and print or save the Page created.

Non-Functional Requirements:

- **Template,** provides a set of *Layout Templates*.
- **Sensible Defaults**, sensible defaults should be determined based on current settings and previous runs of the printing wizard.

Functional Requirements:

- **Printer,** printer selection.
- **Layout,** *Layout* selection.
- **Layout Editing,** *Layout* editing support.
- **Print Preview,** user may view the preview.
- **Save,** allow user to save the print setup.
- **Print,** starts the print process.

7.9 Style Wizard/Editor

Styles can be created and customized using the Style Wizard and the *Style Editor*.

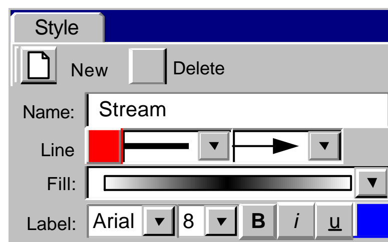


Figure 11: Style Editor Component

The *Style Editor* is one of the two ways styles can be modified in uDig. Figure 11 shows an example of the style editor being used to edit a line style.

Functional Requirements:

- **Wizard Style Creation,** create a new style using a style wizard. Wizard allows user to choose a base style from a pool of "basic" styles and then modify the style.
- **Style Editing,** a style's attributes can be edited after creation.
- **SLD Styling supported,** SLD styles can be created and edited.

7.10 Catalog View

The *Catalog View* is a user interface that provides access to the *Local Catalog*. The *Catalog View* allows the user to define and modify the *Data Sources* known locally, as well as providing functionality for adding the *Data Sources* to *Projects*.

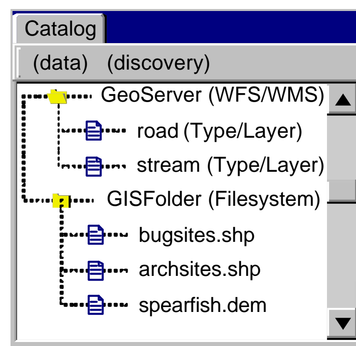


Figure 12: Catalog View Component

The example shown in Figure 12 lists a server with defined (layers, road and stream) and a filesystem (local or remote) is also listed as part of the *Local Catalog*. The *Layers* can be added to a *Project*. New servers and folders may be added to the *Catalog*. The *Catalog* (not the user interface component) is persistent and can be exported and transferred to other systems.

Non-Functional Requirements:

- **Tree View,** displays repositories as a list, and services as branches of a *repository*.
- **Drag and Drop**, integrate with *Viewport* to allow quick visualization of data.

Functional Requirements:

- **Add Data Wizard,** a wizard for adding a new *Data Repository*.
- **Remove Data Repository,** remove a *Repository* from the *Data Man*ager.
- **Refresh Data Repository,** refresh the service entries of a *Repository*.
- **List Repositories,** lists the *repositories* and ping rate of the *Repositories*.
- **List Services,** provide a list of services for each *Repository*.

8 Core Module Requirements

The Core module is central to the operation of uDig and contains the classes that Extensions, Rendering and User interfaces operate through. The *Context Model*, the *Layer Manager* (which is the controller for *Context Model*) and the *Selection Model* and *Manager* all are parts of the Core module.

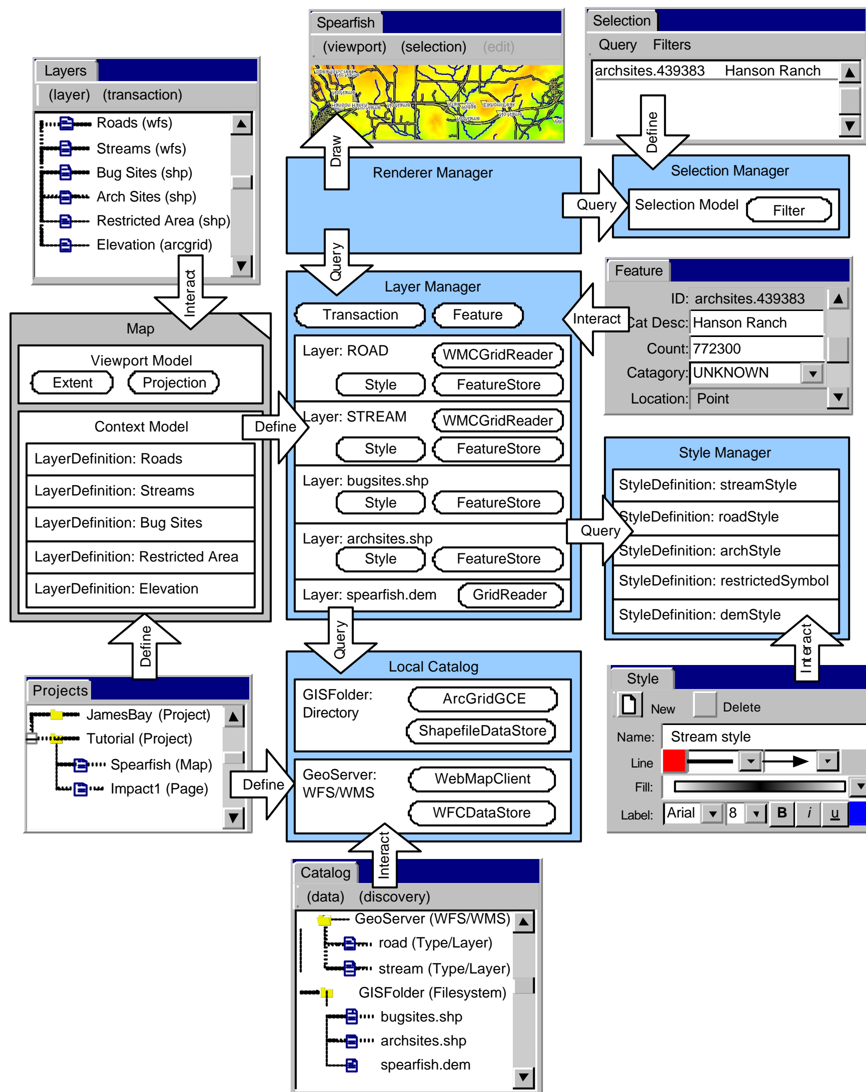


Figure 13: Model Interaction

Section 5.4 provides a high-level discussion of the Core Module that explains how it fits with the rest of the system.

8.1 Layer Manager

Layer Manager is responsible for governing all interactions with the *ContextModel* of a *Map*. The *Layer Manager* controls transaction and editing services. As a result most plug-in interaction with GIS information will occur via the *Layer Manager*.

Non-Functional Requirements:

- **Opaque Locking,** hide Locking from the end user.

Functional Requirements:

- **Controller,** acts as Controller for *ContextModel* in the MVC sense.
- **Layer Access**, provides *FeatureStore* and *GridCoverage* access. Each *Layer* represents the results of a *LayerRef*. A *Layer* may provide several avenues to resolve the same GIS information.
- **Locking,** manages *Feature* locking.
- **Transactions**, manages *Transactions* for editing.
- **Editing**, maintains the *Features* currently being edited.
- **Selection**, maintains a *Selection Manager* to manage the current Selection.

8.2 Context Model

The *ContextModel* is used to manage the layer data inside the *LayerManager*. It retains information about each layer to be displayed and other information pertaining to all of the layers, such as SRS and Extent.

Non-Functional Requirements:

- **Model,** *Layer Manager* "controls" the *ContextModel* in a Model-View-Controller sense.

Context will keep track of decorators, such as a legend, scale bar or compass. This would allow it to be passed down to either the printer or the screen renderer.

Functional Requirements:

- **LayerDefinition**, including both user GIS information and Decorators (like a Legend, or Scalebar) arranged into a common Z order.
- **Metadata,** encapsulates Metadata such as abstract, title and author.
- **Events,** provide notification when modified.
- **Persistence,** can be saved to disk and transported to other workspaces.

8.3 Selection Manager

Maintains a *SelectionModel* for a *Map*, used in conjunction with the *LayerManager* by the *RendererManager* to determine screen appearance (Selection Rendering). Third Party Plug-Ins may also use Selection information as a means of limiting operation scope.

Non-Functional Requirements

- **Scalable,** scalable selection solution for large datasets.
- **Instant Feedback,** sub-second notification of selection activity.
- **Latency**, selection should take less then 5 to 10 seconds for alpha.

Functional Requirements

- **Map Selection,** selection by map interaction.
- **Table Selection,** selection by table interaction.
- **Query Selection,** selection by "Query" user interface.
- **Plug-in Access,** programmatic access for operative plug-ins.

8.4 Selection Model

This is the model controlled by the *Selection Manager*. It captures selection information as a series of *Filters* associated with each *Layer*.

Functional Requirements:

- **Filters**, associated with each *Layer.*
    - **Feature ID,** used to capture individual selections.
    - **Extent,** used to capture spatial selection on *Map.*
- **Invert**, inverts can be formed quickly with the use of "not (filter)".
- **Integration**, strong interactions are expected with the *RenderStack* both in updating the appearance of the *RenderStack* via events, and in quickly determining selection updates via Bounding Box queries against the Raster images.

8.5 Style Manager

The *Style Manager* is used to manage all of the *Styles* within uDig. The *Style Editor* (see Section 7.9) can edit many of the *Styles*. In order to keep track of available *Styles*, the *Style Manager* needs to monitor the getCapabilities responses from WMSs.

Non-Functional Requirements:

- **WMS Style Editing,** styles residing on WMSs are read only, but can be copied to another location (to disk, then edited.)

Functional Requirements

- **Styles Availability,** aware of all styles currently available.
- **Style Sources,** aware of the origin of each style (on disk, WMS, etc.)
- **Request,** will return a style if it is requested.
- **Style Updates,** can receive new or changed styles.
- **Style Editing,** can create new styles and modify existing styles.
- **Buffering,** will retrieve new styles when it is informed of their location.

9 DATA DISCOVERY AND ACCESS MODULE REQUIREMENTS

Data Discovery and Access is responsible for fetching and compiling service metadata and connection information into a catalog; *LocalCatalog* performs this. The Data Access creates objects for reading and writing to the services. Examples of a service may be a local file or a remote server.

Data Discovery and Access is complicated in a distributed environment. The existing data discovery specifications are based on the OGC Catalog Specification. The Web Registry Service is an example of the OGC Catalog Service implemented as an Open Web Service specification (OWS).

In order to capitalize on these developments we will need a strong object model following the OGC conventions.

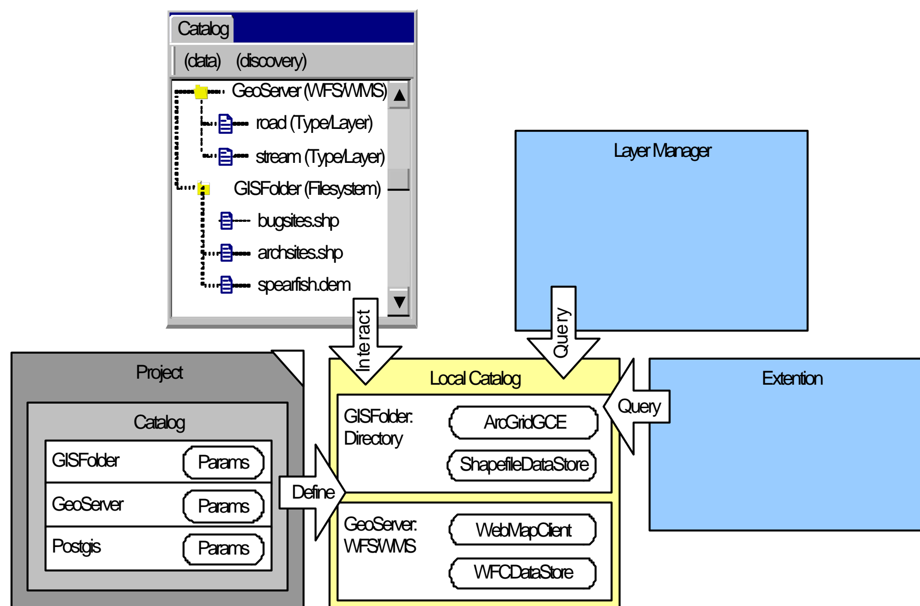


Figure 14: Catalog Interaction Diagram

The following requirements are for the Data Discovery and Access module in general.

Non-Functional Requirements:

- **Standard Naming**, from OGC/ISO Specifications.

Functional Requirements:

- **Metadata,** catalog Metadata lookup functionality.
- **Filter,** consistent query construct.

The existing GeoAPI and Geotools development communities both have partial implementations of our requirements.

9.1 Local Catalog

The *Local Catalog* serves as a central repository of data and server information.

Non-Functional Requirements:

- **Ease of Data Location,** intent is for the user to be separated from the data source; so they need as little technical knowledge as possible.
- **Security,** name/password should be left out of the export/share.

Functional Requirements:

- **Servers,** store server connection information for sharing between projects.
- **Data Directories,** store data directories for sharing between projects.
- **Metadata,** provide access to metadata on Servers/Data Directories.
- **Data Discovery,** provide enough information for a user to define a new layer in their context.
- **Persist Settings,** permit exporting and sharing *DataStore* connection information.
- **DataStores Management,** lookup actualized *DataStores* that are in use.
- **Missing Data,** entries referred to by imported projects should be maintained, allowing the user one location to correct data connection information.

Local Catalog has a strong interaction with the preferences maintained by the local installation of uDig for the current user.

9.2 Web Feature Client

Web Feature Client is a cornerstone of the uDig application, allowing interaction with Transactional Web Feature Serve rs as part of the OWS workflow.

Non-Functional Requirements:

- **Latency**, needs to be minimized, forcing us towards a streaming GML.
- **Throughput,** expected to be network bound.
- **Scalability,** address through bigger pipe, or by multithreading several requests.
- **Additional Formats,** GeoServer currently supports "text/gml" and "gzip/gml"; additional formats such as binXML should be discoverable at runtime.

Functional Requirements:

- **DataStore,** use *DataStore* for WFS communication. The Geotools DataStore API is a sufficient abstraction to capture operational use of a WFS client.
- **Equivalency Metadata,** allow for metadata describing alternate OWS processing streams
- **Catalog Services,** catalog information needs to be accessible allowing for integration with the *Local Catalog*.

Over the course of application development we may receive a new WFS specification incorporating the additions for a subset of GML3.

9.3 Web Map Client

The uDig application requires access to WMS content. WMS is a mature concept with four deployed standards and a common extension.

Non-Functional Requirements:

- **Latency,** cannot be safely constrained as part of a WMS; we need to make use of continuous feedback.
  (JUMP uses a rotating timer to indicate ongoing WMC activity)
- **Throughput**, important when making many small requests
- **Scalability,** the printing process must scale up to practical GIS printer resolutions.
- **Grid Coverage Exchange,** WMS Client Services can be presented at the low-level GCE API provided by GeoAPI according to the GO-1 draft specification. Note this API is slaved to Catalog Services and supports limited data discovery, and lacks an event model.
- **Additional Formats,** GeoServer currently supports "image/png", "image/jpeg" and "xml/svg"; additional formats should be discoverable at runtime.

Functional Requirements:

- **WMS1.0 – 1.1.1 Support,** uses getCapabilities, getMap, FeatureInfo, DescribeLayer methods to access WMS servers.
- **WMS Post,** allows for WMS requests to be sent over POST rather than GET, providing more functionality.
- **SLD Support,** is used to define how WMS renders Layers. Should be capable of offering a WMS as an alternative SLD, and browsing the SLDs available on the WMS.
- **SMS Browsing and Referencing,** can refer Web Map Server to SLDs on SMS Servers.
- **Equivalency Metadata,** allow for metadata describing alternate OWS processing streams.
- **Catalog Services,** catalog information needs to be accessible, allowing for integration with the *Local Catalog*.
- **Rendering,** WMS Client services should be presented behind the Rendering API for ease of integration with the *Rendering Stack*. Such an interface can provide a cache and asynchronous notification

Implementing GCE Exchange as a Geotools technological initiative provides WMS Client code with extensive testing offline of the main uDig project.

9.4 Database Access

Database Access must be supported by uDig. Both raster images and *Features* must be accessible from databases. These requirements primarily focus on features because the requirements in the Raster File Format section (next section) cover most of the requirements for Database image retrieval. All Rasters, including images on Databases, are obtained using the GridCoverageExchange.

Non-Functional Requirements:

- **Latency**, minimal due to *DataStore* streaming Feature Reader model.
- **Throughput,** expected to be network bound.

Functional Requirements:

- **DataStore,** use *DataStore* for Database access. The Geotools DataStore API is a sufficient abstraction to capture operational use of a GIS Database.
- **Catalog Services,** catalog information needs to be accessible, allowing for integration with the *Local Cata*log.
- **Equivalency Metadata,** allow for metadata describing alternate OWS processing streams.
- **Reprojection,** allow for metadata describing alternate OWS processing streams.

9.5 Raster File Formats

Raster File Format support is provided by the *GridCoverageExchange* API. The GridCoverageExchange API is a joint undertaking by Geotools and GeoAPI and is included in the GO-1 initiative.

Non-Functional Requirements:

- **Extensible,** it should be simple to add new File format support.
- **Latency,** minimal due to the fact that files will be on disk or cached to disk. Worst case will be in the situation that the file must be streamed from an HTTP server for the initial load.

Functional Requirements:

- **GeoTIFF,** support for GeoTIFF 1.0 file format
- **ArcGrid**, support for ESRI's ArcGrid file format
- **ASCII Grid,** support for the ASCII Grid file format
- **Data Discovery,** should be possible to look up *GridCoverageExchange* implementations in a registry or catalog.
- **Metadata,** *GridCoverageExchanges* must have metadata associated with them explaining which file formats they support. This metadata must be sufficient to build a user interface allowing users to specify raster data.

9.6 Shapefile Format

Shapefile stores non-topological geometry and attribute information for spatial features in a data set.

Non-Functional Requirements:

- **Latency**, minimal due to *DataStore* streaming Feature Reader model.
- **Throughput,** local shapefile access must provide immediate response.

Functional Requirements:

- **DataStore,** use *DataStore* for Shapefile access. The Geotools DataStore API is a sufficient abstraction to capture operational use of a GIS Database.
- **Data Discovery,** catalog information needs to be accessible allowing for integration with the *Local Catalog*.
- **Equivalency Metadata,** allow for metadata describing alternate OWS processing streams.

10 Rendering Module Requirements

The Rendering module uses the uDig Extension Framework. The *Rendering* module's user interface component is the *Viewport* and its primary responsibility is rendering user-defined maps. The second crucial responsibility is to provide visual editing and selection interaction.

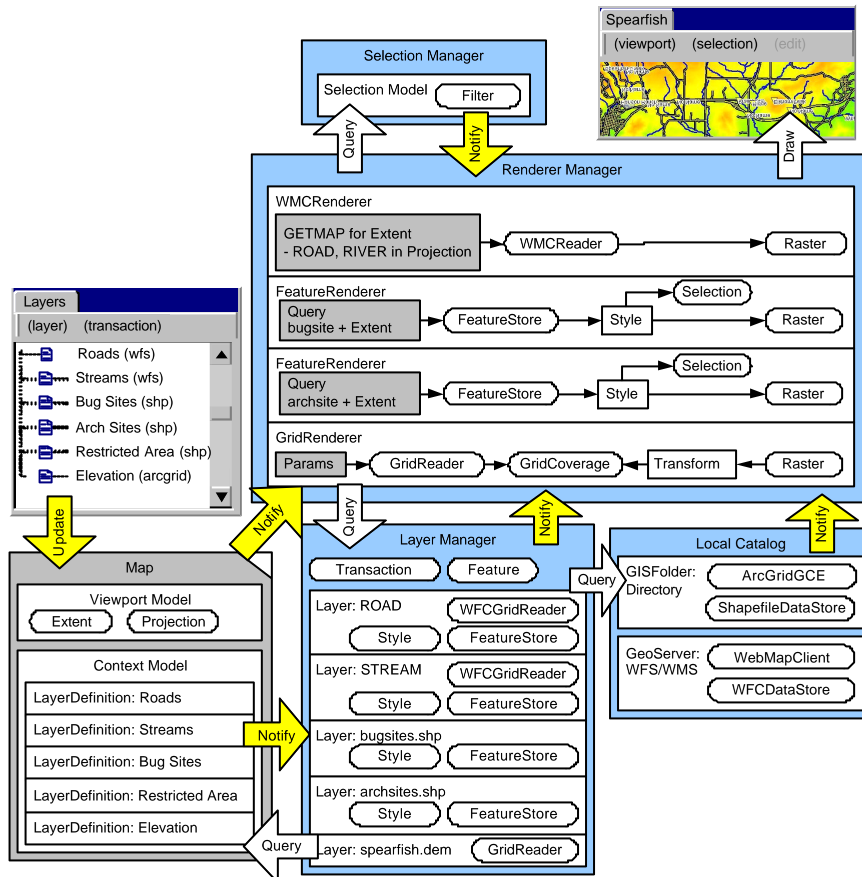


Figure 15: Rendering Interaction Diagram

This section lists the requirements for each of the critical components in the Rendering module.

10.1 Rendering Manager

The *Rendering Manager* is responsible for determining which renderer types should be instantiated to satisfy a request, and determines what changes need to be made during the system's lifetime.

For example, a different renderer may be required for a layer that is editable than for a layer that is not editable.

Non-Functional Requirements:

- **Extensible,** must have a simple process for adding new renderer implementations.
- **Multiple Layer Scalability,** as the number of layers increases the Rendering module continues to respond to the user's actions in a timely manner. When the number of layers gets very large a linear decrease in performance is acceptable.
- **Viewport Size Scalability,** as the size of the *Viewport* increases, the Rendering module can continue to respond to user's actions in a timely manner. As the resolution of *Viewports* nears printer resolutions, the *RenderingManager's* decision-making will become similar to the *PrintEngine's*.

Functional Requirements:

- **Viewport Controller,** acts as the Controller for the *Viewport Model* as visualized by the *Viewport Editor*
- **Layer Access,** acts as an intermediary between the *Viewport Editor* and *LayerManager*.
- **Event Handling,** receives events from *ContextModel*.
- **RenderStack Management,** decides which *Renderers* to use when defining the *RenderStack* for the current *ContextModel*.
  - **Renderer Selection,** chooses a layer renderer based on aspects such as visibility, selectability, and editability.

    ```
    For the same Layer a WMCRenderer may be used for rendering speed, while a
    WFCRenderer would be required for editing.
    ```
  - When choosing a layer renderer the tradeoffs between server- and client-side reprojection should be considered.
  - The choice of layer renderer is also dependent on the size of the viewport and the number of layers that need to be rendered.
  - **WMCRenderer Aggregation**, several layers may be combined into a single *WMCRenderer* request.
  - **Decorator Renderers,** support layers such as Legend and Scalebar
- **Viewport Selection,** provides hooks to query the *RenderStack* for selection information.

10.2 Viewport Model

The *Viewport Model* is the data model for the UI *Viewport* component (see Section 7.2). The Area of Interest, CRS, and decorators are all maintained in the *Viewport Model*. The *Viewport Model*, like most models, raises events when it is modified.

Functional Requirements:

- **Extent,** provides the Area of Interest (AOI)
- **Coordinate Reference System,** describes the projection associated with the *Viewport*
- **Events,** raises events when model data is modified
- **Decorators,** Maintains the list of decorators used in the *Viewport* and their state. For example, a grid for editing may be turned on and have a spacing of 5 kilometers between grid lines.

There may be multiple *Viewports*; each displaying different areas of the data set.

10.3 Renderer

A *Renderer* is responsible for rendering a layer to a Graphics2D for printing, or producing a Render Image for the screen.

Non-Functional Requirements:

- **Viewport Size Scalability,** as the size of the *Viewport* increases, the Rendering module continues to respond to the user's actions in a timely manner.

Functional Requirements:

- **RenderImage,** suitable for rendering on screen. This is often a *RenderImage* although decorator or *GridCoverage* layers may choose to provide a derived RenderImage directly.
- **Graphics2D draw,** suitable for use with printing.
- **Layer Visibility,** responds to layer visibility status changes.
- **Progress Notification,** is required for drawing offline, and timing screen refreshes.
- **Completion Notification,** is required for use with printing.
- **Selection Renderer,** for layers that allow selection, an additional (optimized) *Renderer* for selection may be used.
- **Selection Query,** provides a Filter for a given extent capturing any selected content.

10.4 RenderStack

RenderStack contains the *Renderers* and is responsible for merging all the Render Images created by the *Renderers* with JAI, and updating the viewing area.

Non-functional Requirements:

- **Immediate Selection Feedback,** allows *Selection Manager* to provide immediate feedback based on the RenderImages.

Functional Requirements:

- **RenderImage Merging,** merges the results of the *Renderers* together in a specified z-order.
- **Update Display Widget,** notifies display widget to paint the final raster.
- **Extent Query,** provides layer and filter information for the generation of tooltips, or additions to the *Selection Model*.
- **Renderer Ownership,** contains all *Renderers*.
- **Notification,** handles notification from *Renderers* to interested parties.

10.5 FeatureRenderer

The *FeatureRenderer* accepts a *FeatureSource* from a *Layer* and generates a *Query* based on the *Viewport Model*. The results of the *Query* are styled and rendered to *RenderedImages* as Java2D shapes.

The *FeatureRenderer* is used to render Database, Shapefile and WFS *DataStores*.

Non-functional Requirements:

- **Performance,** render a feature set in 3-5 seconds
- **Progressive,** displays the rendered data progressively during rendering rather than waiting for the rendering to complete before displaying

Functional Requirements:

- **Feature Rendering,** renders features from a feature store using a single *Query* based on the current *Viewport Model*
- **Reprojection,** between the following coordinate systems (CS) as required:
  - **Server CS**, original CS used to store features natively by the *DataStore*.
  - **Data CS**, the CS Feature data arrives from the *DataStore*. This will be either *Viewport* CS for *DataStores* that support reprojection, or Server CS for *DataStores* that do not support reprojection.
  - **Viewport CS**, the CRS used by the *Viewport*; units and orientation are expressed in Geographical terms.
  - **Java2D CS,** the coordinate system used by Java2D for *Rendered Image*; units measured in pixels.
  - **Device CS**, the final coordinate system used by the screen or printer. Each "unit" is a pixel of device-dependent size.
- **Styling,** styles features with a Third-Party Plug-In or Default SLD styling
- **Selection Renderer,** provides a slaved renderer that captures selected features onto a separate RenderImage.
  - The slaved renderer will also be capable of making a modified *Query* (*Viewport Model* + Selection Filter) when used for printing.
  - **Selection Styling,** render selected *Features* based on a selection SLD.
- **Event Handling,** handles the following events:
  - **Context Notification**, changes to styling or visibility.
  - **Viewport Notification,** changes to Extent or SRS.
  - **FeatureStore Notification,** changes to Features due to editing.
  - **Selection Model Notification,** changes to Layer Selection.

10.6 WMCRenderer

The *WMCRenderer* accepts a *WMCRequest* from one or more consecutive *Layers* and generates WMS GETMAP request(s) based on the *Viewport Model*. The results of the GETMAP request are rendered to RenderImage.

Non-functional Requirements:

- **Immediate Response,** immediate response is required for zooming and panning operations (even if the response is from a cache or transparent).
- **Latency,** buffer previous requests allowing for immediate feedback; continuous feedback is required during WFC requests.
- **Performance,** non-blocking generation of Buffered Image; notification will be used to report on progress and produce redraws.

Functional Requirements:

- **WMC Rendering,** renders a RenderImage from a WMS using one or more GETMAP requests based on the current *Viewport Model*
- **GETMAP Request,** produced from one or more *Layers*. Several requests may be issued and tiled together based on the limitations of the backing WFS.
- **Multiple Layers,** the *WMCRenderer* can handle several *Layers*
- **SLD,** used to specify the styling for the GETMAP request
- **Layer Visibility,** a composed WMS request only requests *Layers* that are set as visible
- **RenderedImage,** the results of a GETMAP request are rendered onto a RenderImage for display. If the WMC is able to provide a rough renderer based on cached data while waiting for the GETMAP request, offline notification can be used.
- **Notification,** provided as content arrives from one or more GETMAP requests.
- **Graphics2D draw**, to support printing, several GETMAP requests will need to be requested at printer resolution, and tiled together as a series of Graphics2D draw image calls.

10.7 GridCoverageRenderer

The *GridCoverageRenderer* takes a GridCoverage and derives a RenderImage based on the required reprojection.

Non-functional Requirements:

- **Immediate Feedback**, during a zoom or pan operation response should be immediate
- **Latency**, latency on initial GridCoverage creation should be immediate for local files; continuous feedback is required for remote files.
- **Instant Feedback,** when zooming in, the raster pipeline will attempt to give immediate feedback by performing a manual zoom on the selected area (within one second). It will then go and get the actual requested object (within three to five seconds).

Functional Requirements:

- **GridCoverage Rendering,** renders a GridCoverage. The *Viewport Model* is used to project the GridCoverage to the Screen Coordinate System (the coordinate system used by the screen.)
- **SLD,** style the input as dictated by an SLD
- **Graphics2D draw,** the RenderImage is drawn directly to a Graphics 2D. The projection to Screen/Device Coordinates will provide a re-sampling of the original GridCoverage.

11 Printing Module Requirements

Similar to the Rendering module, the Printing module is an extension that is critical for satisfying the uDig project requirements. The Printing module has the functionality required for printing user-defined maps. The Printing module has a strong dependency on the Rendering module because it uses *Renderers* for rendering.

The crucial difference between Rendering and Printing is that we cannot use RenderImage and JAI to combine the output of *Renderers*. Instead, the Graphics2D provided by the Java2D Printing API must be used.

The size and resolution of the Java2D Printing Graphics2D provides an interesting challenge for some of the renderers. In particular, WMCRenderer needs to issue several GETMAP requests and tile together the result at the resolution of the Printer.

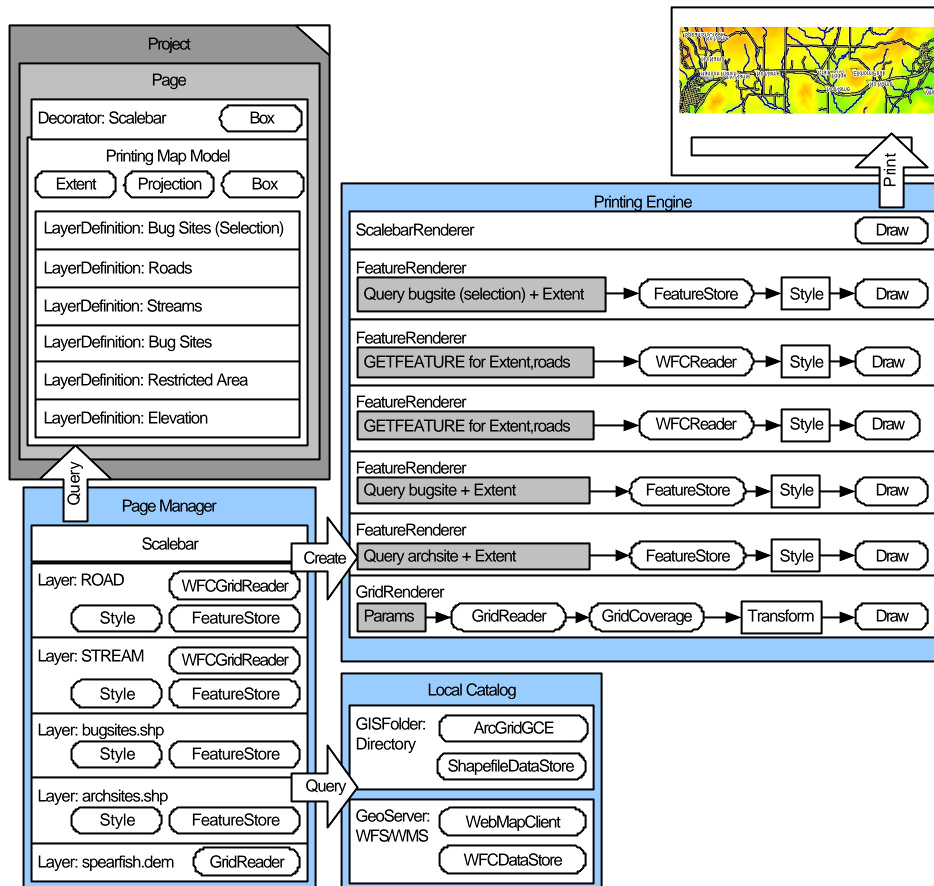


Figure 16: Printing Interaction Diagram

11.1 Layout

A layout represents the way graphics are arranged on a *Page*. *Layouts* usually feature a *Viewport* and several additional decorators.

Functional Requirements:

- **Template,** a specification of how a *Page* can be organized
- **Provided Templates,** a set of templates are provided so the user can easily create a *Page*.
- **Defaults,** provide a default layout (map, legend, scalebar, compass)
- **Persistence,** user must be able to edit/save (may be supplied by allowing export as *Layout* option for *Page* editing)

11.2 Page

Page is intended to allow a user to save a page that they have printed so that they can print it again, or perhaps even send it to someone else who can then print it.

Functional Requirements:

- **Instance,** layout acts as a Template/Prototype for making one of these
- **Printing Context Model,** manages *Printing Context*, similar to *LayerManager's* use of *Context Model* and *Viewport Model*
- **Persistence,** saves knowledge of the printer and the layout (this may not be portable)

11.3 Printing Context Model

The *PrintingContextModel* is derived from a *ContextModel* used for display and is joined with data from the *ViewportModel* and the *SelectionModel*.

Functional Requirements:

- **Layer Data,** encapsulates the layer data as *LayerRefs*. *LayerRefs* have style information.
- **Decorator Data,** encapsulates decorators that are integral to the map (the equivalent decorator data contained by the *Viewport*.) These decorators are different from the decorators contained in the *Layout*
- **Viewport Data Model,** contains the Area of Interest and CRS.

A *Page* may contain several *PrintingContextModels* based on its original *Layout*.

11.4 Printing Engine

The printing engine takes a *Page* and a printer and begins a specific rendering pipeline.

Functional Requirements:

- **Page,** provides *Layout* and *Printing Context Model* for output
- **Printer,** provides access to a Java2D PrintJob (may also be provided by a PDF library)
- **Local Catalog,** constructs what is to be printed from a *PrintingContextModel* using Metadata and Connection information from *Local Catalog*
- **Renderers,** constructed from *LocalCatalog* and *PrintingContextModel*, provide output of Spatial information to a Graphics2D from the Printer
- **Operation Control,** provide ability to cancel any ongoing operations

Printing Engine is "smart;" it determines whether a given WMS and WFS share the same source, and can use either to retrieve the same data. For printing, this will allow it to use WFS when available for the accuracy of vector data.

May need to separate out WMS requests into separate calls tiled together on the client side at native printer resolution.

12 uDig Extension Module Requirements

The ability to extend the uDig project is one of its central measures of success. A strong extension Framework is seen as a key provision facilitating an open source community. By providing an extension framework, the initial outlay to join the community is reduced, and the administrative overhead associated with contribution is reduced.

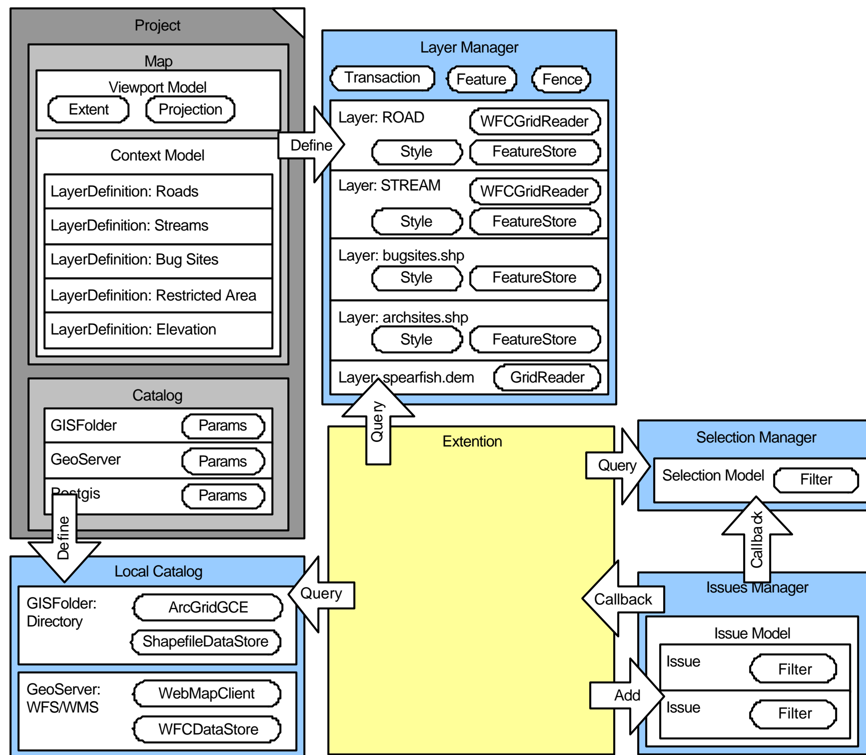


Figure 17: Extension Interaction Diagram

A plug-in (extension) is developed by accessing and communicating with the components exposed to the plug-in developers. The components are sometimes referred to as hooks and extension points. The primary extension points available to developers are the following:

- LayerManager
- ContextModel
- SelectionManager
- SelectionModel
- RenderManager
- Renderer
- StyleManager
- Style

To aid the development of Third Party Plug-Ins, a uDig Extension Framework will be provided that allows the development of Plug-Ins through sub classing. For example, a plug-in developer could extend a data operation class that would contain the functionality for obtaining data, and the plug-in developer would only have to provide the operation functionality.

Non-Functional Requirements:

- **UI Guidelines,** strong guidelines for UI creation/modification
- **Quickstart,** helpful classes and framework that allow developers to create a plug-in in under 30 minutes including download time.
- **Modular,** modular Plug-In extension mechanism, with version and dependency tracking.
- **Application Integration,** complete integration with the application, at both a programmatic and user-interface level.

Functional Requirements:

- **Data Access,** provided via *LayerManager*, or *Local Catalog* depending on plug-in intent
- **Data Modification,** connection to the user's Transaction is maintained via access to the *LayerManager*. Although a plug-in may provide its own Transaction when working against the *Local Catalog*.
- **Viewport Model,** provides access/modification of extent and projection
- **Selection Model,** access to/modify current selections
- **Issues Listing,** access/modify issues list providing unified plug-in callback
- **User Interface,** define views, add menu items/toolbars, key short-cuts
- **Operational Control and Feedback,** cancel or pause/restart a running operation, generate progress events
- **Dependencies and Versioning,** installation metadata to minimize conflict.
- **Local Results,** located locally for operation results.
- **Threaded,** framework has constructs and managers so that threaded plug-in operation is simple for the plug-in developer.

13 SUMMARY

This document has listed the requirements of the uDig application. From the application requirements, use-cases (not listed in this document) were derived. The application has been divided into components, which were grouped into logical modules. The use-cases were used to generate functional and non-functional requirements for each component.

The requirements listed in this document will be used as a basis for the following development processes:

- **Developers Guide,** the requirements will be referenced continually by the uDig application developers throughout development.
- **Acceptance Testing,** the non-functional requirements will be used to create acceptance tests. Acceptance tests will be used to verify that the non-functional requirements are satisfied.
- **Functional Testing,** the functional requirements will be used to create functional tests. The functional tests will be used to verify that the functional requirements are satisfied.
- **Design Documents:**
    - **Application Design,** the design of the uDig application in general. The system design must be driven by the functional requirements so that the final design will satisfy the application functional requirements of the application.
    - **W FS Design,** the design for the WFS client used by uDig. As with the system design, the WFS design will be driven by the functional requirements.
    - **User Interface Design,** parts of this document discuss user interaction requirements. These will be the basis of the User Interface design.
    - **Extension Framework Design,** the requirements for the Extension Framework will be used as the basis for the extension framework design.