The Use, Configuration and Implementation of the OME Image Table Tools

Harry Hochheiser
Computer and Information Sciences
Towson University
hhochheiser@towson.edu

May 25, 2007

Contents

Chapter 1

Introduction

The Open Microscopy Environment (OME) Image Table tools have been designed to support the creation of grid-based displays of images that are described by orthogonal dimensions of metadata. This document will describe the motivation behind these tools, how they can be used by end users, how installations can be configured and customized to meet the needs of specific installations, and (finally), how they features are implemented.

This document proceeds in levels of increasing detail. Many readers will not need to go beyond the introduction and the use of the system, as given in this chapter and Chapter 2. It is hoped that the document as a whole will provide enough information for the creation, and maintenance of site-specific installations.

The later sections of this document include substantial discussion of the internal details of how OME is implemented. Although this document is designed to stand-alone, there may be some details that may be clarified by examining background material and other documentation at the OME website (http://wwww.openmicroscopy.org).

1.1 A Motivating Example

Under the leadership of Minoru Ko, The Developmental Genomics and Aging Section of the Laboratory of Genetics at the U.S. National Institute on Aging has been studying gene expression in mice embryos at various stages of development [Yoshikawa et al., 2006]. This work has led to a set of microscopy images, with each image displaying the expression associated with a single probe at single time point in development. Images may also be associated with user-generated categories that describe the expression patterns.

The image meta-data can be used to places images in multiple, orthogonal dimensions. By "orthogonal" we mean that these dimensions are independent: classification in one perspective does not in any way constrain classification in any other. Furthermore, each of these dimensions may have hierarchical structure.

The data set from [Yoshikawa et al., 2006] has three orthogonal dimensions:

- *Development Stage*: Possible values: 1-cell, 2-cell, 4-cell, 8-cell, morula, or Blastocyst
- *Genetic Background*: The genetic background associated with a given image is defined hierarchically. Each image corresponds to a given genetic probe. Probes are grouped by gene, with each gene typically having sense- and anti-sense probes.
- *User Categorization*: Examination of images may lead to the creation of mutually-exclusive categories that describe the expression pattern.

Given these orthogonal classifications, images can be displayed in a grid, with genes/probes in the rows and developmental stages in the columns. Such a display was used in the original publication of this data [Yoshikawa et al., 2006]

1.1.1 Goals

The goal of the OME Image Table tools is to provide an interactive web-based tool based on the grid model from [Yoshikawa et al., 2006].

This tool would provide the following functionality:

- A grid-based layout of thumbnail images based on orthogonal characterizations
- The ability to add color-coded labels indicating the annotation of image from a set of mutually-exclusive labels.
- Search facilities for limiting the image display to a single gene.

- Detail displays for displaying large views of images, along with metadata.
- Links to external resources, such as the NIA Mouse Gene Index [1] [Sharov et al., 2005]. These links would allow users to retrieve the annotated sequence of the gene corresponding to columns of interest in the grid display.

The remainder of this document will describe how these tools can be used and configured, and how they are implemented. Throughout this document, the motivating example on the NIA mouse development images [Yoshikawa et al., 2006] will be used to illustrate the concepts and details involved.

[1] `http://lgsun.grc.nia.nih.gov/geneindex/`

Chapter 2

Using the Table Viewing Tools

The tools for viewing NIA annotated images include support for laying images out in a table based on annotations from orthogonal dimensions, color-coding based on classifications with respect to a chosen set of categories, category group, detail displays for annotated images, and other bells and whistles.

In terms of the example used in this document, the dimensions used for the table display will generally by the development stage (columns) and the gene/probe combinations for each image (rows). A possible categorization suitable for color coding might be expression localization, taking on values such as nuclear, cytoplasmic, and mosaic.

2.1 Image Table Viewing Tools

The Image Table Viewing Tools are designed to act as interfaces for image repositories established to supplement research publications. As a result, the web pages are designed to be used in a stand-alone mode, distinct from the traditional OME login. Alternatively full OME installations may be customized to add menu selections for viewing annotated data.

The anticipated usage scenario is that users would come across a gene of interest in the NIA Mouse Gene Index (Figure 2.1). A link on that page would lead to an OME display of images relevant to that gene (Figure 2.2).

Images associated with this gene are displayed in a two-dimensional grid. The columns are values of the developmental stage. Rows are probe values, grouped by the gene that each probe is associated with. In this case, there is only one gene: if there were others, they would be visible in the left-most column. The vertical extent of each gene will encompass the rows for all associated probes.

As noted in Section 2.2, only images that have been annotated as being publishable will be included in this display. Any other images will not be displayed, even if they meet the criteria for the grid.

Rows and columns will only be shown if there are images in those rows/columns. The developmental stage has additional values including 1-Cell, 2-Cell, etc. that are not shown because there are no images for those stages associated with the gene Krt2-8.

The types used for the rows and columns are specified in the template used to create the display: see Chapter 3 for more details.

Clickable links on this page can be used to explore throughout the OME database and external databases when appropriate. Specifically, the gene name is a link back to the NIA Mouse gene index. Pull-down menus for the category group, rows, and columns may be provided: if they area available, they can be used to change the dimensions plotted along the axes, along with the categorization criteria used for color coding. A separate pull-down for category can be used to limit the display to images from only one of the categories. For example, users might want to see only those images that have nuclear localization . If the category group selection is changed, the associated list of categories will be refreshed automatically. Pressing "Update Display" will lead to a new table based on the choices selected. Finally, a search box (labeled "Gene" in the example) can be used to filter this display for specific values in one of the rows/columns.

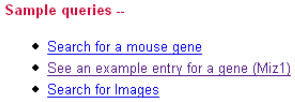

Figure 2.1: Example links: either of the first two links will lead to the Image Table View

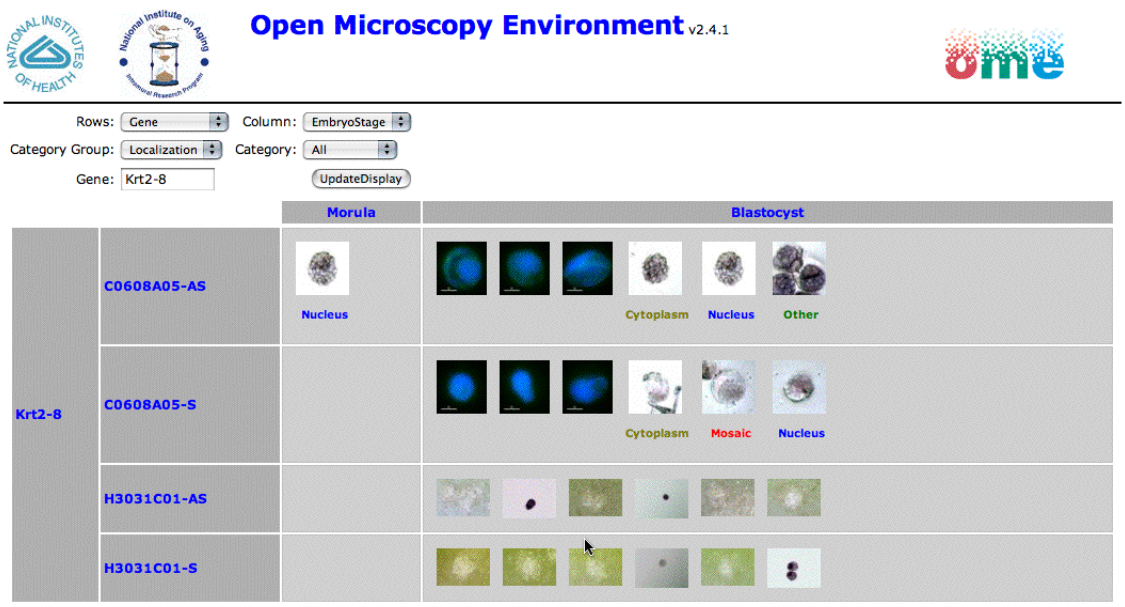

Figure 2.2: An example image table, with images for Krt2-8

Additional detail about any image can be retrieved by clicking on the thumbnail. (Figure 2.3).This display shows annotations for the image in terms of various annotation dimensions, and in terms of all available categorizations. If any publications are associated with the image, they will also be displayed, along with outlinks to PubMed or other bibliographic references.

The category group links on the main image grid lead to displays of all of the images associated with that category. Similar behavior can be achieved by selecting a category from the "Category" pull-down at the top of the page and pressing "Update Display" (Figure 2.4). Note that this page includes images for multiple genes. Each of the gene names is a link to the appropriate page in the NIA gene index, thus closing the circle.

2.2 Annotation Tools

Before images can be viewed in the 2D grid, they must be annotated. Values for each of the dimensions to be used in a categorization, along with any categorizations, must be provided by domain experts. In our example, each image must be associated with a probe, a gene, a developmental stage,a localization, and other values.

This annotation process often involves higher-order domain-specific constraints. For example, a given probe may be associated with a gene, and any gene may be associated with multiple probes. Such constraints requires special consideration during configuration (Chapter 3) and implementation (Chapter 4), and will only be described briefly here.

Specifically, the first step in any annotation is to select the "Search for images to annotate" link. This link leads to a pop-up that can be used to identify images of interest. Once the images are selected, the popup will close, and the annotation page will refresh, with the first image displayed prominently at the top of the page (Figure 2.5

To annotate this image, the user selects a probe, an embryo stage, and a localization from the various pull-downs and then presses "Save & Next" to bring up the next image. If there are any existing annotations for the image, they are selected as the default values of the pull-downs.

To create a new category for the category group on the page (in this instance, "Localization"), users simply type the name of the category into the appropriate field and press "Add Categories" to save the new category. If there are multiple category groups on the page, new categories can be added for each one with one submit, simply by filling in all of the desired values before pressing "Add Categories". Addition of categories does not change the current image or create any annotations; the user must still select the appropriate values and save their choices.

The shaded box to the right of the classification inputs can be used to create new probes: users simply type a new probe name, select a probe type, select a gene, and press the "create probe" button. The page will be refreshed, and the "Probe" input field in the classification inputs will be set to contain the new probe.

Finally, this page contains a checkbox labeled "Suitable for publication?" If this button is checked, the image will be annotated with a notation that indicates that it is publishable. Only those images that are thus labeled will be visible in any table displays.

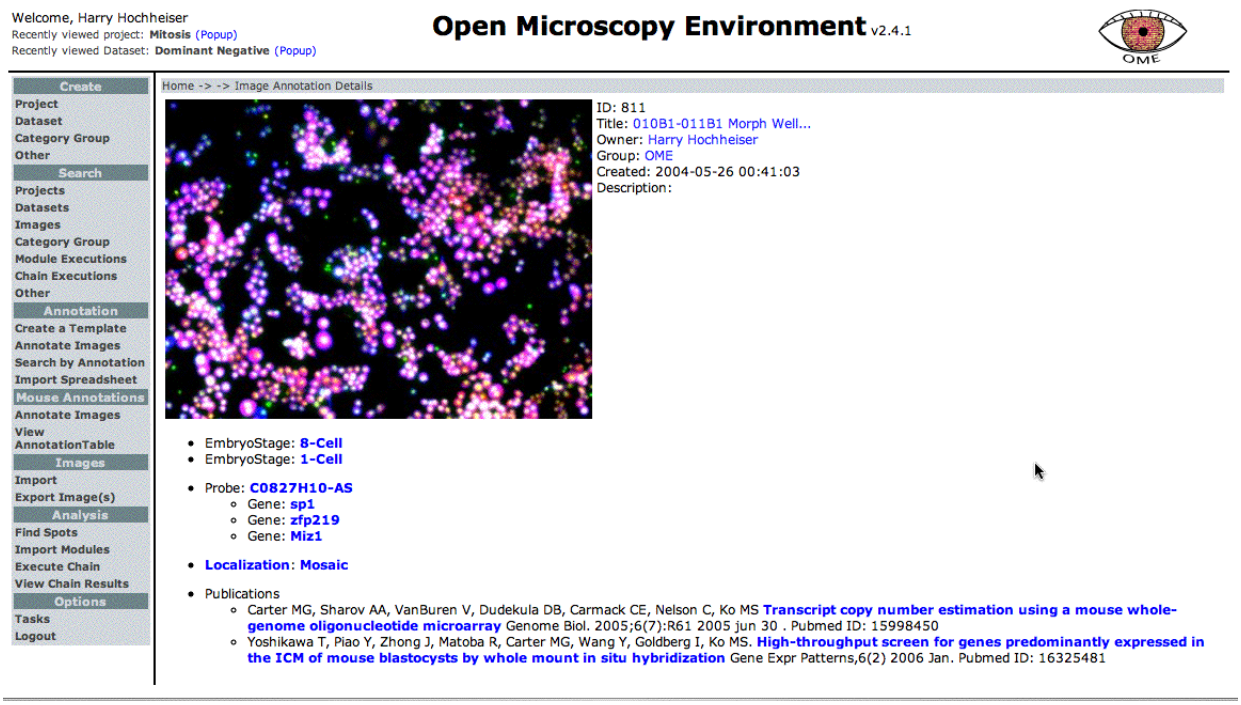

Figure 2.3: Details for a single image

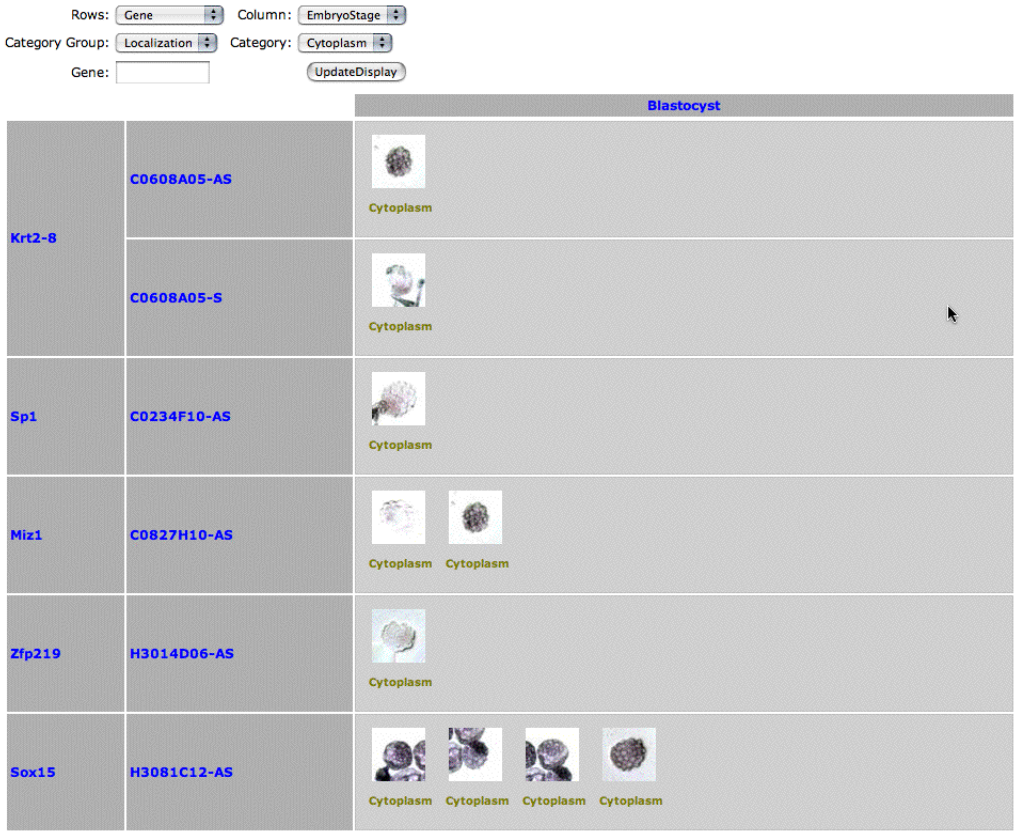

Figure 2.4: Details for a single image

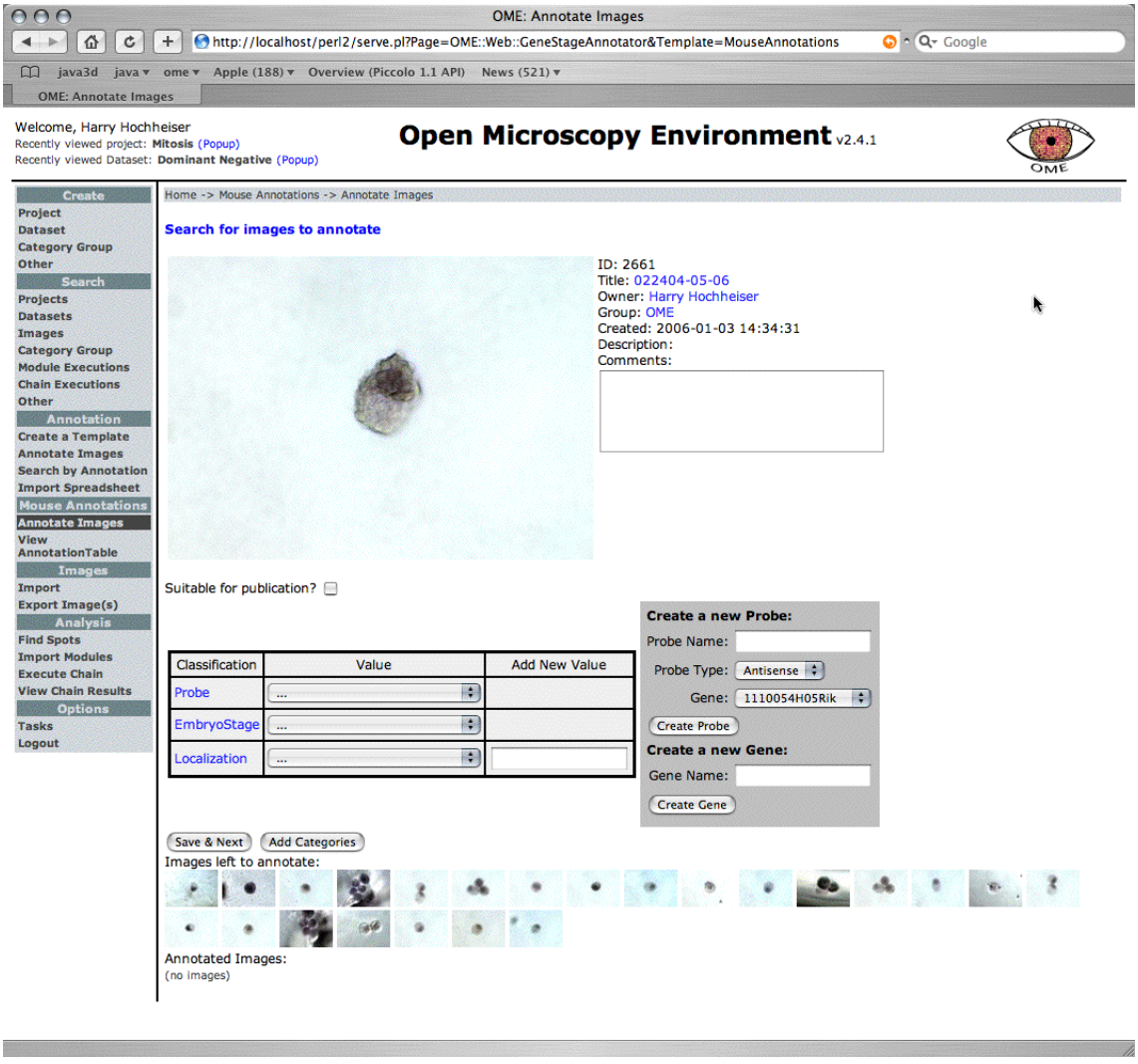

Figure 2.5: An example image table, with images for Krt2-8

Chapter 3

Configuring OME's Image Table Tools

Configuring an OME installation to act as a public image repository involves several steps which will be described in detail in this chapter:

1. Definition of an appropriate data model.
2. Implementation of that model in OME's *Semantic Types*
3. Modification of html template files and some Perl code to meet customized needs.
4. Bulk and/or Manual entry of images and relevant annotation data into the website.

As this document is intended for administrators who will be installing and maintaining an OME-based image repository, familiarity with Unix file structures, text editors, basic XML, and some Perl is assumed. This document also assumes an understanding of the OME system in general and the use of OME's "semantic types" in particular. This document will describe some of the terminology necessary. [1] .

This document will use the running example of Minoru Ko's mouse embryo data (Section 1.1) to explore the appropriate concepts.

3.1 Getting and Installing OME

OME can be downloaded from `http://cvs.openmicroscopy.org.uk/`. Look for the "Server Infrastructure and Web Interface" link listed under "OME Stable Packages". Installation notes and guidelines can be found at `http://www.openmicroscopy.org/install/`.

When installing OME, take particular care to make sure that you answer 'y' to the prompt about enabling guest access. This will configure the system to allow access to certain data views, even if the user has not logged in with a registered account.

3.2 Data Model Definition

As described in the introduction to this document (Chapter 1), the image table viewer is based on a hierarchical, model of orthogonal classifications of images. By *orthogonal*, we mean that the classifications are independent. By *hierarchical* we mean that each of these independent dimensions may have multiple levels of categorization.

Defining the dimensions that will be used for categorizing images, the attributes of these dimensions, and the hierarchical relationships between the dimensions will be the first stage in configuring the table display tools.

This is largely a conceptual task, best done by domain experts, who will determine which characteristics are most important, and how they should be displayed on the screen. Once these decisions are made, administrators and developers will implement these descriptions of data as OME `semantic types`.

The data model will often extend beyond the immediate needs of the table display to include additional descriptors that may be appropriate for meeting other data management and annotation needs.

As data model definition is an important step in defining how data is stored, managed, and interpreted, it should be undertaken thoughtfully and carefully. Although OME data models can be updated and revised, doing so may introduce additional challenges.

[1] For further details, see `http://www.openmicroscopy.org`.For an introduction to the concepts underlying OME, see `http://www.openmicroscopy.org/concepts`. An academic description is also available [Goldberg et al., 2005]

The display of mouse embryo images involved two main orthogonal dimensions: `EmbryoStage` is the stage of development, with values including 1-cell, 2-cell, 4-cell, 8-cell, Morula, and Blastocyst. This dimension does not involve any hierarchical sub-dimensions: each developmental stage exists independently of the others.

The second dimension - the genetic background- is hierarchical, consisting of *genes*, each of which is associated with one or more *probes*.

Two other data types play a role in the table display:

- *External Link*: a link to a related web resource, such as the Mouse Gene Index link for a given gene.
- *Publishable*: a boolean value, indicating whether or not an image is suitable for external publication.

A gene is identified only by its name. An embryo stage has two attributes: a name and a descriptions. Probes have names and types, which are described by yet another data type: the *probe type*. An ExternalLink has several attributes, describing the URL and templates needed to identify the appropriate external resource.

Additional modeling will be needed to determine which (if any) annotations will be used for color-coding images. Color-coding can be done on the basis of discrete categories . In the above example, the *Localization* of gene expression is used for color coding.

3.3 Data Model Implementation: Semantic Types and Category Groups

There are two forms of data models that must be created: semantic types and category groups. Semantic Types are used for the axes of the table display, while category groups are used for color-coding.

3.3.1 Semantic Types

Most Data in OME is described through the use of *semantic types*: structured data types described in a relational model. OME Semantic types exist are defined at different scopes of reference: global, dataset, image, and feature. For the purposes of this image table views, all semantic types will be assumed to be global in scope.

As semantic types are relational in nature, they will often include references to instances of other types. One item from a given type may be associated with many items from a second type, or arbitrary relationships between types, allowing many items from each type to be associated with many items of the other types. Such "many-to-many" relationships require declaration of additional mapping types that exist specifically to indicate the connections between objects.

Semantic types are defined using the "eXtensible Markup Language" (XML), using schema (data definition languages) defined specifically for use in OME. The necessary details of XML use and the schema will be shown by example in this document.

Each semantic type will define how each of its attributes can be stored in the underlying relational database. The syntax for specifying these details will be specified here as well.

In some cases, instances of the semantic types will be defined in advance. XML fragments creating these instances are defined as "CustomAttributes" in the OME file:

The semantic type definitions and their instances must be stored in an "OME" file, the outline of which is given in Figure 3.1.

The definition of individual types and instances is best demonstrated with some elements of our running example. The *EmbryoStage* semantic type, used to indicate the developmental stage associated with the image, is given in Figure 3.2. Note that the definition includes the name of the type, the granularity ("AppliesTo" attribute, set to "G" for global, "D" for dataset, "I" for image, or "F" for feature), a description , and several elements. Each element has several components

Name the name of the elements

DBLocation the location in the relational database where the element is to be stored. Usually, this will take the form of *TypeName.ElementName*, where *TypeName* is a version of the semantic type name ("EMBRYO_STAGE" for the "EmbryoStage" type) , and *ElementName* is the element name given above.

DataType the type of the element. Can be an SQL primitive data type, or "reference", indicating a map to another class - an SQL foreign key.

RefesrTo If the *DataType* of the element is "reference", "RefersTo" indicates the table for which a foreign key will be stored.

```
<OME
    xmlns=''http://www.openmicroscopy.org/XMLschemas/OME/FC/ome.xsd''
   xmlns:xsi=
     ``http://www.w3.org/2001/XMLSchema-instance''
   xmlns:STD=
     ``http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd''
   xsi:schemaLocation = ``
     http://www.openmicroscopy.org/XMLschemas/OME/FC/ome.xsd
     http://www.openmicroscopy.org/XMLschemas/OME/FC/ome.xsd
     http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd
     http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd''>
  <SemanticTypeDefinitions
      xmlns=``http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd''
      xsi:schemaLocation=''``http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd
       http://www.openmicroscopy.org/XMLschemas/STD/RC2/STD.xsd''>

  <!-- Semantic Type Definitions Go here -->

 </SemanticTypeDefinitions>

 <CustomAttributes>

 <!-- Custom Attributes Definitions Go here -->
 </CustomAttributes>
</OME>
```

Figure 3.1: The shell of an ome file

```
 <SemanticType
         Name="EmbryoStage"
         AppliesTo="G">
         <Description>Embryo's Stage of growth.</Description>
         <Element
             Name="Name"
             DBLocation="EMBRYO.STAGE.NAME"
             DataType="string"/>
         <Element
             Name="Description"
             DBLocation="EMBRYO.STAGE.Description"
             DataType="string"/>
     </SemanticType>
```

Figure 3.2: Definition of the *EmbryoStage* Semantic Type

```
<EmbryoStage
    ID="urn:lsid:foo.bar.com:EmbryoStage:456832">
  <Name>1-Cell</Name>
  <Description>Single Cell</Description>
</EmbryoStage>
```

Figure 3.3: Creation of an instance of the *EmbryoStage* Semantic Type

```
<SemanticType
  Name = "ImageEmbryoStage"
    AppliesTo = "I">
  <Description>A mapping between embryonic stages and images. Not
    really a many-to-many map, as an image is taken at exactly one
    embryonic stage. However, the definition of an image is not
    easily extended.
  </Description>
  <Element
      Name = "EmbryoStage"
      DBLocation = "IMAGE_EMBRYO_STAGE.EMBRYO_STAGE_ID"
      DataType = "reference"
      RefersTo = "EmbryoStage"/>
</SemanticType>
```

Figure 3.4: The definition of the *ImageEmbryoStage* Semantic Type

This definition must be included within the `<SemanticTypeDefinitions>` element in the ome file.

Instances of a SemanticType can be created via appropriate XML markup included in the `<CustomAttributes>` section of an OME file. These instances will be most appropriate when defining a data model that associates each image with one of several defined type instances.

For example, we may decide that *EmbryoStage* will have one of several values, including 1-Cell, 2-Cell, 4-Cell, 8-Cell, morula, and blastocyst. The definition of the instance for the 1-Cell case is given in Figure 3.3.

Each semantic type instance has one attribute and several elements. The attribute- "ID" is a life science identifier[2] - an identifier for a datum that is in theory globally unique. Although support for generation and resolution of these identifiers is in the plans, they are currently not directly supported in OME. Thus, this field should include a constructed unique ID based on your domain name. Thus, installations at Internet domain "foo.bar.com" might have an ID of the form "urn:lsid:foo.bar.com:EmbryoStage:456832", where the last digits are a unique number for the given instance.

The elements of the Semantic Type instance correspond directly to the elements contained in the Semantic Type Definition. Thus, an embryo stage instance will have 2 elements: a name and a description.

As described above, "many-to-many" relationships that can be used to define arbitrary links between items require the use of a special *mapping* type. In fact, a mapping type will be needed to connect any two levels in any of the hierarchical dimensions used in an image table display.

These mapping types will generally have a name that indicates both of the types being associated. The word "Map" will often be part of the type name. There are two types of these maps. The first will map between some Semantic Type and a component of the core OME system. Continuing with our example, the mapping between *EmbroyStage* and Images is handled with the mapping type *ImageEmbryoStage*, which is defined as given in Figure 3.4. Note that this type contains an explicit reference to an instance of the *EmbryoStage* type. The reference refers to the appropriate type by using the *FOREIGN_TABLE_NAME_ID* for the second part of the *DBLocation* attribute. The reference to an image is created by the granularity of "I", which creates a link to an image.

The other kind of mapping class contains two references, indicating a mapping between two semantic types, usually both of global granularity. Thus, *ProbeGene*, as defined in Figure 3.3.1, maps from instances of the *Probe* type to instances of the *Gene* type.

These types can and should be used to define the non-trivial paths from an annotation to an image that will create the hierarchical dimensions. . "Gene", "ProbeGene", "Probe" and "ImageProbe" can be used to specify a

[2] http://lsid.sourceforge.net

```
<SemanticType
    Name= "ProbeGene"
    AppliesTo = "G">
  <Description>Defines the relationship between Probes and Genes.</Description>
  <Element
      Name= "Probe"
      DBLocation = "PROBE_GENE_MAP.PROBE_ID"
      DataType = "reference"
      RefersTo = "Probe"/>
  <Element
      Name= "Gene"
      DBLocation = "PROBE_GENE_MAP.GENE_ID"
      DataType = "reference"
      RefersTo = "Gene"/>
</SemanticType>
```

Figure 3.5: The *ProbeGene* SemanticType

path from genes to images. Paths such as these will be explicitly created in the templates for annotating and viewing images.

The complete data model used for the annotations and display of NIA/DGAS images can be found in the MouseAnnotations.ome file in the main OME distribution. A list of Semantic Types defined in this file is given in Figure 3.6

Your data model will need to contain the types necessary for definition of your hierarchical dimensions, all appropriate mapping types, and any instances of the types that will need to be created prior to any image annotation. We strongly suggest that you study the MouseAnnotations.ome file, and the relationship between its contents and the notes in this document, before building your data model.

Once you have created an appropriate file based on the framework in Figure 3.1, you can use the OME command line tool to import the file into OME. Importing the file will create appropriate database tables and instances matching the contents of the OME file. To do this, type `ome import` *`filename`*, where *filename* is the file to be imported (with appropriate path information as needed). The command line tool will ask you to log in to OME. Once you have successfully authenticated, the file contents will be imported.

3.3.2 Category Groups

A category group is a set of discrete, mutually exclusive annotations: an image may be associated with exactly one category from any given group. Categories and category groups are examples of lightweight, simple annotations similar to tags used in photo web sites. As a simnpler form of annotation, category groups, categories, and links between images and categories are defined semantic types.

The OME Image Table Tools use category groups for color-coding. In our example, *Localization* is the category group that is used to color code the images (Figure 2.2).

3.4 Customization of Perl code and HTML Templates

3.4.1 Guest User Access

The table viewing tools were originally intended for use as a public repository: external links would lead users to these repositories based in OME. As this access would be read-only, the available OME functionality would be restricted to the viewing of relevant data. Access to object creation, searching, etc. would be prohibited.

In order to provide this restricted functionality while still providing the full set of web tools to authenticated users, the OME installation and Web tools have been modified to support authenticated and non-authenticated access.

The OME installation process includes the option of allowing guest access. If this option is chosen, a guest user account will be installed. Specifically, an experimenter with a name of "guest", first name of "Guest", a last name of "User", and password "abc123" will be created. A corresponding configuration value will be added to the system configuration table. After this is done, login requests for the guest user will succeed if the configuration flag is set to be true.

EmbryoStage values are 1-Cell, 2-Cell, 4-Cell, 8-Cell, Morula, and Blastocyst

ProbeType Sense and Anti-sense

Probe name, and type

Strain counterpart to a probe, used in worm experiments

Gene only contains a name.

ProbeGene many-to-many mapping from probes to genes. Generally, each probe will be associated with only one gene, but there may be multiple probes for a given gene.

StrainGene similar mapping between strains and genes.

ExternalLink a name, external id, and a URL for a link (to any sort of object) in an external database. ****

GeneExternalLink, [ProbeExternalLink, and StrainExternalLink mapping classes between various objects and external links. Not that the use of mapping classes supports the creation of multiple external links for any given object.

ImageProbe, ImageStrain, and ImageEmbryoStage maps that associate images with instances of types. In general, there would probably be only one instance of any of these types for any given image, but this is not enforced.

Publication A basic description of a research paper, complete with PubMed linking information.

ImagePublication a many-to-many mapping class linking images and publication.

PublicationStatus a flag indicating whether or not an image is suitable for public consumption.

ImageProbe, ImageStrain, and ImageEmbryoStage have image granularity, all others have global granularity.

Figure 3.6: SemanticTypes defined in MouseAnnotations.ome

3.4.2 Creating the Browse Template

GeneProbeTable.tmpl, found in src/html/Templates/Browse in the main OME distribution, is the main HTML Template for the image annotation table display.

The main purpose of this file is to specify the data paths that might be used for displaying images in the table. Specifically, GeneProbeTable.tmpl contains the following line:

```
<TMPL_VAR NAME="Path.load/types-[Gene:ProbeGene:Probe(ImageProbe)
  EmbryoStage(ImageEmbryoStage)]">
```

This template variable specifies two possible paths that can be used to get to an image. Details of these types can be found in the data model discussion.

- Gene, ProbeGene, Probe, (ImageProbe)
- EmbryoStage, (ImageEmbryoStage)

Each list starts with a semantic type name and follows it with the name of a mapping semantic type that maps between the first class and the third item on the list. This alternation of data type and mapping type continues until the end of the list. Parenthesized types indicate types that can be used to map directly from the previous type to images.

Alternative annotation table layouts can be created by copying GeneProbeTable.tmpl to a new file name in the same directory, and then modifying the template variable to include the desired paths.

GeneProbeTable.tmpl includes two template files that contain the bulk of the layout: ImageAnnotationHeader.tmpl and ImageAnnotationFooter.tmpl. Although these files can be modified to revise the layout of the annotation table, great care should be undertaken before any changes are made. As modifications to the template contents might adversely impact the operation of the code, consider reading the implementation notes before making any changes.

3.4.3 Creating the BrowseTemplate Database Entry

Once the template file has been created, an BrowseTemplate table in the OME database must be created. Log into OME, select create/other from the menu. and select BrowseTemplate from the pull-down menu. The Name and full path for the template must be provided in the name and template fields, respectively.

After the BrowseTemplate entry has been created, the group id for the associated module execution should be set to null. See Guest Access Configuration for a discussion of the group ID issue.

3.4.4 Creating the Detail Template

The detail template will be used to display a large image, along with annotation information for that image. The template will display which semantic type annotations will be displayed, and in which order. All category group annotations for the given image are displayed in any image detail template.

GeneProbeTable.tmpl, found in src/html/Templates/Display/One/OME/Image in the main OME distribution, is the main HTML Template for the image detail table display. As with the image annotation table template, this file contains some type specifications and an inclusion of a common footer. The type specifications are as follows:

```
<TMPL_VAR NAME="Path.load/types1-[ImageProbe,Probe,ProbeGene,Gene]">
<TMPL_VAR NAME="Path.load/types0-[ImageEmbryoStage,EmbryoStage]">
```

These entries specify the paths to be used to go from the image to some top-level annotation type. In a manner that is analogous to the types specified for the table template, these entries start with a mapping class of image granularity that maps the current image to an entry of the next type in the list. These types are followed by another map and another ST, repeating until the top-level ST is found.

In addition to providing a path to the types of interest, these declarations also imply a layout. The last ST in the list is displayed at the outermost level, with nesting for each contained ST. Thus, the first entry above implies that the Gene will be at the outer level with Probe nested underneath.

The number after "Path.load/types" provides an ordering: the path labelled 0 will be displayed first, 1 second, etc. Entries must be labelled consecutively, starting with 0.

Alternative annotation table layouts can be created by copying GeneProbeTable.tmpl to a new file name in the same directory, and then modifying the template variable to include the desired paths. As with the table templates, the layout can be modified by changing ImageAnnotationFooter.tmpl, but any such changes should be made carefully.

3.4.5 Creating the DisplayTemplate Database Entry

Just as the annotation table template has a BrowseTemplate entry, the detail display template must have a DisplayTemplate entry. In fact, both entries must have the same name, as the template names is used to determine which detail template is accessed from image links on the table display.

In the example above, a BrowseTemplate entry named GeneProbeTable.tmpl was created. The corresponding DisplayTemplate entry can be created by selecting "DisplayTemplate" from the pull-down on the create/other screen.

The following values must be provided:

- name
- arity is 'one'
- mode is'ref'
- template is the full path to the template file.

After the DisplayTemplate entry has been created, the group id for the associated module execution should be set to null. See Guest Access Configuration for a discussion of the group ID issue.

3.4.6 Module Execution Groups

OME's access control mechanisms use the group id of a module execution to determine who can access data associated with that module execution. Specifically, before access to an attribute is granted, the module execution for that attribute is checked to see if it has a group id that matches the current user.

This can cause problems for guest access, as attributes such as AnnotationTemplate, BrowseTemplate, or DisplayTemplate are often created with group ids set to those of the currently logged-in user. For example, this will happen if these objects are created through the web interface.

Proper guest usage of the NIA pages may require manually identifying the instances that have inappropriate group IDs in their mexes. Once identified, these instances should have their group IDs set to null.

3.4.7 Link Structure

After the above steps have been created, the display can be accessed via a call to OME::Web::ImageAnnotationTable:

http//localhost/perl2/serve.pl?Page=OME::Web::ImageAnnotationTable&Template=GeneProbeTable &Columns=EmbryoStage&Rows=Gene&CategoryGroup=Localization&Gene=Krt2-8

The CGI variables "Columns" and "Rows" must refer to type names from the start of one of the lists given in the annotation table template. These variables determine which variable goes in the row and which in the column. Obviously, they must have different values. The category group is optional: if none is provided, the images will not be color-coded by category.

The "Gene" parameter is an example of filtering. In addition to specifying rows and columns, The variables at the start of the type paths (in this case, EmbryoStage and Gene) can be used to filter the range of values that will be selected. Specifying "Rows=Gene" and "Gene=Krt2-8" implies that the only rows shown will be those corresponding to Gene Krt2-8. Similarly, "Columns=))EmbryoStage((" and "))EmbryoStage((=1-Cell" would limit the display to one column containing 1-Cell images. Multiple values, separated by commas, can be used to display multiple values: "Gene=Krt2-8,zfp219".

The link for image details is somewhat simpler:

http://localhost/perl2/serve.pl?Page=OME::Web::ImageAnnotationDetail&ID=439&Template=GeneProbeTable

Where ID is the OME image ID and Template is the name of the template being used.

3.4.8 Standalone operation

Configuration of the annotation display for standalone operation - that is, without login - requires some minor modifications to the OME source code and database.

These changes take two forms. Enabling guest access is necessary for non-authenticated users. Configuring of the home page, menus, and links is necessary for providing pages that do not have the full OME menus and headers.

3.4.9 Disabling Login

The guest access configuration is described in more detail in Section 3.4.1. To allow non-authenticated users, re-run the OME installation and select 'y' when asked if you want to enable guest access.

3.4.10 Configuring the Default Table Display

The file TableBrowse.pm, under src/perl2/OME/Web, provides a short-cut that can be used to access the image annotation table via abbreviated links. By providing default values for the template name, rows, and columns (at the top of the file), TableBrowse.pm allows for access to the table display with minimal additional parameters: simply the filter values for the rows or columns and a category group, if desired.

This file should be edited to provide the appropriate values for the $TEMPLATE, $ROW, and $COLUMN values for the installation.

3.4.11 Configuring the Header Builder

The HeaderBuilder.pm file under src/perl2/OME/Web/TableBrowse contains specification for a custom header to be used for the standalone installation. This file should be customized as need be. Several variables at the start of the file indicate links to various background home pages: these links are included, along with images, in the header. Special attention should be paid to the $HOME_LOCATION url. This URL, which is associated with the OME logo, will return to an empty version of the TableBrowse.pm page, allowing users to parameters for a new display.

The contents of this page should be configured to meet the needs of the configuration.

3.4.12 Configuring Web Display Defaults

Standalone installations should not have the full set of menus and links found in the generic OME web interface. Modifications to AccessManager.pm in src/perl2/OME eliminate the side menu bar and provide the custom header for guest users.

- In the getHeaderBuilder() procedure, the clause that currently refers to `OME::Web::GuestHeaderBuilder` (ie., the clause that is executed if the session is a guest session) should be modified to point to an alternative file that provides a header. The implementation in `OME::Web::TableBrowse::HeaderBuilder` provides an example.
- The getMenubuilder() procedure should be modified to return an appropriate menu for guest users. The easiest solution is to simply have this code return undef for any guest sessions. This will remove the menu bar for any pages served during guest sessions.
- getFooterBuilder() will create an optional footer for each pages. As currently configured, the page footer specified in `OME::Web::TableBrowse::FooterBuilder` will present a footer with links and logos for the National Institute on Aging. This page should be revised and customized to meet site-specific needs. Alternatively, to deliver pages without a footer, simply set the $FOOTER_FILE variable to be undef.

3.4.13 Configuring the home page

The default OME install writes an index.html page to the web server's DocumentRoot directory. The default contents of this page redirect users to the OME::Web::Home page. The contents of this page should be replaced with some approrpiate links, perhaps to some instantiation of the `OME::Web::TableBrowse` page.

3.4.14 Display Table Options

`OME::Web::ImageAnnotationTable` has two variables that should be set to fit the specific needs of each installation. Both default to zero, meaning the options will not be supported:

$ENABLE_ROW_COLUMN_CHOICES If this variable is set to a non-zero value, the image annotation table pages will contain pull-down menus that will allow users to choose the variables that are on each axis. All variables must still be specified within the template: these choices will simply let you rotate the table around a diagonal. This choice is not recommended for general use.

$ENABLE_CATEGORIES Setting this variable to a non-zero value will lead to the inclusion of pull-down menus for choosing a category group for color-coding. One pull-down menu will be used to choose a category group, while the other will be used to filter out the display to include only one of the values of the chosen category group.

3.5 Configuring the Image Annotation Page

These notes will use a "how to" approach to describe the annotation tools: a set of instructions for configuring the tools should capture the major pieces of how they work and interact.

Note that the model here is that we are going to associate images with semantic types on two orthogonal dimensions: probe/gene and embryo stage. We will also support the tagging of images with values from one or category groups.

3.5.1 Creating and Instantiating Data Types

Use of the NIA Annotation tools starts with the creation and instantiation of the data types specified in the data model.

To create the data types, the MouseAnnotations.ome file must be imported, perhaps via the OME commander.

Instances of the various types - namely probes, genes, and the mapping between them - can be created via OME's spreadsheet importer.

Finally, any desired category groups must be created. This can be done via the OME web page, by selecting "Create/other". Note that categories can be created in advance, but they need not be: the image annotation page includes facilities for adding categories to groups.

3.5.2 Creating the Annotation Template File.

Like most other pages in the OME Web interface, the NIA annotation page is created by a web template. ProbeStage.tmpl, in the src/html/Templates/Actions/Annotator directory, is the template used for the NIA annotation.

Much of this page is straightforward and similar to the annotation pages used in OME's Custom Annotation page. There is a space for a large display of the current image, tools for selecting additional images to annotate, pull-downs for selecting values of STs, facilities for creating new probes/genes, and displays of thumbnails for images that have and have not yet been identified.

In most cases, ProbeStage.tmpl can be used unmodified. There are two entries in the template file that specify the types of annotations that can be done on the page:

```
<TMPL_VAR
    NAME="DetailSTs.load/-[Probe:ImageProbe,EmbryoStage:ImageEmbryoStage]">
```

The "DetailSTs.load" entry includes two or more sets of values inside of the square brackets. The values are separated by commas, and each value is a series of types leading from some outermost type to an image: image is the implied last member of each list.

Thus, the above list indicates that two dimensions of annotation will be possible: 1) going from Probe to ImageProbe and then to image. 2) Going from EmbryoStage to ImageEmbryoStage to image. Note that these lists alternate between object classes and mapping classes. The web interface code takes advantages of this, showing only the objects to the user.

```
<TMPL_VAR NAME="CategoryGroup.load/id=[Localization]">
```

This variable specifies the names of zero or more category groups that can be used to annotate the images.

Both of these variables can be modified to fit the needs of the local installation. Other elements in the template can be modified and reordered as needed, but extreme care should be taken in modifying an HTML form fields or template entries. The HTML::Template documentation can be a useful resource for any non-trivial changes to the template.Test carefully and frequently - preferably after having backed up the original file.

3.5.3 Creating the AnnotationTemplate Entry in the Database

Once the template file has been created, an AnnotationTemplate entry in the OME Database must be created. To do this, log in to OME and select "create/other" from the left-hand menu bar. A name for the template must be provided (in the "name" field), along with the file system path for the template itself (in the "Template" field).

The module execution record for this newly created entry must be manually adjusted to have a group id of null. See Guest Access Configuration for details.

3.5.4 Link

Once the entry has been created in the annotation_template file, the annotation page can be accessed via the following URL:

`http://hostname/perl2/serve.pl?Page=OME::Web::GeneStageAnnotator&Template=MouseAnnotations`

Alternatively, if you don't need the facilities in the ProbeStage.tmpl template that support adding new probes, you can use

`http://hostname/perl2/serve.pl?Page=OME::Web::ImageDetailAnnotator&Template=MouseAnnotations`

In both cases, replace "hostname" with the name of your computer and "MouseAnnotations" with the name of the annotation template entry.

As this implies, ImageDetailAnnotator.pm and GeneStageAnnotator.pm are the modules that implement the annotation logic. GeneStageAnnotator.pm provides functionality for creating new probes, which is not included in ImageDetailAnnotator.pm. ¿

3.6 Importing Images and Annotation Data

Once the above steps have been completed, appropriate images must be imported. Relevant annotations.can either be entered via the annotation interface (Section 2.2) or entered in bulk from a spreadsheet. Bulk annotation via spreadsheet may require writing or modifying Perl code. However, this approach may be necessary for creating links to external web resources. See Section 3.6.2 for details on bulk import of annotations.

3.6.1 Importing Images Manually

Images can be imported to OME via the OME command-line tool. Simply typing `ome import` *`filenames`* will start the process. Note that this can take some time, as appropriate statistics must be computed for each image thus uploaded.

3.6.2 Importing Images and Annotations in Bulk

The general approach for bulk importing of image annotations involves creating a spreadsheet with the appropriate data and using the contents of that spreadsheet to create instances of semantic types that will be associated with each other, or with images, as necessary.

The approach to be described in this document is similar to, but distinct from, the OME Spreadsheet Importer `http://www.openmicroscopy.org/custom-annotations/spreadsheet_importer.html`. The spreadsheet importer assumes that all annotations are stored as categories and category groups: *ad hoc* collections defined outside of the Semantic Types.

As annotations for the OME Image Table Viewer must create both instances of Semantic Types, and of the associated mapping types, the approach is somewhat more complicated. These notes will describe the details of what must be done at a somewhat high-level: remaining information will be found in the *custom* directory of the main OME server distribution.

In particular, the *dgas-patch.pl* Perl script contains example code for configuring the OME server, importing images, and importing associated metadata. This script also contains code for configuring various elements of the OME server as appropriate. The associated *README* file, in the same directory, contains notes describing the use of the script - which is itself well-documented.

These notes will give a high-level introduction to the components that must be in place before the script can be executed, along with a high-level of what the code does. Please read these notes carefully before attempting to write your own versions, and *carefully backup all data* before attempting to run this script or any derivative.

These notes assume that all images to be imported are stored in a single directory, which can be specified (hard-coded) into the import patch file. They also assume that a spreadsheet file containing annotations has been created. Assuming the data model used in our running example, this spreadsheet file will contain the following columns:

Image the name of the image to be annotated. This should be the same as the prefix of the filename for the actual image file. Thus, if the file name is `foo.tiff`, this column should have the name `foo`.

Gene The name of the gene associated with the image

Probe The name of the probe associated with the gene.

EmbryoStage the name of the EmbryoStage, with values being limited to those that were defined in the Mouse-Annotations.ome file

ProbeType the type of probe - sense or antisense.

3.6.2.1 OME Server Configuration

The notes in *dgas-patch.pl* describe how the script enables guest access, configures templates, and imports type definitions from a file like MouseAnnotations.ome. Although these steps will generally not cause any problems if they are repeated, customized versions of *dgas-patch.pl* can omit them if needed.

3.6.2.2 Image Import

The Perl variable $IMAGE_DIR specifies the name of a directory. This directory is assumed to contain image files in the TIFF format, with filenames of the form `name.tif`. Image files in this directory will be imported into the OME repository. Each image will be given a name as determined by the filename: `image103.tif` will be associated with the OME Image object with name `image103`. This will be the image name that must be used in the spreadsheet containing the annotation data.

3.6.2.3 External Links

External Links are defined by two semantic types. The *ExternalLink* type contains a description, an external ID, a URL, and a link to a template. The template is an instance of `ExternalLinkTemplate`, which as a name, and a template string. The template string contains a URL containing the dedicated string " ID .".

If an *ExternalLink* contains a full URL, that URL will be used as the link for the object. If however, a link is not provided, the ID for the *ExternalLink* will be used in place of " ID " associated template, in order to find a URL for the item in question. This scheme allows multiple external links to point to the same base *ExternalLinkTemplate*, which would be the only item that would require updating in case of changes to the underlying web resource.

The patch file contains the variable $MGI_TEMPLATE_PATTERN, which specifies the link

`http://lgsun.grc.nia.nih.gov/geneindex5/bin/giU.cgi?search_term=~ID~` As being the template for the external link associated with a gene. Each gene will be associated with a *GeneExternalLink* instance, which will map between the *Gene* and the *ExternalLink*, which will contain the name of the gene as the ID.

The patch file will use this template pattern to create an instance of the *ExternalLinkTemplate*. As each row in the annotation file is processed, an appropriate *ExternalLink* will be created, referring to this template.

3.6.2.4 Annotation Import

The meat of the annotation import occurs in the `importAssociations()` procedure in the the patch script. This procedure starts by loading in the various semantic types needed, creating module execution and global module instances[3], and creating the appropriate external link template.

Once this initialization and set-up code is complete, the Perl module `Spreadsheet::ParseExcel` is used to read through the spreadsheet one line at a time. For each row, the appropriate values are read, and semantic types are created for each column. In addition, instances of linking types are created as need be. Specifically, the following steps are executed for each for.

- The image specified by the image name is loaded.
- An object is created (if necessary) for the associated gene.
- An object is created (if necessary) for the external link for that gene.
- A *GeneExternalLink* instance is created, mapping the gene object to the external link.
- A *Probe* instance is created.
- A *ProbeGene* instances is created, mapping the probe to the gene.
- An *ImageProbe* instance is created, mapping the probe to the image.
- An *EmbryoStage* instance is created.
- An instance of *ImageEmbryoStage* is created.
- A instance of *PublicationStatus* is created, setting the *Publishable* field to 1, so the image will be displayed.

Note that these instances will not be created if they already exist. Two strategies are used to avoid this duplication:

[3]An OME *module* is a computational unit that creates some data, a *module execution* is a record of a specific invocation of that module.

1. The `factory->maybeNewAttribute()` call will only create a new item in the database if a matching item does not already exist.

2. Loops that examine links between images and other types will search for a link to an appropriate target, and that target will be used if found.

Note that this script does not include any category group annotations. To see examples of how images might be associated with categories, look at `!OME::Web::GeneStageAnnotator`. This file calls on `OME::Tasks::CategoryManager->class` to classify images: given an instance of a category, and an image, this call will create an appropriate classification.

Although the details of the annotation will vary from one installation to the next, depending on the content of the data models, the code in this patch file should serve as a model that can be customized to meet the needs of any OME installation.

Chapter 4

Implementation

OME is a complex system, consisting of many layers of code for object-relational mapping, web user interface presentation, interaction with a standalone image server, and other functionality.

This document will not (and, in fact, cannot) describe the architecture and implementation of OME as a whole. Documents available on the OME web site present a solid introduction to various components. Specifically,

- For an overall introduction to the OME Perl server, take a look at the getting started guide: `http://www.openmicroscopy.org/getting-started/`.
- The "newbie" guide (`http://www.openmicroscopy.org/newbie/`)provides an overview of many implementation details.
- The Perl API documentation provides additional details (`http://www.openmicroscopy.org/api/perl/`).
- OME mailing lists (with archives) can provide useful information (`http://www.openmicroscopy.org/getting-involved/`).

The remainder of this chapter will discuss some specific issues that may be relevant to providing a publicly available, image table view of OME data. This document assumes a working knowledge of the general outlines of the OME system, along with some familiarity with object-oriented code.

All Perl module references are relative to `src/perl2` in the OME distribution. Thus, `OME::Web::AccessManager.pm` is found in `src/perl2/OME/Web` .

These notes will not provide a comprehensive design/implementation description: rather, they are intended to provide a roadmap to help developers orient themselves in the code, which should have further details.

4.1 Guest User Access

OME is generally an authenticated system: we assume that users are given an account by a systems administrator, and this user/name password combination is used to access the system.

For the purposes of the image table viewer, guest access is allowed. As described in Section 3.4.1, an option provided during installation of OME can be selected to support guest access. If this access is chosen, an appropriate guest user account will be created.

This section describes some components of the OME system that interact with the guest user facilities.

4.1.1 Session Management

`OME::Web::SessionManager` and `OME::Web` contain the bulk of the code that is used to get guest logins off and running. As `OME::Web` is the ancestor of all web pages, it provides the basic login verification code, which proceeds as follows:

1. If a cookie or URL parameter includes a session key, the user is authenticated and can proceed.
2. Otherwise, an attempt will be made to login with the hard-coded guest name and password.
3. `OME::Web::SessionManager` will allow a login with the hard-coded user name and password only if the system has been configured to allow guest logins.

Please note that any system that is intended for public use should not have a system-specific user named 'guest". As that user name will be restricted to use in this particular circumstance, any changes to the associated configuration (i.e., modifying the password) could cause problems.

4.1.2 Access Manager, Menus, Headers, and Footers

Although the table viewer is specifically intended for guest access, the underlying OME installation can be used as needed by authorized users. This requires two sets of menus, headers, and footers: one for authorized users, and one for guests. `OME::Web::AccessManager` provides call that are used to return the correct headers and footers, based on whether or not the current session is a guest session.

As currently implemented, this module contains procedures that return a default menu and header to authenticated users and an alternative to guest users. Code in modules such as `OME::Web::GuestMenuBuilder` and `OME::Web::DefaultMenuBuilder` provides examples of this differences, while `OME::Web::TableBrowse::FooterBuilder` illustrates a custom footer used in a guest installation for the National Institute on Aging.

`OME::Web::DefaultMenuBuilder` also supports the possibility of selectively adding menu option for annotation and display of annotations if the variable `ENABLE_MOUSE_ANNOTATIONS` in this file is set to be non-zero, a new menu section will be added, with selections for viewing the annotation table and annotating images.

4.1.3 Template Manager

As described in Chapter 3, the OME HTML pages are heavily template-driven. Introductions to this design can be found at `http://www.openmicroscopy.org/custom-annotations/`.

`OME::Web::TemplateManger` provides a centralized, common access points for retrieving any of the template instances used for constructing the OME web interface. This class provides methods for getting list of HTML templates, and specific instances for browsing, annotating, and displaying data.

4.1.4 Authenticated

`OME::Web::Authenticated` should be used as a base class for any Web functionality that should be restricted to authenticated users. This class provides one procedure: `getTemplate`. If the user is authenticated, this procedure provides the template that is appropriate for authenticated users. If not, `undef` is returned. This causes Web.pm to provide a page indicating that access is not allowed.

4.2 Table Viewing Tools

There are two parts to the annotation viewing tools: the annotation table, which displays a grid of images, and the annotation detail display, which provides a large rendering of the image along with annotation details.

As described in Chapter 3, both of these tools require the creation of templates and associated database entries. Furthermore, these facilities are used in pairs, linked by common template names. Thus, an annotation table template with name "foo" will always be used with a display template named "foo".

These implementation notes will provide a high-level overview of code and related facilities, including templates and their contents, cascading style sheet support, JavaScript, and other details.

4.2.1 Image Annotation Table and Table Browse

The code that generates the tabular display of image annotations is complex and at times a bit convoluted. Like the image annotation detail display, this code engages in arbitrarily nested hierarchies, which make the use of HTML Templates complicated at best. To further complicate matters, the annotation table does this in the context of a table layout, making things even more complicated. Thus, this code contains a good deal of HTML layout.

Furthermore, the annotation table generation takes a somewhat unusual view on OME object retrieval. Most OME web pages are built around the idea of displaying one or more "root" objects and then rendering objects associated with them. For example, an image might have attributes, classifications, module executions, etc. rendered along side of it. OME's data retrieval tools make this sort of reference-walking very easy.

Unfortunately, this approach does not work very well for a table-based display. Instead of starting from one single OME object - say, for example, a gene- and walking to a set of images, each cell in an annotation table contains the set of images that are associated with a value for the row and a value for the column. Thus, we cannot simply walk references. Instead, we must do our own querying and navigating of data structures.

Add some layout template trickiness, customization of image renderers, css definitions for clarifying display, and even a little bit of XMLHTTP request, and you get a fair bit of functionality. These notes will attempt to make sense of it all.

4.2.1.1 HTML Templates

As described in the description of configuring annotation viewing, the template for the annotation table contains a single variable that has 2 or more paths from a root semantic type to an image. This variable is found in `GeneProbeTable.tmpl`, which simply wraps the variable declaration with calls to include `ImageAnnotationHeader.tmpl` and `ImageAnnotationFooter.tmpl`.

The header file is straightforward, containing two JavaScript references and two hidden input variables. The JavaScript references load the sarissa[1] JavaScript library for platform-independent XML-HTTP requests, and the `categories.js` file, which includes code to handle XML-http requests made by this page. Details below.

The footer file is more complicated. Details will be described below, but this file can be divided roughly into several sections:

1. Optional pull-down menus for selecting the variables to be used in the rows and columns of the display.
2. Pull-downs for the category group and category.
3. A text entry field for the row.
4. A button for updating the display based on the above inputs.
5. A conditional section specifying either (a) an error message or (b) the contents of the table. If there is no error, the table consists of one or more rows of column headers, followed by one or more rows of data. Each row of data starts with zero or more headers.

Note that both the pull-down menus for the variables to be used in the rows and columns and the selections for category group/category are optional . If the variable `ENABLE_ROW_COLUMN_CHOICES` in `OME::Web::ImageAnnotationTable` is set to be non-zero, these pull-downs will be incorporated. If not, they will be omitted. This is accomplished via the conditional `TMPL_IF` mechanism, using the template variable `HasRowColumnChoice`.

Similarly, the variable `ENABLE_CATEGORIES` in `OME::Web::ImageAnnotationTable` is used to indicate whether or not the category/category group widgets should be displayed. For these components, the `TMPL_IF` mechanism will look for the definition of the variable `HasCategories`.

Various CSS tags are used to describe how the components should be rendered. These tags, and the use of the HTML template tags, will be described below

4.2.1.2 Initialization

The initialization of the `ImageAnnotationTable` code is straightforward. There are several instance variables that are used to store state necessary for the creation of the table. A quick summary of these variables is given here: more details will be provided below:

rows the name of the root variable for the rows in the table.

columns the name of the root variable for the columns in the table.

CategoryGroup the (optional) category group used for labeling individual images.

Category the (optional) category within the category group. if specified, the display will be limited to images associated with this category.

Template the name of the display template database entry that is being used to create the table layout. See the configuration notes for discussion of the database entries.

rowValue the value used to filter the rows: ie., the Gene for which items are being displayed.

rowSwitch a flag that will be set if the user has typed something in the rowValue text input field.

returnPage The name of the page that will be used to render the displays of images associated with a given category. This will usually, but not always, be `OME::Web::ImageannotationTable`.

[1] http://dev.abiss.gr/sarissa

4.2.1.3 Get Page Body and the Bulk of the Table Details

Like other descendants of `OME::Web`, the image annotation table code has `getPageBody` as its main entry point. However, unlike many other files, `OME::Web::ImageAnnotationTable` does not do the bulk of its work in this procedure. Instead, `getPageBody` does some boiler-plate work of retrieving the template, calls getTableDetails, and then outputs the appropriate resulting HTML.

This approach provides for the possibility of using the image annotation table code within some other page. As described above, the annotation tables are not based on HTML templates. This makes it very difficult to recursively call on the renderer to layout an annotation table. The `getTableDetails` procedure provides a means by which other pages could include a table: they can simply create a new instance of the image annotation table and call getTableDetails with the appropriate arguments. This call will return a chunk of URL which can then be assigned to an HTML template variable.

One example of this can be found in `OME::Web::TableBrowse`, which provides simplified access to the image annotation table. This is done by pre-populating default values of the template, row, and column. These values are then passed in to the template and used in a call to `getTableDetails`, allowing for a simpler, pre-populated approach to the annotation table.

This approach may not be as clean and elegant as a fully template-driven implementation, but it provides support that would not otherwise be available.

4.2.1.4 getTable Details

The arguments to getTableDetails are as follows:

- container is the OME::Web object that is calling this code. This value is needed to access the CGI parameters.
- which_tmpl is the name of the current template.
- returnPage is the page that should be used to render displays of images from a specific category. Usually, this will be the name of the page that generated the container.

The population of the table begins with the identification of the template entry in the BrowseTemplate semantic type, and the instantiation of that template.

The next few lines retrieve that value of the parameters that specify the variables used in the rows and columns. The next step is to compare the value specified for the row with the previous row (if it is set). If the previous row ('prevRows') is not the current row, then the rowSwitch instance variable flag is set.

This flag is used to clear out specified values for the current row/column if the value for the row is switched using the pull-down. For example, if a current display has Rows is the type Gene and Columns is EmbryoStage, the argument Gene=Krt2-8 might also be specified. If a subsequent request involves changing Rows to EmbryoStage and columns to Gene, the request for Gene=Krt2-8 is no longer relevant. In this case, the previous rows and the requested rows would be different and the flag would be set. Eventually, the getObjects procedure will use the value of this flag to determine whether or not the Gene=Krt2-8 specification should be used in building the table.

The template field "rowFieldName" is then set to match the name of the variable being used for the rows. This field is used to label a text input box used for filtering on possible values of the rows. This field, which is directly above the "update display" button, will always be labelled with the name of the current row. The text field itself corresponds to the template variable rowValue. Thus, if the display indicated that Gene was the type for the Rows and that Gene=Krt2-8, the rowFieldName would be Gene and the rowValue would be Krt2-8.

These fields are also checked to see if values of 'None' have been provided. If so, the class variables are not set, indicating that no Row/Column has been specified. If the Column is omitted, images will be shown based on the row value provided. A Row value must be provided: if it is not, an error will be reported.

Continuing through getTableDetails (additional info will be provided below as needed), getTypes and getChoices look at the types provided in the HTML template in order to populate the "Columns" and "Rows" pull-downs that let users specify the contents of the table.

Subsequent calls to getCategoryGroups and getCategories populate the category group and categories pull-downs.

The next chunk of code handles outputs and error conditions. First, the CGI parameter 'Base' is checked. If it is set, no further output is generated.

This behavior is used to provides a means of using the ImageAnnotationTable as a "return base". By setting this parameter, links can return to a blank page that can easily be adjusted and resubmitted to generate a table layout. This approach is used, for example, in the OME::Web::TableBrowse::HeaderBuilder class, to create a link associated with the OME logo. See the configuration notes for more details.

Assuming the Base parameter is not set, the code continues to check to see if the rows and columns specify the same value. If they do, and error message is generated. If not, a call to renderDims generates the HTML for the table.

After renderDims, there is some logic for managing the input text field that specifies the filter for the rows. Essentially, we want to populate that text field with the right value if (a) a value has been given and (b) the Type used for the rows has not changed. Case (b) can be further broken down to two cases: if there is no previous row, it has not changed. If there is, and it is not equal to the current row, then it has changed, and the input field should be left blank, by not setting the template variable.

The next step involves checking the return value from renderDims. If it is not set, then an error message is returned, indicating that no data was available. This is handled in the HTML template with a TMPL_IF block: if there is an error message, it is printed, and the table layout (given in the TMPL_ELSE clause of that TMPL_IF) is not generated.

4.2.1.5 getTypes and getChoices.

The getTypes procedures parses out the type specification in the HTML template and creates a hash, storing the parsed results. Furthermore, the instance variable ImageMaps will be updated to contain an entry mapping type names to any associated mapping types. The keys of the resulting hash are the first variables in each of the type lists. For each key, the value is a reference to an array including the full list of types for that entry. Thus, if there was an entry in the path list of the form Gene:ProbeGene:Probe(ImageProbe), the resulting hash would have an entry for Gene containing the array (Gene,ProbeGene,Probe). Furthermore, the instance variable ImageMaps will be modified to contain the value ImageProbe for the key Probe.

Given the results of getTypes, getChoices reads through the keys and uses them to build up the contents of the columns and row pull-downs specified by the Rows and Columns template variables. The 'selectedRow'/'selectedColumn' fields are set for an entry if it matches the value for the current row/column. Thus, if the URL or CGI parameters specify that this page has EmbryoStage in the column, the entry for EmbryoStage will have selectedColumn set, making it appear selected in the resulting HTML output.

Note that `getChoices` will only be called if the module variable `ENABLE_ROW_COLUMN_CHOICES` is set to be non-zero. If this variable is non-zero, `getChoices` will be called, and the appropriate template variable `HasRowColumnChoice` will be set, leading to the inclusion of these pull-downs. If this variable is not set, the pull-downs will be left off.

4.2.1.6 get category groups/getcategories..

getCategoryGroups basically finds all of the category groups and renders them in a list of options, which is then returned back to be placed into the template data hash as categoryGroups/render-list_of_options.

If a CategoryGroup parameter is set in the CGI params, that group is identified and set as the default value of the pull-down. Note that the CategoryGroup can be passed either by name or by OME ID.

getCategories finds the categories within the selected group, rendering them in the category pull-down. If a category has been specified in the CGI parameters, and if it belongs to the current group, it will be set as the default value. This will apply for situations where the display of all images from a given category are shown. Note that, as with groups, categories can be specified by name or id.

4.2.1.7 CategoryGroups & XMLHTTP-Request

The category group pull-down uses JavaScript to manage an XMLHTTP-request to update the category list: when the category group is changed, a request to the server will update the list of associated categories, without requiring a complete round-trip for the page.

The code for managing this is found both in the ImageAnnotationFooter.tmpl template file, and in the categories.js file in the JavaScript directory. When the category group pull-down is changed, the "changeCategoryGroup" procedure is called. This procedure then makes a call to the DBObjRender code for rendering a CategoryGroup in select mode.

Before describing the relevant template, it should be noted that this application represents a change of focus for DBObjRender. Previously, this module had been used for generating components of web pages, as specified in Templates, as opposed to being called directly. Modifications to DBObjRender made it suitable for use in a stand-alone context, as required for the XMLHTTP-request. Specific modifications include changes to createOMEPage to not provide HTML headers, and to getPageBody to process the request. The net effect of these changes is to provide for the return of a textual snippet of HTML that can be put into a div element, as opposed to returning a whole page.

The specific HTML snippet to be generated can be seen by following the parameters given to the renderer. According to categories.js, the mode is select and the type is CategoryGroup. Looking at the select.tmpl template in src/html/Templates/System/Display/One/CategoryGroup, the template generates an HTML select block with a name and Id of "Category", and the associated list of categories populating the select items.

When this request is processed and a result received, the JavaScript code takes the result and puts it into the span in ImageAnnotationFooter.tmpl with the ID "catSelect". Thus, the entire select - both the options and the select block itself - are replaced via DOM manipulations. Earlier efforts in changing just the options in the select block did not work so well. Apparently, DOM manipulations inside of such blocks often don't work.

This use of XMLHTTP-request depends upon the sarissa JavaScript library.

4.2.1.8 renderdims: Building the Table

The construction of the annotation table is a complex and somewhat convoluted process. The specifications for this layout included the goal of not including any empty rows or columns: if there are no entries at all in a given row or column, that row should not be shown. This essentially forces a two-pass strategy: we cannot render the column headers or any of the rest of the table until we know what data is present.

The first part of renderDims retrieves the objects associated with the rows and columns: as this may involve walking down trees of linked objects, we simply start with any roots (either explicitly specified or implicitly as all values of a given type) and build the tree of descendants leading to the type before the image. This tree is then flattened into an array for ease of processing.

Given these lists of leaf objects for the rows and columns, we populate the cells by retrieving the data for each. We also build hashes to tell exactly which rows and columns are not empty.

If we have actually retrieved some data, we populate column headers and then the body of the document.

Each of these pieces will be discussed in more detail below

4.2.1.9 Retrieving the objects

getObjects looks at the type for the type parameter to decide which root objects should be shown. For example, if Rows="Gene" and "Gene=Krt2-8" is found in the CGI parameters, getObject will identify that as the root. If not, all genes will be returned.

If the "type" parameter is not provided - as will happen if the specification for Columns is omitted - getObjects will simply return an array containing a single value to be used to indicate the absence of columns. This will be used by subsequent code to indicate that no columns are to be used in the display.

The mechanism for identifying root objects is actually handled by the getRoots procedure, which parses out a comma delimited list, identifying root objects by id or name, retrieving them from the database, and returning a list.

4.2.1.10 getTreeFromRootList, getTree, and flattenTree

These procedures recursively descend down the type path, starting from a set of roots, building data trees, and then flattening them into lists. These procedures are extensively documented inline: some high-level discussion is given here for flavor.

The general idea of the tree construction is that each of the root objects will be a key in a hash. The value for the root object will be a hash containing all of the associated objects of the next type in the type path. Thus, if we have a path containing Gene, ProbeGene, and Probe the top-level hash will contain genes, each Gene will point to a hash containing an entry for associated probes, etc. Mapping classes are used to build the tree, but are not explicitly rendered in the tree.

Items are stored in the tree with keys that are formed by combining the name and id of the item. This is a necessary hack to ensure that items get rendered in a semantically-meaningful manner (1-cell before 2-cell, etc).

getTreeFromRootList starts off this process, getTree does the recursion, and flattenTree recurses through to turn the result into an array. For a given input tree, the resulting array will contain one entry for each path from root to leaf. These paths will themselves be arrays, containing one entry for each (including roots and leaves) from root to leaf. As the tree is flattened, the ID numbers are stripped off of the hash keys.

4.2.1.11 Retrieving the image data

populateCells does the work of iterating over the row and column entries to find appropriate image data. Using the path types information (see inline documents for getAccessorName) and the contents of the flattened tree, the leaf objects for each given cell are identified and stored in hash: keyed first by row and then by column.

The retrieval is done by building up a hash containing all relevant criteria for both columns and rows. In either case, the criteria will include an entry for each of the parenthesized matching types found in the type specification. So, if we have a row type of the form WormLocation(ImageWormLocation):WormLocationStrain:Strain(ImageStrain), and instances of WormLocation and strain, the following criteria will be used:

1. ImageWormLocation.WormLocation.Name = ¿ name of worm location
2. ImageStrain.Strain.Name =¿ name of strain

Note that the relationship between worm location and strain (ie., the existence of a correspondence between a given location and strain) has already been established by the set of queries that identified the contents of each row (see description above).

The images are stored in a nested hash, keyed first off of a key for the row contents and then off of the column contents. The keys used are simply concatenations of all of the elements in the given row/column entry.

If the columns are unspecified, the second level of the hash will be keyed by a single value.

Before a given list of images is returned, it is filtered by category group and category: if category and a group are specified, only images that match those criteria will be retained: others will be eliminated.

getActiveList is used to take the flattened data lists for rows and columns and to filter those that do not have any related images. The result should be arrays containing path arrays for those rows/columns that have image data to display.

At this point, the required data has been retrieved, and the table can be rendered.

4.2.1.12 Rendering the table.

To see how the table is rendered, we go back to the code in renderDims. if there are images that need to be rendered, the code starts by finding the size of the row entries, with a call to getHeaderSize.

This value is needed to determine how many columns will be needed to display the headers for the row, and it is equal to the number of types in each row. Thus, if the rows contains probes and genes, the header size will be 2.

This value is used to populate the template variable named rowHeaderCount. In the ImageAnnotationFooter.tmpl template, this variable names an HTML COLGROUP that specifies the column widths to be "0*" - occupying only as much space as is needed. Note that this may not work well on all browsers.

The headers for the table columns are populated via populateColumnHeaders. This somewhat convoluted chunk of code is complicated by the desire to avoid repetition. If I have Genes and probes as my column types, and I have 3 probes associated with a given gene, I would like to have the column entry for that gene span the three columns for the probes. To do this, I examine each column name to see if I have something that matched what came before. if it does, i do not have any output. If it does not, i find out how many times it occurs (via getRepeatCount) and set an appropriate colspan correctly. populateColumnHeaders also leaves the appropriate number of columns empty (via populateEmptyColumnHeaders), in order to leave columns for the row labels. The appropriate layout code is found in the template in the loop with the template name "columnHeaders".

All column names are given as output that takes one of three forms, in decreasing order of priority: 1) a link to the appropriate ExternalLink, 2) the OME object detail or 3) plain text.

Columns are omitted if they were not requested in the original CGI request.

populateBody creates the bulk of the annotation table, populating the template variable "cells". The structure is similar to populateColumnHeaders - particularly in terms of having labels span rows. However, this procedure must build up labels and contents for each row. The label logic is analogous to populatecolumnHeaders, and then populateRow iterates over the cells in order to pull out the images and render them, via a call to getRendering.

4.2.1.13 Rendering the cell contents

getRendering renders a set of images, possible relative to a given category group. This is done by calling renderArray on the associated renderer, with a set of images, an appropriately populated data hash and template argument.

If a category group is provided, the images will be sorted by category - see sortImagesByCG for details. This procedure finds the relevant category for each image in the category group, and sorts the images by name of the category.

The data hash passed to the renderer contains specification of the Rows and the Columns , the template name, the return page, and the category group if specified. These items are used in the templates and custom rendering code for each image.

The details for the rendering can be seen in the mode of the renderArray call: "color_code_ref_mass_by_cg". This template, found in src/html/Templates/System/Display/Many/OME/Image, looks very much like others for rendering images. There are two loops - one for the color-code ref -map, and one for the st annotation display. The

ref_st_annotation_display.tmpl (found in Templates/System/Display/One/OME/Image) sets up the image map with the upper-left-hand corner going to the generic object detail page with the rest going to the ImageAnnotationDetail page for the image. This is (one instance) where the Template parameter to the hash will get passed through to provide the appropriate information for the ImageAnnotationDetail link.

The color_code_ref_map_by_cg contains the usual image and map statements found in ref_map templates, along with additional fields for the category label. These fields are populated by special-purpose code in `OME::Web::DBObjRender::__OME_Image.pm`. This code looks for the "classificationColor" field in the image rendering requests. If it is found, a color from a list is chosen and an appropriate CSS statement is generated (see getClassificationColor). Parameters from the CGI, including the template, rows, columns, category group, and category are used by getLinkParams to build an appropriate link.

4.2.1.14 CSS Notes

There are several CSS declarations defined in "src/html/ome2.css" that are used in ImageAnnotationFooter.tmpl and related templates. These declarations establish color and spacing for the image annotation table, so any changes should be tested carefully.

The "ome_category_border" class is particularly important. This class is used for the DIV that wraps every image in a set of images in the table. the "flow: left" entry guarantees that images in a cell will flow nicely in a cell, adjusting layout as necessary.

4.3 Image Annotation Details

This note describes the image annotation detail display implemented in `OME::Web::ImageAnnotationDetail`.

The NIA Annotation Detail works off a template that specifies zero or more paths from images to semantic types. Each of these paths takes the form

`<TMPL_VAR NAME="Path.load/types<NUM>-[<MapST1>.<DetailST1>,<MapST2>,<DetailST2>, ...<DetailSTn>]>`

- NUM is an integer 0, 1, ... indicating the order in which items will be displayed. Expect to end things badly if indices are omitted or repeated.
- MapST1 is an image granularity ST that maps images to instances of DetailST1, a type that contains some value of interest.
- MapST2n is a mapping ST between DetailST(2n-1) and DetailST(2n+1).

Thus, the entrry `!<TMPL_VAR NAME="Path.load/types1-[ImageProbe,Probe,ProbeGene,Gene]">`
indicates that ImageProbe is an image-granularity ST that maps to Probes, and ProbeGene maps from there to Gene.

Given these specifications, there are three stages of displaying the annotation details: image display, ST annotation display, and Category Group Annotation display.

4.3.0.15 Displaying the image.

The ImageAnnotationFooter.tmpl template contains an HTML template for rendering the image:
`!<TMPL_VAR NAME=image/render-large_no_comments>`
The getImageDisplay procedure parses out this template variable and calls the renderer to render the image according to the specified mode - in this case, large_no_comments.

The mode specified must correspond to a template file stored in src/html/Templates/System/Display/One/OME/Image. In this case, the template file is large_no_comments.tmpl.

4.3.0.16 Displaying the Semantic Type Annotations

The display of semantic type annotations is the most potentially confusing part of this code.

The first part - parsing of the template variables is relatively straightforward. The procedure getPaths parses the template variables, identifying those that specify paths. They are examined in order of their indices, and each is split into an array of the constituent pieces. References to these arrays are stored in a result array, and a reference to that is returned to the caller.

The output HTML for these paths is generated by the procedure getDetail, which calls getPathDetail for each path and builds up an HTML string that displays all of the resulting annotations.

A note on the philosophy of the code might help clarify the details of the implementation. Most of the OME Web code is highly template driven, leaving the Perl code to simply populate template variables. As a result, HTML is extremely rare. For all of the usual reasons about separations of concern, this is a good thing.

Unfortunately, this mechanism breaks down for the annotation detail display. In this display, we want to use arbitrarily nested lists to describe containment: an image might be annotated with Type A, which is associated with one or more Bs, and each B has one or more Cs, etc.

Specifying this sort of arbitrary nesting in an HTML template might be possible, but it is not straightforward. One possibility would be to attempt some sort of recursive template, but it's not clear if this will work.

Given this difficulty, it seemed much more straightforward to implement the layout in Perl. To that end, getDetail calls getPathDetail to get HTML strings for each of the path entries, and simply concatenates them. Thus, these procedures are operating entirely in HTML land, devoid of templates. The result of getDetail is then stuffed into a template variable.

getPathDetail starts from a root object Image of type "OME::Image" and walks down the contents of each path. Each iteration peels a map type and a detail type off of the path list. The map type is used to find map instances associated with the root. If there are no map instances, the procedure returns.

If, however, some maps are found, there are two possibilities: if the end of the path has been reached, the target of each map is identified, the maps are placed in an HTML list. If, however, there are more entries in the path, then each map object is taken as a root for a jumping off into a recursive call to getPath Detail.

This recursive calling makes it crucial that the path type array is passed in by value, not by reference. Given Perl's semantics, these value arrays are copied, giving each recursive call its own copy of the array, which can be modified as necessary. This may be potentially inefficient, but no more so than explicitly copying elements to create new arrays. Avoiding this inefficiency might require elimination of the recursion.

Note that each level of recursion creates a new nested HTML list.

Where appropriate, the objects in the list are populated by calling getObjURL. This procedure uses a two-step operation to generate a link for an object. First, the type of the object is used to infer a link to any ExternalLink instances that may be associated with that object. If one is found, it is used. If not , the OME object detail URL is used.

4.3.0.17 Category Group Annotation Display

The category group annotation display is handled by getClassificationDetails. The implementation is straightforward: `the OME::Tasks::CategoryManager->getImageClassification` call is used to retrieve all of the classifications for the image, calling getClassificationDetail on each one. getClassificationDetail generates the necessary HTML, including links to the category group and the category as available.

Unlike the Semantic Type annotations, the category group annotations are laid out using HTML template code. Each annotation populates the "catGroup" and "catValue" template parameters, which are then appended to a list used to populate the "categoryAnnotations" template loop.

4.3.0.18 Publication Display

Publications associated with the image are handled by getPublicationDetails, which renders each of the publications in a list, using appropriate templates to layout each publication.

4.4 Annotating Images

The code for annotating of images can be found in `OME::Web::ImageDetailAnnotator`. This code is very similar to the code in `OME::Web::CG_Annotator`: in both cases, the code populates a template with image to be annotated, followed by fields for specifying annotation values, and finally lists of images that remain unprocessed, and should still be processed.

The template provided for the gene/probe/development stage example used in this document can be found in `src/html/Templates/Actions/Annotator/ProbeStage.tmpl`. After starting with some hidden fields necessary for tracking the current state of the form, this template renders the current image, provides a checkbox for indicating whether or not the image is suitable for publication, and then populates a table with the semantic types and category groups that will be used for annotating the images. This table is followed by control buttons for saving details on the current image and adding categories, and then lists of selected images that have not yet been annotated, and those that have been annotated. An optional section dealing with the creation of genes and probes will be describe below in Section 4.4.2.

4.4.1 Image Detail Annotator

`OME::Web::ImageDetailAnnotator` starts off by loading the category groups and semantic types that are specified in the template. Semantic types to be used are specified in terms of the immediate type, and the mapping type that goes from that type to an image. Thus, the inclusion of "Probe:ImageProbe' in the DetailSTs field of the template indicates that instances of the Probe semantic type should be provided for selection in the pull-down menu, and they will be linked to Images via instances of the ImageProbe semantic type.

Similarly, category groups of interest are loaded from the "CategoryGroup.load" parameter list in the template.

Once these details are loaded, `populateImageDetails` is called to display the image currently under consideration.

If the "Save and Next" button was pressed, the appropriate annotations will be created (via calls to `annotateWithSTs` and `annotateWithCGs`, and the current annotation will be set. Similarly, if the "Add Categories" button was pressed, an appropriate category will be created.

The pull-down entries for category groups and semantic types will be created by calls to populateAnnotationSTs and populateAnnotationGroups.

4.4.2 Gene Stage Annotator

Annotation tasks will often require extended functionality that adds domain-specific logic. `OME::Web::GeneStageAnnotator` builds upon `OME::Web::ImageDetailAnnotator`, adding facilities to create genes and probes. Note that `GeneStageAnnotator` is not a subclass (as it probably should be) - instead, it is a parallel class with much cut-and-paste code reuse. Ideally, this code would be more cleanly reused through subclassing.

`GeneStageAnnotator` differs from `ImageStageAnnotator` in the presence of the `populateProbeFields` procedure, which calls `populateProbePulldown`, and in the `createProbe` and `createGene` procedures. The `populate...` procedures fill in a box presenting a field for a new probe name, a pull-down for selecting a gene to go with the new probe, and a text box to create a new gene. Given appropriate values, `createProbe` and `createGene` will build instances of these types, allowing the user-driven evolution of the database. Once genes and probes are created, they can be used to annotate images.

Bibliography

[Goldberg et al., 2005] Goldberg, I., Allan, C., Burel, J.-M., Creager, D., Falconi, A., Hochheiser, H., Johnston, J., Mellen, J., Sorger, P., and Swedlow, J. (2005). The open microscopy environment (ome) data model and xml file: open tools for informatics and quantitative analysis in biological imaging. *Genome Biology*, 6(5):R47.

[Sharov et al., 2005] Sharov, A. A., Dudekula, D. B., and Ko, M. S. (2005). Genome-wide assembly and analysis of alternative transcripts in mouse. *Genome Res.*, 15(5):748–754.

[Yoshikawa et al., 2006] Yoshikawa, T., Paio, Y., Zhong, J., Matoba, R., Carter, M. G., Wang, Y., Goldberg, I., and Ko, M. S. (2006). High-throughput screen for genes predominantly expressed in the icm of mouse blastocysts by whole mount in situ hybridization. *Gene Expression Patterns*, 6(2):213–224.