

A Modular Architecture for Client-Based Analysis of Biological Microscopy Images

by

Sheldon Y. Chan

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

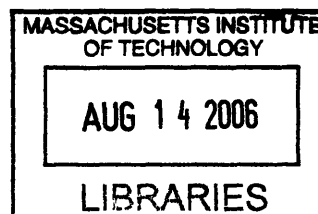
[June 2006]
May 26, 2006

© 2006 Massachusetts Institute of Technology. All rights reserved.

Author.....
Department of Electrical Engineering and Computer Science
May 12, 2006

Certified by.....
Peter K. Sorger
Professor
Department of Biology
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



ARCHIVES

A Modular Architecture for Client-Based Analysis of Biological Microscopy Images

by

Sheldon Y. Chan

Submitted to the
Department of Electrical Engineering and Computer Science

May 12, 2006

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The modular client architecture created for the Open Microscopy Environment (OME) enables developers to easily integrate client-based analyses into the experimental workflow typical of biological imaging. This architecture utilizes a componentized, pluggable framework to provide ease of integration and scalability while lowering the total cost of ownership for the OME client. The application programming interface (API) for connecting external analyses is designed within this modular architecture taking advantage of declarative plug-in extensions to automatically recognize new analyses. Ease of integration with the OME client allows users to analyze multi-dimensional images with a repertoire of analyses and persist derived data into OME.

Thesis supervisor:

Peter K. Sorger
Professor
Department of Biology

Acknowledgements

The Open Microscopy Environment is a joint effort between the Sorger Lab in MIT's Department of Biology, the Swedlow Lab in the Wellcome Trust Biocentre at the University of Dundee, Scotland, the Image Informatics and Computational Biology Unit at the National Institute of Health in Baltimore, and the Laboratory for Optical and Computational Instrumentation at the University of Wisconsin-Madison.

I could not have complete this thesis without the technical help and advice of Peter Sorger, Jeremy Muhlich, Tony Scelfo, Erik Brauner, Jason Swedlow, Chris Allan, Jean-Marie Burel, Joshua Moore, Josiah Johnston, Harry Hochheiser, Anne Carpenter and Michael Lamprecht.

I would also like to thank Melissa Chan, Yvonne Chan, David Jackson, Bill Fienup, Johnny Boy, Brock Arnold, Jon Salz, Darcy Kelly, and Laura Snow for putting up with me through this project.

This project is dedicated to my parents, Cheuk and Wing Chan.

The work described in this thesis was supported by MIT CDP grant #P-50-GM68762 and ICBP grant #5-U54-CA112967-02.

Table of Contents

1 Introduction	13
1.1 Open Microscopy Environment.....	14
1.1.1 OME Data Model.....	17
1.1.2 Server-Side Analysis Modules.....	17
1.2 User requirements	18
1.3 Project evolution	18
1.4 Software Dependencies.....	19
1.4.1 Eclipse	19
1.4.2 CellProfiler	20
1.5 Workflow.....	21
2 Componentization.....	24
2.1 Current Architectural Overview	25
2.2 Goals & Constraints.....	26
2.4 System Componentization.....	27
2.4.1 Physical Repackaging.....	28
2.4.2 Modified Initialization Sequence.....	30
2.4.3 Component Registration.....	31
2.4.4 Resource Location.....	32
2.4.5 AWT Interoperability.....	33
2.5 Summary	33
3 External Analysis.....	36
3.1 Overview	36
3.2 Design Considerations	38
3.3 External Analysis.....	40
3.3.1 Shoola Component Awareness	40
3.3.2 Message Event Handling.....	42
3.3.3 Shoola Abstraction.....	42
3.3.4 Local Image Management	43
3.3.5 Analysis Plug-in Extension Points.....	44
3.3.6 Managing Third-Party Analyses	46
3.3.7 Executing Analyses.....	46
3.3.8 Threading Model & Monitoring Progress	47
3.4 Storing Annotation Data	48

3.4.1 Design Considerations	48
3.4.2 Remote Data Storage.....	51
3.4.3 Local Data Management.....	53
3.5 CellProfiler	55
3.6 User Interface	57
3.6.1 Model, Listeners, and AnalysisBrowser Classes	58
3.6.2 Analysis Selection	58
3.6.3 Settings Modification	60
3.6.4 Monitoring Progress.....	61
3.6.5 Viewing History & Importing Data	63
4 Data Visualization & Manipulation	66
4.1 Data Visualization.....	66
4.2 Time-Series Analysis and Cell Tracking	68
5 Conclusion.....	71
5.1 Workflow.....	71
5.2 Future Work.....	73
Appendix A: Source Code & Documentation.....	75

List of Figures

FIGURE 1. The three-tiered software architecture for the Open Microscopy Environment (OME) begins with the collection of images and experimental meta-data. This image and meta-data is then stored remotely on a Perl server that is accessible via a Java-based software client.	16
FIGURE 2. This software workflow summarizes some of the functionality that my thesis project worked to provide while simultaneously satisfying all software requirements. This includes being able to easily retrieve images, run third-party analyses, store the data locally or remotely, and render or analyze analysis data.	22
FIGURE 3. The Java-based OME client (Shoola) was originally architected as a set of loose Java packages that were roughly organized into logical components.	24
FIGURE 4. Components are shown here logically clustered into functional groups and loosely layered according to component inter-dependencies.	28
FIGURE 5. The inter-dependencies of individual components are shown in higher detail in this plug-in dependency diagram. The third-party and core dependencies are at the bottom with increasing business logic for Shoola layered on top.	30
FIGURE 6. Individual components, or “agents,” are declared in the container.xml file. This example shows the Viewer component being identified to Shoola, the main Java class implementation for the component, and an XML file containing optional parameters for the Viewer agent.	31
FIGURE 7. Resource location within the Eclipse environment requires overcoming boundary conditions imposed when components are loaded in separate classloaders. This diagram illustrates the pathways traversed when a utility class in the core Shoola component is called by the DataManager to convert images into icons. The IconManager needs to locate the resources within the DataManager’s classloader.	32
FIGURE 8. The external analysis component’s UML class diagram shows the relationships between classes for the logic to recognize new analyses, store data, and render a user interface. This class diagram also includes the relationship to the CellProfiler external analysis.	37
FIGURE 9. This is an isolated class diagram of the org.openmicroscopy.shoola.analysis plug-in, that provides image retrieval, management of third-party analyses, and historical logging of previous analyses run on a client.	40
FIGURE 10. This class inheritance diagram for the “agent” class in the analysis plug-in shows the simplicity of implementing a new Shoola component.	41

FIGURE 11. The ImageManager serves as the actual implementation class that arbitrates between requests made by analyses through the AnalysisManager API and the OME server. Retrieved images are stored in a local image repository on the client, allowing for analyses to subsequently run on the saved images.	44
Figure 12. XML for the CellProfiler analysis plug-in's extension to the Analysis plug-in's extension-point. The Analysis plug-in has previously defined a set of fields that it requires and this CellProfiler plug-in simply provides the requested information.	45
FIGURE 13. Three scenarios for data storage were considered. In (A) analysis data would be transient from one run to another. (B) Every piece of analysis data is stored on the remote server. (C) Hybrid storage structure would store a copy of everything locally, but allow selective data importing to the OME server.	49
FIGURE 14. This XML semantic type definition defines a CellArea to the server to be a feature that contains a tag and floating point value.	51
FIGURE 15. This shows the XML definition for the CellProfiler external analysis module that establishes proper semantic data input and output. This definition on the server allows for future analysis data to be properly attributed to this module on the OME server.	52
FIGURE 16. This is sample spreadsheet data suitable for the spreadsheet importer. There are five CellArea data points for the image with an ID of 5. The Tag column references each distinct cell object and the actual cell area can be found in the last column.	53
FIGURE 17. The local Datastore class diagram expresses the relationships for classes that are predominantly used to parse and retrieve data saved from analyses.	54
FIGURE 18. The CellProfiler plug-in and its simple class interactions with the Matlab plug-in are shown in this figure.	55
FIGURE 19. This class diagram for the user interface shows the relationships between classes. The AnalysisBrowserModel is the model, panels are views, and listeners act as controllers as per the Model-View-Controller (MVC) paradigm.	57
FIGURE 20. A screenshot of the analysis selection screen shows the meta-data for the selected image or data set. Installed and available analyses are listed along with potential list of sub-analyses in a tree. Descriptions of analyses are provided by third-party plug-ins.	59
FIGURE 21. A screenshot of the analysis settings screen shows available channels of the current selection. Drop-downs enable a user to match channels with analysis inputs.	61
FIGURE 22. A screenshot of the status screen shows comprehensive progress information and a meter for a measure of completion.	62
FIGURE 23. The history screen shows a list of historical analyses performed by this client along with relevant time-stamp, analysis, input, and a path to output data.	63
FIGURE 24. The LoViewer is a data visualization tool built to extract and screen OME data for interesting trends and relationships. This screenshot demonstrates its visualization capabilities on data that was saved by a CellProfiler external analysis.	67
FIGURE 25. This figure shows a time-series graph tracking cell division in a movie. Each individual circle indicates the frame and the pixel location of the cell. The solidity of the lines indicate the probability of the connection.	69

FIGURE 26. This figure revisits the software workflow summarizing some of the functionality that was implemented by my thesis project. Pathway 1 indicates image download from the OME server, 2 shows analysis results from an external analysis, 3 and 4, respectively, show remote and local data storage of analysis data. Pathway 5 shows LoViewer visualization of local or remote data stored, and 6 shows data manipulation using tracking algorithms.....72

CHAPTER 1

Introduction

Technology for quantitative image analysis has made significant strides over the past decade; however, software-based information management and total workflow integration has lagged significantly behind. This has left biologists with the ability to perform accurate measurements on microscope images while forcing them to continue to keep their data in spreadsheets or scribbled inside notebooks. This ad-hoc management of data disassociates it from the original images, makes it difficult to search, and magnifies the task of finding data relevant to the research being done. Software to manage this image workflow, perform advanced image analysis, and handle the results in a consistent manner, has fallen behind the pace of software for extracting values from individual images.

The typical biological workflow involves preparing a sample, collecting images, performing image analysis, and recording the data from the analysis. Each of these respective tasks requires keeping track of experimental metadata, information about the microscope optics, information about the analysis algorithms, and the image analysis data. This biological workflow requires a software architecture that supports advanced image analysis tools and linking images, meta-data, and analysis results. Such an architecture also needs to provide a way to access and view that data in a more effective manner than by current ad-hoc methods. In this way, a biologist could store microscopy images and utilize third-party image analysis software while maintaining the relationship between data and images. Additionally, this analysis data would be easily accessible and

visualizable. With such a system, a microscopist would not have to manually link images with image analysis data, and high content screening would be simplified.

My thesis has been to develop an architecture that will support the described workflow and implement it as software components. This architecture is embedded within the Open Microscopy Environment (OME) client to provide a complete solution to managing data, analysis tools, and analysis data. It allows a biologist to retrieve images from a central repository, run analysis tools on those images, and then store the results back into the OME data repository. This data is easily accessible and can either be manipulated using data-mining algorithms or visualized to provide a quick means to screen analysis data.

This document provides the design and implementation of this modular external analysis architecture for image analysis. Chapter two provides an in depth look at the task of componentizing the existing OME architecture. Chapter three presents a complete look at designing and implementing support for external analysis, along with an actual sample analysis application. Chapter four looks at some of the real applications of this architecture. Finally, chapter five reviews how the architecture described by this thesis supports the workflow described earlier and considers some future areas of development.

1.1 Open Microscopy Environment

The Open Microscopy Environment¹ (OME) is an open-source software project that was started at MIT to aid in the quantitative analysis of biological images through a database-driven system. Currently, this project is a joint effort between groups at the Massachusetts Institute of Technology, the Wellcome Trust Centre at the University of Dundee, the National Institute of Health (NIH) in Baltimore, and the University of Wisconsin at Madison. The current implementation of OME is being developed by this international consortium to solve issues of information loss associated with manual image management and analysis, and to provide an effective workflow for biologists. However, there are significant challenges when dealing with the rapidly developing areas of bioinformatics and microscopy. Idiosyncratic requirements driven by continually

¹ Open Microscopy Environment. <http://www.openmicroscopy.org.uk>.

evolving biological semantics and experimental details are demanding increased flexibility from bioinformatics software. While the number of file formats and data ontologies for image analysis applications continue to expand, very little effort has been vested in integrating these applications.

The primary focus of OME has been to develop software and protocols to store image data in a common ontology while preserving the meta-data specific to an experiment, equipment, or software used to process the images². Meta-data includes information such as magnification of the optics, the set of filters employed, and even the model of the microscope where the images were collected. The common set of semantics that is used by OME have been derived from existing ontologies including the Medical Subject Headings (MeSH), the Microarray Gene Expression Data Society (MGED), and the Minimal Information About a Microarray Experiment (MIAME) effort³. When images are collected using modern microscopes and stored in OME, meta-data remains associated with the images and is not lost to a researcher's notes. A secondary focus of OME has been to leverage these OME data representation semantics in its server-side image analysis engine. The server allows analysis modules to communicate with each other in a data-centric manner. The development of a universal language for storing multi-dimensional microscopy images and associated meta-data¹, provides a common way to read and write data regardless of which analytic module is used. As a result, analytic modules can be combined into chains to perform automated multi-parametric analysis of a series of images.

The OME system is implemented as a three-tiered architecture (FIGURE 1) with a remotely networked Perl server and a local Java client (Shoola). The server is restricted to running on Unix- or BSD-based operating systems, whereas the client is supported on most Java-friendly systems. The server is responsible for all image and data warehousing

² I. G. Goldberg, C. Allan, J. Burel, D. Creager, A. Falconi, H. Hochheiser, J. Johnston, J. Mellen, P. K. Sorger and J. R. Swedlow, "The Open Microscopy Environment (OME) Data Model and XML file: open tools for informatics and quantitative analysis in biological imaging," *Genome Biology*, vol. 6, no. 5, pp. R47.1-R47.13, 2005.

³ J. R. Swedlow, I. Goldberg, E. Brauner, P. K. Sorger, "Informatics and Quantitative Analysis in Biological Imaging," *Science*, Apr., p. 100-102, 2003.

on the OME system. In particular, images imported from the microscope are stored into the image server (OMEIS) and analysis or associative data is stored in the data server (OMEDS). The server also provides functionality to run Matlab analyses on imported images. The Java client is primarily limited to viewing images and has some ability to annotate images with additional information.

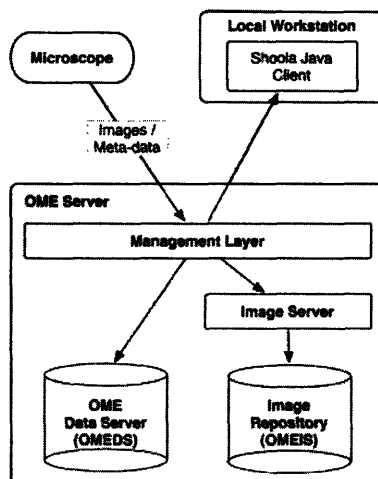


FIGURE 1. The three-tiered software architecture for the Open Microscopy Environment (OME) begins with the collection of images and experimental meta-data. This image and meta-data is then stored remotely on a Perl server that is accessible via a Java-based software client.

Within this tiered architecture, the Open Microscopy Environment's built-in image analysis engine has the ability to run a series of server-side quantitative analyses on large sets of images and store analysis results within the common ontology; however, the current implementation of OME can only leverage analysis applications and image tools designed to process biological images if they are available on the same Unix platforms that are supported by the server. Unfortunately, integrating analyses with the server is a complex process that has a steep learning curve, and even were integrating analyses with the server straightforward it is not an option for software on the Windows operating systems, which predominates in commercial image analysis applications. This thesis will cover my alternative to this server-side analysis through the creation of a client-based analysis architecture that is capable of interacting with third-party analyses. The Shoola client is not without its own shortcomings, but the client provided the best starting point

to fill the gap between OME and existing academic and commercial analysis packages, to easily perform research specific analyses on a personal workstation.

1.1.1 OME Data Model

The OME data model is based on the principle of strongly typing data to semantic type objects. These semantic types are simply a name or label given to a piece of information to describe it on some particular level – global, dataset, image, or feature. Globally defined types are used describe data that can apply to an entire biological experiment. Dataset- and image-level types refer to data that can only be applied to dataset and image-wide characteristics. Feature-level types apply to data regarding objects within an image. Regardless of the level, these semantic types remain universally and uniquely interpretable in OME after they are defined. For example, an `ImageCellCount` is a semantic type that is used to describe an image level feature of how many cells are contained within an image. Any data that is typed as an `ImageCellCount` will always be interpreted by OME to mean the number of cells contained in the associated image.

This original data model offers the flexibility for supporting new types that may not be known *a priori*, but it requires careful and complete declaration of types when you are ready to store data. This hierarchical model for representing data provides a direct correspondence with experimental biological data in a human-readable form. These types are user defined by an XML file that needs to be imported onto the server before they can be used. The names for new semantic types must be unique from existing server semantic types to prevent a collision in namespace. This is why it is critical to have a clear definition of types the first time they are declared to the server.

1.1.2 Server-Side Analysis Modules

As a part of the server-side analysis chains, modules can be defined on the server to have a particular set of inputs and outputs. For example, the `StackStatistics` module takes newly imported files as an input and outputs a set of statistical measurements on the images. Associating input images with data, and linking to the analysis module used to define how the data was derived, is crucial in supporting data provenance with the

architecture described by this thesis. Data provenance is particularly important with image analysis since data such as cell location in an image can differ depending on the exact algorithm applied. Modules can be declared on the server but the actual analysis does not have to be run as part of the server analysis chains. This will allow us to associate data from external analysis tools with the original images while maintaining data provenance.

1.2 User requirements

There are two use cases for performing analysis in the Open Microscopy Environment. The first is to run analyses within a server-centric system as modeled by the original server implementation of OME. In this case, a user can run a pre-defined set of analyses on a large set of images. A second case involves being able to efficiently run a diverse set of analyses, including research-specific code and client-side commercial applications, on small sets of images. Both of these scenarios demand that the means to integrate these different types of analyses are efficient, simple to implement, extensible, and organized within a data-model.

What follows from these use cases are a set of requirements for my project and OME system to be able to interact with third-party analyses in an extensible manner. This implies that the work needed to integrate new analysis modules must be low and that the system should scale with the addition of more analyses. Secondly, these analyses should be able to run on more than just the platforms supported by the server. Finally, the data that is generated by these analyses should be storable in an OME-compatible form, such that it is viewable and manageable by any OME-aware application.

1.3 Project evolution

This project began in July 2005, following lengthy discussions with Erik Brauner of the Sorger Lab and Jason Swedlow of the Swedlow Lab. In September of 2005, I visited the University of Dundee for an OME conference, where the design and feature set for an external analysis architecture on the client was solidified. Through the end of December until February, discussions with Zachary Pincus of the Theriot Lab and members of the

Sorger Lab reinforced the fundamental need for a client-side analysis architecture. Finally, a programming interface to the OME server was created in December 2005 and a full analysis architecture was completed the following March.

1.4 Software Dependencies

Two software packages, Eclipse and CellProfiler, have been used in the course of the work for this thesis to either support the architecture or to demonstrate how this architecture will work in a production environment. Eclipse was used to lay the support for the client architecture and CellProfiler was used to demonstrate how typical image analysis software fits into this architecture.

1.4.1 Eclipse

Componentized architectures are often applied to software systems to improve the ability to easily replace failing components, enhance or modify the functionality of a system, or increase the flexibility for developing new components⁴. Existing application frameworks for creating component-based applications, such as Apache Struts⁵ are readily available to help server-based applications achieve a componentized architecture. Similarly, plug-in based technologies such as Eclipse⁶ aim to make componentized and extensible, client-based applications easier to create.

At the heart of Eclipse are two fundamental technologies – OSGi⁷ bundles and plug-in extension-points. OSGi is a specification that outlines a set of standards for a component-oriented computing environment. The Eclipse implementation of an embedded OSGi microkernel provides a way to control the lifecycle – installation, activation, execution, and shutdown – of individual bundles, or plug-in components. The second technology

⁴ R. Seacord and L. Wrage, "Replaceable Components and the Service Provider Interface," [Online document], 2002 Jul, [cited 2005 Dec 01], Available HTTP:

<http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tn009.pdf>

⁵ Apache Struts. <http://struts.apache.org/>

⁶ Eclipse. <http://www.eclipse.org>

⁷ OSGi. <http://OSGi.org/>

comprises plug-in extension-points⁸ which form an interlocking layer for components. This layer is the glue that allows individual plug-in bundles to extend the functionality of other bundles.

In addition, the Eclipse framework surrounding extension points provides plug-in bundles with the ability to be dynamically aware of other components. Components can therefore recognize and utilize other components at will. Another aspect of the Eclipse framework is that all plug-ins operate under a lazy load paradigm in which plug-in bundles are activated on an as-needed basis. This principle of lazy loading means that dependencies are only loaded into memory when they are explicitly referenced or instantiated. The delay in loading dependencies prevents unreferenced or unused objects from even being loaded,⁹ potentially improving memory consumption and startup performance.

The OME Java client when I started this project was comprised of a set of logical components that were loosely packaged into “agents.” This layout formed the basis of a package-level componentization, but failed to provide the benefits of a fully componentized architecture and therefore made adding new functionality difficult. Additionally, the existing architecture had neither a clearly defined process for creating new agent components, nor a well-defined application program interface (API) to facilitate component replacement. This thesis project leveraged Eclipse to satisfy some of the extensibility requirements that were outlined in Section 1.2.

1.4.2 CellProfiler

A large number of academic and commercial image analysis applications exist. CellProfiler¹⁰ is an open-source image analysis project spearheaded by Anne Carpenter of the Whitehead Institute that runs a series of Matlab analyses on microscopy images and generates quantitative data about cell morphologies. Some other common analysis tools

⁸ A. Bolour, “Notes on the Eclipse Plug-in Architecture,” 2003 July, [cited 2005 Dec 01], Available HTTP: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

⁹ “Lazy Loading of Dynamic Dependencies,” [cited 2005 Dec 01], Available HTTP: <http://docs.sun.com/app/docs/doc/817-1984/6mhm7pl1h?a=view>

¹⁰ CellProfiler. <http://groups.csail.mit.edu/vision/cellprofiler/>

include Definiens Cellenger¹¹, Bitplane Imaris¹² and Metamorph.¹³ This particular project focused on the use of CellProfiler to demonstrate the functionality of the external analysis architecture developed through the course of this thesis.

My selection of CellProfiler as a reference image analysis engine was based on two criteria. First, it uses Matlab for mathematical processing and Matlab is one of the most accessible mathematical tools. Thus, establishing clear support for Matlab was a prudent course. That would ensure support for any future Matlab-based analyses or custom Matlab scripts outside of CellProfiler. Secondly, the availability of Anne Carpenter's development team for support throughout this process made CellProfiler an excellent candidate for integration.

1.5 Workflow

This thesis project was organized into two main tasks – componentization and development of an API for external analysis. These tasks collectively addressed the requirements established for this project in Section 1.2. The task of componentization helps to provide a modular architecture for client-side external analysis, and promotes extensibility and scalability. Secondly, the framework for an external analysis architecture supports integration of third-party analyses into the OME architecture. Finally, embedding this new architecture into the Java client supports a larger number of platforms than the existing server-side analysis infrastructure. FIGURE 2 provides a workflow that served as the fundamental goal for this thesis. Satisfying the requirements that have been established and supporting the features outlined ensured that such a workflow is possible.

¹¹ Definiens Cellenger. <http://www.definiens.com/products/cellenger.php>

¹² Bitplane Imaris. http://www.bitplane.com/products/imiris/imiris_product.shtml

¹³ Metamorph. <http://www.moleculardevices.com/pages/software/metamorph.html>

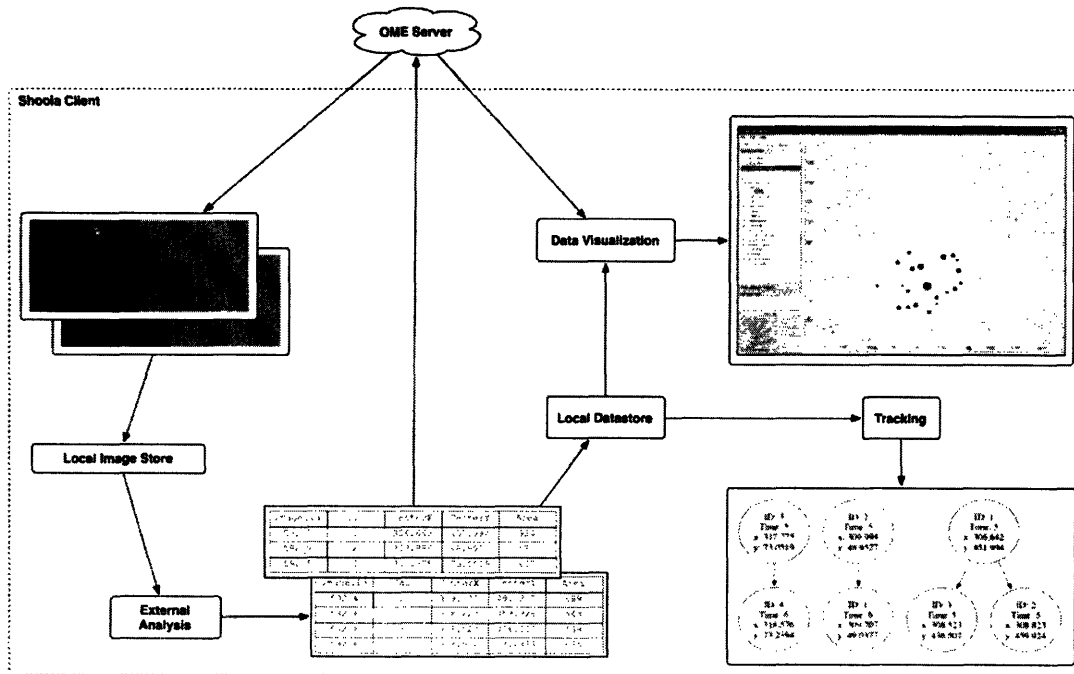


FIGURE 2. This software workflow summarizes some of the functionality that my thesis project worked to provide while simultaneously satisfying all software requirements. This includes being able to easily retrieve images, run third-party analyses, store the data locally or remotely, and render or analyze analysis data.

This workflow begins with the Shoola client retrieving the planes for multi-dimensional images from the server. The client will be able to store these images locally. In turn, with an external analysis system, these images can be analyzed for relevant feature data by a third-party tool to produce data that can be stored either on the server or locally by Shoola. Finally, the data is easy to visualize and manipulate. This thesis will outline the basic elements to promote this workflow to satisfy our goals, while simultaneously ensuring all our requirements are satisfied.

CHAPTER 2

Componentization

This chapter provides a look into the original Shoola client architecture and analyzes how it could be augmented to support an extensible architecture by architectural re-componentization using the Eclipse platform. Particular obstacles encountered in the process of componentization will be discussed, and design decisions will be reviewed for satisfying the software requirements established in Section 1.2.

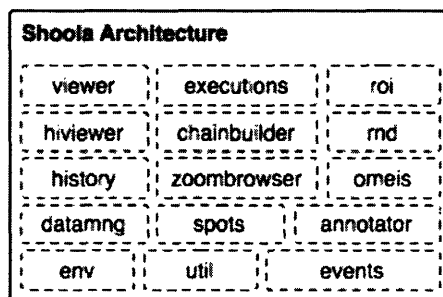


FIGURE 3. The Java-based OME client (Shoola) was originally architected as a set of loose Java packages that were roughly organized into logical components.

A clear separation between individual components adds flexibility to the client and allows for a means to upgrade or extend the client by easily adding, removing, or modifying individual bundles. An example of the flexibility of such an architecture is support for multiple image browsers with differing functionality. The original Shoola client has a single image browser (*hiviewer*) that is tightly coupled with the rest of the system. If we wanted to integrate a custom browser designed specifically to view images of plates, a device that contains many biological samples that are closely positioned relative to each other, a developer would have to augment the existing browser with the

new features. However, under a pluggable and componentized architecture based on Eclipse and OSGi, and a well-defined API, such a component could be developed separately and automatically recognized when it is installed with the Shoola client.

2.1 Current Architectural Overview

The Shoola client is a collection of package-level “agents” that are logically organized within Java package namespaces. For example, the image viewing components of Shoola are located in a series of packages under `org.openmicroscopy.shoola.agents.viewer`. Similarly, the hierarchal viewer used to view thumbnails of a collection of images is found under `org.openmicroscopy.shoola.agents.hiviewer`. This introduces the first of two issues that arise from the existing architecture. Despite intending to be component-based, the Shoola architecture depends on the developer to appropriately organize all of their components into a logical hierarchy.

This package-level organization is purely by convention since a developer can arbitrarily opt to use any package for an extension. This loose organization of code within specific packages may help to identify logical components, but without a stricter enforcement of component boundaries or a formal interface between components, the responsibility falls on future developers to determine whether or not code from a particular package is relevant to their needs.

This leads to the second problem of global access or awareness of a particular class even if the class was not meant to be accessible outside of its parent component. Java class and method scoping can address some of these issues, but fails if a component spans multiple packages, as is the case with Shoola. This is very apparent within the original version of Shoola, for example, where the Data Manager component contains ten separate packages. There is no way for the Data Manager to exercise complete access control over its classes that span these ten packages. FIGURE 3 attempts to show that the current architecture is composed of a collection of coded components, however, interactions between components are not clearly delineated. Moreover, the code does not prevent one

component like the `viewer` from seeing or interacting with the internal code for something like the `env` package, which manages user and session state with the server.

The first issue of determining where code belongs can be addressed by supplementary documentation by the developer. However, in consideration of new components, such as external analysis and data visualization tools, the need for a simple means to work on individual components rather than the entire client at one time provides a significant benefit. This is beneficial if new components don't require changes to existing components and can thus be inserted into a running version of the client. Additionally, isolation between components can provide better performance by allowing individual components to be loaded on demand rather than forcing everything to load on startup.

2.2 Goals & Constraints

Based on problems with the original OME client, relative to the requirements set forth in chapter one, there are three primary goals for system re-componentization. The first and most important directive for long-term and large-project extensibility is to clearly define a layered architecture within the Shoola client. This can only be achieved if a directed acyclic graph of component dependencies can be drawn between all logical components. This follows that there is also strict dependency enforcement between logical components. Otherwise, future developers are prone to violate the dependencies, reverting the client to a brittle non-hierarchical.

The second goal of componentization is to allow the independent development of self-contained components. This independence implies that the addition of new components merely requires appending them to the directed acyclic dependency graph, and will not require a full recompilation unless something within that specific component's dependency tree has changed. This also allows for individually compiled components to be dynamically inserted into an installed client that has compatible base components. For example, in this model, the multiple packages that make up the Data Manager component are fully self-contained, with their dependencies on other components clearly defined.

Finally, the last goal of componentization is to provide an architecture with an extension mechanism that supports better resource management. A declarative extension mechanism would allow components to add functionality to each other without requiring compilation or direct access to internal classes. For example, components could contribute menu entries to the Data Manager through extension points, and the Data Manager would not actually need to be aware of these components until runtime. A plug-in model supporting lazy-loading of an unlimited number of individual plug-in bundles on-demand provides for a scalable architecture upon which to build a complex client.

These goals, however, have several additional constraints beyond the requirements already outlined in this thesis that must be imposed in order to produce a successful product. A client with a reworked architecture demonstrating full componentization must provide as much functionality as the original client, if not more. This means that all components need to exhibit the same user-visible behavior as they did previously. The case for system componentization is weakened if the resulting client does not have comparable functionality, resource (disk and memory) utilization, and performance characteristics, since re-componentization does not produce visible changes to an end-user.

2.4 System Componentization

To re-componentize Shoola, I reconciled the previous package-level componentization model of the OME Java client with the replaceable component model to determine how to transform the former into the latter. The package-level componentization appeared to satisfy our requirements outlined in Section 1.2 at first glance, however, under scrutiny the dependencies between packages were more tightly coupled than expected. I leveraged the Eclipse framework to recast “agent” components as Eclipse plug-ins in order to create a more versatile client. As part of this process, I defined component dependencies and removed circular dependencies that can restrict extensibility.

2.4.1 Physical Repackaging

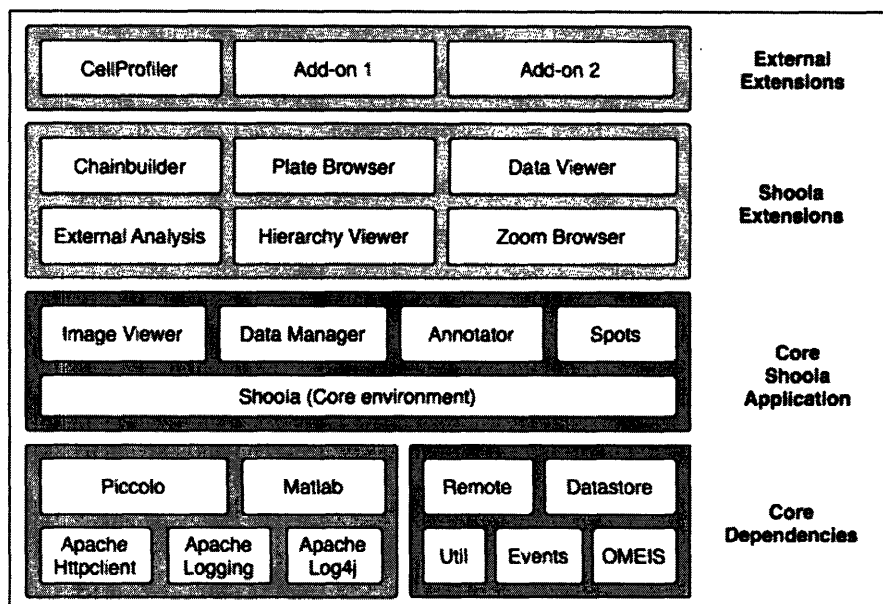


FIGURE 4. Components are shown here logically clustered into functional groups and loosely layered according to component inter-dependencies.

The first and most significant step in componentization was the physical re-packaging of loosely delineated agents into more strictly defined components. The dependencies on third-party applications such as Apache utilities – HttpClient and Logging – were easily bundled into their own plug-ins. Most of the package namespaces were preserved by packaging corresponding packages into plug-ins of the same name. Each plug-in is usually defined by three main characteristics: one or more Java packages containing component source code, a single Java class defining the plug-in bundle activator to manage the plug-in’s lifecycle, and a manifest file defining this plug-in’s supported inter-dependencies. An optional XML file is used to define and declare extension points that allow individual bundles to extend each other. The resulting componentization can be seen in FIGURE 4, with a full plug-in dependency diagram in FIGURE 5. Each individual block in FIGURE 4 equivalently represents both a self-contained logical component and corresponding Eclipse plug-in. Since each plug-in in Eclipse makes a literal declaration of all of its requisite dependencies and exportable packages, this allows us to easily create and enforce a directed acyclic dependency graph among components.

The layering of components in FIGURE 4 corresponds to the direction of the directed acyclic dependency graph between components. On the bottom-most tier, core dependencies are divided into two sections – third party dependencies from Apache and other software packages not developed by the OME development team, and fairly generic components that can be used by future Shoola components, but were developed with OME in mind. Above this layer reside the core components of the Shoola application. This layer contains the logic for the initialization of additional extensions, and provides the most primitive functionality for viewing images and managing data sets.

The Shoola Extensions layer provides some added functionality to the basic Shoola components. This is also the layer in which the external analysis components and various data visualization tools exist. Finally, at the top of the entire diagram are a series of extensions upon plug-ins or other add-ons that can build off of underlying layers of functionality. The crux of this diagram and of this repackaging is that each layer is only dependent on all of the layers below it to function correctly. Components can be removed from higher layers without affecting anything below it, and more importantly can be easily added to higher levels.

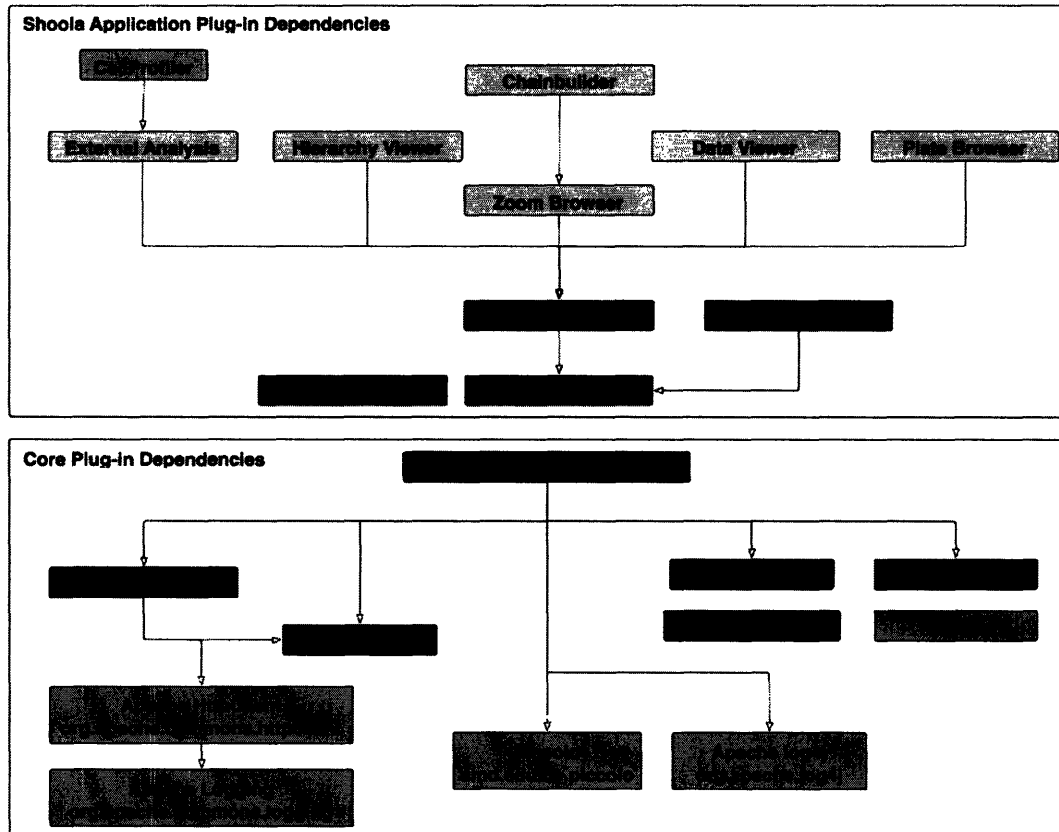


FIGURE 5. The inter-dependencies of individual components are shown in higher detail in this plug-in dependency diagram. The third-party and core dependencies are at the bottom with increasing business logic for Shoola layered on top.

An example of this characteristic of understanding dependencies can be seen in FIGURE 5 outlining the Hierarchy Viewer. From this graph, we can see that it depends on the Data Manager, the Annotator, and presumably most of the components in the core dependencies. However, if we wanted to make a change to the Chainbuilder, we would not have to touch the Hierarchy Viewer. In contrast, making changes in the Data Manager could force a cascading set of changes to all dependent plug-ins.

2.4.2 Modified Initialization Sequence

The restructuring of the original Shoola client to take advantage of the Eclipse plug-in architecture required specific modifications to the client initialization sequence. The original bootstrap sequence for Shoola started with a main class that spawned a thread

from which the rest of the client would run and then promptly exit. This design, originally intended to allow for the grouping of threads, caused a failure when trying to bootstrap within the Eclipse framework. When Shoola's parent thread exits and orphans its child process, Eclipse assumes Shoola has quit based on the death of the parent thread that it contains. In turn, this causes Eclipse to exit rather than re-parent the underlying child. This process of creating a sub-thread on startup was eliminated and Eclipse was allowed to bootstrap the Shoola client by creating an Eclipse Rich Client application. This newly created Eclipse application replaces the calls that were used by the original startup thread, allowing Shoola to be bootstrapped completely by Eclipse and supported by its dynamic plug-in bundle loading capabilities.

2.4.3 Component Registration

Under the original Shoola client, adding new agents, or components, involved modifying of a single XML file (`container.xml`) to append an entry with the following fields: the name of the agent, the class name of its implementation of the `Agent` interface, and the name of an XML file containing preferences and information for resource location. For example, declaring the viewer as a component in Shoola required the addition of the XML snippet in FIGURE 6.

```
<agent>
  <name>Viewer</name>
  <class>org.openmicroscopy.shoola.agents.viewer.Viewer</class>
  <config>viewer.xml</config>
</agent>
```

FIGURE 6. Individual components, or “agents,” are declared in the `container.xml` file. This example shows the Viewer component being identified to Shoola, the main Java class implementation for the component, and an XML file containing optional parameters for the Viewer agent.

This single file contained the information for every component added, and was centrally located, such that it was globally visible and modifiable.

To accommodate the addition of new components, the former method of registering “agents” was augmented to take advantage of the Eclipse plug-in extension-point mechanism. The `org.openmicroscopy.shoola` plug-in now provides an extension-point that accepts declarations of agents. The arguments are a name, the agent interface

implementation, and an optional XML preferences file, corresponding to the fields of the original `container.xml`. However, these declarations are now all self-contained within the declaring plug-in. For example, the viewer plug-in can now declare the same information to the `org.openmicroscopy.shoola` plug-in, and this information does not need to be manually appended to a single XML file.

2.4.4 Resource Location

The original design for Shoola made resource location relatively easy since all classes for the entire application were loaded into a single classloader, classes and resources could easily find each other. However, the Eclipse plug-in architecture adds a level of complexity, when it comes to locating specific resources across components, because each individual plug-in is loaded within its own plug-in classloader. With n distinct classloaders for each individual component plug-in, and a strict division between plug-in classloaders, this poses a problem when a class needs to find a resource in a foreign classloader.

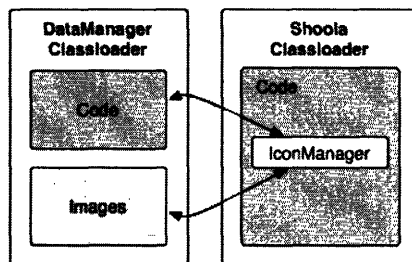


FIGURE 7. Resource location within the Eclipse environment requires overcoming boundary conditions imposed when components are loaded in separate classloaders. This diagram illustrates the pathways traversed when a utility class in the core Shoola component is called by the DataManager to convert images into icons. The IconManager needs to locate the resources within the DataManager's classloader.

This problem arises in cases such as the `IconManager` implementation offered by the `org.openmicroscopy.shoola` plug-in to load icon files that are located in the respective components' namespaces. For example, the Data Manager component uses this `IconManager` class to retrieve icons that are actually stored within the Data Manager plug-in. However, FIGURE 7 illustrates the problem with this approach since the actual code for retrieving the icons is contained in the `org.openmicroscopy.shoola` plug-in

classloader and the desired resources are located in the `org.openmicroscopy.shoola.datamng` plug-in classloader. This lack of co-location between resources and consuming code was addressed by passing a reference for the resource's classloader into the method, so that classes such as the `IconManager` will know how to locate the resources.

2.4.5 AWT Interoperability

The Shoola client was written entirely in AWT and Swing to provide menu and image interfaces. However, the Eclipse platform introduces an alternative user interface engine called the Standard Widget Toolkit¹⁴ (SWT) that is analogous to Swing / AWT. SWT takes advantage of native operating system implementations to create more aesthetically pleasing and professional looking graphical widgets. Since Eclipse provides a simple API to create sophisticated graphical user interfaces, use of SWT was considered for the Shoola client. However, both SWT and AWT need to run their event loops on the first thread on a Macintosh system and this results in conflicts. Some other options were considered to see if any SWT code could be used. The first option involved specifying a Windows environment for the release of the newly repackaged version of Shoola. However, this fails one of the constraints outlined in Section 2.2, since it would fail to provide functionality comparable to the original client. Another option would be to port all Swing and AWT code to SWT. Porting would have required a significant amount of effort and entailed a high degree of risk by relying totally on SWT, a fairly immature widget toolkit. Ultimately, the decision was made to retain all original Swing and AWT code, and make sure that SWT was not exposed to component code.

2.5 Summary

The Eclipse-based approach to componentizing the Shoola client provides a defined dependency graph between individual components, supporting an extensible and easily decipherable code base. Resulting components can be managed and developed independent of any other component. Additionally, the plug-in platform provided by

¹⁴ Northover, Steve, "SWT: The Standard Widget Toolkit," 2001 March, [cited 2006 Mar 28], Available HTTP: <http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html>

Eclipse provides better resource utilization with on-demand resource loading with minimal overhead. These characteristics satisfy our goals and requirements of componentization and provide a flexible basis to build a modular external analysis architecture.

CHAPTER 3

External Analysis

The previous chapter provided a comprehensive look into the design and execution of a plan to componentize the Shoola client to provide a more extensible architecture. This chapter will make use of this componentized architecture to bring external image analysis to the Shoola client, with a particular focus on the integration of the CellProfiler image analysis tool. Several alternative designs will be considered with extensive analysis of the design that I chose to implement. Particular emphasis is placed on how the mechanism for adding external analyses interacts with Shoola and supports storage of annotation data. This chapter will also highlight CellProfiler integration and present a sample interface to this external analysis architecture.

3.1 Overview

External analysis of OME data requires support for four basic functions: interfacing with third-party analysis applications, image retrieval, data storage, and data retrieval. Third party analysis tools need a way to register with the external analysis component and to retrieve images programmatically from the server. Subsequently, data derived from analysis must be stored in an OME-compatible form. This data needs to be easily accessible for further analysis, visualization, or raw viewing. Finally, a clear abstraction to the OME system will reduce the complexity of communicating with the OME server and total cost of ownership for building and maintaining analysis modules for the client.

The external analysis component that was implemented handles the dynamic installation and removal of third-party analysis modules and provides a straightforward interface with

the OME system. The CellProfiler image analysis tool is used as an example to demonstrate how a developer would interact with the Shoola system.

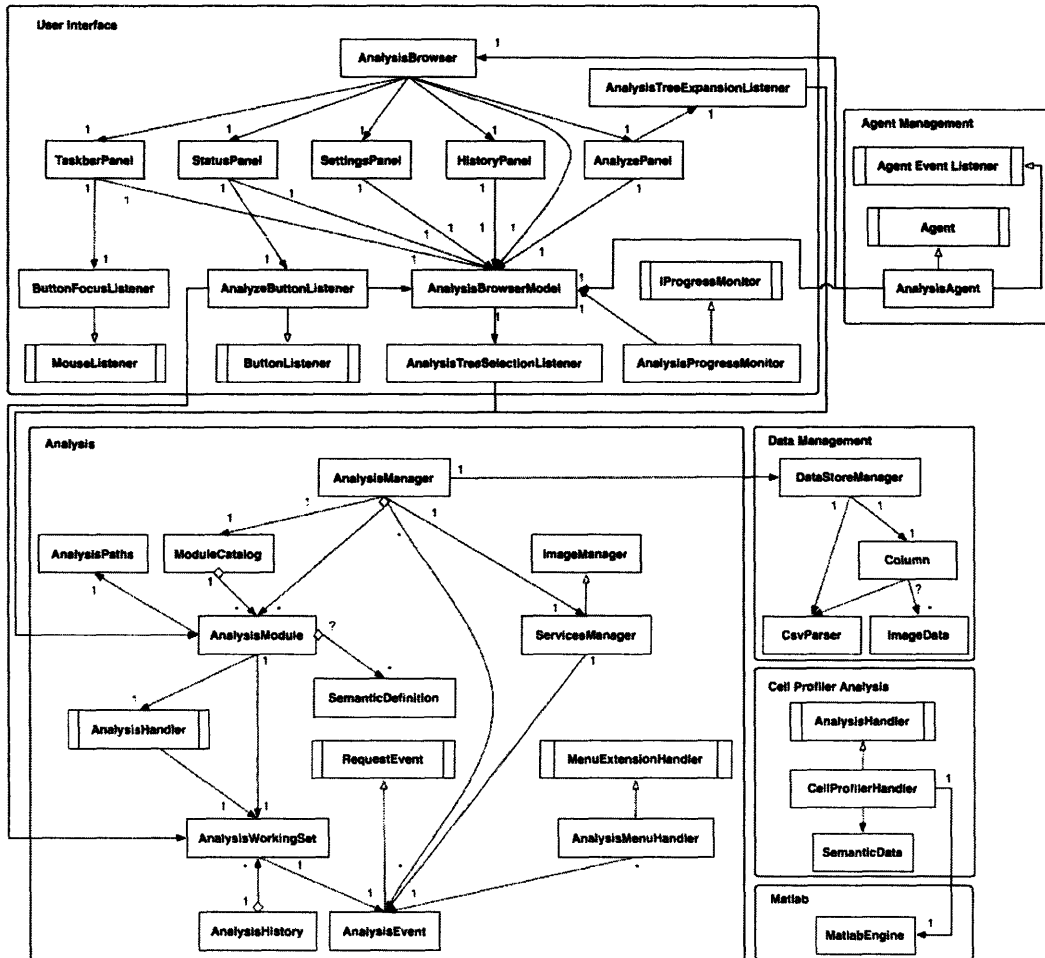


FIGURE 8. The external analysis component's UML class diagram shows the relationships between classes for the logic to recognize new analyses, store data, and render a user interface. This class diagram also includes the relationship to the CellProfiler external analysis.

FIGURE 8 provides a global look at all of the pieces of the external analysis module by means of a class diagram. The classes are roughly divided into the logical parts of the system. The classes at the top of the diagram comprise the user interface, while the lower-left and upper-right groups, respectively labeled Analysis and Agent Management, constitute the core of the external analysis architecture. The three groupings along the lower right-hand side of the diagram are plug-in components that interact with the

external analysis architecture. The Data Management plug-in provides local data storage (Section 3.4.3), while the Matlab and CellProfiler Analysis (Section 3.5) components refer to an example third-party analysis.

3.2 Design Considerations

Several alternative designs for an external analysis architecture were considered based on the requirements outlined in Chapter one but were all found to be lacking. The first approach involved leaving analysis as it exists on the server as a system of chainable analysis modules. Commercial analysis applications that normally run on workstations would be integrated as individual analysis modules on the server. This approach would allow us to perform high-content screening on large sets of images. However, unless the analysis programs run on Unix- or BSD-based systems, appropriate licenses are available, and hardware requirements are met, this approach fails our requirement of portability of analyses on a number of different platforms. Additionally, integrating all external analysis on the server would drive the OME project towards a dumb terminal network¹⁵ that would under-utilize the increasingly powerful workstations that are already available.

The second approach involved integrating a more complex and extensive server framework to call external analyses through a distributed system. The OME server would act as the gateway to Windows-based machines running target software. This might allow users to take advantage of existing commercial and academic applications, but it would force users to have intimate knowledge of the server in order to add custom analyses. The steep learning curve for developing modules on the server and the specific nature of some analyses made it dubious whether such a distributed system would work.

The last alternative approach to create an analysis framework involved leaving the client architecture as it existed, but attempting to find hooks in the existing code to integrate

¹⁵ “What is Client / Server?” 1997, [cited 2005 Dec 01], Available HTTP: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnproasp/html/thebackgroundtoclientserver.asp>

commercial analysis applications. This would satisfy our use cases, but fail the requirements for a scalable and extensible architecture. This method would increase the complexity of the client, add to the operating cost for using OME, and do little to assist new developers in adding their own analysis mechanisms to OME.

The design that I implemented is a hybrid of both client and server technology to provide a complete external analysis solution. The flexible, Eclipse-based Shoola client described in chapter two was used as the basis for development. The external analysis mechanism was developed as a plug-in component in this client and serves as the direct interface to third-party analysis tools providing an API to retrieve images, call desired analyses, and store and retrieve data both locally or remotely. Third-party analyses can contribute wrapper interfaces via plug-in extensions to easily add functionality to Shoola. This design satisfies our requirements for extensibility, near-platform independence, and data-awareness.

3.3 External Analysis

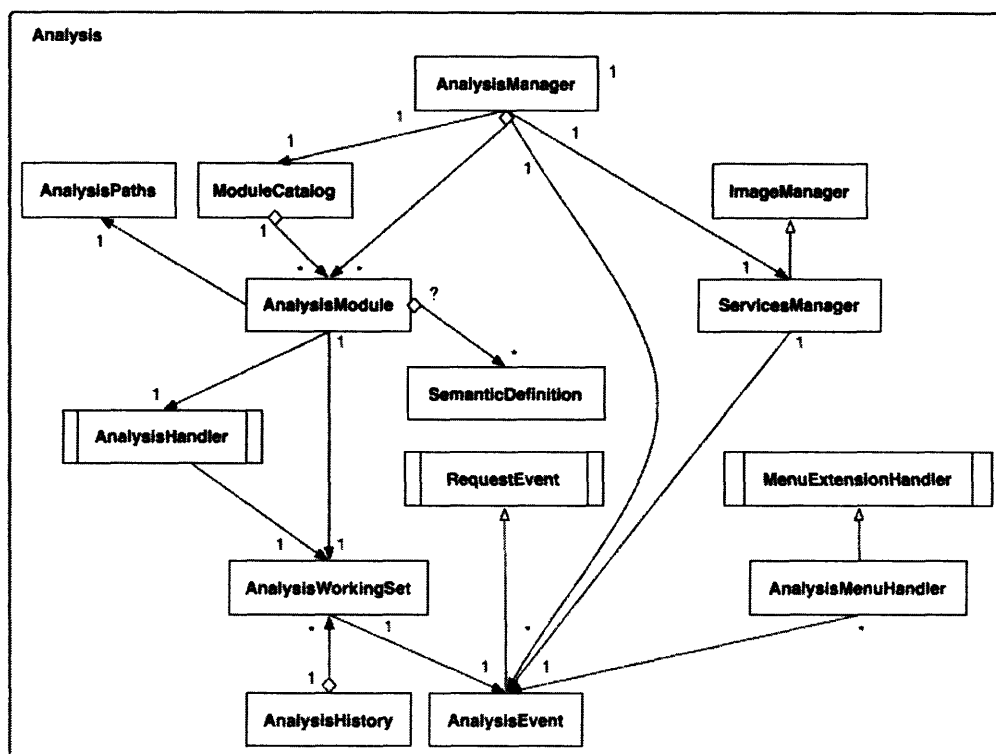


FIGURE 9. This is an isolated class diagram of the `org.openmicroscopy.shoola.analysis` plug-in, that provides image retrieval, management of third-party analyses, and historical logging of previous analyses run on a client.

FIGURE 9 shows the class diagram for the `org.openmicroscopy.shoola.analysis` plug-in containing the core logic to retrieve images and manage third-party analyses. This section will go through the different classes in this diagram and describe how they interact with the Shoola client.

3.3.1 Shoola Component Awareness

One of the first tasks in developing an external analysis framework as a component of the Shoola client was registering it properly as a component and ensuring that the proper entries were added to menus in the correct spots. To register this component, an `AnalysisAgent` class was created and announced to the Shoola client as a new component. This `AnalysisAgent` implements both the `Agent` and `AgentEventListener`

classes, as can be seen in FIGURE 10, so that Shoola will know how to interact with this component.

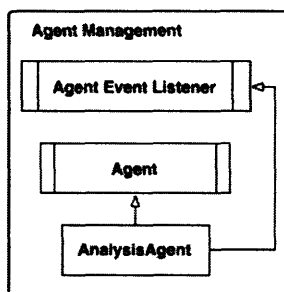


FIGURE 10. This class inheritance diagram for the “agent” class in the analysis plug-in shows the simplicity of implementing a new Shoola component.

This concept of an Agent is based on the original architecture, where components can register themselves for access to OME resources, with the exception to the actual registration mechanism. When Shoola initializes the `AnalysisAgent` class, it provides it with access to the `Registry`, a catalog of Shoola services that have already been instantiated and authenticated for the user. In particular, these services provide access to server data including images on the Image Server. Additionally, as a type of the `AgentEventListener` class, this component can register with the `EventBus`, the underlying event mechanism to process and shuttle messages, to receive messages to initiate an analysis. Finally, this analysis plug-in employs the modified plug-in registration discussed in Section 2.4.3, by declaring the `AnalysisAgent` class and a null argument for the preferences XML field to the `org.openmicroscopy.shoola` extension point.

Once Shoola can recognize this analysis plug-in as a component within its architecture and provide it with access to OME services, the analysis component needs to expose the appropriate menu items to the end user to allow for callback. Since the Data Manager serves as the launching point for managing and manipulating image data sets, this component was modified so that it will dynamically accept extensions to add menu items to its context menus. The extension point accepts a single class of type `MenuExtensionHandler` so as to provide a list of `MenuItem` objects and retrieve

associated `AgentEvents` for a corresponding `MenuItem`. The `MenuExtension` class located in the `org.openmicroscopy.shoola.agents.datamng.extension` package provides the actual logic to handle these contributions. With this new Data Manager extension point, the analysis plug-in can contribute the `AnalysisMenuHandler` class, its implementation of the `MenuExtensionHandler` class. This class provides a submenu containing a list of all analyses that are registered with the analysis plug-in. Upon the actual selection of an analysis, the `MenuExtension` class fires an associated `AnalysisEvent` object for the given menu item onto the event bus so that the analysis component can respond.

3.3.2 Message Event Handling

Message handling between components within Shoola is managed via the `EventBus`. This bus is an implementation of a single-threaded asynchronous completion token pattern and serves as a conduit between Shoola agents. The `EventBus` provides a framework for registered agents to fire objects that are subclasses of `AgentEvent` and, similarly, allows for components to selectively register themselves to listen for specific events. In the case of this analysis component, it waits and listens for the `DataManager` to fire an `AnalysisEvent` object. This `AnalysisEvent` object contains all the needed information to determine whether the item selected is a single image, a set of images, or a set of sets. When such an event is received the analysis component renders its user interface, which will be discussed in Section 3.6.

3.3.3 Shoola Abstraction

One of the underpinnings of any good architecture is providing a good abstraction to complex systems. Likewise, it is important to develop a good external analysis module that can provide a consistent interface to OME's plethora of services, without having to fully understand the nuances of the entire OME system. The `AnalysisManager` at the top of the class diagram in FIGURE 9 is the focal point for providing this API. This manager class exposes Shoola functionality in a straightforward manner and provides the

functionality for analysis management that third-party developers can use by providing a comprehensible abstraction on top of the existing OME-Java code.

The `AnalysisManager` relies on the `ServicesManager` class, which in turn extends `ImageManager`, to perform any underlying Shoola function. Chief among the available image functions are retrieving image planes and thumbnails. In particular, the `ImageManager` abstracts the authentication with the OME Image Server using Apache `HttpClient`, in order to recursively retrieve image planes from the server. Through the `ServicesManager`, the `AnalysisManager` also provides facilities to add semantic types, store annotations, notify the server of new external analysis modules, and decipher events passed from the `DataManager`. Developers can take advantage of this single point of contact, the `AnalysisManager`, to find the appropriate services to interact with the OME server.

3.3.4 Local Image Management

To run an analysis on images that are retrieved by the `AnalysisManager`, planes for an image are downloaded and stored locally. Local storage ensures speed and reliability during analysis. The actual retrieval of image planes begins with a call to `ImageManager` with the ID of a given image, dataset, or project ID as defined on the OME data server. When the `ImageManager` receives a request, it translates the request to the OME image server to retrieve an image stack. An image stack refers to 5-dimensional images stored in OME's Z/T/C image model that can consist of z-planes, multiple time-points, and channels. FIGURE 11 provides a simplified view of the role of the `ImageManager` as it mediates requests from the `AnalysisManager` and the OME data server. As soon as the `ImageManager` retrieves the image identifier from the data server and meta-data about how many planes, time-points or channels the image has, it makes a request to retrieve all planes from the image server and stores them to the local image repository. Alternatively, images can be streamed on-demand in the background while analysis is run in the foreground.

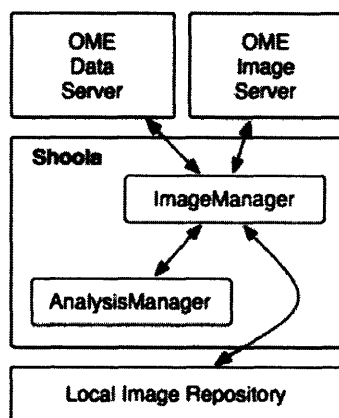


FIGURE 11. The ImageManager serves as the actual implementation class that arbitrates between requests made by analyses through the AnalysisManager API and the OME server. Retrieved images are stored in a local image repository on the client, allowing for analyses to subsequently run on the saved images.

The local image repository is physically located in the local user's workspace under the relative path of `.metadata/.plugins/org.openmicroscopy.shoola.analysis/input`. Planes are stored on disk as TIFF images, with each plane named as an aggregation of the plane's image ID, position in the z-stack, and time point, if applicable, such that it takes the form: `imageid_zposition_timeposition.tiff`. When an analysis is initiated upon a specific image ID or set of IDs, the appropriate image planes are moved to a time-stamped folder within the input folder and the file pointers are handed to the analysis code. Upon completion the planes are returned to the repository. This isolation of working set images is an indirect requirement of image analysis tools such as CellProfiler that expect pointers to directories of images.

3.3.5 Analysis Plug-in Extension Points

The Analysis plug-in provides a series of extension points to enable third-party developers to integrate their own analyses. These extension-points work by pre-defining a set of data that will be required and contributing plug-ins, such as third-party analyses, provide the actual data to the extension point. The `module.exsd` file in the `org.openmicroscopy.shoola.analysis` plug-in is the XML schema for this extension point. It defines two types of element declarations: modules and semantic types. FIGURE

12 is an example of the contribution from the CellProfiler analysis' plug-in extension to the analysis extension-point.

```
<extension
  point="org.openmicroscopy.shoola.analysis.module">
  <module
    class="org.openmicroscopy.cellprofiler.CellProfilerHandler"
    description="CellProfiler cell image analysis software is
      designed for biologists without training in
      computer vision or programming to quantitatively
      measure phenotypes from thousands of images
      automatically."
    name="Cell Profiler External Analysis"
    subanalyses="true"
    xmlFile="definitions/CellProfilerModule.ome">
    <semanticTypes
      description="XML file containing CellProfiler Semantic Type
        definitions that are not already defined in the
        base OME install"
      xmlFile="definitions/CellProfilerSemanticTypes.ome"/>
    </module>
  </extension>
```

FIGURE 12. XML for the CellProfiler analysis plug-in's extension to the Analysis plug-in's extension-point. The Analysis plug-in has previously defined a set of fields that it requires and this CellProfiler plug-in simply provides the requested information.

The first elemental extension for modules allows a developer to specify a series of five attributes: `class`, `name`, `xmlFile`, `description`, and `subanalyses`. The `class` attribute is the canonical name of the implementation of the abstract `AnalysisHandler` class. This class defines the name of the new analysis, as well as a method to run the analysis that accepts pointers to the image planes, an `AnalysisWorkingSet`, and an `IProgressMonitor` as arguments. The `name` attribute simply indicates the proper name for this analysis. The `xmlFile` is the relative path to an OME module definition file that defines the analysis to the OME server and declares its relevant inputs and outputs. This also allows all future annotation data generated by this analysis to be associated with this server module. The `description` attribute is a short description of the analysis being contributed. Finally, the `subanalyses` attribute is a Boolean value to indicate whether a single analysis algorithm is being contributed or if the analysis provides multiple algorithms. CellProfiler is one such multi-algorithm analysis tool.

The second extension element is the `semanticTypes` element. Since all annotation data in OME is based on the concept of semantic types, the only way to store data from individual analyses is if the required data type, or semantic type, exists on the server. This

extension point element only has two attributes: a `description`, and an `xmlFile` path which points to the OME file defining the semantic types that are used by this analysis. The `module` and `semanticTypes` element maintain a one-to-many relationship.

3.3.6 Managing Third-Party Analyses

When the analysis component receives an `AnalysisEvent` from the Data Manager, the extension points are dynamically read and stored into `AnalysisModule` objects. Since the act of reading the extensions does not force the entire plug-in bundle to be loaded, this is a lazy operation and `AnalysisModules` serve as proxy objects until they are actually needed. When an actual analysis is selected, the corresponding `AnalysisModule` proxy object is used to manage the retrieval of its `AnalysisHandler` implementation and installation on the server.

Since this system supports an unlimited number of third-party analyses, new analysis modules need to be managed in an organized manner. The `ModuleCatalog` was created to perform pre-installation checks to determine whether or not a module has been installed. This catalog maintains a list of all modules that have been registered on the server by the Shoola client. This process is skipped if the module has already been installed; however, if the module is not found in the catalog then both the module's definitions and semantic types, as defined by the extension, are imported to the server using the `AnalysisManager` to perform a remote import. This catalog of installed analyses is implemented as a serialized Java `List` that contains all installed `AnalysisModule` objects.

3.3.7 Executing Analyses

When an analysis is requested on a selected image or data set, an `AnalysisWorkingSet` is generated. Simultaneously, the appropriate images are downloaded to the local image repository and passed to the corresponding `AnalysisModule`. The `AnalysisWorkingSet` object manages the entire state for a single instance of an analysis run from start to finish. It includes all the relevant information for finding inputs and outputs, the type of analysis to use, and how to locate the images that were downloaded.

As discussed in the previous section, the `AnalysisModule` contains information on how to access the actual analysis and execute it on a given input. After the `AnalysisModule` receives an `AnalysisWorkingSet` and module installation is complete, the appropriate `AnalysisHandler` is instantiated. By using the information in the `AnalysisWorkingSet`, the handler can easily find the images in the local image repository as well as the desired output location for its analysis data.

Upon the completion of an analysis that has been executed, the `AnalysisWorkingSet` is time-stamped and saved to the `AnalysisHistory` log. This retains all of the relevant data that would theoretically be necessary to repeat the analysis. This log is a Java serialized `List` object containing `AnalysisWorkingSet` objects of past analyses that have been run. Since the `AnalysisWorkingSet` contains all of the metadata about a particular analysis run, it is ideally suited to encapsulate the historical data.

3.3.8 Threading Model & Monitoring Progress

A user's expectation for an interactive system has been an important motivating factor during development on a client-side external analysis module. Long running tasks usually leave a user with a frozen or unresponsive user interface, leaving them incapable of doing anything else except wait for the system to return from performing whatever synchronous process was executed. Certain functions in image analysis are computationally intensive and thus prone to long running time, which inevitably affects the responsiveness of the user interface. To avoid this problem, tasks that could potentially take a long time to execute were encapsulated in an Eclipse `Job`. These `Jobs` are an improved version of the Java `Thread` class, offering the same asynchronous behavior of a Java `Thread` and also a means to schedule threads relative to one another.

For any potentially long-running task, putting the task into the background provides a better experience to the end user by providing a more responsive interface. However, even if these tasks are threaded and running asynchronously, frustration can still arise in the absence of feedback. Users often suspect that their software has crashed when it takes

a long time to run, or requires a significant amount of processing power. To mitigate this problem, all `Jobs` and most of the methods in the analysis architecture can accept an `IProgressMonitor` as a means to monitor the progress of a running task. An implementation of this Eclipse interface can be found in the UI package as the `AnalysisProgressMonitor` class. This class accepts feedback from the method about the amount of progress completed, or even what is currently being done. This progress can be expressed to the user either as textual feedback or as a progress bar. For example, the `runAnalysis` method in the `CellProfilerHandler` is one function that accepts an `IProgressMonitor` as a parameter. As a `CellProfiler` analysis executes, information such as which image is being processed is provided to the progress monitor. Consequently, the user interface can update itself providing the user with information about which image is currently being processed.

3.4 Storing Annotation Data

One of the final steps in completing the analysis architecture was to persist analysis data in some form or another, such that it could be retrieved at some future time for review, visualization, or further analysis. In order to achieve these goals of client-side external analysis and visualization of semantic data, this data needed to be storable both locally and remotely to the OME server. The design I implemented satisfies the requirements and provides an adequate method to persist data.

3.4.1 Design Considerations

There are several requirements for storing data within the OME framework. First, data needs to be maintained within the OME data model for semantic type definitions. This means that every piece of data has to be associated with a pre-defined semantic type on the server. Strict typing requires encapsulating data with more information about the data format when manipulating data. When storing and retrieving to the OME server, semantic types add overhead that is proportional to the amount of raw data being worked. Second, all of the data must be persisted in a form that supports future analysis either by the same analysis module or by another module. Similarly, it should be viewable in either a generic

browser or through specialized data viewers specific to the analysis modules. Third, an OME user must be able to selectively store data onto the server from a local analysis. Without this capability, the added value of providing client-side external analysis to OME is severely diminished. Finally, the process for persisting and retrieving data should keep the OME client-server topology in mind, such that neither performance nor usability is diminished. With these constraints and requirements, several designs were evaluated for storing all, a subset, or none of the analysis data (FIGURE 13).

The first case, where no data is stored on the server, fails our requirement for allowing a user to store external analysis data remotely. On the other hand, if all analysis data were stored on the server, this would satisfy most of our requirements but impact efficiency and usability. For example, the CellProfiler external analysis can generate an enormous amount of data that is redundant or not sufficiently useful that it needs to persist on the server. CellProfiler's association between filenames and image data is unnecessary since OME already maintains its own association between images and files on the image server. Likewise, obscure metrics such as complex Zernike moments may be uninteresting to the OME user and pointless to store permanently.

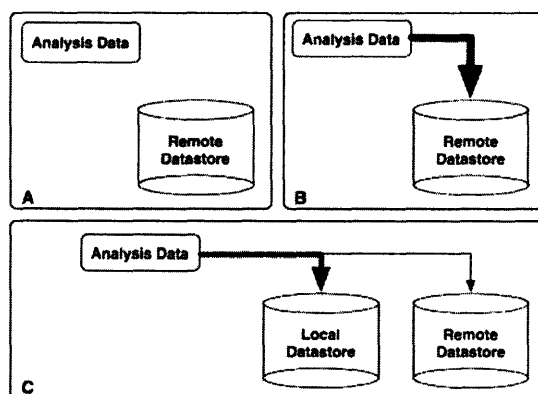


FIGURE 13. Three scenarios for data storage were considered. In (A) analysis data would be transient from one run to another. (B) Every piece of analysis data is stored on the remote server. (C) Hybrid storage structure would store a copy of everything locally, but allow selective data importing to the OME server.

Generated analysis data can contain a significant amount of intermediary data that can be used for more analysis or visualized, but is often uninteresting to other OME users.

Storing all of these intermediary results can lead to significant amounts of data flowing to and from the server for the simplest of operations, which could potentially degrade both performance and usability, failing our last requirement. Finally, for every specific type of data that gets stored on the server, there must be a pre-defined semantic type, or a new semantic type must be created. All of this can lead to a flood of semantic types that may be too narrowly focused on a specific type of analysis and fail to maintain the universality of meaning for a semantic type.

Our third design involves selectively persisting subsets of the overall data created by a client-side external analysis. Specific data would be extracted from the set of results and associated with the appropriate semantic types. This data subset would then be stored on the server using the OME server API. This allows for associating data with images and persisting shareable data on the OME server. However, this case does not persist all data, including intermediary data, which can be used by future analysis or viewed by specialized viewers on the client. If we go one step further, we can augment this case to handle the storage of remaining data in either a specialized format on the same client, or within a uniform data persistence layer. This would allow for maximum flexibility and satisfy our requirements. We can suitably ensure that the final requirement for performance and usability is kept under consideration by maintaining this hybrid storage on both the client and the server, and by finding a suitable balance in the amount of data selected for storage as semantic types on the OME server. This design provides a simple, balanced solution to our requirements. Only universally interpretable semantic types are stored on the server, ensuring all relevant data that should be public is stored on the server. Intermediary data is stored in a locally accessible mechanism on the client to provide easy access to all data generated by an analysis in case a user later decides that this data is valuable. This local data tends to be vulnerable to data loss based on tendencies to backup servers more often than workstations. However, primary data should always exist on OME servers.

3.4.2 Remote Data Storage

As described in Section 1.1.1, all data stored on an OME server is strongly typed. This means that all server data types are predefined prior to storage and will thereafter be universally recognized within the OME system. In the previous section we came to a design decision to selectively store subsets of analysis data onto the server. In order to do this, the semantic types need to be defined in an XML file and imported onto the server.

```
<SemanticType Name="CellArea" AppliesTo="F">
  <Element Name="Tag" DBLocation="CELL_AREAS.TAG" DataType="string"/>
  <Element Name="Value" DBLocation="CELL_AREAS.VALUE" DataType="float"/>
</SemanticType>
```

FIGURE 14. This XML semantic type definition defines a *CellArea* to the server to be a feature that contains a tag and floating point value.

If we take a closer look at the semantic type definition in FIGURE 14, the XML contains a declaration of the name, along with associated elements for the actual data fields. In this case, a *CellArea* has two elements – tag and value – of type string and float, respectively. The attribute `AppliesTo="F"` on the first line indicates that this semantic type refers to features. Within the hierarchy of OME data structures, a feature is a single characteristic of an object within an image. In the case of *CellProfiler*, a feature could represent a cell in an image and *CellArea* would be a semantic type associated with this feature. There are also semantic types such as *ImageCellCount* that apply to an entire image and are thus given the attribute `AppliesTo="I"`.

To store these semantic types on the server and associate them with the particular external analysis that was run on it, a module must be defined on the server, similar to semantic types, that specifies what input and output semantic types it expects. In section 3.3.5 on the analysis plug-in's extension points, we mentioned the module definition file in passing. An example of an actual module definition for the *CellProfiler* external analysis can be seen in FIGURE 15. This module definition contains a declaration of inputs and outputs, one of which corresponds to the *CellArea* semantic type as defined above in FIGURE 14.

```

<AnalysisModule
  ModuleName="Cell Profiler External Analysis"
  ModuleType="OME::Analysis::Handlers::NoopHandler" ProgramID=""
  ID="urn:lsid:openmicroscopy.org:Module:34">
  <Declaration>
    <FormalInput>
      Name="Files"
      SemanticTypeName="OriginalFile" Count="+">
      <Description>The original image files</Description>
    </FormalInput>
    <FormalOutput>
      Name="CellArea"
      SemanticTypeName="CellArea" Count="*">
      <Description>Feature Cell area</Description>
    </FormalOutput>
    ...
  </Declaration>
</AnalysisModule>

```

FIGURE 15. This shows the XML definition for the CellProfiler external analysis module that establishes proper semantic data input and output. This definition on the server allows for future analysis data to be properly attributed to this module on the OME server.

One way of storing semantic type data and associating it with a client-side analysis module on the server is by utilizing existing OME-Java package that provides an XML-RPC layer to store semantic type objects directly. However, analyses such as CellProfiler generate millions of points of data potentially creating significant memory requirements, since OME-Java requires individual Java objects to be created for every data point. My implementation of the analysis component takes advantage of the OME server's built-in Perl-based spreadsheet importer¹⁶ for semantic types. This importer was not publicly accessible from the OME-Java programming interface, so an Apache HttpClient connection directly to the server was used to pass the data file. The importer accepts a tab-delimited file formatted with an image identifier specified as the first column, and the headings of the following columns in the format: `semantic_type.semantic_element`.

Two modifications were made to the importer to store data correctly in a strongly typed OME format. The first was to augment the importer so that the data could be associated with a particular analysis module. This involved modifying the `SpreadSheetImportPrompt.pm` file to accept an additional `Module` parameter. If a valid module exists then any data associated with that particular session would be associated with the outputs of that module. The second modification was support for semantic types for features, since the importer was originally designed for image-wide semantic types

¹⁶ http://openmicroscopy.org/custom-annotations/spreadsheet_importer.html

only. A simple modification to the `SpreadSheetReader.pm` file was made to enable recognition of features. Files imported to the spreadsheet importer implicitly require a `Tag` element for each corresponding semantic type associated with a feature. Additionally, this `Tag` element should be uniquely defined by an integer that is associated with an image.

Image.id	CellArea.Tag	CellArea.Value
5	1	667
5	2	439
5	3	832
5	4	526
5	5	298

FIGURE 16. This is sample spreadsheet data suitable for the spreadsheet importer. There are five `CellArea` data points for the image with an ID of 5. The `Tag` column references each distinct cell object and the actual cell area can be found in the last column.

For example, to import five points of data for the `CellArea` feature on an image identified by an id of 5 the imported file would be formatted such that the first column contained the image identifier, as in FIGURE 16. The second column would contain the `Tag` element for the `CellArea` semantic type, where each `Tag` value corresponds to a unique cell in the image. Finally, the last column would be the `Value` element, which is the actual area of that particular cell.

3.4.3 Local Data Management

A local data store was therefore created to handle the storage of remaining data in tab-delimited files. This data store is physically located in a time-stamped folder in the local user's workspace under the relative path of `.metadata/.plugins/org.openmicroscopy.shoola.datastore`. Inside this time-stamped folder are two sub-folders, `FeatureData` and `ImageData`. To store data, `DataStoreManager` exposes a method via the `AnalysisManager` that accepts parameters for either feature or image data type and a time-stamp. Subsequently, this method provides a `FileWriter` for external analyses to write their data directly to a handle properly localized in the data repository.

The format for the stored data is similar to the OME spreadsheet importer requirements, but is significantly more flexible. The only requirement on the format of the data is that the first column lists the `Image.id`, and if the data is for features, then the second column lists the `Tag`. If the image happens to be part of a time-series, then the image id is suffixed with an underscore followed by the time-point. All remaining column headings are unconstrained. This allows for flexible storage and generalized logic to read the data store. The `DataStoreManager` provides a public API to access this data in a read-only fashion.

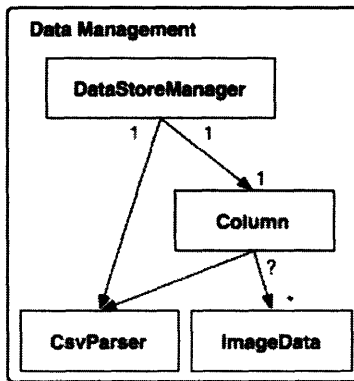


FIGURE 17. The local Datastore class diagram expresses the relationships for classes that are predominantly used to parse and retrieve data saved from analyses.

Requests to read data from the local data store are based on the timestamp of the analysis and on the type of data desired – feature or image data. From the class diagram in FIGURE 17, we can see the dependencies between the classes needed to manage the local data store. The `DataStoreManager` performs the search for the requested files and parses the column headings from the original file to produce `Column` objects. Adhering to the lazy-load model for code development, these `Column` objects are proxy objects that contain information about a particular column in a file, but do not read the data from the file until the data from the column is actually requested. As soon as a `Column`'s values are accessed, the data file is read and the data is either returned as a `List` of values corresponding to the entire column, or processed into an `ImageData` object if data for a single image id is requested. An `ImageData` can contain a mapping between the `Tag` and respective values, or an even more complex set of nested maps if the image id requested

is part of a time-series. In the latter case, the map returned would be the time-points mapped against tag-value maps.

3.5 CellProfiler

There are two specific aspects unique to getting CellProfiler running – interacting with Matlab and storing output data.

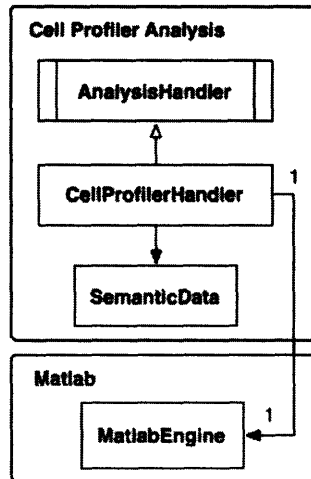


FIGURE 18. The CellProfiler plug-in and its simple class interactions with the Matlab plug-in are shown in this figure.

Since CellProfiler is a collection of Matlab scripts, the biggest challenge in getting CellProfiler to work with the new architecture was getting Shoola's Java-based architecture to interact with Matlab. In order to do this, an open source package called JMatLink¹⁷ was used to provide the interface between Java and Matlab through Java Native Interface (JNI). The package was compiled on both Windows and Mac OS X platforms so that CellProfiler analysis could transparently run on both platforms. Once native calls could be made to Matlab, understanding the actual CellProfiler architecture was essential to integrating it with Shoola.

CellProfiler is a collection of Matlab analysis scripts that are strung together into an analysis pipeline, where outputs of one script are inputs to another. Additionally, a

¹⁷ JMatLink. <http://jmatlink.sourceforge.net>.

Matlab-based UI drives the entire analysis and manages the script execution with all the correct parameters. After working with the developers of CellProfiler, the interface for CellProfiler was subverted, allowing scripts to be run headless by directly setting a number of parameters. All of this work to set the correct CellProfiler parameters in Matlab is encapsulated in the `CellProfilerHandler` class, which manages creation of a Matlab instance and sets the appropriate parameters to enable pipeline analysis.

The subversion of the CellProfiler UI takes advantage of the fact that CellProfiler is capable of running in batch mode and distributing workloads to a Matlab cluster. When CellProfiler runs in this batch mode, it runs without a user interface and a script file that loops through the input images and feeds them to each individual script in the pipeline drives analysis. This is identical to the model of how the `CellProfilerHandler` class provides analysis to Shoola users without requiring them to interact with the CellProfiler interface.

Persisting the data from an analysis is equally important as being able to generate the data itself. As we have seen, raw CellProfiler data should be storable as both image and feature data, locally and remotely. The implementation requires the raw data to be slightly modified so as to be compatible with both remote and local data stores. On the server, only specific columns of data from the results are processed as governed by the module outputs defined by the CellProfiler plug-in. The `SemanticData` class in FIGURE 18 is used to process the spreadsheet data that results from a CellProfiler analysis to map appropriate columns of data to OME semantic types. This class parses out remotely storable columns of data, and ensures that they are properly formatted as per section 3.4.2. This means that image data must have an appropriate image id column, and column headers are renamed to semantic type names. Similarly, feature data must also have an image id column, and `Tag` columns for each semantic type stored. This same class also makes minor modifications to store all data into the local data store. It ensures that there is an image id column, and optionally a `Tag` column for feature data.

3.6 User Interface

A simple user interface was created to demonstrate the use of the external analysis architecture described in this thesis. The interface provides a means to execute CellProfiler analysis pipelines within the Shoola architecture. This particular interface was created using Java Swing and AWT and is co-located in the external analysis plug-in. The structure of the interface follows the Model-View-Controller¹⁸ (MVC) design pattern to provide versatility and abstraction. The MVC pattern defines a separation program functionality where the model defines internal data structures and state of the program, the view defines how the model is rendered, and the controller performs actions in the program that affect the model.

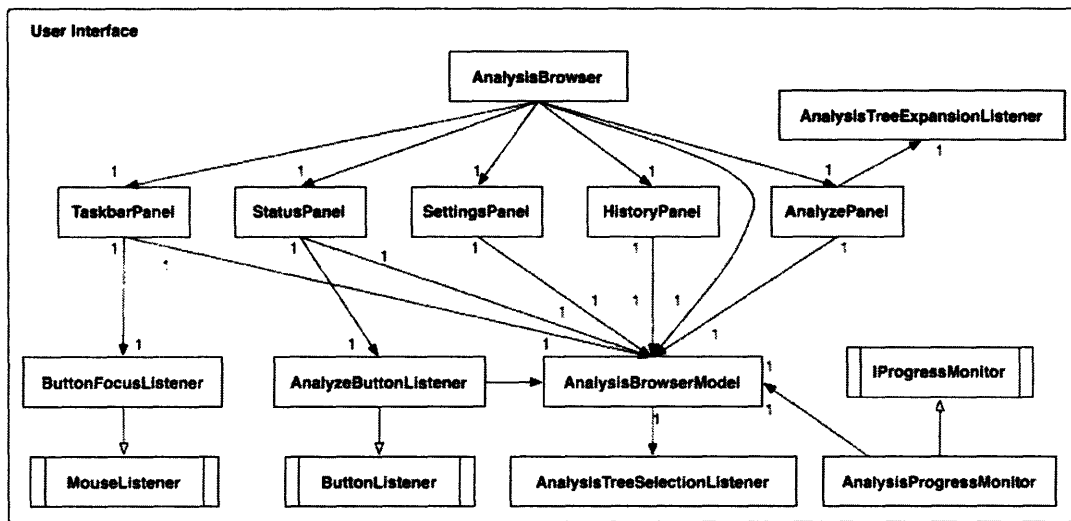


FIGURE 19. This class diagram for the user interface shows the relationships between classes. The **AnalysisBrowserModel** is the model, panels are views, and listeners act as controllers as per the Model-View-Controller (MVC) paradigm.

The relationships between classes in the user interface are presented in the class diagram in FIGURE 19. This diagram highlights the relevant classes that will be discussed by this section in further detail.

¹⁸ Java BluePrints – Model-View-Controller. <http://java.sun.com/blueprints/patterns/MVC-detailed.html>

3.6.1 Model, Listeners, and AnalysisBrowser Classes

The role of the model is central to the MVC paradigm. It is responsible for managing the application state, responding to queries about state, exposing the underlying business logic, and notifying views of changes. The `AnalysisBrowserModel` class provides this functionality by maintaining the variable parameters for a particular analysis state. This model class can be updated by the views and by the application, and appropriately fire updates to respective views. The `AnalysisBrowserModel` implements the `Observable` interface, to allow other UI components to register and listen to state changes that occur. Individual panels, discussed in the following sections, correspond to the views in this model that render content and allow user interactivity.

The controllers in this system are manifested in the form of listeners: `AnalysisTreeSelectionListener`, `AnalysisTreeExpansionListener` and `AnalyzeButtonListener`. These listeners define how the interface interacts with the external analysis architecture by mapping an action to model updates and selecting the correct views for response. The tree selection and expansion listeners manage the rendering and expansion of the tree that lists the available analyses to an end user. This requires queries to the `AnalysisManager` for available analyses and updating the model to render the analysis panel. The button listener initiates the image analysis by forcing the download and caching of images to the local image store and generating an `AnalysisWorkingSet` that is passed to the `AnalysisManager` for analysis execution.

Finally, the `AnalysisBrowser` class bootstraps the views in this interface when the analysis plug-in receives an event on the `EventBus` from another component in Shoola. This parent container is responsible for rendering the page selector found on the left-hand side of the screen and responds to updates to the model to determine which panel to render on the right-hand side of the screen.

3.6.2 Analysis Selection

The workflow for running an external analysis begins after a user has selected a particular image, data set, or project to analyze in the Data Manager. FIGURE 20 shows the two-

paneled user interface that is rendered by the `AnalysisBrowser` upon selection. The left-hand side-panel, generated by the `TaskbarPanel` class, allows users to navigate the analysis interface. The right-hand panel contains a description of the selected image or image sets, a list of all registered analyses and sub-analyses, and a short description of the currently highlighted analysis.

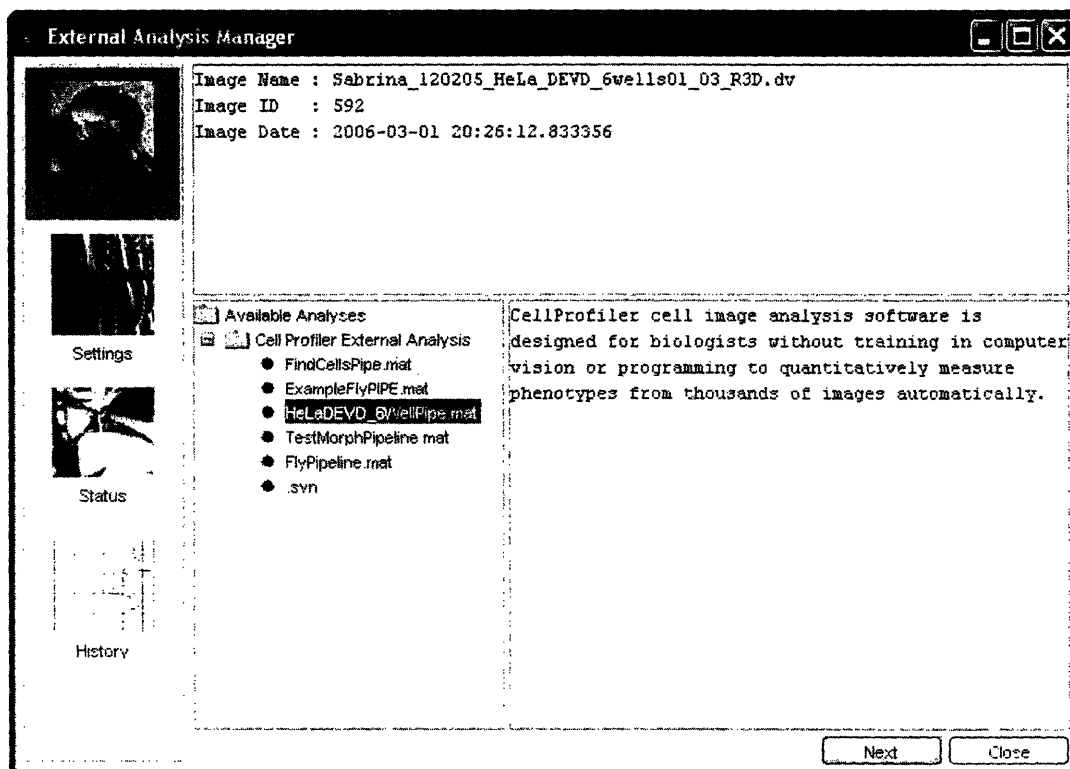


FIGURE 20. A screenshot of the analysis selection screen shows the meta-data for the selected image or data set. Installed and available analyses are listed along with potential list of sub-analyses in a tree. Descriptions of analyses are provided by third-party plug-ins.

Meta-data information about the image or images selected is gleaned from the event that is passed in to create the `AnalysisBrowser`. This event is stored by the `AnalysisBrowserModel` and used to display the information at the top of the screen. The tree of analyses is rendered through a `JTree` and populated using the expansion and selection listeners described in the previous section. The description of the highlighted analysis is retrieved from the extension point that is declared by the external analysis

plug-in. The analysis module happens to be CellProfiler, so the description is defined by the CellProfiler plug-in's extension to the external analysis plug-in.

3.6.3 Settings Modification

Images stored in OME can have multiple channels, or wavelengths. In the next step in preparing an analysis is to identify which image channels are available and select which ones to analyze. This could be performed intelligently by examining the meta-data information associated with the image to determine which filters were used on the image and automatically matching them up with the inputs to the analysis. However, for simplicity the Settings panel shown in FIGURE 21 provides manual association between channels and inputs. This panel also provides a thumbnail of the first image in the channel, defaulting to the first z- and time-points in the case of image stacks or time-lapse movies, respectively. These channel thumbnails are retrieved using the `ImageManager` via the `AnalysisManager`.

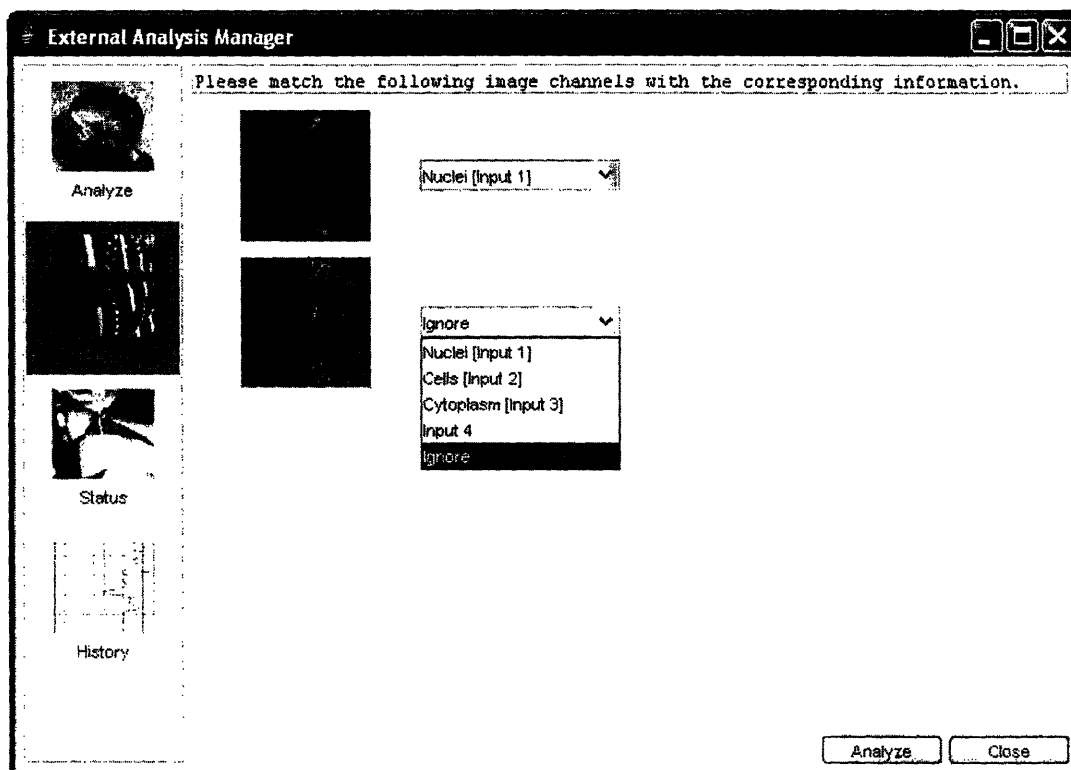


FIGURE 21. A screenshot of the analysis settings screen shows available channels of the current selection. Drop-downs enable a user to match channels with analysis inputs.

The drop-down menus allow the user to match each thumbnail channel with an input or completely ignore the channel. After a user has identified the desired pairings, the mapping between channel and input is stored in the `AnalysisBrowserModel` in preparation for the analysis. At the bottom of this panel is the `Analyze` button that initiates the external analysis module's code that subsequently kicks off the full image analysis. As described in Section 3.6.1, the `AnalyzeButtonListener` is the controlling mechanism in this interface. It processes and constructs an `AnalysisWorkingSet` that can be used to perform an analysis by reading parameters, such as the channel mapping, from the model.

3.6.4 Monitoring Progress

This user interface was built on the paradigm of providing feedback often and with sufficient detail to prevent users from wondering what is happening under the covers.

This also reduces user frustration and provides a more responsive dialog for debugging issues that might arise.

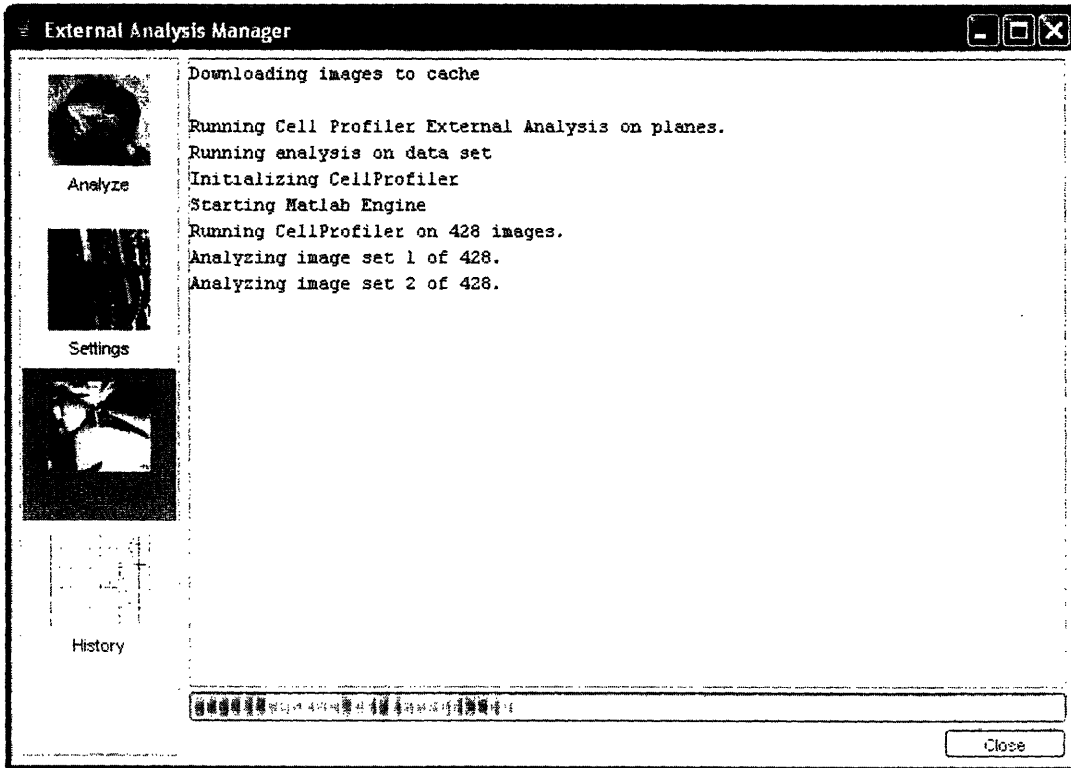
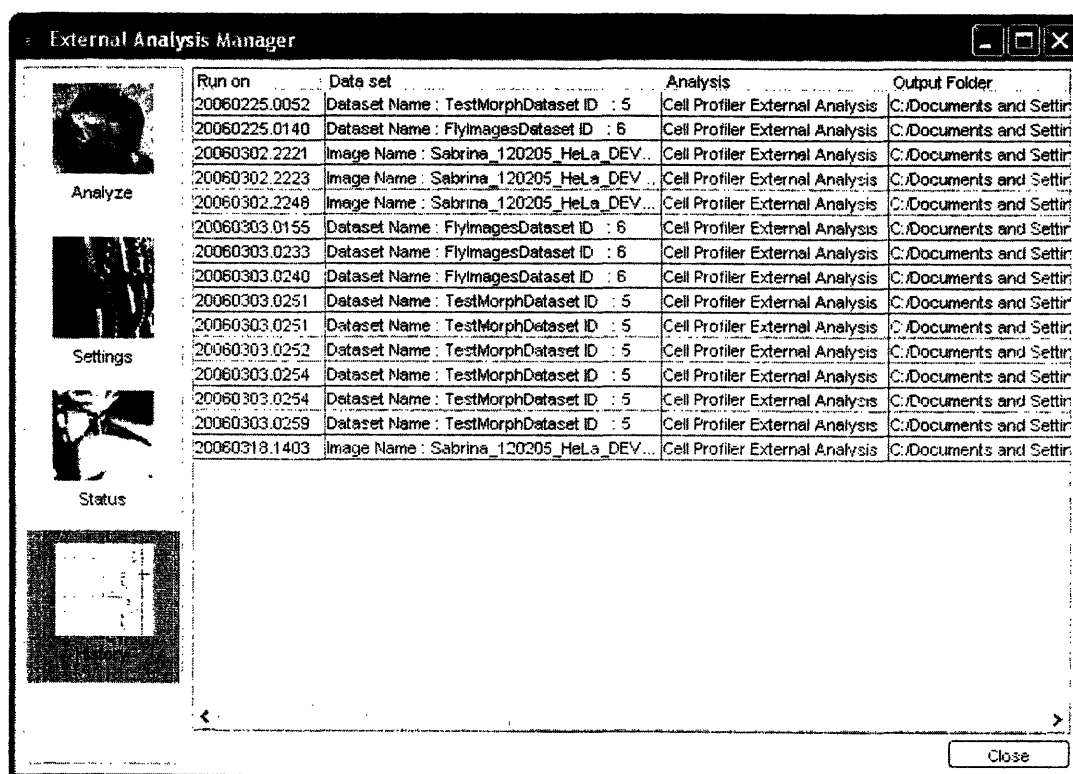


FIGURE 22. A screenshot of the status screen shows comprehensive progress information and a meter for a measure of completion.

The status panel provides a streaming frame of text that describes ongoing analysis processes. Since image analysis duration varies from minutes to hours, it is important to a user to know that a particular analysis has not crashed and that the software is continuing to work on the user's behalf. The `AnalysisBrowserModel` drives the text that is produced in the status panel; in turn, the model class gets its status by passing the `AnalysisProgressMonitor` as an argument through respective stages of the analysis. This progress monitor class allows receiving classes a way to output status to the user by dynamically updating the status panel when it receives a change, avoiding user frustration (see FIGURE 22).

3.6.5 Viewing History & Importing Data

Following the completion of an analysis, the user is presented with a final screen that provides a comprehensive listing of all analyses that have been executed on this particular client, including the one which just finished. This panel includes a timestamp, a description of the data that was analyzed, the name of the analysis, and the actual path pointing to the output analysis data in the local data store. This information, shown in FIGURE 23, provides a pointer back to when and how a particular piece of data was analyzed and allows us to establish provenance on a particular piece of analysis.



Run on	Data set	Analysis	Output Folder
20060225.0052	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060225.0140	Dataset Name : FlyImagesDataset ID : 6	Cell Profiler External Analysis	C:\Documents and Settings\...
20060302.2221	Image Name : Sabrina_120205_HeLa_DEV...	Cell Profiler External Analysis	C:\Documents and Settings\...
20060302.2223	Image Name : Sabrina_120205_HeLa_DEV...	Cell Profiler External Analysis	C:\Documents and Settings\...
20060302.2248	Image Name : Sabrina_120205_HeLa_DEV...	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0155	Dataset Name : FlyImagesDataset ID : 6	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0233	Dataset Name : FlyImagesDataset ID : 6	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0240	Dataset Name : FlyImagesDataset ID : 6	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0251	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0251	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0252	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0254	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0254	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060303.0259	Dataset Name : TestMorphDataset ID : 5	Cell Profiler External Analysis	C:\Documents and Settings\...
20060318.1403	Image Name : Sabrina_120205_HeLa_DEV...	Cell Profiler External Analysis	C:\Documents and Settings\...

FIGURE 23. The history screen shows a list of historical analyses performed by this client along with relevant time-stamp, analysis, input, and a path to output data.

Another important feature of this panel is a means to import the selected analysis data from the local data store to the server. Simply right-clicking on an entry provides a menu to import the data into the OME server. The data used to populate this table is retrieved from the AnalysisHistory log, as discussed in Section 3.3.7. The actual data that gets imported to the server is the subset of all data that has been strongly typed to semantic

types already defined by the particular analysis module. Through the use of the `AnalysisManager`, this data is imported using the modified spreadsheet importer discussed in Section 3.4.2.

CHAPTER 4

Data Visualization & Manipulation

Chapters two and three provided a comprehensive view of an external analysis mechanism within a componentized architecture augmenting the OME system with the ability to perform third-party image analyses within a unified framework. This chapter will briefly discuss some of the direct consumers of the componentization and external analysis architecture.

4.1 Data Visualization

The best active demonstration of the architecture presented by this thesis is the LoViewer, a visualization tool that has been designed to work with OME and has been implemented to work on top of the plug-in architecture. The LoViewer is a visualization component developed by Tony Scelfo in the Sorger Laboratory at MIT in collaboration with the Open Microscopy effort. It was originally intended to visualize semantically typed data stored on the server. The interface for this visualization tool can be seen in FIGURE 24.

This semantic type viewer is built as a plug-in within the componentized model and takes advantage of the extension points to register itself with the Shoola client. Similarly, the viewer plug-in extends the Data Manager to provide a menu to access and launch this visualization tool. As described in previous chapters, by developing within the componentized architecture, the data visualization tool can easily be added or removed from Shoola clients and possesses a clear graph of class dependencies. This provides a

straightforward understanding of how this plug-in fits in and interacts with other components.

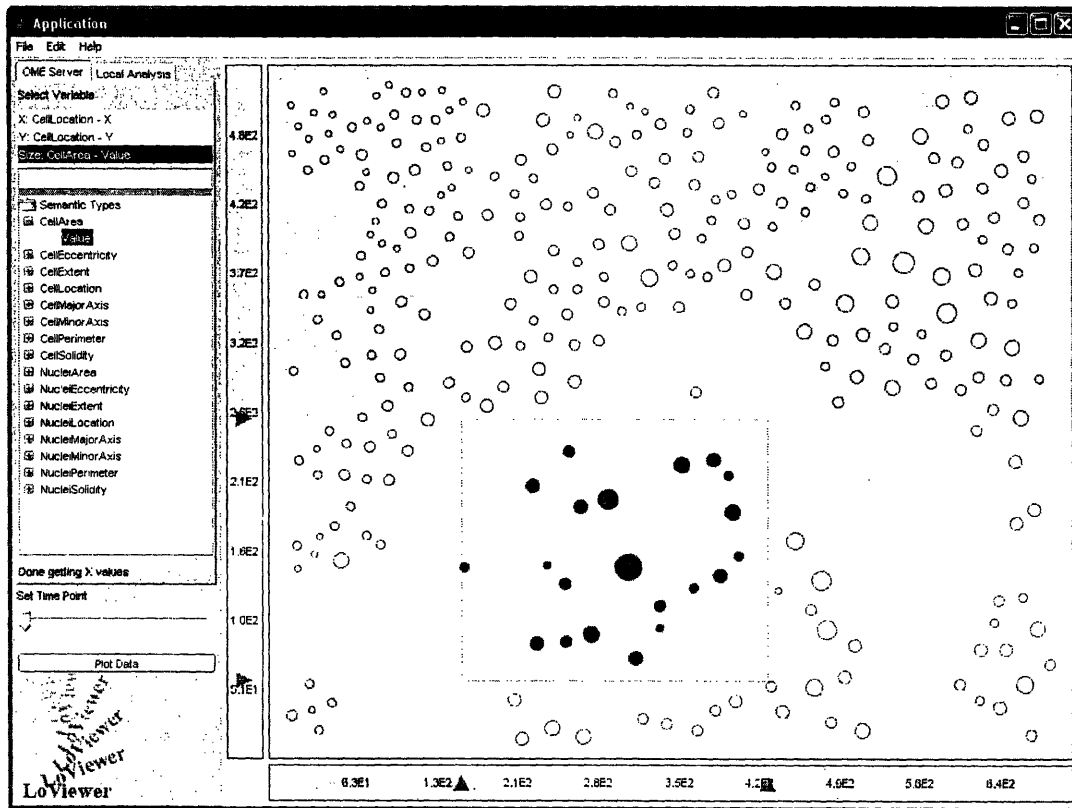


FIGURE 24. The LoViewer is a data visualization tool built to extract and screen OME data for interesting trends and relationships. This screenshot demonstrates its visualization capabilities on data that was saved by a CellProfiler external analysis.

The most fundamental product of the external analysis architecture is the sheer amount of feature data produced by image analyses such as CellProfiler. This data viewer takes advantage of the fact that such a large volume of data is more effectively analyzed visually than by other means. Reading highly structured data from the remote storage facilities on the OME server, as well as the relatively unstructured data that is stored in the local data repository, allows this tool to perform its visualization. This provides users of OME an invaluable way to quickly and effectively review image analysis data, as well as the flexibility to view and work with different types of data.

4.2 Time-Series Analysis and Cell Tracking

Another application of this architecture has led to the creation of more advanced algorithms to manipulate data generated by external analyses. For example, feature data produced by CellProfiler on a time-series movie can provide data at the discrete time points, but these results give a disjointed picture of the overall flow of the movie. In particular, one of the areas of research within the Sorger Lab involves studying the division of cells by live-cell time-lapse imaging. To understand this data it is necessary to track individual cells, accounting for cell splits and merges, and maintain a family tree of parents and daughters.

A cell-tracking module was written to take advantage of cell location and area data calculated by CellProfiler, and stored in OME, via a regressive cell-tracking algorithm. This algorithm is a modified version of one introduced by Withers and Robbins¹⁹ that uses distance and overlap ratios to link cell lines within a movie through splits and merges. The modified algorithm provides regressive correction of simple linkage errors that occur when cells appear or vanish between frames, some due to division and others due to death. FIGURE 25 provides a visual graph of the data for several cell-lines in a movie being tracked over a period of six time points. It provides a complete picture of the cell splits and demonstrates the regressive matching of orphaned children with potential parents.

¹⁹ J.A. Withers and K. A. Robbins, "Tracking Cell Splits and Merges," *IEEE*, pp. 117-122, 1996.

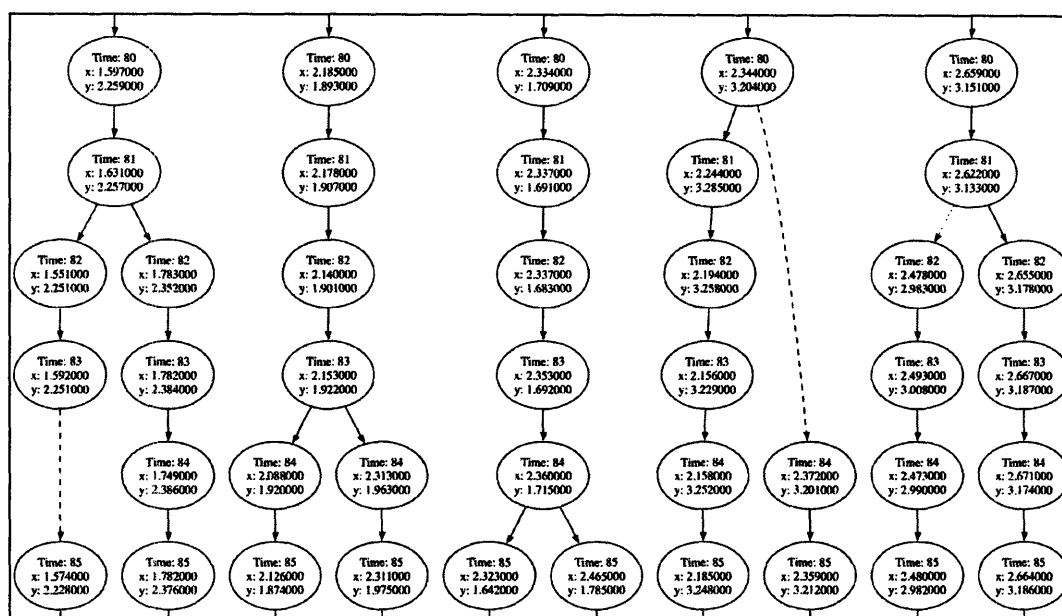


FIGURE 25. This figure shows a time-series graph tracking cell division in a movie. Each individual circle indicates the frame and the pixel location of the cell. The solidity of the lines indicate the probability of the connection.

This tracking module was written as plug-in to the componentized-Shoola architecture, taking advantage of the remote and local data store facilities, similar to how the LoViewer visualization tool reads and displays data. This tracking effort underscores the idea of how results from external analyses can be easily accessed in OME by data manipulation tools or algorithms. This provides biologists with an effective set of techniques for image analysis that fit nicely within a user's workflow in OME.

CHAPTER 5

Conclusion

This thesis describes a modular architecture that improves the extensibility of OME client software as well as a means to perform image analysis on client workstations using remote and local data stores. It demonstrates some of the different use cases for this project, and exactly how this architecture can augment a user's ability to process, visualize, and manipulate image data. This chapter will revisit the model of workflow introduced in the first chapter and how the components described in this thesis support this workflow. Additionally, since there is substantial opportunity for future work in the field of biological microscopy, some topics for further study will be discussed.

5.1 Workflow

Revisiting the OME workflow (FIGURE 26), we can now see the logical progression and development of these pieces throughout this thesis, and how these pieces support this workflow and satisfy the requirements that had been set forth. We can see that each of the red-highlighted blocks within Shoola, such as the External Analysis and Local Data Store, are logical plug-in components within the componentized architecture. The pathway marked with ① shows the retrieval of planes for a desired set of images using the External Analysis component's API to interact with the OME image server (FIGURE 26). In pathway ②, third-party analyses can analyze images in the local image store to produce resulting analysis data.

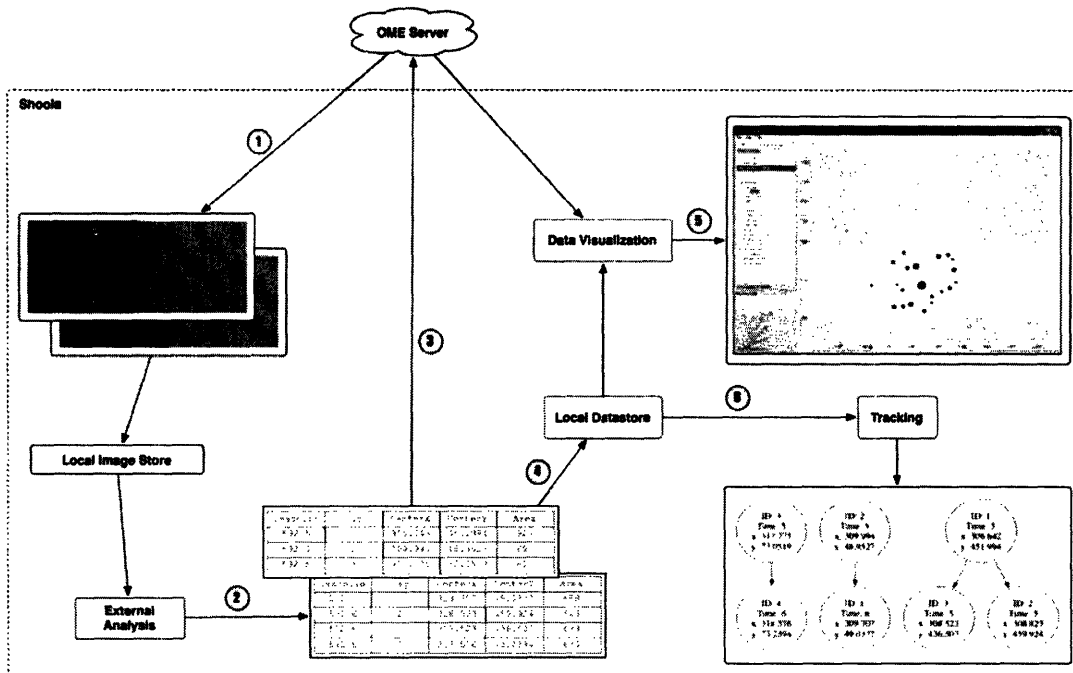


FIGURE 26. This figure revisits the software workflow summarizing some of the functionality that was implemented by my thesis project. Pathway 1 indicates image download from the OME server, 2 shows analysis results from an external analysis, 3 and 4, respectively, show remote and local data storage of analysis data. Pathway 5 shows LoViewer visualization of local or remote data stored, and 6 shows data manipulation using tracking algorithms.

Once this data is generated, the analysis architecture provides the ability to store analysis data, either on the remote server ③ via the spreadsheet importer or in the local data store ④ through the `DataStoreManager`. Finally, as covered in chapter four, some of the user-visible applications of the external analysis architecture – data visualization and tracking – can take advantage of the stored analysis data in pathways ⑤ and ⑥.

I have discussed three requirements for building an external analysis architecture. The first requirement was scalability and extensibility of the architecture. Chapter two described a componentized architecture in Shoola that allows us to create a scalable and extensible client. Individual block components in FIGURE 26 build upon this componentized architecture. Second, this system needed to be available to more platforms than those supported by the server. By developing an analysis architecture on the Shoola client, image analysis is available on any platform that can run a supported

Java Virtual Machine (JVM). The number of platforms that support running a JVM is significantly more than the number of platforms supported by the OME server. Third, data must be stored in such a way that it can be easily retrieved, visualized, or manipulated. Chapters three and four show exactly how this external analysis architecture supports this functionality, and we can see where they fit into the workflow in pathways ③, ④, ⑤ and ⑥ in FIGURE 26.

5.2 Future Work

Much of the work that has been done in this thesis has been based on the suggestions and requirements of biologists. There are two future directions for this project: architectural work and extensions on the analysis framework.

One area of work may be to re-design the agent initialization framework. The current initialization sequence forces agents to be loaded on start-up, and then allows the Eclipse platform to handle the initialization of any remaining bundle dependencies. This initialization sequence can be rewritten such that all agents, or components, are lazily loaded. Another interesting project would be to package plug-ins into features and take advantage of the Eclipse's update management architecture. This would allow for easier management of client systems and the ability to quickly roll out new features.

In the area of analyses, augmenting data manipulation by providing a facility to users to chain analyses together between disparate tools, by a data-centric model similar to the server analysis chains, would help improve automation of the screening process. Finally, building out new analyses or integrating with more third-party analyses would be a clear demonstration of the architecture built out in this thesis and provide biologists with invaluable access to additional tools.

APPENDIX A

Source Code & Documentation

The complete source code with associated Javadoc documentation can be accessed via subversion on the OME project servers:

Source (view only):

<http://cvs.openmicroscopy.org.uk/svn/boston/>

General project documentation:

<http://www.openmicroscopy.org.uk/>

Code for the external analysis components are located under the `org.openmicroscopy.shoola.analysis` folder. CellProfiler related source is under `org.openmicroscopy.shoola.cellprofiler`. Required Matlab and local data store plug-ins are in `org.openmicroscopy.shoola.matlab` and `datastore` respectively. Similarly, the cell tracking plug-in and all algorithm implementations can be found in `org.openmicroscopy.shoola.tracker`. All sources in subversion tree is in componentized architecture hierarchy, with the first level being the root folder for individual plug-ins (with the exception of the OME-JAVA folder).