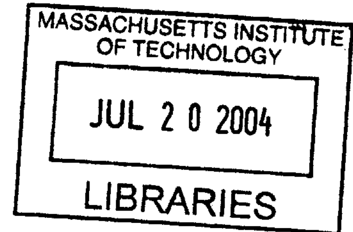
Visualization and Management of Large Biological Imaging Datasets

by

Jeffrey C. Mellen



Submitted to the Department of
Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 2004

© 2004 Massachusetts Institute of Technology.

Author ....
Department of

Certified by...

Associate Professor
Department of iter Science
Supervisor

Certified by..

Peter K. Sorger
Associate Professor
Biology
ervisor

Accepted by..................

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

BARKER

Visualization and Management of Large Biological Image Datasets

by

Jeffrey C. Mellen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Open Microscopy Environment (OME) image browser enables biologists to quickly analyze, manipulate and modify large imaging datasets. The browser includes a variety of features that facilitate image classification, annotation, visualization, and organization. The browser displays image metadata using a variety of techniques, including visual cues, context-sensitive overlays, and color-coding. The application explicitly supports visualization of screening datasets, but also supports multidimensional images, as well as standalone images. When integrated with the rest of the applications of the OME client software, the browser allows users to view images in greater resolution, analyze multiple dimensions, and in future releases will support analysis routines.

Thesis co-supervisors:

Dennis M. Freeman
Associate Professor
Department of Electrical Engineering
And Computer Science

Peter K. Sorger
Associate Professor
Department of Biology

Acknowledgements

OME is a joint effort between the Sorger Lab in MIT's Department of Biology, the Wellcome Trust Biocentre at the University of Dundee in Dundee, Scotland; the Institute of Chemistry and Cell Biology at Harvard Medical School, the Image Informatics and Computational Biology Unit at the National Institutes of Health in Baltimore, and the Laboratory for Optical and Computational Instrumentation at the University of Wisconsin.

I couldn't have completed this project, or this thesis, without the assistance and advice of Chris Allan, Erik Brauner, Jean-Marie Burel, Andrea Falconi, Denny Freeman, Ilya Goldberg, Harry Hochheiser, Dan Rines, Peter Sorger, Jason Swedlow, and especially Doug Creager. Thanks for the opportunity and the codebase. The movie will be made.

I would also like to thank Matthew Aichele, Yuriy Brun, Sheldon Chan, Bren Cox, Evan Davidson, Jessica Mellen, John Ritchie, Rahul Sarathy, Cathy Shaw, Tony Scelfo, Tony Walters, Josh Yardley, and Sarah Zhou for tolerating me throughout the project.

This project is dedicated to my wonderful parents, Tim and Peggy Mellen. You finally get to see some return on your investment now!

Table of Contents

1 Introduction 11
1.1 Collecting user interface requirements 13
1.2 Project evolution/summary 14
1.3 The OME Project 14

2 The User Interface 17
2.1 The OME Client 17
2.2 Browser Components 21
2.3 Displaying Images 24
2.4 The Heat Map & scalar variable display 30
2.5 Image Classification 34
2.6 Annotations 36

3 Design & Implementation 37
3.1 Browser Overview 37
3.2 The Browser Agent and application-level classes 40
3.3 Core browser classes 44
3.4 Thumbnails & drawing 49
3.5 Event handling, actions and behavior factories 53
3.6 Heat & Color maps 58
3.7 Integration: Loading a dataset 61

4 Supporting Large Image Datasets 65
4.1 Semantic zooming 65
4.2 Color coding 68
4.3 Contextual Layouts 71

5 Sample Datasets 75
5.1 ICCB full plate 75
5.2 A sample dataset of 5D images 78

6 Conclusion....................81
6.1 Future Work....................81

Appendix A: Code & Documentation URLs....................83

Appendix B: Complete Source File List....................85

List of Figures

Figure 1.1: OME 2.2 architecture 15
Figure 2.1: A screenshot of the entire OME client 18
Figure 2.2: The OME image viewer 19
Figure 2.3: The Annotator module 20
Figure 2.4: An empty browser window 21
Figure 2.5: The heat map window 23
Figure 2.6: The color map window 23
Figure 2.7: The phenotype editor 24
Figure 2.8: The image browser displaying a screen 27
Figure 2.9: Quantum treemap and heatmap mode 28
Figure 2.10: The image magnifier 30
Figure 2.11: Heat map Boolean mode 34
Figure 2.12: Classifying an image 35
Figure 2.13: An image annotation 36
Figure 3.1: The class hierarchy of the image browser 38
Figure 3.2: The BrowserAgent hierarchy 41
Figure 3.3: Agent event classes 43
Figure 3.4: Application-level classes 44
Figure 3.5: The browser core classes 46
Figure 3.6: Thumbnail classes 51
Figure 3.7: Paint method classes 52
Figure 3.8: The browser event & action classes 55
Figure 3.9: Heat map classes 60
Figure 3.10: Color map classes 61
Figure 4.1: Paint method classes (inheritance) 67
Figure 4.2: Color map classes II 70
Figure 4.3: Layout method classes 73
Figure 5.1: The 384-well ICCB dataset 76
Figure 5.2: A small 5D image dataset 78

CHAPTER 1
Introduction

In recent years, quantitative analysis of biological images has become a common technique for testing scientific hypotheses. Biological images are now frequently the subjects of numerical analysis and the basis of experiments[1] in chemical screening and in modeling intracellular processes. Researchers can now take high-resolution movies of cells in five dimensions, and run screening experiments using thousands of compounds at a time, to verify biological models and determine compound behavior. The hardware and methods used to generate the terabytes of imaging information necessary to conduct these experiments have become more sophisticated and more efficient[2]. However, the software used to manage, classify and analyze this data has not matured at the same pace.

Currently, biologists who work with large image datasets manage them in an ad-hoc manner, using whatever tools are available. Often, they save libraries of thousands of images in folders on a normal filesystem, identifying each unique image only by filename. To take notes about a particular image or series of images, they use a notebook or spreadsheet, which may or may not be in sync with the dataset. To search for images that meet a certain criteria, they usually scan through their notes, or scan through the images themselves with a viewer

---

[1] Swedlow, J. R., Goldberg, I., Brauner, E., Sorger, P.K. "Informatics and Quantitative Analysis in Biological Imaging." Science, Vol. 300 (April 4, 2003); 100-2.

[2] Yarrow, J.C., Feng, Y., Perlman, Z.E., Kirchhausen, T., Mitchison, T.J. "Phenotypic screening of small molecule libraries by high throughput cell imaging." Combinatorial Chemistry & High Throughput Screening, June 2003; 279-86.

such as Metamorph[3], one at a time. This workflow and these ad-hoc tools are imprecise, inefficient, and ill suited for analyzing entire datasets as a whole.

To better analyze the volumes of data generated by imaging experiments, biologists need a tool that can quickly extract and display information about hundreds, even thousands of images at a time. Such a tool would ideally allow the biologist to identify and organize images by phenotype, create annotations on the fly, and display relevant image metadata at the dataset level. This tool would have several advantages over current ad-hoc solutions. First, the tool would unite an image and its metadata. A biologist would no longer have to search for image information in auxiliary files or in a notebook; an image's data would instead appear in the same application and same location as the image itself. With images and their metadata united, a biologist could more quickly discover correlations between quantitative variables, and between variables and phenotypes. Such an application would also enable the biologist to perform comparative analysis of a large number of images, often by visual inspection. Finally, this tool would allow a biologist to more easily gather and share quantitative information about a large set of images, and review imaging experiments more easily.

My thesis project has been to develop such a tool. Over the past several months, I have developed an image browser capable of displaying hundreds of images and their associated metadata at the same time. The browser provides tools for the biologist to classify images and organize images by phenotype, and quickly compare images over an unlimited number of quantitative variables. It also allows a biologist to create and save freeform notes about specific images. Finally, as a component within the OME (Open Microscopy Environment) client, the browser provides access to more sophisticated image analysis tools and databases.

This document describes the feature set, design and implementation of the image browser in detail. Chapter 2 contains a tour of the application, describing all the

[3] MetaMorph, Universal Imaging Corporation. http://www.image1.com/products/metamorph.

features of the image browser's user interface. Chapter 3 summarizes the design and structure of the image browser at a source module level. Chapter 4 details how the image browser allows a biologist to better analyze large datasets in particular. Chapter 5 analyzes how effectively the browser interacts with real image datasets. Finally, Chapter 6 proposes future directions for the browser. The remainder of this section recounts how the image browser has evolved into its current form, and quickly describes the larger OME project.

1.1 Collecting user interface requirements

Before designing any modules or writing any code, I conducted an informal survey with the biologists in the OME project about what features would enable them to more effectively analyze large volumes of data. These discussions in November 2003 led to the idea of an image browser, originally intended to facilitate analysis of chemical and biological *screens*. Screening is the concurrent analysis of hundreds of molecules or hundreds to thousands of cells using an automated microscope, following exposure to some perturbation, such as treatment with a small molecule. In each screen, the microscope images an entire *plate*, which contains an array of *wells*, typically arranged in a standardized 96 or 384-well format. In this manner, a biologist can systematically test if cells react to libraries of compounds by the hundreds. Our idea was that the image browser would help the scientist by displaying the images in well order; images onscreen would appear in the same location as on the plate, thus making the dataset more familiar and accessible to the screener.

I added a number of features to the proposed browser based on discussions with other biologists, particularly Jason Swedlow at the University of Dundee, and Dan Rines at MIT. Dan & Jason had developed their own ad-hoc methods for annotating and classifying images, but wanted something much more solid, and much more convenient. They convinced me that an image browser needed to organize and group images by phenotype, and that an image browser should display and support the creation of annotations. Peter Sorger, my advisor in the Biology department at MIT, suggested the idea of a heat map to compare images quantitatively. I adopted the idea of semantic zooming, treemaps, and an image

magnifier after researching several applications that organize and display photo albums—particularly PhotoMesa[4] from the University of Maryland's Human-Computer Interaction Laboratory, and the Piccolo 2D API from the same research group.

After development began, I visited the University of Dundee in January and explained the concepts and features of the image browser to the biologists there. They confirmed that many of the proposed features would be useful, and suggested that categorizing phenotypes by color would be useful in order to compare two phenotypes at once. I agreed. After that point, the proposed feature set was frozen, and all efforts turned to development.

1.2 Project evolution/summary

The design and development of the image browser began in November. I developed a basic design for the browser component throughout December, and tested several components that the browser relies on (particularly the Piccolo[5] 2D drawing framework) throughout the month. The entire development team of the OME client met in Dundee in January, at which point we agreed on how client-side components would interact, and on additions to the interface between client applications and OME servers. At that point, I conducted extensive development and testing, particularly throughout March and April. The image browser is now in the final stages of testing; its feature set is final, and bug fixing will continue until the final release of OME 2.2, due at the end of this month.

1.3 The OME Project

The Open Microscopy Environment (OME) is an open source project with a suite of tools that analyze, annotate, share, and store imaging data for a variety of applications[6]. It is a joint effort between MIT, the Institute for Chemical & Cell Biology (ICCB) at Harvard, the Wellcome Trust Centre at the University of

---

[4] PhotoMesa, University of Maryland Human-Computer Interaction Lab (HCIL). http://www.cs.umd.edu/hcil/photomesa/.

[5] Piccolo, HCIL. http://www.cs.umd.edu/jazz/.

[6] Open Microscopy Project. http://www.openmicroscopy.org; http://docs.openmicroscopy.org.uk.

Dundee, the National Institutes of Health (NIH) in Baltimore, and the University of Wisconsin at Madison. OME's ultimate purpose is to allow biologists to perform experiments and analysis on image data they collect from microscopes, in an automated and versatile manner.

The core of OME is two data sources: the OME data server (OMEDS) and OME image server (OMEIS). OMEDS is a relational database that contains information about experiments, image sets, analysis modules, annotations, and data histories. It also contains a Perl layer that converts the relational data in the underlying database into objects that client applications can create, use and modify. OMEIS stores binary image data, and selectively serves regions of interest (ROIs) and thumbnails of multidimensional images to client applications. A Java-based client and web-based client allow users to view images and manage metadata. The image browser is a component of the Java client. An overview of the current OME architecture is shown in Figure 1.1. More information about the project and the architecture of the entire OME system can be found at http://docs.openmicroscopy.org.uk (see the Appendix for a complete list of URLs pertaining to the project). The next section begins with an overview of the Java client.

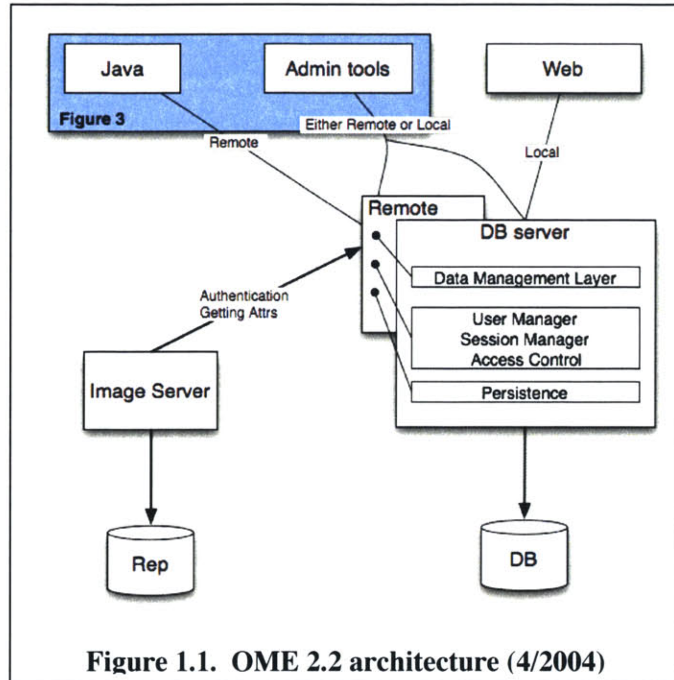


Figure 1.1. OME 2.2 architecture (4/2004)

CHAPTER 2

The User Interface

This section is a complete tour of the image browser. The browser is a component within a larger OME client, so this section will begin with an overview of the other components in that application. A discussion of the individual UI components of the image browser will follow. The remainder of the section will focus on how the user interface reflects the dataset, how a user interacts with the components of the UI, and how the dataset view displays image metadata. Keep in mind that this section discusses these components from the perspective of the user; Chapter 3 details the components from the programmer's point of view.

2.1 The OME Client

The image browser is one of several user interface components embedded within a larger application, the OME client, codenamed "Shoola." The other visual components within Shoola are the Data Manager, which allows users to view their work in a hierarchical tree; a Viewer, which allows users to view and manipulate their images at full resolution; an Annotator, which allows users to make notes about datasets and images; and information panels, which provide quick summaries about individual projects, datasets, and images.

The enclosing user interface of the OME client appears after a user logs into OME. A single window appears, containing multiple embedded windows that can be minimized, maximized, and closed independently, as shown in Figure 2.1. Those embedded windows are not children of the operating system's window

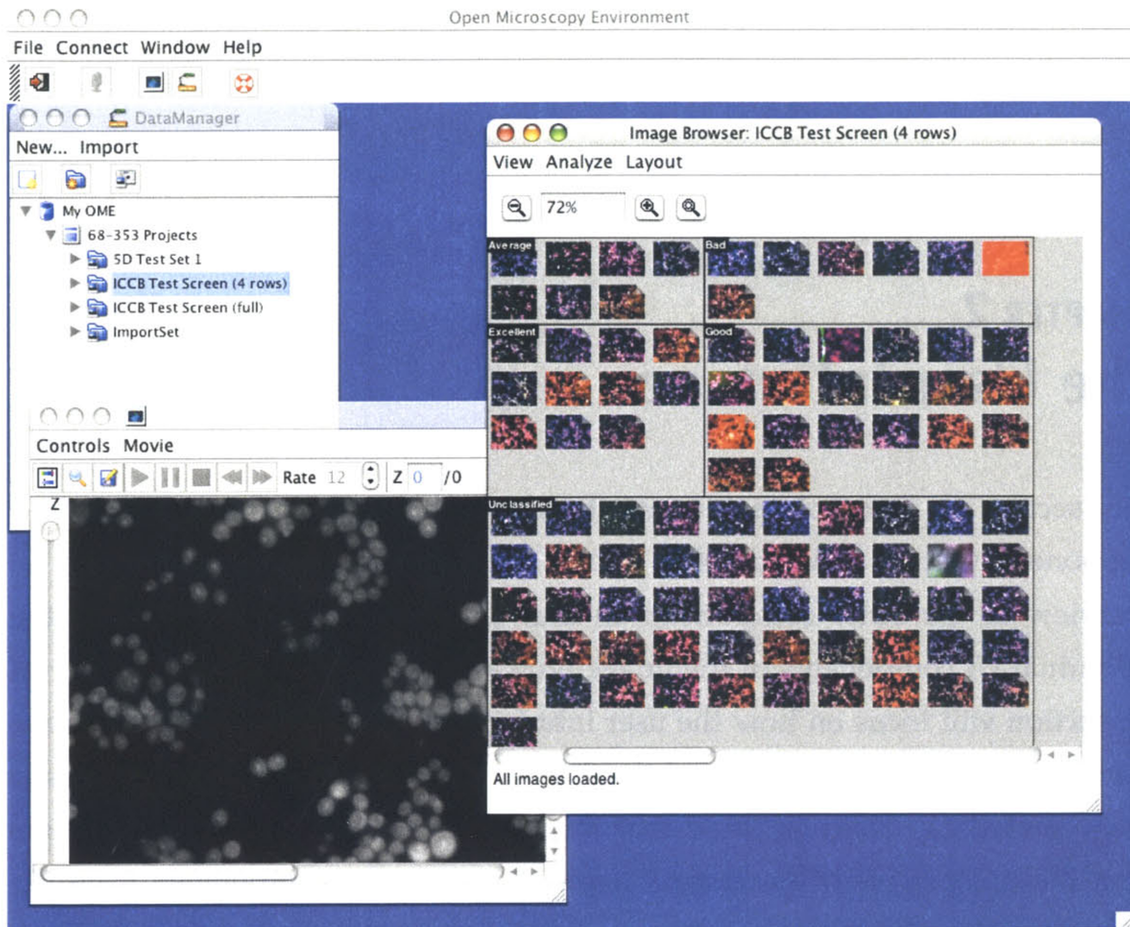


Figure 2.1. A screenshot of the entire OME client.

manager, but rather of the parent window. This configuration is commonly known as a MDI (multiple-document interface) view. Other popular applications that use MDI include America Online, and Adobe Photoshop. One advantage of MDI is its simple representation within an operating system's window manager; an MDI application only adds one child to the window manager. A user can manipulate Shoola and other MDI applications more easily in an operating system context. However, with that convenience comes some loss in flexibility. Biologists who may want to organize the image browser and multiple viewer panels across multiple screens cannot do so, as the modules are all enclosed within a single window. Whether or not Shoola remains an MDI application is a subject currently up for debate by the OME development team.

The lone child window that appears when a user loads Shoola is the Data Manager, which is the top left window in Figure 2.1. The Data Manager contains

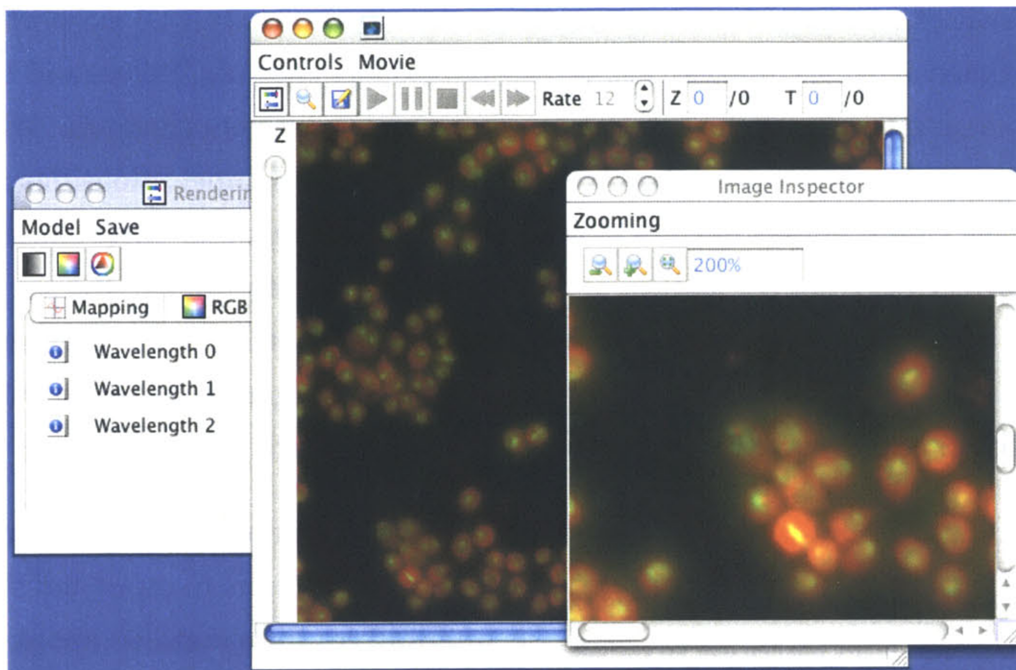


Figure 2.2. The OME Image Viewer.

a tree view of the entire project hierarchy on a remote OME server. Initially, the tree view is only displayed at project depth. However, by expanding each project, a user can see a list of datasets contained within that project. Clicking on the arrow next to that dataset will reveal the complete list of images that are members of that dataset. Double-clicking on any entry will trigger an information panel, which contains basic information about projects, datasets, and images, including image dimensions, project and dataset size, and that object's name. Right-clicking on any object within the data manager will trigger a menu with a list of actions pertaining to that resource. A user can trigger the image browser by right-clicking on a dataset and selecting "Browse" from the popup menu. Similarly, a user can also trigger the viewer by right-clicking on an image and selecting "View."

The viewer is a component that displays images stored in OME at full resolution, and offers users a wide variety of filters and analysis tools. The main purpose of the viewer, shown in Figure 2.2, is to allow the biologist to clearly isolate qualitative features within an image. Thus, it has much more UI support for image analysis and view customization than the browser, through a variety of palettes, image views, and menus. Whereas the image browser only shows a

thumbnail of an image in two dimensions, the viewer has full UI support for multi-dimensional 5D images. The viewer can display multiple Z slices and time points of an image, as well as reveal the image under different channels. While the browser displays thumbnails with a default set of channel intensities and colors, the viewer allows the user to manipulate these intensities. Most critically, the browser only shows images at a limited level of resolution—the viewer allows a user to view an image at high resolution. However, the browser is better suited for classifying images by phenotype, and organizing images within a dataset, once they have been inspected within the viewer.

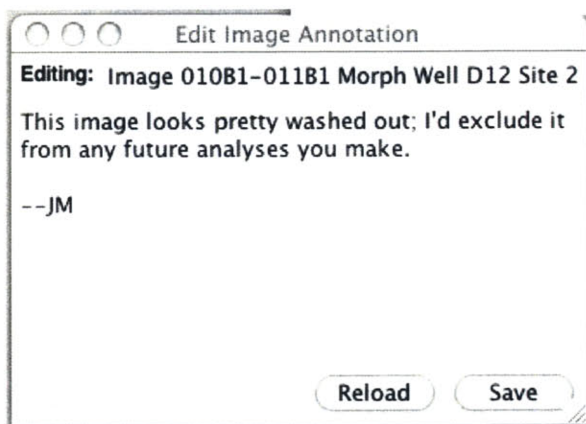


Figure 2.3. The Annotator module.

Finally, the Annotator is a component which allows users to input and store qualitative information about an image, such as notes, general observations, and even messages to other biologists. The Annotator is simple, and can be triggered by both the data manager and the browser. As shown in Figure 2.3, it contains a simple text box with the latest annotation for a particular dataset or image, and buttons to either save an updated annotation or restore the previous one. Currently, the Annotator supports the display of a single annotation per image, although multiple annotations (from different users) may be necessary in the future. However, the freeform nature of an annotation makes it easy for multiple users to collaborate and make notes about the same dataset or image.

The data manager, viewer, and parent UI were developed by Andrea Falconi and Jean-Marie Burel at the Wellcome Trust Center at the University of Dundee in Scotland. I helped the Dundee team develop the annotator. Future releases of the OME client will include graphical analysis and experiment visualization tools, developed by Harry Hochheiser at the National Institutes of Health in Baltimore.

2.2 Browser Components

The image browser contains several UI components, which are all children of Shoola's parent window. Obviously, the main component is the browser itself, which organizes and displays image information within a dataset. In addition, a heat map window allows a user to quickly identify outliers in analysis and survey quantitative image information graphically. A color map window enables a user to quickly identify the phenotypes of a particular image. There is also a phenotype editor, which allows a user to create categories and phenotypes for image classification. The remainder of this section is a brief overview of what these components look like; their function will be discussed in future sections.

2.2.1 The browser window

The browser window is the principal component of the image browser. It displays thumbnails and provides the mechanism for selecting, opening, and interacting with individual images. It contains four main components: a menu bar, a toolbar, the dataset view, and a status bar. An empty browser window is shown in Figure 2.4. The title bar always contains the name of a dataset; this allows users to identify between multiple active datasets. The Shoola UI supports multiple open browser windows, although it does not support multiple concurrent views of the same dataset.

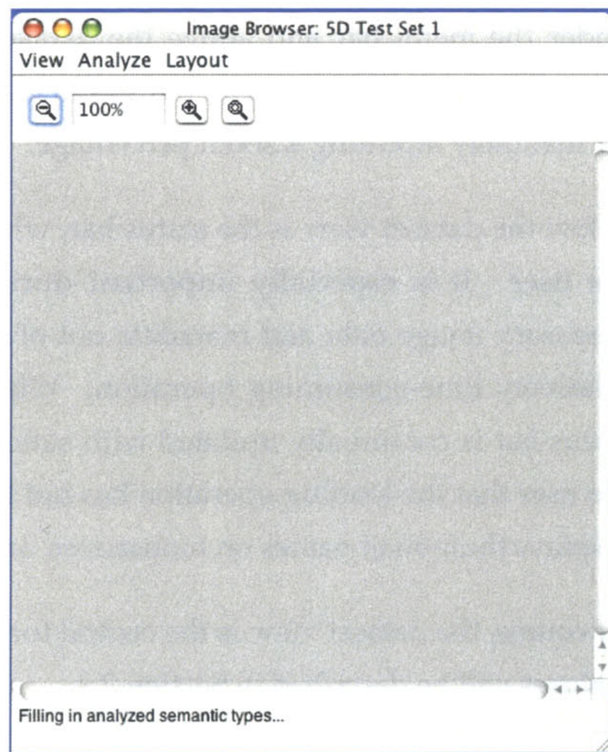


Figure 2.4. An empty browser window.

The menu bar currently contains three menus: View, Analyze, and Layout. The view menu allows users to specify what additional information they would like displayed on top of each thumbnail. Each entry in the view menu is an overlay

that can be turned on or off; its current status is indicated with a check mark. Current view options include well number (for screening datasets) and an annotation indicator. Future indicators accessible through the View menu include those for displaying image dimension, site number (for screening datasets), and image ID. The Analyze menu allows a user to display the other components in the browser—the heat map window, the color map window, and the phenotype editor. The option to turn the image magnifier on or off is also located in the Analyze menu. Finally, the Layout menu contains all possible layouts for a particular dataset. The default layout is always an option, as is an item for organization by phenotype. Possible future layouts include graph and a custom, freeform layout mode.

Users manipulate the zoom level of the browser through the toolbar, located under the menu bar and above the dataset view. The toolbar contains three buttons related to zoom: zoom out, zoom in and zoom to fit; as well as a text field for manually inputting a zoom percentage.

Below the dataset view is the status bar, which displays a variety of messages to the user. It is especially important during dataset loading. Retrieving all necessary image data and metadata out of the OME system is, unfortunately, a relatively time-consuming operation. While the browser loads that data, the status bar is continually updated with status messages. These messages inform the user that the loading operation has not timed out. In addition, the status bar displays thumbnail names on mouseover, and the status of selection operations.

Of course, the dataset view is the central focus of the browser window, and all its features will be described in Section 2.3.

2.2.2 The heat map window

The heat map window is the front end to the heat map tool, which color-codes each image based on its value for a particular variable. This provides users with a method to easily identify outliers and infer phenotypes. Figure 2.5 illustrates the window's three basic components. The first is a tree, listing the available scalar parameters to supply to the heatmap algorithm. The list of parameters is

inferred from the OME database during the browser window loading sequence, and contains every Boolean or scalar-valued semantic element for which images in the dataset have some non-null value. The second component is the heatmap legend, which contains a color band, dataset minimums and dataset maximums, and combo boxes for supplying additional parameters to the heatmap algorithm.

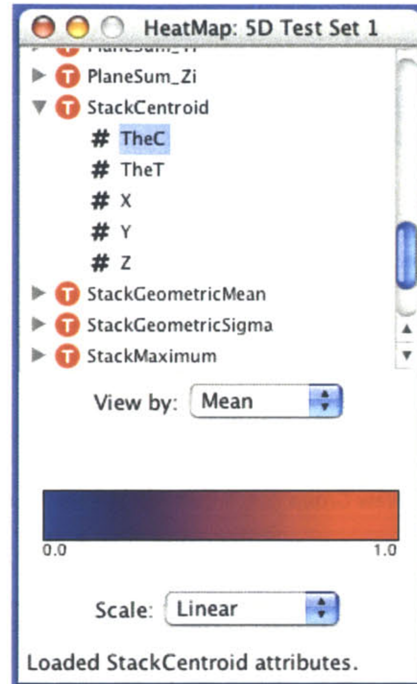


Figure 2.5. The heat map.

2.2.3 The color map window

When the color map is active, thumbnails are color-coded according to their phenotype within a certain class. The color map contains a legend of which colors correspond to which phenotypes. A single combo box controls the currently displayed class, while a list containing color boxes and phenotype names explain the relationship between color and image type. Figure 2.6 contains the color map for an Image Quality class.

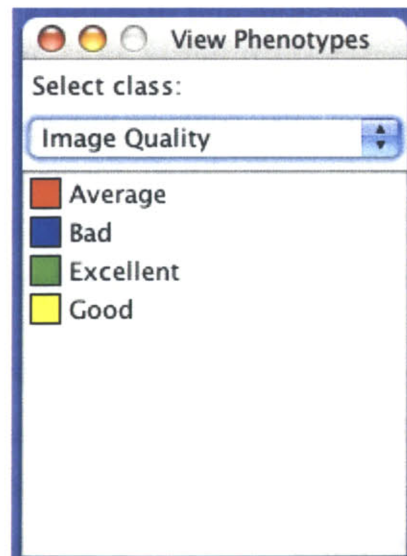


Figure 2.6. The color map.

2.2.4 The phenotype editor

The last peripheral component of the image browser is the phenotype editor, which allows users to create classes and phenotypes for images within a particular dataset. The editor is shown in Figure 2.7. The editor contains two lists: a list of classes, and a list of phenotypes for a particular class. Selecting a class in the left list will reveal and/or change the list of phenotypes in the right list. Users can create and edit both classes and phenotypes by clicking on the buttons below the lists. Doing so will trigger the class and phenotype creator, which allows a user to input the name of a phenotype or class, and an associated description. Descriptions may contain

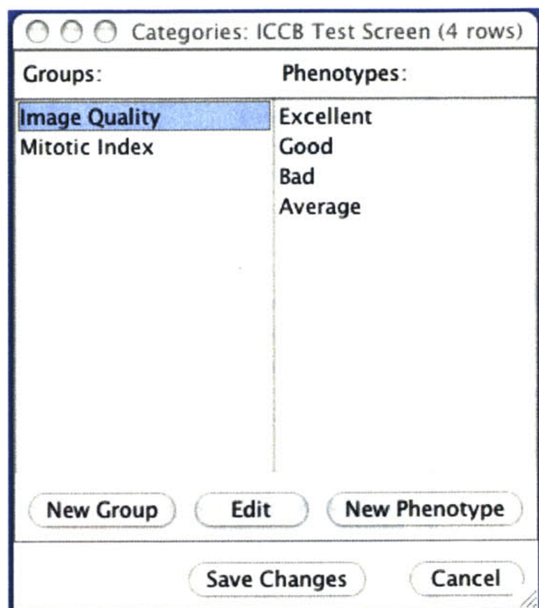


Figure 2.7. The phenotype editor.

informal criteria for classification, or information about the phenotype itself, but they are semantically equivalent to annotations—simply text to guide biologists. Phenotypes and classes are saved directly to the database, and thus, there is a small amount of lag that occurs when loading the editor, and when committing changes with the "Save Changes" button. Pressing the "Cancel" button will abort any changes made to the phenotype list.

2.3 Displaying Images

The dataset view displays all images in a particular dataset. It does so by representing and drawing a thumbnail of each image. Then, depending on the current mode, it organizes these thumbnails according to a specific layout. The dataset view conveys metadata to the user through image overlays, and allows a user to more closely inspect an individual image through an image magnifier. This section summarizes each visual component of the dataset view, and describes how biologists may interact with them.

2.3.1 Thumbnails & overlays

Thumbnails are the fundamental units of the image browser. They can represent an individual image, or in screening context, an individual well. The browser can group and rearrange thumbnails to provide contextual information to the user, and draw over the thumbnails to provide additional information about the underlying images. Aside from the background, the thumbnails are the only items the user sees within the dataset view, so they also are the basis for user interaction with the image browser.

When initialized, the image browser loads all thumbnails in a dataset from OMEIS and stores them in memory, in order to avoid making excessive calls to

the image server. The image browser retrieves the thumbnails by calling OMEIS's `GetThumbnail`[7] function; thus, OMEIS determines the size, color balance, black level, and contrast of the displayed images. OME's Importer attempts to optimize the contrast and color balance of an image for feature visibility, and the browser depends on this best-effort computation to display visually coherent thumbnails. The one limitation of this default thumbnail extraction behavior is in calculating thumbnails for multidimensional images. The default plane extracted from a multi-dimensional image is the middle Z-slice from the first timepoint in a 5D image. Unfortunately, this may not be the most meaningful plane in the entire image stack. However, using the Viewer, a user may correct this by marking a particular plane (and a channel configuration) as the default. It should be noted that the `GetThumbnail` function supports multiple parameters, such that the browser can retrieve different planes or larger thumbnails than the default. However, extracting images with this method is very time-consuming, which is detrimental when opening large datasets.

A user can interact with a thumbnail in several ways. Moving the mouse over a thumbnail will either trigger the magnifier (see Sections 2.3.4 and 4.1.1) or display the name of the image in the status bar. Right-clicking on a thumbnail will bring up a popup menu, which lists further options for viewing, classifying and manipulating the underlying image. A user may select multiple thumbnails in the same method that a user in graphical file manager would select multiple files. In this manner, a user can classify multiple images at once. The browser currently lacks support for dragging the thumbnails within the dataset view. While this feature would be appropriate for quickly classifying images in treemap layout mode, and for allowing a user to organize a dataset to his or her liking, I did not consider drag support critical, especially with the other analysis and classification tools contained in the browser.

Finally, the dataset view makes extensive use of overlays, graphical cues drawn atop thumbnails, to display image metadata. These graphical cues can be zoom-

---

[7] The complete OMEIS API can be found at http://docs.openmicroscopy.org.uk/api/omeis/index.html.

dependent or static, can take the form of icons, shaded rectangles, or text, and can even respond to user input. For example, selecting "Well number" in the View menu (for a screening dataset) will display the well number in the upper-left corner of each thumbnail. Overlays can be zoom-sensitive; certain overlays will only appear or react to user input if the zoom level is above a certain threshold. The color map, heat map and annotation discussion will reveal more overlays and how they can rapidly provide the user with quantitative and qualitative information about a dataset.

2.3.2 Layouts

The image browser can currently organize thumbnails within a dataset in four different ways, using four different `LayoutMethod` subclasses: by a maximum pixel width layout method (`MaxWidthLayoutMethod`); a maximum-number column layout method (`NumColsLayoutMethod`); in the configuration of a chemical screen (`PlateLayoutMethod`), or by using treemaps to group thumbnails by phenotype (`QuantumGroupLayoutMethod`). The image browser determines a default layout method for each dataset by verifying whether or not a dataset represents a chemical screen. If the dataset contains mages of a chemical screen, images within a dataset will have associated `ImagePlate` attributes in the OME database; non-screen images will lack those attributes. If the image browser finds these attributes, it will order the images in a plate layout configuration; otherwise, the thumbnails will be organized in an n-by-8 grid, where n is some number of rows.

The plate layout method, shown in Figure 2.8, attempts to simulate the layout of images within a chemical screen as accurately as possible. By keeping track of well addresses in the dataset, the browser determines the number of rows and columns that are required to recreate the screen, and then assigns a location for each image thumbnail based on its well letter and number.

There are a number of subtleties in processing screening datasets. First, the wells likely do not have a one-to-one correspondence with the images in the dataset. More often than not, there are multiple images for a particular well, taken at

Figure 2.8. The image browser, displaying an entire 384-well screen.

different sites within the well by the scanning microscope. Thus, a single thumbnail may represent multiple images. A small triangle-shaped overlay that gives a thumbnail a "folded" look notifies the user that a well has multiple images. The browser only displays one image per well at a time. A user can change which image to display by using the image magnifier to select a different site. Finally, the well number overlay is only available for screening datasets. It is not shown by default; a user must select "Well Numbers" from the View menu to see the well number of each image.

The other major layout method, the quantum grouping layout method, is more complicated, and merits its own section.

2.3.3 Quantum Treemaps & Organization by Phenotype

The default method to organize thumbnails by a phenotype is by employing a *quantum treemap*. The benefits of quantum treemaps are the subject of section 4.3.

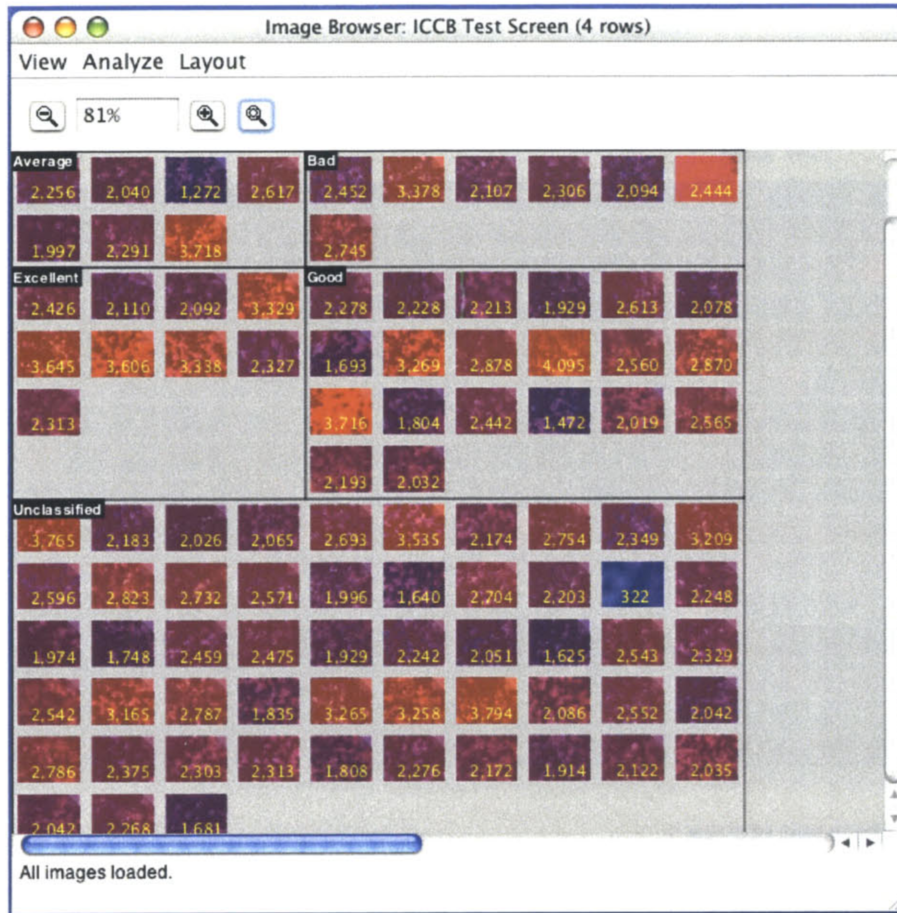


Figure 2.9. A screenshot of quantum treemap (and heatmap) mode.

What is important to mention now is that the quantum treemap algorithm simply divides a rectangular space into regions, into which visual objects can be placed. The sizes of the regions are roughly proportional to the number of objects that they contain. In the browser context, the algorithm accepts sets of thumbnails—each set containing thumbnails of like phenotype—and divides the browser window into regions of like phenotype. Selecting the "Arrange by Phenotype" menu item from the Layout menu will yield a treemap layout similar to that shown in Figure 2.9. The menu item contains a list of classes by which the browser can group thumbnails. Selecting one of the classes will initiate the quantum treemap algorithm, over sets of images with like phenotype within that class. When the algorithm completes, thumbnails of like phenotype appear grouped together. A tab in each region informs the user of the phenotype of that region. Unclassified images appear in the "Unclassified" region. These are the same images that are not colored when the color map is active.

Quantum treemap layout provides an additional and useful UI benefit—allowing a user to see the quantitative variance of images within a phenotype, and allowing users to see information about two classes of phenotypes at once. For example, given the two classes "Image Quality" and "Mitotic Arrest," a user can organize thumbnails into various phenotypes in the Mitotic Arrest class with a treemap, and identify good examples of images within a particular phenotype by selecting the Image Quality class in the color map. As quantitative information sometimes determines the phenotype of an image, using the heatmap in treemap layout mode can allow the biologist to identify misclassified images, pick out outliers out of a certain phenotype, and visualize the correlation of other scalar variables to phenotype. See Section 2.4 for more information about the heat map, and Section 4 about how the combination of treemap and layout views is especially valuable in analyzing large datasets.

2.3.4 The image magnifier

The image magnifier is a tool that allows a user to quickly investigate an image within the dataset without having to open the viewer. It provides a higher resolution view of a particular thumbnail, and displays information about the image that is normally hidden in the dataset view. Furthermore, it allows a user to quickly and easily perform certain operations on a thumbnail, such as annotation, property editing, classification, and opening within the viewer.

The default action when a user moves the mouse cursor over a thumbnail is for the image magnifier to appear over that thumbnail. An example of this is shown in Figure 2.10. In order to prevent the magnifier from appearing unintentionally, the browser waits 500 milliseconds to prompt the magnifier. If the user's cursor has not left the boundaries of the dataset view, and has not left the boundaries of a single thumbnail, the magnifier will appear.

The magnifier in Figure 2.10 is an example of a magnified image within a screening dataset. Several bits of information that do not normally appear in the dataset view (due to space constraints) appear in the magnifier. For example, the well number is prominently displayed in the upper-left hand corner of the magnifier. In addition, the magnifier displays the total number of images in a well and the currently selected image within the well. There are conceivably additional applications and features for the magnifier. When more analysis code is merged into the Shoola client, the magnifier could display image features, and display the dimensions of an image. This practice of displaying additional metadata at higher resolutions is called *semantic zooming*, and there are additional examples of this technique elsewhere in the browser (also see Section 4.1 on why semantic zooming is beneficial).

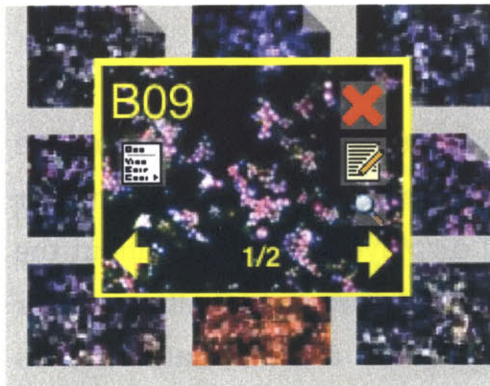


Figure 2.10. The image magnifier.

The icons in the magnifier, laid atop the image, provide the same functionality as right-clicking on an image. The icon on the left side triggers the popup menu for a particular thumbnail; a user can classify the currently zoomed image in this manner. The paper icon on the right side triggers the image annotator. Below the annotation icon is a magnifying glass icon. Clicking on this icon instructs the browser to load the zoomed image in the image viewer. Finally, the red X in the upper right hand corner closes the magnifier; this may be necessary to view neighboring images obscured by the magnifier window. Unlike thumbnails, the magnifier is not subject to overlays generated by the heat map and color map, so it is useful for displaying the original image when a thumbnail is color-coded.

2.4 The heat map & scalar variable display

The heat map component allows a user to quickly compare images quantitatively, using color to indicate the value of a particular variable in an image. If an image is colored blue, its value for a particular scalar is low

compared to the rest of the images in the dataset. If an image is colored red, its value for a particular variable is relatively high. In addition, the heat map has a Boolean mode, which colors images based on whether an image meets a certain criterion.

Comparing the relative values of a variable for all images is a useful tool in biological image analysis, especially for chemical screening. For example, one common variable in analyzing biological image is maximum image intensity. Proteins stained with GFP will fluoresce if they are in high levels. In this case, images with higher maximum intensities (in the GFP emission wavelength) likely indicate higher protein levels, which is useful for analyzing and isolating cell processes. Using color to show quantitative differences between images allows biologists to quickly survey a dataset and determine an image's relative value. This is not a new technique; everything from weather maps to news magazines to instant messenger clients use color to indicate quantitative difference. However, it is invaluable in analyzing large image datasets.

A user opens the heat map by selecting the "HeatMap" item from the Analyze menu. The heat map initially displays a hierarchy of all semantic types with a scalar or Boolean element that is *image-granular*; that is, types that can have unique values for every image. A semantic element and enclosing semantic type will only be included in the heat map's tree (for a particular dataset) if at least one of the images in the dataset has a non-null value for that element. For example, all images have a Pixels attribute, and have non-null values for the SizeX and SizeY elements; thus, the heat map places the Pixels type in the tree. Because the image browser must rely heavily on the database for information about available semantic types and their applicability to the dataset, building the heat map for a particular dataset takes significant processing time. Thus, to make user interaction with the heat map as smooth as possible, this analysis occurs during the initial dataset load.

There are three types of objects displayed in the tree: Types, symbolized by a red icon with a "T", scalars, represented by a pound sign icon, and Boolean values, represented by a yin-yang icon. The heat map does not classify and assign colors

to string or object values; this is better suited for the treemap layout and for the color map classifier. Expanding a type reveals more subtypes, or scalar or Boolean elements. Clicking on a scalar element initiates heat map scalar mode, whereas clicking on a Boolean element initiates heat map Boolean mode. As these two modes are sufficiently different, I describe them in their own subsections.

2.4.1 HeatMap scalar mode

By clicking on any scalar element within the heat map's hierarchical tree, a user initiates scalar mode. This sets off a chain of processing, which can take several seconds, depending on the size of the dataset. First, the image browser must contact the OME data server (OMEDS) to retrieve all attributes of the element's enclosing semantic type, if it has not done so already (the browser will cache all instances of a type once they are loaded). Once all the elements are retrieved from the data server's database, the heat map agent analyzes the values of each image, extracting the minimum, mean, and maximum values across the dataset. This analysis sets the range of possible data values—the "cold," or blue value, is the minimum scalar value for all images in the dataset, and the "hot," or red value, is the maximum for the dataset. Finally, for each image, the heat map determines a color based on either linear or logarithmic interpolation between the "cold" and "hot" color. The heat map finally maps each thumbnail to a particular color, and for each thumbnail, instructs the browser to draw a semi-transparent rectangle overlay of that color. The end result, as earlier shown in Figure 2.9, is that each image has a relative "temperature," which is apparent by visual inspection.

The heat map offers several options to fine-tune coloring and handling multiple attributes per image. First, there are different scales a user can use to color a dataset. Different scales control how the heat map interpolates color for each image. The default scale is linear. Using a linear scale, an image that has a value that is exactly the average of the entire dataset will be colored purple—the average color between red and blue. A logarithmic scale is also available. Using the logarithmic scale, color is assigned by comparing the log of the value of a

particular image to the log of the minimum and the log of the maximum. Color is interpolated based on how close the log value is to either extreme. The logarithmic scale is useful for comparing images whose values for a particular variable may differ by orders of magnitude. A standard deviation/mean scale will be included in a feature release of the image browser.

It is often possible, even likely, that multiple instances of a single semantic type will correspond to a particular image. For example, the number of StackMax attributes (the maximum intensity computed in three dimensions) for a multidimensional image, is proportional to the number of timeslices and different channels in an image. The heat map, however, must assign a single color to an image, regardless of this many-to-one relationship. Currently, the heat map combines these attributes in a manner determined by the user. The default is to use the elemental mean of the attributes. For example, when determining the value to use for displaying the StackMax.Maximum for a multidimensional image, the heat map will take the average of the Maximum element of all associated StackMax attributes. However, using a combo box in the heat map, a user can also elect to use the minimum value, median value, or maximum value as the value assigned to a thumbnail. More sophistication may be necessary for multidimensional images, but this strategy works well for three-dimensional (X, Y, channel) screening images.

Finally, enabling heat map scalar mode will activate an overlay that draws the numerical value of a scalar atop a thumbnail. Because the image browser prints this in a small font, the overlay is only visible at 75% zoom and above. This is another example of *semantic zooming*; the scalar value only appears once a user has zoomed in past a certain threshold. In contrast, the color overlays assigned by the heat map are not semantic; the image browser draws them at all resolutions and levels of zoom.

2.4.2 Heat map Boolean mode

Boolean mode is similar to scalar mode, but displays different information. The method of obtaining relevant attributes is the same as in scalar mode, as the image browser must contact the OMEDS to retrieve information about each image. However, once this occurs, processing is complete. If the value of a Boolean variable for a particular image is true, the image is colored green. If it is false, the image browser colors it red. Figure 2.11 shows a dataset colored in Boolean mode. Unlike scalar mode, there are no scale parameters, as Boolean classification is binary and discrete. The lone complication is when there are multiple instances of a semantic type for a particular image. Currently, if the 'true' values outweigh the 'false' values, the image is marked as "true" and vice versa. It is possible, and may be more desirable, to provide a color blend, or a true/false count; however, this is not currently implemented in the system. It should be noted that using red and green to represent false and true is a strategy adopted from JUnit, the popular Java testing facility, although JUnit may have borrowed that color scheme from traffic lights. In any case, it is recognizable and distinctive enough to suit the purpose of Boolean mode.

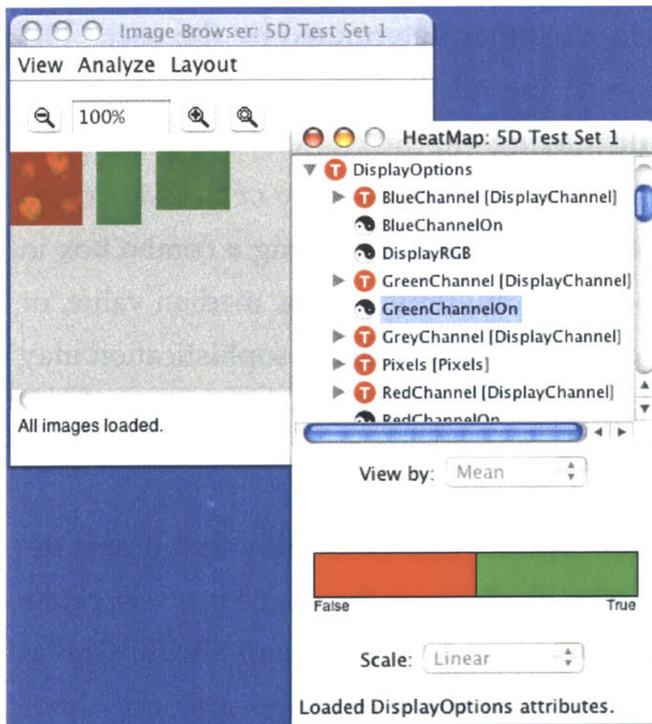


Figure 2.11. Heat map Boolean mode.

2.5 Image Classification

The image browser uses three components for image classification: the phenotype editor, the color map, and the thumbnail popup menu. As described in Section 2.2.4, the phenotype editor allows a user to create classes and phenotypes. The color map provides a legend linking colors and phenotypes

within a dataset. The final piece of the puzzle is the thumbnail popup menu, which allows a user to assign phenotypes to a particular image. This is a very useful feature because the vast majority of biological image data is hand-annotated.

To activate the thumbnail popup menu, a user must either right-click on a thumbnail (or group of thumbnails), or click on the menu icon of the magnifier. The popup menu, shown on Figure 2.12, contains several items. To classify an image, a user must click on "Categorize." Clicking on "Categorize" displays the list of phenotype classes that belong to a dataset. Clicking a group pops up the available phenotypes for that class, and clicking on a phenotype classifies the selected image (or images) as that particular type. The classification occurs in the database immediately.

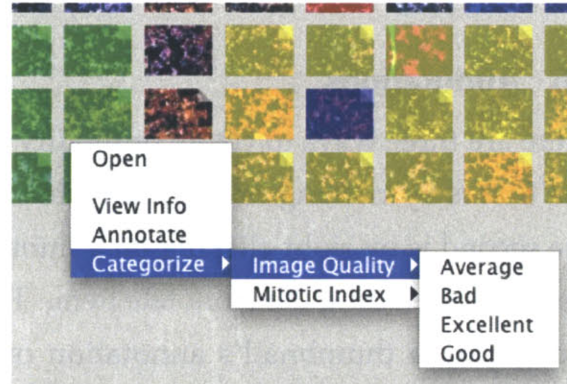


Figure 2.12. Classifying an image.

Both the color map and quantum treemap layout rely on these classifications. Section 2.3.3 discusses treemap layout has already been discussed in detail; this section focuses on the color map. A user triggers the color map by selecting the "View Phenotypes" item in a browser's Analyze menu. The color map then appears above the browser. A user must then select a class to view. When a class is selected, the color map agent displays a mapping between colors and phenotypes in its color list, and then instructs the image browser to color-code the thumbnails in accordance with that mapping. A user can change the phenotype of an image when the color map is active; this change will be reflected in the thumbnail. The color map explicitly supports 32 different phenotypes per class; it will assign distinctly different colors for the first 32 phenotypes. After 32, the color map will issue a random color for each subsequent phenotype.

Changes to phenotype classes and phenotypes themselves made using the phenotype editor will be appear immediately in the other image classification

tools. If quantum treemap mode is active, updated phenotypes will appear renamed. Additional colors will appear in the color map if a user adds additional phenotypes. The only exception is that a user will need to manually reorganize a quantum treemap using the Layout menu to reflect changes in a phenotype class.

2.6 Annotations

There are three ways to create or modify an annotation with the image browser. The first is by clicking on the annotate icon (below the red X) in the magnifier. The second is by right-clicking on a thumbnail, triggering the thumbnail popup menu, and selecting the Annotate item. The third, illustrated in Figure 2.13, is by clicking on a thumbnail's annotation overlay. Each method will trigger the Annotator module, which allows a user to create or modify annotations.

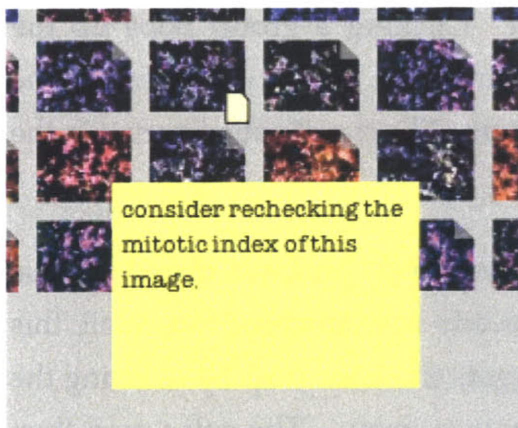


Figure 2.13. An image annotation.

A user can determine which thumbnails have annotations by selecting the Annotations item from the View menu. This prompts the image browser to draw a small icon overlay in the lower right corner of each thumbnail that has an associated annotation. The overlay itself is another example of semantic zooming; it is drawn only at 50% zoom or higher. However, unlike other overlays discussed thus far, the annotation overlay is an active *overlay node*, and responds to user input. Moving the mouse over the annotation icon will pop up a text box that looks like a "Post-It Note," which displays the first few lines of the associated annotation. Clicking on the icon or the dynamic text box will trigger the Annotator, and allow the user to modify that annotation. These active overlays allow the user to quickly view and subsequently edit annotations, without excessively cluttering the dataset view.

CHAPTER 3

Design & implementation

The previous chapter described *what* the image browser displays to a user. This chapter describes *how* the browser component works, at the system level. The first half describes the modules in the image browser from the top down, working from the top-level application classes to the browser components down to thumbnails and thumbnail action classes. I will cover several patterns used throughout the browser application in detail, including behavior factories, and top-down event handling. This section concludes with a thorough summary of how the browser modules work together to load a dataset.

3.1 Browser overview

Like the rest of the OME client, the image browser is written in Java. The browser itself is composed of more than 140 separate classes; the total number of classes that make up the client is about 350. The classes in the browser fall into several groups: a set of application-wide classes which serve as the glue between the rest of the OME client and the browser, a collection of browser UI classes, user event handlers and behavior factories, layout managers, overlay drawing methods, image metadata storage classes, and packages for the heat map and color map widgets. Figure 3.1 shows a schematic of the browser class hierarchy.

The browser relies on three external frameworks for the user interface and for communicating with the other major OME components. For displaying UI widgets, the OME Java client uses Swing, the standard user interface toolkit for Java. The dataset view uses Piccolo, an open-source 2D graphics toolkit built atop Java2D, authored by the HCIL at the University of Maryland.

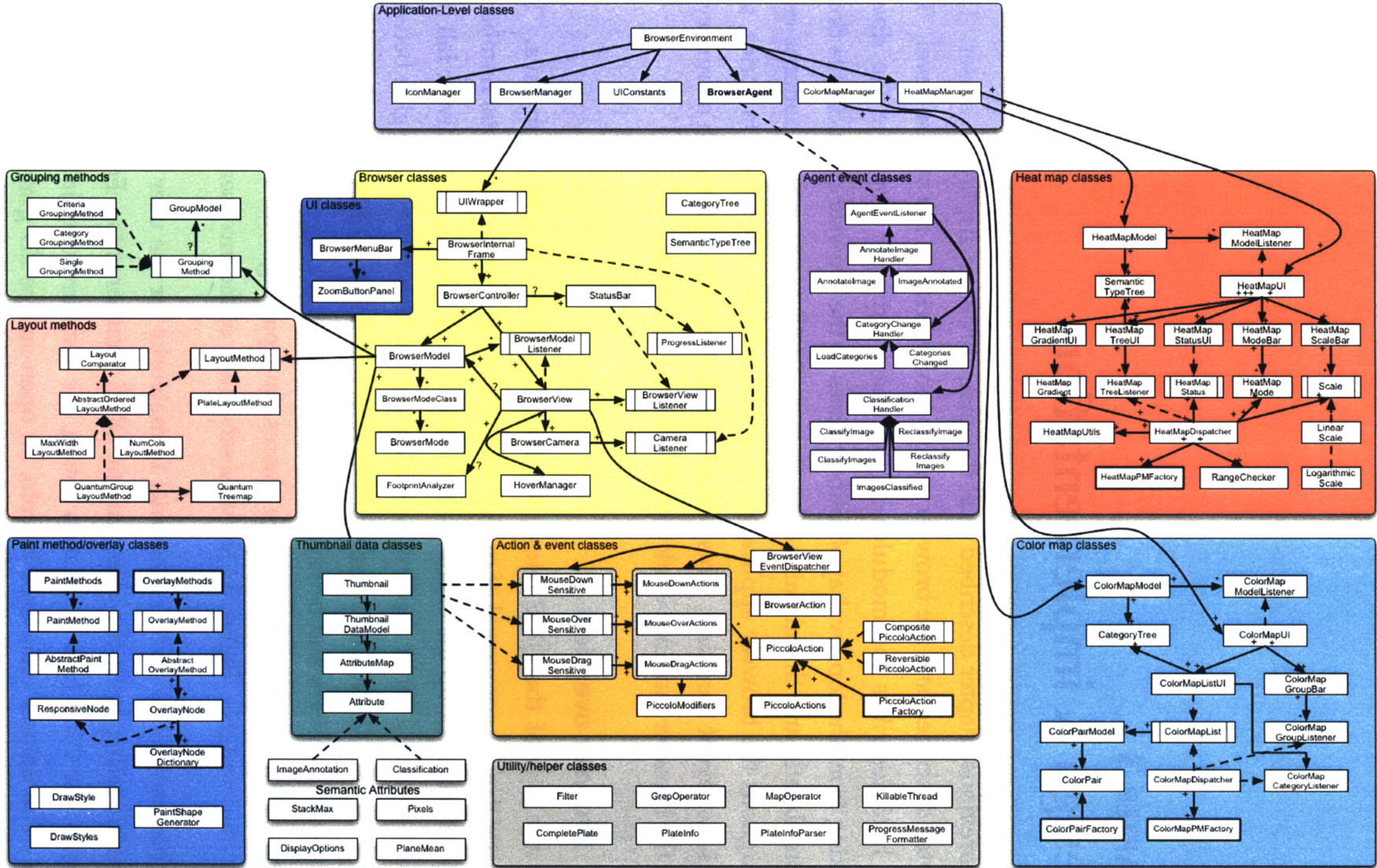


Figure 3.1. The class hierarchy of the image browser. For clarity, all usage relations are shown.

Finally, to extract image data and metadata from OMEDS and OMEIS, the image browser relies on the OME remote framework, a Java layer that abstracts XML-RPC communication with the data server, manages object persistence, and regulates data transport.

The browser's use of Swing is straightforward. The parent UI widget of the entire client is a `JDesktopPane`[8], and agents within the enclosing application are `JInternal-Frames`. This is the normal way to build MDI (multiple-document interface) applications in Java (see to Section 2.1 for a more detailed discussion on the benefits and drawbacks of MDI). The image browser, heat map, and color map components are all `JInternalFrames`. Each frame contains a various configuration of buttons and drop-down boxes, and is fairly straightforward, using the only default UI widgets provided by Swing.

The dataset view uses the Piccolo graphics framework for drawing overlays and rendering images. I chose Piccolo instead of standalone Java2D for several reasons. First, when displaying large datasets, performance is of the utmost importance. Piccolo is optimized for handling scenes with high primitive counts, drawings with many objects, and uses Java 1.4's drawing facilities wherever possible, which enable a system to employ hardware acceleration where needed. More importantly, Piccolo makes several common tasks easier for the programmer, such as zooming, panning, mouse event handling and organizing many graphics primitives into a hierarchy.

In Piccolo, every drawable object is a node—an instance of the `PNode`[9] class. Nodes can have children and parents, and are viewed though a camera, represented by the `PCamera` class. Piccolo includes separate nodes for simple shapes, images, and text. Nodes can be added to a viewable layer, and can respond to mouse events, such as clicking, dragging, and hovering. The camera controls the zoom level and center of projection; that is, which point in the

---

[8] The complete Java API can be found at http://java.sun.com/api. The specifications for each Swing component can also be found at that site.

[9] The Piccolo API, including `PNode`, `PCamera`, `PPickPath`, and `PImage` classes, can be found at http://www.cs.umd.edu/hcil/piccolo/download/piccolo/api/.

drawing space corresponds to the center pixel within the browser viewport. Piccolo makes a 2D programmer's life much easier by abstracting the acts of panning and controlling the zoom level to methods in the `PCamera` class. In addition, the Piccolo canvas facilitates event handling by constructing directed acyclic graphs (DAGs) in response to user input, such as a mouse click. These graphs, called `PPickPaths`, contain a list of all possible `PNodes` that could have captured that event. The browser takes advantage of this data structure to streamline event handling, as described in Section 3.5.

Finally, to retrieve information from OMEIS and OMEDS, the image browser interacts with the remote framework, a collection of classes that abstract database calls and URL constructions to the image server. The Java remote framework bundles database calls in XML-RPC messages, sends them to the active data server, unmarshals the response into Java objects, and forwards those objects back to clients. In addition to communication and transport classes, the remote framework contains Java representations of OME core types (such as a Project, Image, Dataset or Feature) as well as semantic types and attributes. The image browser makes extensive use of attributes to display metadata, and must query the remote framework frequently in order to retrieve metadata from and store metadata to OMEDS.

3.2 The Browser Agent and application-level classes

The OME client is made up of several independent modules called *agents*. The data manager, viewer, and browser are all examples of agents. Agents send and receive messages to and from other agents, communicate with the data layer to retrieve metadata from OMEDS and pixel data from OMEIS, and allow modules to have common access to filesystem and application-wide resources, such as icons, configuration information, and a unified error-handling framework.

The browser agent (`BrowserAgent.java`)[10] plays several roles. First, it receives messages from the data manager, and sends messages to the Annotator,

[10] The complete OME client reference API, including documentation of all browser class & method references, can be found at http://sorger-g51.mit.edu:8009/shoola. The source code is available at

Classifier and Viewer agents. It is also the one module in the browser application that has access to the data layer; thus, all requests for image data or metadata must be routed through the `BrowserAgent`. It must also provide access to resources and information that the browser needs, such as zoom icon buttons, and the browser configuration file on disk. Finally, and most importantly, it is in charge of loading all thumbnails and immediately relevant image metadata from the remote servers when a dataset is first opened.

3.2.1 The BrowserAgent within the OME client

By implementing the OME client's `Agent` interface, the `BrowserAgent` class serves as the conduit between the internal modules of the image browser and the external resources and components of the OME client. The `Agent` interface contains setup and teardown methods, as well as the `setContext` method, which provides the browser agent with access to resources shared by the entire Shoola application. This access comes in the form of a Registry object, the lone parameter to the `setContext` method. The registry contains references to the remote data service, remote image service, a semantic type resolver, an icon manager, an error notification module, and a reference to the enclosing UI component (`TopFrame`) of the entire Shoola application. Each agent is supplied with a registry when the OME client application is first initialized; the *agent container* (a module that initializes each agent at application startup) ensures this.

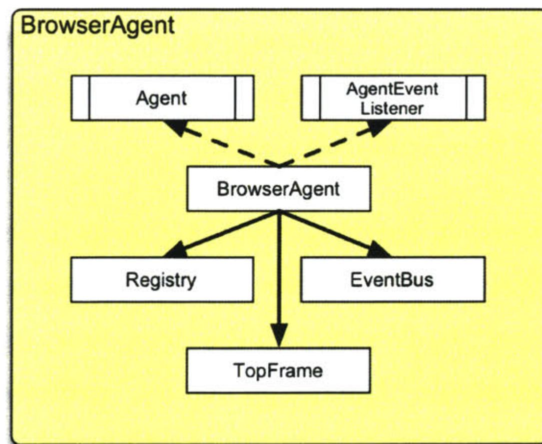


Figure 3.2. The BrowserAgent hierarchy.

The `BrowserAgent` also implements the `AgentEventListener` interface, which allows it to listen to messages from other agents. How this is done is the subject of the next subsection.

---

http://cvs.openmicroscopy.org.uk. Follow instructions on how to retrieve the latest CVS snapshot of the code; there is also a web view where users may inspect the most recent source code.

3.2.2 Message handling

One of the modules contained in the registry is the event bus (`EventBus.java`). The event bus is a message queue that all agents in the application share, and is the mechanism by which one agent can talk to another. To receive messages from other agents, an agent must selectively subscribe to a certain class of messages, using the event bus's `register()` method. This method takes an `AgentEventListener` (the browser agent is one) and a message class as parameters. This instructs the event bus to forward any messages of that class in the queue to that agent.

The browser agent listens for (and registers with the event bus for) four types of event-driven messages: `LoadDataset`, `ImageAnnotated`, `Images-Classified`, and `CategoriesChanged`. Each message type is a separate class, whose instances contain the parameters of the message. Whenever another component generates a message of those types, the browser agent will react to it. For example, consider when a user clicks on the "Browse" popup menu entry in the Data Manager. Clicking on "Browse" instructs the data manager to post a `LoadDataset` message to the event bus. As the BrowserAgent has subscribed to listen to those messages, it receives it, extracts the ID of the dataset to load, and then loads the dataset.

Likewise, the image browser posts messages (of different types) to the event bus. When a user clicks on the image open icon, the browser agent posts a `LoadImage` event to the event bus, embedding the image ID in the event as a parameter. The image viewer, configured to listen for these messages, responds by triggering the viewer window and loading the specified image.

In some cases, the browser agent might require a response to a particular message. For example, consider the case when a user clicks on the annotate icon in the magnifier. The browser agent posts an `AnnotateImage` message to the event bus, which forwards it to the Annotator agent. The annotator then, of course, pops up its editing box and allows a user to create or change the annotation. However, the browser needs to know when editing is complete, in

order to display an annotation icon and the new annotation itself over the target thumbnail. Any agent that requires such a response can attach a *handler* to a message, and listen for a response. For example, prior to posting the `AnnotateImage` message, a browser agent attaches a `AnnotateImage-Handler` object, which contains the code that should be executed when database processing is completed (in this case, the code updates annotation overlays). When the Annotator finishes updating the OME database with new annotations, it wraps the `AnnotateImage` object in an `Image-Annotated` response, and posts it to the event bus. When the browser agent receives the response, it executes the response's `complete()` method, which executes the original handler code. In this manner, agents can respond and post messages to each other asynchronously. This request handler-response mechanism is an established design pattern, known as an *asynchronous communication token* pattern, or *ACT*. The browser agent uses ACTs when creating new annotations, assigning new phenotypes, and editing phenotype classes, although not when launching the viewer.

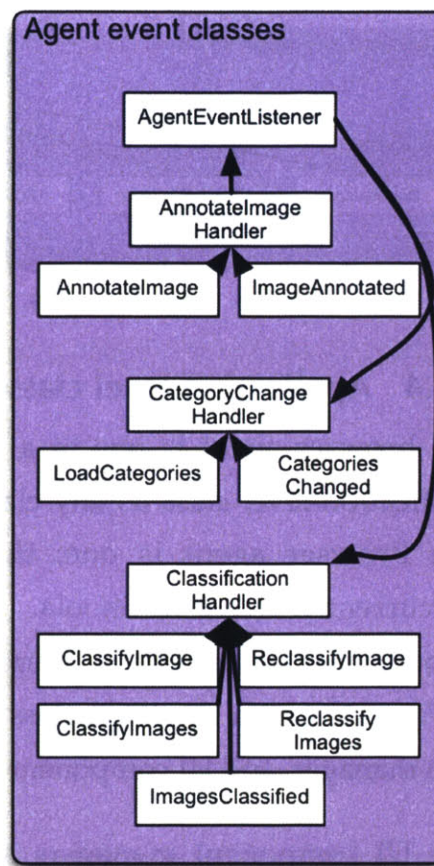


Figure 3.3. Agent event classes.

3.2.3 Loading a dataset

Because the browser agent is the only component within in the image browser to have access to the Registry, and thus the only component to have access to OMEDS and OMEIS, it is the only component capable of loading a dataset. Loading a dataset is a complicated operation, and discussing it in detail here will sideline the discussion of the image browser's design. The process of loading a dataset is covered fully in Section 3.7.

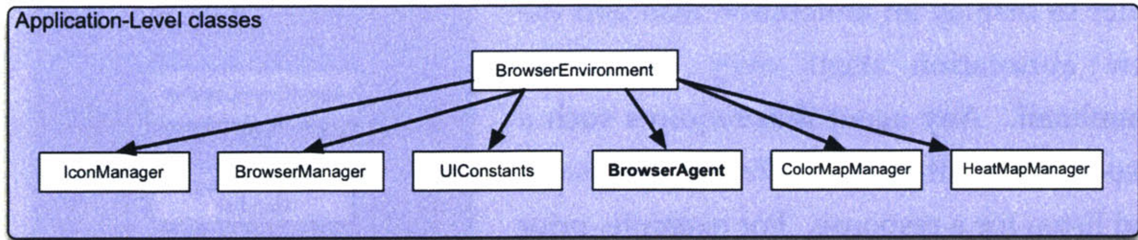


Figure 3.4. The application-level classes.

3.2.4 Application-level classes

The browser agent is one in a set of *application-level classes*. I define an application-level class as any class that should only have one active instance. The browser agent is one; there shouldn't be multiple browser agents concurrently active in Shoola. Every application-level module is accessible through the `BrowserEnvironment` class. The `BrowserEnvironment`, shown in Figure 3.4, is a singleton class that contains references to the browser agent, icon manager, and UI component managers.

The UI component managers are application-level classes that control the visibility of the browser's user interface widgets. The `BrowserManager` class maintains a list of active browser windows. The `HeatMapManager` keeps track of a heat map's state, as well as all the heat map models for each browser window. Similarly, the `ColorMapManager` maps browser windows to color map models, and keeps track of the color map's visibility.

Any module in the browser application that needs to query the UI component managers, retrieve information from the BrowserAgent, or send messages to other components within the OME client must acquire an instance of the `BrowserEnvironment` class. This is fairly easy to do; a module simply needs to call the environment's `getInstance()` method. That module can access the browser agent and component managers by then calling accessor methods on the environment instance.

3.3 Core browser classes

A browser window is a visual representation of a dataset. A dataset, in OME terms, is the parent object of any number of images, and a child of any number of

projects. Each browser window displays information about a specific dataset, visualizes the set of images that belong to that dataset, and allows a user to edit and create metadata about the dataset and its member images.

Whereas there can be only one application-level class, the application supports multiple browsers. A user can view multiple browser windows open at the same time, as long as each browser window displays a different dataset. For each browser window, there is one set of core browser objects, shown in Figure 3.5. Most prevalent is the browser's viewer, which displays thumbnails and overlays, and responds to user input. The viewer is a façade atop a browser model, which contains a browser's state, as well as references to the dataset. There is a camera class for each viewer that manages its perspective and draws viewer-level overlays. Finally, there are several classes that manage event handling and the viewer's integration within the Swing UI.

3.3.1 BrowserModel

The browser model (`BrowserModel.java`) maintains the state of a browser window and stores all the information that the viewer requires for display. It is the model component of the model-view architecture of each browser window. Other classes use the browser model to query and modify a browser's state and manage a dataset's thumbnails; the viewer class uses the model to determine how to display them.

The `BrowserModel`'s primary function is to store a dataset's thumbnails. In order to do so, it maintains a `Set` of thumbnails, and provides accessor methods for both adding and removing thumbnails. When a dataset is loaded, the browser agent adds a set of preloaded thumbnails to the browser model using these methods. In addition to storing the thumbnails themselves, the browser model maintains a list of selected thumbnails, and exposes mutator methods that mark thumbnails as selected or deselected.

The browser model also contains information about the dataset it represents. Of course, it contains the basic attributes of a dataset within OME, such as its unique ID, name, and owner. However, the browser model also stores a variety of

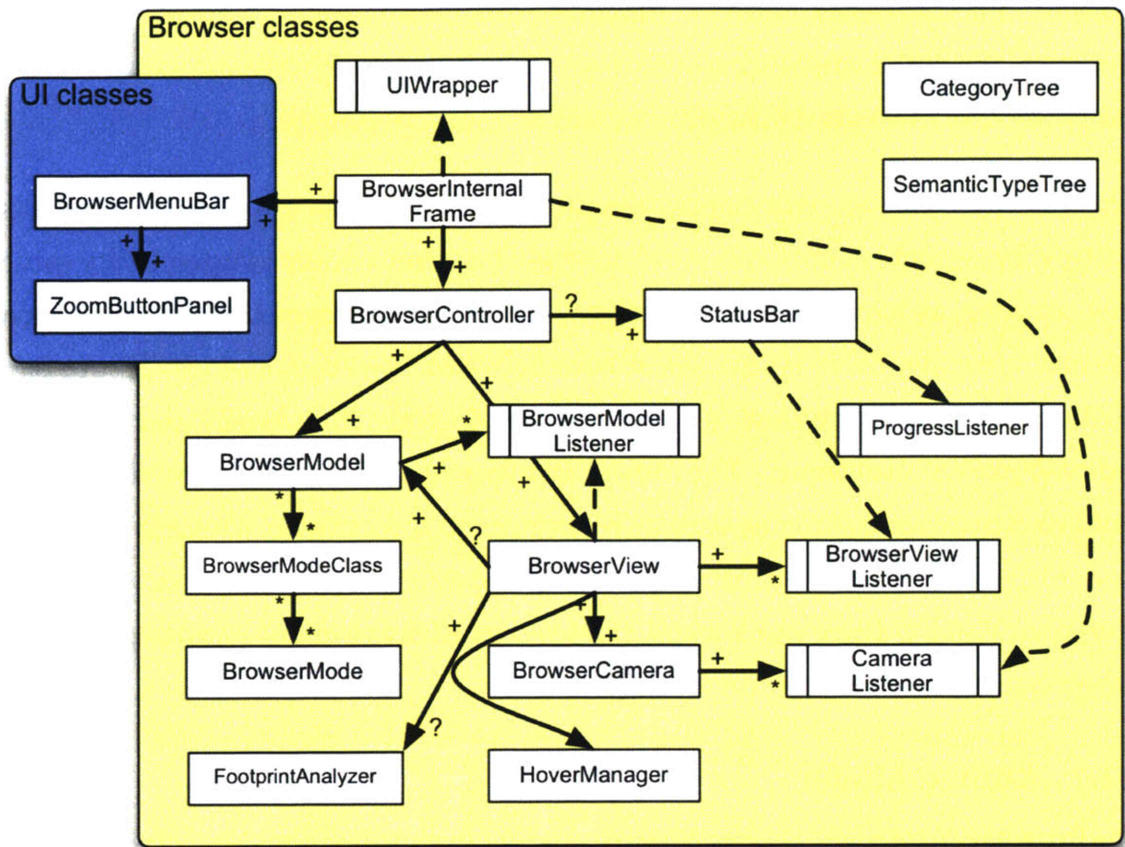


Figure 3.5. The browser core classes.

metadata that other components use. For example, each browser model contains a `CategoryTree` object, which manages the set of phenotype classes and phenotypes (categories) that exist (within OMEDS) for images in a particular dataset. The browser model also encapsulates a `SemanticTypeTree` object, which keeps track of which image-granular semantic types are valid for the dataset. If an image has a non-null attribute of a particular semantic type, that type will exist in the browser model's semantic type tree.

Finally, the browser model maintains browser state. At any time, a browser may be in several of a few modes. Whether or not the magnifier is turned on defines one mode. Whether or not thumbnails have been selected is another. Each discrete category of modes is called a `BrowserModeClass`, each of which has several `BrowserMode` objects. One mode may be selected for each class at a time. Other classes can infer the mode of the browser model through its accessor methods. In addition to modes, the browser keeps track of how thumbnails are

organized onscreen by storing default and current `LayoutMethod` objects. Section 4.3 covers how the `LayoutMethod` objects place thumbnails within the browser viewer.

3.3.2 The Browser View

A `BrowserView` object is the visual component of a browser window. It is a drawing canvas, a Piccolo `PCanvas` object. As such, it is the parent object for all visual primitives in the dataset view, including thumbnails, overlays, tool palettes, and popup menus. In addition, as the canvas and a Swing UI component, it receives both mouse and keyboard input. It is the first object in the dataset view to be notified of user events, and can thus best control how they are processed. Section 3.5 discusses image browser event handling in detail.

At its heart, the browser view is the parent for all thumbnails. This is critical, as Piccolo organizes visual primitives into a hierarchy; a canvas or a layout can be the parent of other nodes, and in turn, those nodes can have other children. The browser view is effectively the root. It has no visible parent (there is a root object to describe the entire scene graph, including the projection and perspective of the canvas), but does have many children—the image thumbnails. Drawing in Piccolo is recursive; if the parent is redrawn, its children will also be redrawn. Thus, while the browser view is responsible for *when* the dataset view is updated visually, it is not necessarily responsible for precisely *how* it is drawn. Indeed, the `paintComponent()` method of the `BrowserView` object is fairly short; Section 3.4 will illustrate how and why the bulk of the drawing logic resides in the `Thumbnail` classes.

The browser model and browser view share a producer-consumer relationship. Although the browser view contains a reference to the browser model, it is not notified directly when changes to the browser model occur. Instead, the browser model contains a set of listeners (that implement the `BrowserModelListener` class), and notifies these listeners whenever it changes. The `BrowserView` object is such a listener. It updates the view in response to a wide variety of changes to the underlying model, including thumbnail addition and subtraction;

mode changes, thumbnail selection, a change in the model's layout and grouping methods, and changes to the set of active overlays.

For a more detailed example, consider the case when a set of thumbnails is added to a browser model. This is a normal occurrence during the initial load of a browser window. The browser model will signal that such an event occurred by invoking the `thumbnailsAdded` method on all subscribers, with an array of `Thumbnail` objects as the parameter. In response to that invocation, the browser view will add all the thumbnails in the array to the Piccolo canvas; thumbnails are Piccolo nodes and can be children of a canvas layer. The view then consults the browser model as to how the organize the thumbnails, and then draws the dataset. The majority of events invoked by the browser model will trigger such a chain of events; the browser view will check the browser model's latest state, change the layout of its thumbnails if necessary, and then redraw the new pixels to the screen.

There are other functions that the viewer performs independently of the browser model. For example, the magnification level of the dataset is an attribute inherent to the browser view, not the model. Thus, the browser view itself has zoom level accessor and mutator methods. In addition, the view object has a significant amount of logic that makes sure that a dataset "looks right." The browser view calculates the total bounds of the entire collection of thumbnails onscreen, and ensures that a user cannot move beyond those bounds. It also limits the factor by which a user can zoom in and out of a dataset, to prevent thumbnails from appearing too small (for lower zoom factors) or too pixelated (for high levels of zoom). Finally, it contains two nodes that it maintains independently of the model: a background node, and a foreground node. It uses the foreground node to draw the boundaries and phenotype names in quantum treemap mode, and the background node to capture user input.

3.3.3 The browser camera

The `BrowserCamera` class controls the viewport into the browser view's canvas, and also serves as a Piccolo parent to any dataset-level overlays. When a user

zooms into a dataset, some regions of the dataset will be out of view, and others will be in focus. The camera keeps track of the current bounds of the viewport, manages the viewport position on zooming in and out, and notifies camera listeners when the boundaries of the viewport have changed relative to the browser view. The scrollbars of each browser window are updated in this manner; they listen for change events thrown by the camera and change their appearance accordingly. The `BrowserCamera` also contains a `PCamera` object, supplied when the browser camera is initialized. The `PCamera` object is the camera for the browser view. `PCameras` are special objects within Piccolo, in that they maintain a constant coordinate system, even if the scenes (or in this case, browser views) they "project" change coordinates due to zooming or scene translation. The implies that children of a `PCamera` will appear the same size, regardless of the zoom level or viewport location of the underlying Piccolo canvas. For that reason, the `BrowserCamera` manages any visual component that should be drawn at a constant size, such as the magnifier, and the annotation "Post-It" notes.

3.3.4 Support classes

Finally, each browser window contains classes that interact with Swing and with the rest of the browser application. The parent UI component of each browser window is a `BrowserInternalFrame`, a subclass of JInternalFrame. This class encapsulates all the UI widgets of the browser window, including the menu bar, toolbar, status bar, and `BrowserView` object itself. In addition, each browser window has a `BrowserController`, which provides an interface for the browser manager to access the browser model. The other UI components in the image browser, such as the heat map and color map, use the controller to change a browser window's drawing modes, and update the browser model, if necessary.

3.4 Thumbnails & drawing

Thumbnails are the basic units of both drawing and storing information in the image browser. Thumbnails contain more than just pixel data; they contain all metadata associated with the images they represent. Moreover, a Thumbnail

may contain more than one image. Like browser windows, thumbnail objects use a model-view architecture; the view controls how the thumbnail is drawn and how it responds to user input, and the model contains the backing state used to determine how a thumbnail is drawn.

3.4.1 The Thumbnail object

`Thumbnail` objects are the visual representations of thumbnails in the browser. They are instances of Piccolo's `PImage` class. `PImages` are nodes that have embedded images; their bounds are defined by the size of their embedded image, and its image is drawn whenever its parent (in this case, the browser view) is repainted. A `Thumbnail` is also responsible for drawing its own overlays. It does so by maintaining sets of `PaintMethod` objects, which contain additional drawing instructions to be executed during each redraw. Thumbnails also respond to user input events such as mouse clicks and hovers by maintaining a set of behaviors to be executed whenever such events occur. (see Section 3.5 for more details) Thumbnail objects may contain more than one image, as is the case in chemical screening. If a thumbnail has multiple embedded images, it is called a "multiple-mode" thumbnail, and much of its behavior has to change because of its multi-image nature. Finally, `Thumbnail` objects are normally created when the dataset is loaded, once the browser agent has downloaded the image from the image server. However, the image contents are allowed to change; if an analysis module changes the look of a particular image, a thumbnail can be updated to reflect this change.

3.4.2 Thumbnail data models

Each thumbnail contains one or more `ThumbnailDataModel` (TDM) objects, which contains all metadata associated with the thumbnail's embedded image (see Figure 3.6). If a thumbnail is a "multiple-mode" thumbnail, it will have multiple TDMs. The thumbnail data models are the basis for image classification, annotation, and organization, as they are the lone objects in the image browser that contain OMEDS data. TDMs contain basic information about the images they belong to, including image name and dataset ID. More

importantly, TDMs contain AttributeMaps, which are the structures that store metadata from OMEDS.

AttributeMap objects make it fairly easy to infer particular information about all images in a dataset. They form key-value pairs between semantic types (or, more accurately, their names) and any attributes of that semantic type that may apply to an image. For example, consider what happens when a user selects the "Annotations" item from the View menu. How does the browser figure out which images have annotations? As it turns out, Annotation is actually a semantic type with image granularity, and instances of ImageAnnotation are attributes in the database. When a user loads a dataset, if any images within that dataset contain annotations, they will be loaded into those images' attribute maps. So, to figure out whether or not to draw an overlay, all a module needs to do is for each thumbnail, check if there is some non-null value associated with the "ImageAnnotation" ST or name in its AttributeMap. Attribute maps also contain classification and phenotype data (as those are attributes as well), image statistics, and any other metadata stored in the OME database and requested at some point by a browser process.

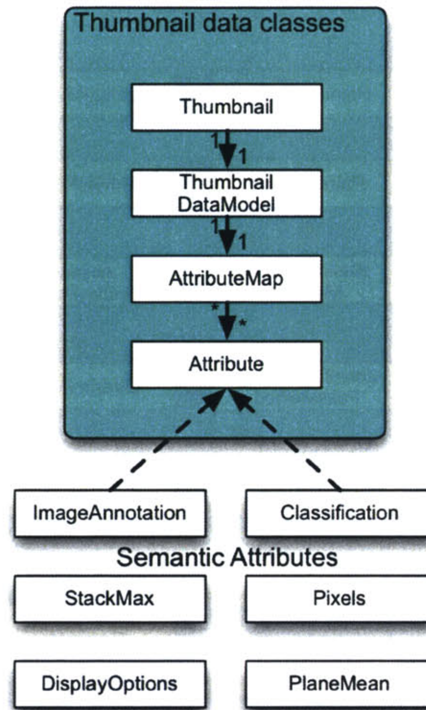


Figure 3.6. Thumbnail classes.

3.4.3 Drawing thumbnails and overlays

Each thumbnail maintains a set of PaintMethod objects. A PaintMethod is an interface with a single method: paint(), which takes a PPaintContext object (supplied to a thumbnail through its paint() method, just like a Graphics object is supplied to a Swing or AWT component) and a Thumbnail as parameters. Each paint method contains logic to determine how to draw an overlay atop a thumbnail. This logic is usually the same for all images in a

dataset, which is why the `paint()` method takes a thumbnail as a parameter. When executed, the logic inside a paint method analyzes the thumbnail or the thumbnail's backing model to determine what kind of shape to draw, if any. This is precisely the module that makes the decision to draw an overlay.

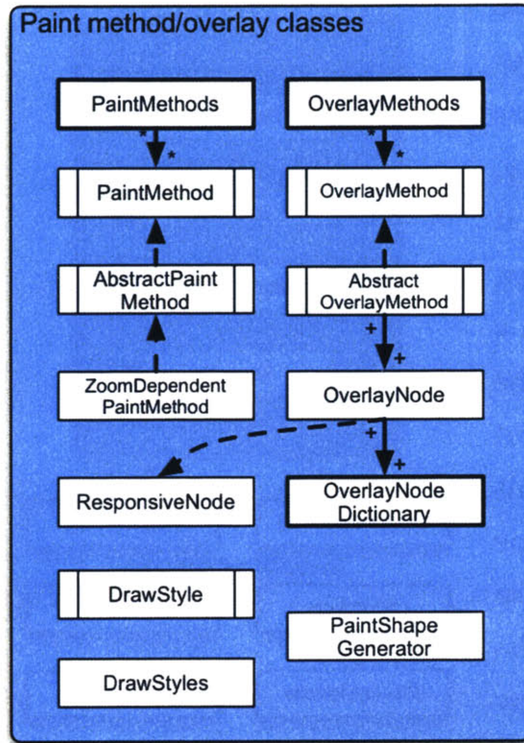


Figure 3.7. Paint method classes.

Because these paint methods can be executed across all thumbnails in a dataset, they are reusable, easy to write, and cleanly abstracted away from thumbnails themselves. Several commonly used paint methods are found in the `PaintMethods` object. One such paint method is responsible for drawing a small graphic atop a thumbnail if it has a valid image annotation—the `DRAW_ANNOTATION_METHOD`. The logic inside this paint method is exactly that outlined in the previous section. When the thumbnail calls `paint()` on that `PaintMethod`, it analyzes the model of the thumbnail supplied as a parameter, and if the model's `AttributeMap` contains an `ImageAnnotation` attribute, it draws the icon to the `Graphics2D` canvas accessible through the `PPaintContext` parameter.

Occasionally, the logic required to determine what kind of overlay to draw can be costly. For example, in heat map mode, there may be a significant number of attributes of type PlaneCentroid associated with an image, especially a multi-dimensional image. Computing the average value to determine which color overlay to draw should only happen once. Thus, some paint methods include caches that map thumbnails to overlays. In this case, the logic to pick a thumbnail color will be executed once per thumbnail. On subsequent calls to `repaint()`, the paint method logic will simply fetch the color mapped to the

thumbnail in its cache, and draw a rectangle of that color atop the image. The design of `Thumbnail` and `PaintMethod` makes this possible, as all thumbnails can contain a reference to the *same* `PaintMethod` object, which in turn can hold references to *all* thumbnails.

Thumbnail objects contain three sets of paint methods—a background set, a middle set, and a foreground set. A thumbnail's paint() method executes the paint() function of every method in the background set, then renders the image, and then calls the paint() function of every method in the middle and foreground sets, respectively. As a result, overlays created by paint methods in the background set will appear behind the image, middle overlays will appear immediately atop an image, and foreground overlays will appear atop both middle overlays and the image itself. All coloring paint methods, such as the heat map and color map methods, are middle overlays. Annotation icons, data values, well numbers, and selection state are drawn in the foreground.

Adding these overlays as Piccolo nodes is a more natural way to add information to a thumbnail. However, Piccolo's performance dropped off as the number of visible nodes in a scene graph increased beyond a certain threshold—around a thousand. This makes sense; each node has an event handler, and has somewhat complicated paint logic. Drawing overlays that don't respond to user input by with only additional Java2D drawing calls eliminates this overhead. In addition, we do not have to worry about removing children (an occasionally messy task), or the overhead of *thousands* of additional objects in the Java virtual machine. Recall that a single `PaintMethod` object can be applied to any number of thumbnails, so its overhead in the JVM is very low.

3.5 Event handling, actions and behavior factories

The image browser adopts an unconventional event handling architecture in the dataset view, a hybrid between bottom-up event handling, and top-down event handling. Event handling in most Swing applications is *bottom-up*, where individual UI widgets are responsible for responding to UI input. For example, a Swing button often contains an *action listener* that executes when it is pressed.

This is in contrast to *top-down* event handling. In a top-down system (any applet constructed in the early days of Java 1.0 is an example), there is a single event handler per application, which determines which object received the user input and responds appropriately.

The advantage of bottom-up event handling is its simplicity and reusability. Top-down event handlers often contain complicated logic, whereas bottom-up event handling allows a programmer to bind reusable actions to individual UI widgets. However, there are also several drawbacks to bottom-up event handling, especially with large datasets. Adding a separate event handler to each UI widget (including thumbnails) adds overhead to the Java virtual machine. More importantly, it is difficult to include code that affects the entire enclosing application with a bottom-up event handler. Moreover, bottom-up event handling doesn't handle modal semantics very well. The browser has several modes that determine the appropriate UI response. For example, when the magnifier is deselected, moving the mouse over a thumbnail should not activate it, and vice versa. This is easier to manage using top-down handling, as the single event handler can determine what action to take based on the current mode.

3.5.1 The BrowserView/Thumbnail approach

The `BrowserView` object combines these two approaches. The thumbnails contain the code that executes the response, but the `BrowserView` assigns that code to each thumbnail depending on the mode, and serves as the event handler for the entire application. In addition, the behavior of a thumbnail in response to user input is more configurable than that of a default Swing widget.

All thumbnails implement two event interfaces (see Figure 3.8 for an illustration): `MouseOverSensitive`, which is analogous to Swing's `MouseMotionListener` interface, and `MouseDownSensitive`, which is analogous to Swing's `MouseListener` interface. These interfaces contain similar semantics ot their Swing counterparts, with one critical addition: accessor and mutator methods for each action. A module can specify what occurs when a thumbnail (or any other `MouseDownSensitive` object) is double-clicked by

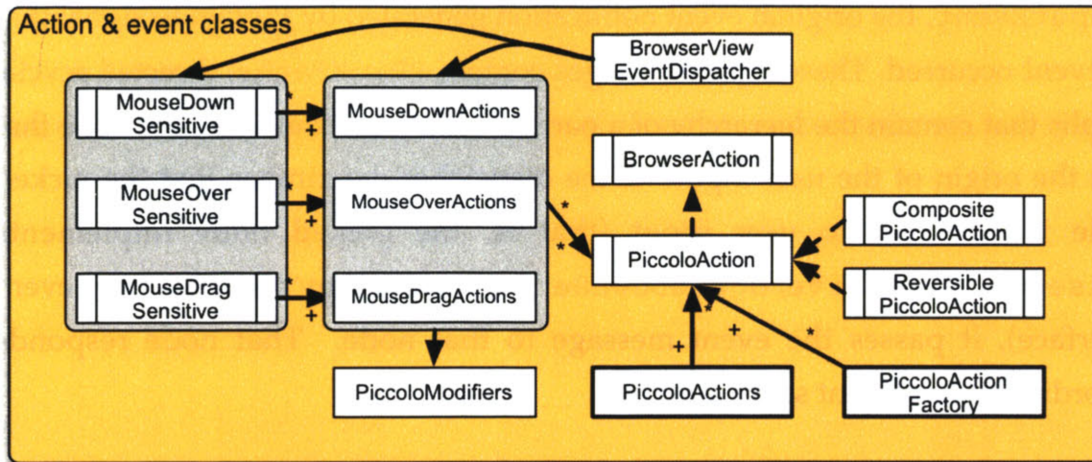


Figure 3.8. The browser event & action classes: a hybrid event-handling approach.

calling its `setDoubleClickAction` method. In this manner, the browser window can change specific UI responses depending on a mode, while leaving others the same. Moreover, the mutator methods take an input modifier as a parameter, so the browser model can dynamically configure actions that occur when a user right-clicks, or shift-clicks a thumbnail.

The browser view object controls the UI responses of each thumbnail. Based on the browser mode (which it finds out about when invoked by the browser model), it selects an appropriate set of UI responses for all or a set of thumbnails. For example, consider the case when a user selects a thumbnail. This indicates a mode change, and requires thumbnails to respond to user input differently. Clicking on another thumbnail should deselect the previously selected thumbnail, as well as select the new one. Shift-clicking should add additional thumbnails to the selected set. Clicking a non-thumbnail region should deselect all thumbnails. The browser view ensures that the thumbnails respond in this manner by passing these actions to their `setClickAction` methods. When a user clicks on a space away from the thumbnails, the mode changes again (no thumbnails should be deselected), and the browser view applies the new set of responses to each thumbnail.

The browser view object employs a `BrowserViewEventDispatcher` object to handle all user input. The dispatcher is the conventional event handler for both Piccolo and Swing events. It can determine the origin of an event by analyzing a

PInputEvent, the original event notification generated by Piccolo to signal that an event occurred. These input messages contain PPickPaths, directed acyclic graphs that contain the hierarchy of a particular node; in this case, the node that was the origin of the user input. If the dispatcher determines that the picked node is sensitive to user input (that is, the picked node implements MouseOverSensitive, MouseDownSensitive, or any other browser event interface), it passes the event message to that node. That node responds according to its current set of actions.

3.5.2 PiccoloActions

The image browser contains a number of classes that encapsulate *actions*, or responses to user input. The base interface is a BrowserAction, analogous to the Swing Action interface. A BrowserAction specifies only one method—execute(). However, the image browser uses the base interface sparingly; most actions implement the PiccoloAction interface. The PiccoloAction interface, which extends BrowserAction, specifies an additional method—execute(PInputEvent). All response logic is in that method. There are several varieties of actions, including a CompositePiccoloAction, which bundles a set of actions together, and a ReversiblePiccoloAction, which contains both execute and undo methods.

The normal pattern for action logic is to extract the picked node from the PInputEvent, cast it as a Thumbnail or appropriate OverlayNode object (casting coherence is guaranteed by the browser view), and then perform some operation on the thumbnail, or execute other module methods using the specified thumbnail as a parameter. For example, consider the popup menu action, triggered by a user right-clicking on a thumbnail. The input event is first processed by the browser view dispatcher, and passed to the picked thumbnail via its respondMouseClick method. The thumbnail's response to a respondMouseClick call is to inspect its set of actions for a response that matches the pair <mouse-click, popup modifier>, and execute that action with the PInputEvent as a parameter. The action bound to the input-modifier pair is the popup menu action. The logic inside the popup menu action again extracts

the picked node, constructs a popup menu based on the thumbnail selected, and draws that popup menu atop the thumbnail. The second node extraction seems redundant, but is required for action reusability. Much like a single paint method can apply to every thumbnail, each thumbnail can use the same `PiccoloAction` event to respond to user input. This allows the image browser to maintain a collection of reusable action classes that are deployable to all thumbnails.

3.5.3 PiccoloActionFactory and other behavioral factories

Many actions require additional state and access to additional modules in order to execute properly. Certain actions are also conditional on additional browser state, not just browser modes. The `PiccoloActionFactory` class constructs these more complicated `PiccoloActions`. As the name suggests, it is a factory class; each method takes in several parameters and generates a `PiccoloAction` object that uses those parameters.

Again, consider the popup menu action. The popup menu is a Swing object, and must be the child of another Swing UI object. However, there is no Swing UI component specified in the `PInputEvent`, or `PPickPath`. Thus, the action is created using the factory's `getPopupMenuAction` class, which takes a `BrowserView` object as a parameter. The parameter is final (constant), so the `PiccoloAction` the method creates can reference it. The popup menu action's `execute()` method does so; after creating the popup menu, it instructs it to appear inside the passed browser view component window.

The action factory is one of several examples of *behavior factories* in the image browser. A behavior factory creates logic classes, which can be applied and executed by any thumbnail or other module. The purpose of such a factory is to abstract action from presentation in the code, and to make certain actions and action patterns more reusable. Other behavior factories in the image browser include the paint method factories in both the color map and heat map, and a popup menu factory. The paint method factories generate logic that determines what color an overlay should take, whereas a popup menu factory generates

logic that determines how a popup menu should be built, in response to what kinds of phenotypes and categories are available in a dataset.

3.6 Heat & Color maps

The heat and color map objects are UI components that are separate from the browser window. They adopt their own MVC (model-view-controller) architecture. They both can access the browser agent independently of the browser window, and can appear even when no browser is visible in the application. However, both the heat map and color map use `BrowserModel` objects to obtain information about a dataset, and the images within that dataset. Thus, they both rely upon some underlying data model to display information to the user. This section will briefly outline the architecture of each component, and how they interact with the browser model and other UI component classes.

3.6.1 The heat map

The heat map, as discussed in the user interface overview, is a component that allows a user to compare images in a dataset based on a single variable. A user chooses a variable to compare by selecting an element from a tree of valid semantic types. When a user selects an element in the tree, the images in the active dataset are analyzed to determine a minimum and maximum value, and then each image is assigned a color based on its variable value. Images whose values are close to the minimum appear blue, and images whose values are close to the maximum appear red.

The heat map contains several model classes (see Figure 3.9 for more details), which contain state for UI components and dataset metadata. The `HeatMapModel` is the backing data model for the entire component. It contains a reference to a data source (a `BrowserModel`), and a tree model of valid semantic types and elements. The `HeatMapModel` builds this tree model by acquiring a list of relevant, applicable semantic types from the browser model (using the `getRelevantTypes()` method), and constructing a `SemanticTypeTree` from that information. A semantic type tree is a data structure that organizes semantic types and elements into a hierarchy, keeps

track of the data type of each element, and facilitates retrieval and storage of attributes retrieved from OMEDS. It also serves as the model for the tree view; the `HeatMapTreeUI` constructs a `JTree` object based on the data types and hierarchy contained in the tree.

The remaining models manage heat map parameters and statistics. The gradient class (`HeatMapGradient`) maintains statistics about the range of a variable, such as its minimum and maximum value. Multiple scale objects (linear, logarithmic) are constructed for each variable, and perform the math that converts a variable value to a color. The heat map component also contains a mode class. Each mode class (`HeatMapMode`) contains a function that determines how a single variable value is assigned to an image, in the event that the image has multiple attributes of like semantic type. Modes include minimum-value assignment, mean-value assignment, median-value assignment, and maximum-value assignment.

Each model backs a corresponding view component. The parent view object of the heat map is the `HeatMapUI`. The heat map UI is responsible for listening to changes in the heat map model, and refreshing and rebuilding its subcomponents in response. It also maintains basic heat map state, such as whether or not the heat map is displaying a scalar or Boolean value.

The `HeatMapUI` contains a `HeatMapGradientUI`, a widget in charge of representing the range of a variable and a color legend to the user. This class draws the color bar and the scale that indicates the range of variable values to the user. The `HeatMapTreeUI`, as mentioned, constructs a Swing tree based on the semantic type tree stored in the `HeatMapModel`. It also responds to user input, particularly node (element) selection. Finally, the `HeatMapModeBar` and `HeatMapScaleBar` allow a user to select a mode and scale, to further refine the mapping between color and variable value.

The `HeatMapDispatcher` class is the nerve center of the heat map. It is responsible for extracting metadata from OMEDS via the browser agent in response to element selection, and for constructing the paint method to apply to

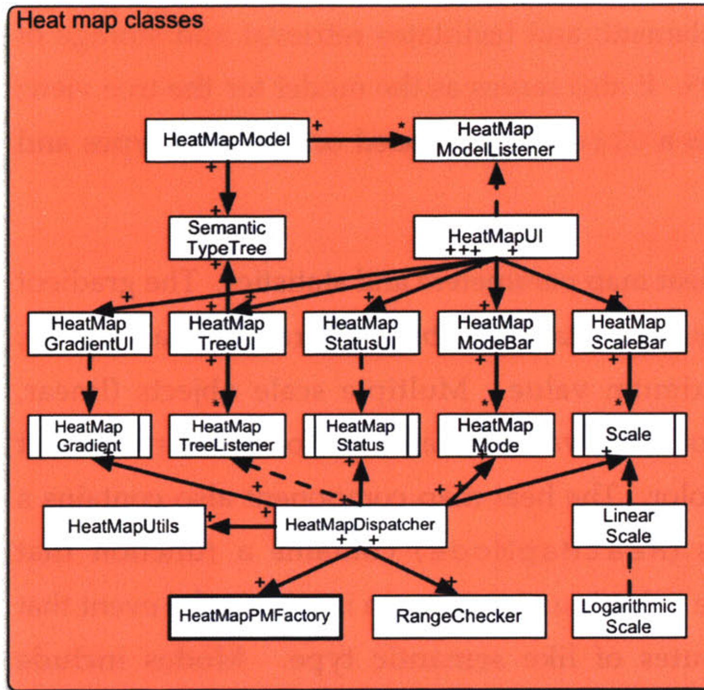


Figure 3.9. Heat map classes.

all thumbnails. Each UI component in the heat map notifies the dispatcher whenever some user input event occurs, such as a mode or scale change, or element selection. When an element is selected, the dispatcher accesses the semantic type tree to determine the semantic element to display. The dispatcher then analyzes the tree to determine how it should structure its queries to OMEDS. Once the dispatcher executes these queries and receives metadata from OMEDS (a process which may take some time, so the dispatcher posts status messages to the heat map UI), it uses the `HeatMapPMFactory` to build an overlay method. The factory constructs a paint method that draws an overlay based on the current mode, scale, and min-max information. Finally, it adds this method to the browser model through its `addPaintMethod` function, which allows a user to apply a `PaintMethod` to all thumbnails at once. Whenever a user changes modes, scales, or elements, the factory must construct and apply a new paint method, and remove the previous paint method from the dataset. Finally, the dispatcher responds to changes in the heat map model itself, by redrawing the gradient and tree UI objects.

Finally, other UI components can interface the heat map using the `HeatMapManager`. The `HeatMapManager` allows modules such as the browser window to launch the heat map view. In addition, the browser agent notifies the heat map manager whenever a dataset is loaded, and constructs a new heat map model. This is covered in more detail in Section 3.7. The manager is embedded in the `BrowserEnvironment` class, as it is an application-level module.

3.6.2 The color map

The color map is the component that colors images by phenotype. It contains a legend that displays both the color and phenotype name, and a combo box for changing the class of phenotypes to investigate. It does not have the same range of options as the combo box, but does not need to display data that is as complicated, or as continuous.

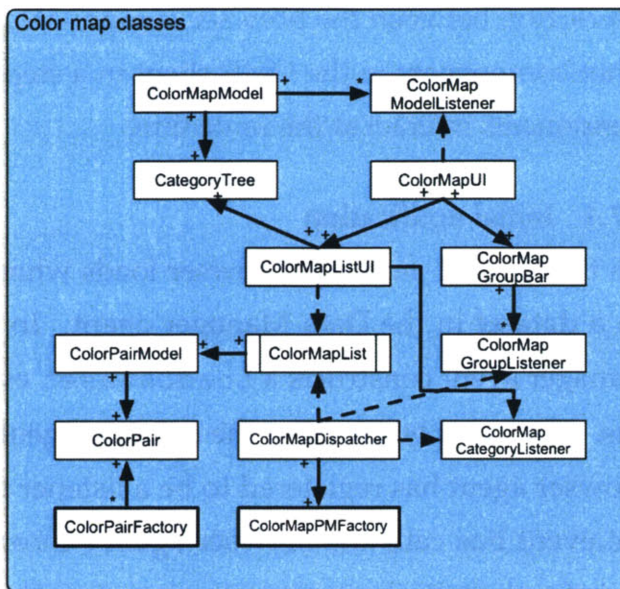


Figure 3.10. Color map classes.

The color map architecture (shown on Figure 3.10) mirrors the structure of the heat map. Like the heat map, it contains a model class for each UI component, UI widgets which visualize those models, a `ColorMapDispatcher` object that responds to user input and controls the look of the other UI widgets, and a `ColorMapPMFactory` that constructs paint methods that apply to all thumbnails. These classes interact in a similar manner to their analogs in the heat map.

The one unique implementation to note in the color map is color selection. Because phenotypes are discrete, the color map must assign a color to a specific phenotype arbitrarily, as opposed to interpolation. Currently, the color map assigns the first 32 phenotypes in a class with a distinct color. However, past that, the color map will start assigning arbitrary, random color values to additional phenotypes. We have not yet tested constructing a dataset (or set of categories) that meet this criteria, but in that case, it would be easier to organize thumbnails by a quantum treemap than classifying by color.

3.7 Integration: Loading a Dataset

The best way to demonstrate how all these pieces fit together is to analyze what happens when the image browser loads a dataset. This analysis will focus on the

interaction between the browser components, and between the browser and the other components in the OME client in a step-by-step approach, based on which components interact at the same time.

3.7.1 Initial notification

On the surface, an image browser loads when a user clicks the "Browse" option on a dataset in the Data Manager agent. In response to the selection, the data manager agent constructs a `LoadDataset` event object, and posts it to the event bus. The event bus notifies the browser agent of the `LoadDataset` event, as the browser agent has registered to be a listener for such messages. More precisely, the event bus calls the browser agent's `eventFired` method, which contains logic for demultiplexing event messages. The logic inside `eventFired` extracts the information from the message—most specifically the ID of the dataset to load, and then calls the browser agent's `loadDataset` method, which starts the loading process.

3.7.2 Initializing the browser window

The browser agent's `loadDataset(int)` method constructs the components of the browser window. It creates a new `BrowserModel`, `BrowserView` and `BrowserController` object, and adds the component to the `BrowserManager`. In addition, it initializes and keeps a reference to the browser window's status bar, so that it may display messages to the browser window while loading data from the OMEDS. Finally, it adds the UI component to Shoola's parent `JDesktopPane`. This is the earliest point when the agent can show some feedback to the user that it is loading a dataset, so it chooses this point to display the window in the client, even though the browser does not yet have any thumbnails or image metadata to show.

Once this occurs, the browser agent starts a thread that loads data from OMEDS and OMEIS. This process runs in a separate thread because this task may take a long time to complete, and blocking UI input until the dataset is fully loaded would be highly undesirable. In addition, a user may want to cancel the loading process. In order to support load cancellation, the loading thread is an instance

of a KillableThread, a class unique to the image browser. A KillableThread is a thread that specifies a kill() method. It is unsafe to call a stop() method on a thread, so Java no longer supports that procedure. However, a KillableThread allows other modules to specify when a thread should die. Calling the kill() method will set a Boolean flag in the thread that the thread should die as soon as possible. The logic inside the thread can choose to heed the flag or not, so the method specification does not guarantee that the thread will stop. However, the threads in the browser agent behave well, and stop processing (i.e., don't make network calls to the OMEDS or OMEIS, or update other browser components) when killed.

3.7.3 Loading image metadata

The first task of the thread is to load the basic data about each image in the dataset, by requesting core image object data from OMEDS. This data includes the name of the image, the default Pixels attribute, and the image ID. It maintains this list of IDs as a parameter to supply to OMEDS when retrieving attributes, such that the agent will only retrieve attributes and metadata pertaining to those images. These IDs effectively act as the argument to a SQL IN clause.

Next, the thread loads all attributes that are essential to the display of the thumbnails in the dataset view. It queries the OMEDS to return all attributes of semantic type ImagePlate, ImageAnnotation, and Classification that belong to any image in the dataset, as well as any CategoryGroup and Category attributes that belong to the dataset. The latter two are the database representations of phenotype classes and phenotypes; a Classification is a mapping between an image and a phenotype.

The browser agent processes these attributes immediately after the OMEDS response. Using the retrieved CategoryGroup and Category attributes, it creates a new ColorMapModel and instructs the ColorMapManager to add and show that model in the color map UI. It also stores a copy of the phenotype class and phenotype hierarchy (a CategoryTree) in the browser model. In addition,

it stores a map between image IDs and both `ImageAnnotation` and `Classification` objects, which will be added to thumbnails later. It also saves `ImagePlate` objects, if they exist, which will be used to determine the dataset's layout later.

However, before layout and thumbnail acquisition, the browser agent loads the definitions of all semantic types with attributes that belong to images in the dataset. This is the most complicated database operation by far, and is clearly the most time-consuming. When complete, however, the browser agent builds a `HeatMapModel` using those valid semantic types, and instructs the `HeatMapManager` to display the semantic type tree within the model if it is active.

The loading thread diverges at this point, based on whether or not the dataset is a based on a standard screen. If it is, the browser agent maps which image belongs to which well, infers the size of the screen from the `ImagePlate` attributes retrieved, and adds multiple images to a thumbnail, if necessary. If it is not, the browser agent simply retrieves the pixel data for all thumbnails from OMEIS, without preprocessing. In both cases, the threads construct a group of Thumbnail objects, add image data from OMEIS, and then add all the image metadata captured thus far to the thumbnail's `ThumbnailDataModel`. Finally, the browser agent adds the thumbnail to the model. When all thumbnails are added, the loading is complete, and the view is instructed to draw all the thumbnails. At this point, the dataset appears to the user for analysis.

CHAPTER 4

Supporting Large Image Datasets

The ultimate goal of the image browser is to provide biologists with the means to effectively analyze large and varied datasets. Thus, an image browser must be able not only to display a large number of images at once, but also to display information to that aids in extracting meaning from image datasets. This section outlines how the image browser combines image and numerical information to enable faster and more precise analysis. I will briefly describe several features, how they facilitate analysis, and how they are implemented in the browser code.

4.1 Semantic zooming

Semantic zooming is simply the selective display of metadata and other information based on the size and resolution of thumbnails. Semantic zooming balances information overload and underreporting metadata, and ensures that metadata is presented to the user in the correct context. When a user zooms out to inspect the entire dataset, he/she likely does not want to be flooded with information about each image. Instead, users likely want to spot trends in the dataset, quickly pick out outliers, and inspect the types of images in the dataset. Similarly, when users zoom in to a specific region or image, they are likely more interested in those images themselves. Because a user focuses on individual images more at higher levels of magnification, the browser displays more detailed information about each image. Multiple components in the browser use this concept to effectively display information.

4.1.1 Magnifier

The magnifier provides more information to a user about a single image in a dataset, and provides the user with a range of image manipulation options. Section 2.3.3 describes the individual features and functions available to the user through the component. The magnifier is a good tool for initially analyzing and identifying images of particular phenotypes within the dataset. For example, because the magnifier displays images at higher resolution, a biologist can quickly confirm that an image in a particular screen is of bad quality, and mark as such using the browser's classification tool. In future releases, the magnifier will also display select image features, such as displaying "spots," regions calculated by an analysis module. Displaying image feature information atop a low-resolution image would lead to information overload; displaying it atop a magnified image of interest makes more sense.

The plumbing behind the magnifier is located in the `SemanticZoomNode` class. The class contains the logic to fetch a higher-resolution version of the target image from OMEIS, manages `PiccoloAction` responses to user input, draws icons to the user and keeps track of icon hotspots, and finally draws information about the thumbnail atop the image. A variety of `PiccoloAction` classes regulate the magnifier's activation. If magnifier mode is on, moving the mouse across a thumbnail will trigger that thumbnail's `MouseOverAction`. This action instructs a browser view helper class, the `HoverManager`, to start timing how long the user has kept the mouse over that thumbnail. If the hover time exceeds 500 milliseconds, the `HoverManager` adds a new `SemanticZoomNode` to the view's `BrowserCamera`, and draws the over the active thumbnail. Note that because the magnifier is a child of the `BrowserCamera`, it will always remain the same size, regardless of the zoom level of the dataset. The magnifier disappears if the user moves the mouse off of its onscreen region.

4.1.2 Contextual image overlays

Contextual overlays are overlays that only appear atop images at a given level of zoom or greater. There are two varieties of contextual overlays in the browser:

overlay graphics (normally drawn by `PaintMethods`), and *overlay nodes*, which are overlays that can respond to user input. Examples of overlay graphics include well numbers, and the variable value in heat map mode. The annotation icon is an example of an overlay node; UI events occur when a user clicks or moves the mouse over the node. Whereas overlay graphics mitigate information overload, overlay nodes mitigate hotspot overload. Any graphical representation sensitive to user input should not have hundreds or thousands of hotspots; sensitive regions become too small, and a user can often accidentally trigger a UI response.

Section 3.4 covers the general construction and drawing of overlay graphics, but omits a few implementation details. First, there exists a special subclass of `PaintMethod`: a `ZoomDependent-PaintMethod` (see Figure 4.1 for a listing of the paint classes). This class is a wrapper for a normal paint method, with two additional parameters: minimum zoom level, and maximum zoom level. The `paint()` method of the `ZoomDependent-PaintMethod` checks the current scale of the drawing context, and executes the paint() procedure of the embedded `PaintMethod` only if the current scale is within the specified zoom range. The `PaintMethod` that draws variable values on images in heat map mode is such a zoom method, with bounds <0.75,+∝>.

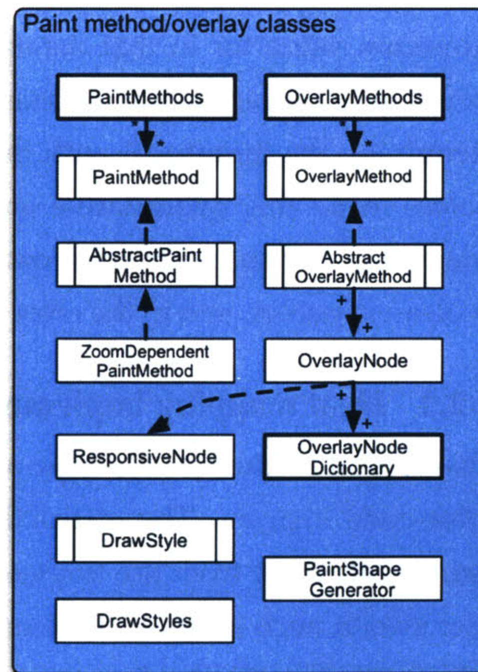


Figure 4.1. The paint method classes. Dotted arrows indicate inheritance; solid arrows indicate usage. The drawing utilities are on the bottom.

Overlay nodes are constructed and added to thumbnails in mostly the same manner as overlay graphics, with a few differences. Each overlay node is an instance of an `OverlayNode` class, which in turn extends `ResponsiveNode`, which contains default responses to all UI events. `OverlayMethod` objects that create the `OverlayNode` are added to the thumbnails, instead of the

`OverlayNodes` themselves. The semantics of an `OverlayMethod` are similar to those of a `PaintMethod`. However, `PaintMethods` do not necessarily have to retain state; they contain code that should be executed every time the canvas is drawn. However, `OverlayMethods` add overlay nodes (which are Piccolo node children) to a `Thumbnail`, which should not happen every time an image is drawn. Thus, an `OverlayMethod` must check if the thumbnail is already displaying the node. In addition, if the thumbnail no longer meets the overlay method criteria, the method has to remove the overlay node from the thumbnail. Aside from that detail, the semantics of an `OverlayMethod` are identical to those of a `ZoomDependentPaintMethod`.

4.2 Color coding

Colors are an ideal way to allow a user to compare images in a large dataset, both by phenotype, by scalar variable, and by Boolean value. Consider an alternative to classification by color. Imagine that the image browser instead drew the scalar values of a particular variable atop thumbnails. It would be much harder to for a user to determine the image with the maximum and minimum value for in that dataset. If a hundred images were visible, a user would have to scan each thumbnail to extract that information, instead of quickly identifying the thumbnail with the designated minimum and maximum color. Colors make this quantitative analysis much easier, and much clearer. The image browser supports color-coding in the heat map for analyzing scalar and Boolean variables, and in the color map for classifying by phenotype.

4.2.1 Heat mapping implementation

This section focuses on the logic of the paint methods generated by heat map to color-code images. The instructions to build and apply paint methods onto thumbnails come from the `HeatMapDispatcher`, which responds to heat map user events, such as semantic element selection or scale selection. When the user selects an item in the heat map's element tree, the dispatcher retrieves all instances of an enclosing semantic type from the database and stores the attributes locally. The dispatcher subsequently uses the `RangeChecker` object to compute the minimum and maximum values of a variable over all images in

the dataset. Finally, the dispatcher instructs the HeatMapPMFactory to construct PaintMethods based on the type of the element selected. If the element is numerical, the dispatcher will invoke the factory's getPaintMethod() procedure. Otherwise, if it is Boolean, it will invoke the factory's getBooleanPaintMethod() procedure. Both procedures return a PaintMethod that the dispatcher then applies to all thumbnails in the active browser model.

The logic inside both paint method varieties is similar. Both method constructors take as parameters the current mode, interpolation scale, minimum/false color, maximum/true color, target attribute name, and target semantic element. The supplied element is the variable to convert to a color. As the elements are all final, they may be accessed from anonymous inner classes, which the method constructors return.

The logic of the Boolean PaintMethods created by the factory is simpler, because Boolean methods do not have retain state (see Section 3.4.3 for an explanation). The paint() method adheres to the PaintMethod specification, taking a PPaintContext and Thumbnail object as parameters. The method extracts the value from the supplied thumbnail by analyzing the thumbnail's AttributeMap. If the target attribute exists in the map, the target element's value will be either true or false. If true, the method will fill a region over the thumbnail with a rectangle of the "true" color (normally green). If false, the method will draw a rectangle of "false" color (normally red) over the thumbnail. If no attribute of the requested type exists, the method will not draw the rectangle.

Scalar paint methods are slightly more complicated, as they cache colors for each thumbnail. Each scalar paint method contains an IdentityHashMap that maps thumbnails to colors. If a thumbnail is not in the map, the PaintMethod will call its setupMethod() procedure. The setupMethod() procedure computes the mapped color, by analyzing the supplied scale, mode, and values of all relevant attributes. Once this color is determined, the scalar paint method draws

a rectangle of that color over the thumbnail, in the same manner as the Boolean paint method.

4.2.2 Phenotype color mapping

The color map draws colors atop thumbnails based on their phenotypes. Its operation is similar to that of the heat map. A `ColorMapPMFactory` object constructs overlay methods in a similar manner to the `HeatMapPMFactory`. The color map factory's `getPaintMethod()` procedure is nearly identical to that of `HeatMapPMFactory`, with two main differences. First, the method does not accept individual colors as parameters like the `HeatMapPMFactory`. Instead, it accepts a `ColorPairModel` that maps phenotypes to colors. Second, the color map factory's `setupMethod()` is somewhat different, although it addresses the same problem as the heat map factory's `setupMethod()`—to accelerate overlay drawing. Phenotypes are represented in the source and in OMEDS by `Classification` objects. The `setupMethod` thus searches for any `Classification` whose embedded phenotype (`Category`) is in the supplied phenotype-color map. An image may have multiple Classification attributes, at most one per phenotype class (`CategoryGroup`). If there are many phenotype classes, finding a classification belonging to the current `CategoryGroup` could be a time consuming operation. Caching the phenotype using `setupMethod()` avoids this bottleneck.

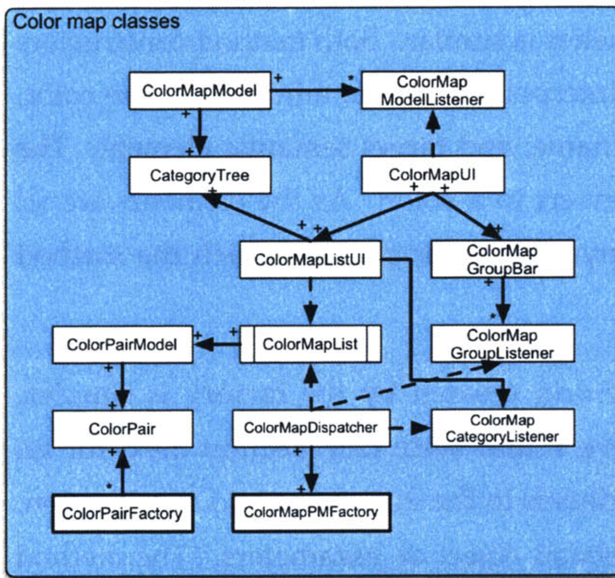


Figure 4.2. The color map classes.

4.3 Contextual layouts

The image browser provides *spatial coherence* when displaying a dataset, by placing thumbnails onscreen in a meaningful, structured manner. First, the browser can display images in a standard screen in plate order. Images that correspond to particular wells appear in the same location onscreen as they would in a plate, allowing a researcher to quickly pick out an image that corresponds to a particular well. The image browser can also group images of like phenotype or like criteria into groups, using a *quantum treemap*. Before describing how the image browser places and organizes thumbnails, I will discuss the structure and benefits of quantum treemaps in more detail.

4.3.1 Quantum treemaps

Quantum treemaps are graphical data structures originally developed by Ben Shneiderman and Ben Bederson at the Human-Computer Interaction Lab (HCIL) at the University of Maryland. *Treemaps*[11] are simply a method to divide a single rectangular region into different-sized subregions. Typically, treemaps are used to compare the size or importance of subsets within large datasets. For example, the SmartMoney MarketMap browser[12] uses treemap layout algorithms to subdivide a single region (the entire stock market) into different-sized market segments. The sizes of the segments are roughly proportional to the day's trading volume within each segment.

One drawback to traditional treemap algorithms is its dual constraint: all subregions must fit into a rectangular area, and the regions must be sized relatively proportional to each other. Thus, regions that are relatively small may have a high aspect ratio, so that all subregions may fit into a rectangular shape. Traditional treemaps have no notion of a minimum width or height, which is required if an application must to display structured information (such as a

---

[11] Bederson, B.B., Shneiderman, B., and Wattenberg, M. "Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies." ACM Transactions on Graphics, 21, (4), October 2002, 833-854.
[12] SmartMoney MarketMap Browser. http://www.smartmoney.com/marketmap.

group of thumbnails) inside the regions. Bederson and Shneiderman solved this problem by introducing quantum treemaps.

Quantum treemaps are used to place *n* identically-sized objects into *r* different regions. The minimum size and height of a region is determined by the size of each object. Establishing a minimum size ensures that regions are wide enough or high enough to contain a single visual object. Bederson uses quantum treemaps to organize photo albums in PhotoMesa, with favorable results.

Whereas PhotoMesa uses quantum treemaps to divide an entire album of photos into subfolders, the image browser uses quantum treemaps to divide thumbnails in a dataset into regions of like phenotype. This division allows biologists to more rapidly find images of a specific phenotype, and make quantitative comparisons of like images, which is especially useful in analyzing large datasets. Grouping subsets of images in large datasets together can allow a biologist to search for additional correlation between phenotypes, by using the heat map to investigate variable values, or by visual inspection. Grouping images together also adds order to a large dataset, which may be relatively unstructured when viewed as a plate or with another default layout.

4.3.2 Layout method implementation

The LayoutMethod family of classes place thumbnails in the dataset view. Each LayoutMethod implements two methods: `getAnchorPoint(Thumbnail)`, which returns the coordinate of the top-left corner for a that thumbnail, and `getAnchorPoints(Thumbnail[])`, which generates a `Map` that maps thumbnails to the determined coordinates of their top-left corners. The first method is used only for `LayoutMethods` that retain state; the `PlateLayoutMethod` is one such method. The other main methods used by the image browser are stateless. In both cases, the `LayoutMethods` do not physically place the thumbnails; instead, the `BrowserView` object uses the coordinates returned by the LayoutMethod objects to place its thumbnail children.

The two simple layout methods for non-screen datasets are very similar. Both the NumColsLayoutMethod and MaxWidth-LayoutMethod assign coordinates to thumbnails in rows from left to right across the canvas until a certain point, then begin placing additional thumbnails in a new row below the previous set.

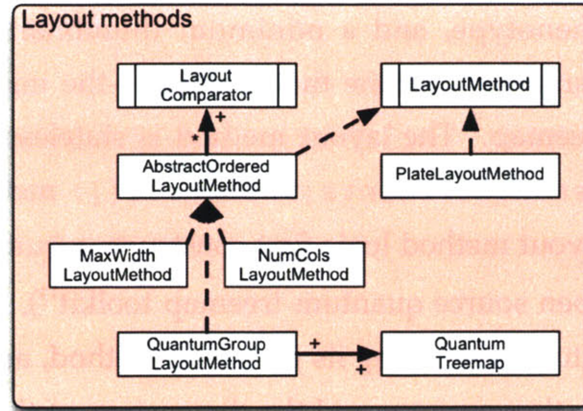


Figure 4.3. The layout method classes.

The NumColsLayoutMethod logic creates a new row every n thumbnails. The MaxWidthLayoutMethod starts a new row when the bounds of the next thumbnail in the row would extend beyond a specified maximum width. Both methods maintain a constant thumbnail order, as they are subclasses of AbstractOrdered-LayoutMethod, which has one function—to sort thumbnails in a list by ascending image ID. When presented with a group of thumbnails to place, both methods use the sorting function to arrange them by their ID, and then begin the coordinate assignment algorithm.

The PlateLayoutMethod is a bit more complicated, as it maintains the number of rows and columns in the plate, as well as a mapping between thumbnails and an index within the plate. The browser agent establishes this mapping when it loads a dataset for the first time. Indexes correspond to locations in the plate, increasing from left to right, then from top to bottom. For example, in a 384 (24x16) plate, the thumbnail corresponding to well B5 will have index 29. The layout method uses this index to quickly assign a coordinate to a thumbnail. The row number is the dividend of the index into the number of rows, and the column number is the remainder. Since all images in a screen are assumed to be the same size, the row and column number determine the coordinate of the thumbnail, which the PlateLayoutMethod returns to the browser view.

The most complicated layout method is the quantum treemap layout method. The constructor for a QuantumGroupLayoutMethod takes a CategoryGroupingMethod, which divides the thumbnails into subsets by

phenotype, and a minimum thumbnail width and height. The latter two parameters define the quantum—the minimum dimension of a region in the treemap. The layout method is stateless, so determines all coordinates in its `getAnchorPoints(Thumbnail[])` method. To calculate the coordinates, the layout method logic first constructs a `QuantumLayout` object (from the HCIL's open source quantum treemap toolkit[13]), divides the supplied thumbnail array into groups using its grouping method, and runs `QuantumLayout`'s algorithm on those groups and the dimensions of the parent container. `QuantumLayout` returns a Rectangle for each group, with position and dimensions that define the visual region for that group. The layout method then uses the rectangles to place the thumbnails. Thumbnails are placed in ascending image ID order within the regions, just like in `NumColsLayoutMethod` and in `MaxWidthLayoutMethod`. Occasionally, the `QuantumLayout` method will generate a different configuration of rectangles for the same groups, but it will always attempt to fill the enclosing container in as balanced and even a manner as possible.

[13] http://www.cs.umd.edu/hcil/photomesa/download/layout-algorithms.shtml.

CHAPTER 5
Sample Datasets

This section briefly discusses how the image browser, in its current state, performs against several real-world biological image datasets. The first is a complete 384-well plate obtained from Harvard's Institute for Chemical and Cell Biology (ICCB). The second is a sample collection of multidimensional images from the National Institute for Aging at the National Institutes for Health.

5.1 ICCB full plate

The first dataset is a complete 384-well chemical screen, containing 648 images. No analysis modules have been run on the dataset to date, so no image metadata is available other than image intensity statistics generated by OME's importer code. The browser, with the complete plate, is shown in Figure 5.1.

The ICCB dataset revealed both strengths and weaknesses of the image browser. First, it highlighted the value of the color map and treemap functions to sort and group the images. The thumbnails look nearly identical at low resolution, differing only in color and occasionally in intensity. Without some sort of graphical classifier, the dataset looks uninteresting and uniform. However, zooming out to see the entire dataset does allow a user to quickly pick out and classify poor-quality images. Even at a zoom factor of 40%, several images clearly appear overexposed, or blurry, or nearly empty. Displaying well numbers atop a thumbnail was more useful than anticipated, as it allows users to keep track of individual images when changing layouts. The magnifier worked well once it was triggered after a delay; it allowed users to easily perform basic image operations and quickly inspect individual images.

Figure 5.1. The 384-well ICCB dataset.

Performance on the large dataset was both a strength and a weakness. Once the image loaded, the browser was very responsive, drawing all images and overlays quickly. Moving the scrollbar updates the dataset viewport at a reasonable response rate, and the apparent speed of the browser increases as zoom increases. Changing image layout is nearly instantaneous, as is switching between categories and preloaded elements in the heat map. However, any operation that required querying the database triggered significant delay. For example, over a 100MB Ethernet connection, and using a dual G5 Mac OS X server to run OMEDS, the metadata from the ICCB dataset took over 90 seconds to load. The 648 thumbnails from OMEIS, required less than a minute to load. Selecting an element in the heat map triggers a database query that may take as long as 45 seconds to complete. This would likely be unacceptable as a first impression. However, these problems are likely not browser-specific. We suspect that XML-RPC is adding significant overhead to any database communication, as is the Perl layer on OMEDS. Although not limited to the

browser, these problems are at their worst when loading and extracting information about a large dataset. If poor data layer performance adversely affects the performance of the browser application, the OME project will need to focus on improving database performance.

Experimenting with the ICCB dataset also revealed several problems that will be addressed in upcoming versions of the image browser. First, more support for multiple images per well is needed in screening. Currently, the only overlay that indicates multiple images in a well is a folding icon. However, this only reflects the presence of multiple images, and not the content of those other images. There is presently no visual cue to indicate that another image in a well has an annotation. Heat mapping and color mapping only take the active image in a thumbnail into consideration when applying color overlays. When a user selects treemap layout, the thumbnails are organized based on the phenotype of the active image in a well, which may be different from the phenotype of another image in the same location.

The other significant problem has to do with color overlays. Currently, the browser does not convert the images in the browser to grayscale before applying a heat map or color map overlay, as the operation makes the browser's repaint operation much too slow. As a result, the overlays, which are partly transparent to show the contents of the image, may blend with the colors of the thumbnail in undesirable ways. One way around this is to make the overlays larger than the thumbnails themselves, and draw them in the background. I have yet to experiment with this technique, but it seems that it would resolve this particular problem.

Other problems the ICCB dataset revealed were relatively minor. Some users complained that the magnifier was too intrusive. Initial versions of the browser contained menu palettes in the Piccolo space, like the web UI image viewer. Users complained that those were often in the way as well, so I added a more traditional Swing menu bar and toolbar to each browser window instead. Current plans are to update the browser to handle even larger datasets, on the order of several thousand images.

5.2 A sample dataset of 5D images

I also tested the image browser on a much smaller dataset, containing just a few multidimensional 5D images. Figure 5.2 shows a screenshot of that dataset. My main conclusion from testing the browser on the 5D images is that the viewer does a better job of describing a multidimensional image than the browser currently does. Right now, the browser only shows a single image slice with a default configuration of channels. This is the thumbnail generated by OMEIS—which for a 5D image is a color-balanced projection of its middle Z slice and first timepoint. However, the region of interest for a 5D image may really be at a different timepoint or different Z slice. A user has to determine this with the viewer, and not the browser. One possible solution would be to support chimographs, or displaying whole stacks of time points and Z slices in a different browser window. Such a strategy, however, would rely of OMEIS to deliver thumbnails of arbitrary planes quickly, which it is not currently optimized to do.

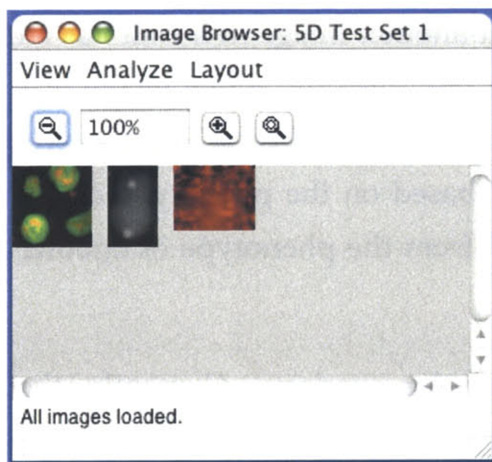


Figure 5.2. A small 5D image dataset.

The other problem revealed by 5D image datasets is with the heat map. Several important image statistics attributes, such as maximum intensity, image centroid, and mean intensity, are characteristics of stacks and planes within the image, and not the entire image. However, a thumbnail represents an entire image, and not a single plane or stack. The heat map tries to address this by allowing a user to pick how a value is assigned when multiple attributes of a specific type exist—as they surely will in the case of stack and plane-level attributes. However, this is really an oversimplification. There aren't many ways around this. One method that has been discussed is to use the image browser code as a template for a stack and plane browser within an image. Another is to allow the user to inspect different timepoints and Z slices in the magnifier. The latter makes the most sense, and will likely be a feature in a future version of the browser.

Despite these limitations, the browser is still useful for larger 5D datasets. The browser is still the only tool within OME that allows a user to annotate an image and view those annotations. The viewer may allow a user to better classify and characterize a 5D image, but the browser still allows the biologist to quickly detect, remember, and use that information.

CHAPTER 6
Conclusion

I have outlined and summarized an image browser for visualizing and managing large datasets of biological images. The features of this tool enable biologists to more quickly extract meaningful information from gigabytes of image data, through classification, annotation, and semantic overlay mechanisms. It includes a user interface that changes behavior based on the current level of detail and zooming factor on the dataset. Finally, it is designed to be adaptive, flexible, and efficient. However, the project is not completely finished. The image browser is in its first incarnation, and there is room for improvement and new features

6.1 Future work

The most important goal in the short-term is to get feedback from users about the image browser, and develop a list of new features for the tool. I recently presented the browser to the biologists working in SorgerLab, and they have already responded with constructive feedback. For example, one researcher suggested a layout method that would order thumbnails by a scalar variable—a graphing layout. Another suggested faded overlays for annotations in images in a well, or a multiple-page look for multiple images in a well. We will be meeting very soon with screeners at the Harvard ICCB to allow them to test the software, get feedback, and consider implementing their suggestions.

Further down the road, we have discussed making the image browser a more generic tool, developed separately from OME. Such a tool would use a flatfile or direct JDBC data connection to extract image data, and could be used to organize

and classify images on a remote or local filesystem. The act of browsing is independent from any paradigm in OME; it may prove useful for the browser to become independent of OME at some point as well.

Finally, the development of the image browser was rapid. Mistakes were inevitably made, and several features, particularly key events, were omitted. Once the entire OME package is released, I will include those omissions, and analyze the source to determine if and where the code should be refactored. All in all, my goal in the next few months (while I am still at MIT) is to improve the code, and leave an application that biologists will actually use. If my software enables a biologist to more effectively conduct research, then this project will be judged a success.

APPENDIX A

Code & Documentation URLs

Both the code for the image browser and its Javadoc documentation can be found on the OME project websites, or through anonymous CVS.

Source (view only):
http://cvs.openmicroscopy.org.uk/cvsweb/

Source (anonymous CVS- HOWTO):
```
% cvs -d ':pserver:anoncvs@cvs.openmicroscopy.org.uk:/ome' login
(Enter anoncvs for the password)
% cvs -z3 -d ':pserver:anoncvs@cvs.openmicroscopy.org.uk:/OME'
checkout OME
```

The browser code is located in the Shoola branch, under org/openmicroscopy/shoola/agents/browser.

Javadoc (OME client, including browser):
http://sorger-g51.mit.edu:8009/shoola/

Javadoc (OMEDS, OMEIS interfaces):
http://sorger-g51.mit.edu:8009/ome-java/

General project documentation:
http://docs.openmicroscopy.org.uk/

APPENDIX B
Complete Source File List

The following is a list of files written and created by the author for the purpose of creating the browser and for completing this Thesis. The root folder of these files from the main OME branch (see Appendix A for code access information) is Shoola/SRC/org/openmicroscopy/shoola/agents.

```
annotator/AnnotationCtrl.java
annotator/Annotator.java
annotator/DatasetAnnotationCtrl.java
annotator/ImageAnnotationCtrl.java
annotator/TextAnnotationUIF.java

annotator/events/AnnotateDataset.java
annotator/events/AnnotateImage.java
annotator/events/DatasetAnnotated.java
annotator/events/ImageAnnotated.java

browser/BrowserAgent.java
browser/BrowserController.java
browser/BrowserEnvironment.java
browser/BrowserManager.java
browser/BrowserMode.java
browser/BrowserModeClass.java
browser/BrowserModel.java
browser/BrowserModelAdapter.java
browser/BrowserModelListener.java
browser/BrowserTopModel.java
browser/BrowserTopModelListener.java
browser/IconManager.java

browser/colormap/ColorBoxLabel.java
browser/colormap/ColorCellRenderer.java
browser/colormap/ColorMapCategoryListener.java
browser/colormap/ColorMapDispatcher.java
```

```
browser/colormap/ColorMapGroupBar.java
browser/colormap/ColorMapGroupListener.java
browser/colormap/ColorMapList.java
browser/colormap/ColorMapListUI.java
browser/colormap/ColorMapManager.java
browser/colormap/ColorMapModel.java
browser/colormap/ColorMapModelListener.java
browser/colormap/ColorMapPMFactory.java
browser/colormap/ColorPair.java
browser/colormap/ColorPairFactory.java
browser/colormap/ColorPairModel.java

browser/datamodel/AttributeMap.java
browser/datamodel/CategoryComparator.java
browser/datamodel/CategoryTree.java
browser/datamodel/CompletePlate.java
browser/datamodel/DataElementType.java
browser/datamodel/PlateInfo.java
browser/datamodel/PlateInfoParser.java
browser/datamodel/ProgressListener.java
browser/datamodel/ProgressMessageFormatter.java

browser/events/AnnotateImageHandler.java
browser/events/BrowserAction.java
browser/events/BrowserActions.java
browser/events/CategoryChangeHandler.java
browser/events/ClassificationHandler.java
browser/events/CompositePiccoloAction.java
browser/events/MouseDownActions.java
browser/events/MouseDownSensitive.java
browser/events/MouseDragActions.java
browser/events/MouseDragSensitive.java
browser/events/MouseOverActions.java
browser/events/MouseOverSensitive.java
browser/events/PiccoloAction.java
browser/events/PiccoloActionFactory.java
browser/events/PiccoloActions.java
browser/events/PiccoloModifiers.java
browser/events/ReversibleBrowserAction.java
browser/events/ReversiblePiccoloAction.java

browser/heatmap/AbstractHeatMapMode.java
browser/heatmap/HeatMapDispatcher.java
browser/heatmap/HeatMapDTListener.java
browser/heatmap/HeatMapFilter.java
browser/heatmap/HeatMapGradient.java
browser/heatmap/HeatMapGradientUI.java
browser/heatmap/HeatMapManager.java
browser/heatmap/HeatMapMode.java
browser/heatmap/HeatMapModeBar.java
browser/heatmap/HeatMapModel.java
browser/heatmap/HeatMapModeListener.java
```

```
browser/heatmap/HeatMapModelListener.java
browser/heatmap/HeatMapModes.java
browser/heatmap/HeatMapPMFactory.java
browser/heatmap/HeatMapScaleBar.java
browser/heatmap/HeatMapStatus.java
browser/heatmap/HeatMapStatusUI.java
browser/heatmap/HeatMapTreeListener.java
browser/heatmap/HeatMapTreeRenderer.java
browser/heatmap/HeatMapTreeUI.java
browser/heatmap/HeatMapUI.java
browser/heatmap/HeatMapUtils.java
browser/heatmap/LinearScale.java
browser/heatmap/LogarithmicScale.java
browser/heatmap/RangeChecker.java
browser/heatmap/Scale.java
browser/heatmap/SemanticTypeTree.java

browser/images/AbstractOverlayMethod.java
browser/images/AbstractPaintMethod.java
browser/images/DrawStyle.java
browser/images/DrawStyles.java
browser/images/ImageAnnotationNode.java
browser/images/ImageAnnotationOverlay.java
browser/images/OverlayMethod.java
browser/images/OverlayMethods.java
browser/images/OverlayNodeDictionary.java
browser/images/PaintMethod.java
browser/images/PaintMethods.java
browser/images/PaintShapeGenerator.java
browser/images/ResponsiveNode.java
browser/images/Thumbnail.java
browser/images/ThumbnailDataModel.java
browser/images/ZoomDependentPaintMethod.java

browser/layout/AbstractOrderedLayoutMethod.java
browser/layout/AspectLayoutMethod.java
browser/layout/CategoryGroupingMethod.java
browser/layout/ConstrainedLayoutMethod.java
browser/layout/CriteriaGroupingMethod.java
browser/layout/FootprintAnalyzer.java
browser/layout/GroupingMethod.java
browser/layout/GroupModel.java
browser/layout/ImageIDComparator.java
browser/layout/LayoutComparator.java
browser/layout/LayoutMethod.java
browser/layout/MaxWidthLayoutMethod.java
browser/layout/NumColsLayoutMethod.java
browser/layout/PlateLayoutMethod.java
browser/layout/QuantumGroupLayoutMethod.java
browser/layout/QuantumTreemap.java (from HCIL)
browser/layout/SingleGroupingMethod.java
```

```
browser/ui/BrowserCamera.java
browser/ui/BrowserInternalFrame.java
browser/ui/BrowserMenuBar.java
browser/ui/BrowserView.java
browser/ui/BrowserViewEventDispatcher.java
browser/ui/BrowserViewListener.java
browser/ui/CameraListener.java
browser/ui/CategoryEventHandler.java
browser/ui/CategoryMenuFactory.java
browser/ui/HoverManager.java
browser/ui/HoverSensitive.java
browser/ui/NeighborFinder.java
browser/ui/PopupMenuFactory.java
browser/ui/RegionSensitive.java
browser/ui/SelectedRegion.java
browser/ui/SemanticZoomNode.java
browser/ui/StatusBar.java
browser/ui/UIWrapper.java
browser/ui/ZoomButtonPanel.java
browser/ui/ZoomParamListener.java

browser/util/Filter.java
browser/util/GrepOperator.java
browser/util/KillableThread.java
browser/util/MapOperator.java
browser/util/StringPainter.java

classifier/AttributeComparator.java
classifier/CategoryCtrl.java
classifier/CategoryEditUI.java
classifier/CategoryUI.java
classifier/Classifier.java

classifier/events/CategoriesChanged.java
classifier/events/ClassifyImage.java
classifier/events/ClassifyImages.java
classifier/events/ImagesClassified.java
classifier/events/LoadCategories.java
classifier/events/ReclassifyImage.java
classifier/events/ReclassifyImages.java
```