

User Interface software development for the WIYN One Degree Imager (ODI)

John Ivens^a, Andrey Yeatts^a, Daniel Harbeck^a, Pierre Martin^{a*}
^aWIYN Observatory, 950 N. Cherry Ave, Tucson, AZ 85719 USA

ABSTRACT

User interfaces (UIs) are a necessity for almost any data acquisition system. The development team for the WIYN One Degree Imager (ODI) chose to develop a user interface that allows access to most of the instrument control for both scientists and engineers through the World Wide Web, because of the web's ease of use and accessibility around the world. Having a web based UI allows ODI to grow from a visitor-mode instrument to a queue-managed instrument and also facilitate remote servicing and troubleshooting. The challenges of developing such a system involve the difficulties of browser inter-operability, speed, presentation, and the choices involved with integrating browser and server technologies. To this end, the team has chosen a combination of Java, JBOSS, AJAX technologies, XML data descriptions, Oracle XML databases, and an emerging technology called the Google Web Toolkit (GWT) that compiles Java into Javascript for presentation in a browser. Advantages of using GWT include developing the front end browser code in Java, GWT's native support for AJAX, the use of XML to describe the user interface, the ability to profile code speed and discover bottlenecks, the ability to efficiently communicate with application servers such as JBOSS, and the ability to optimize and test code for multiple browsers. We discuss the inter-operation of all of these technologies to create fast, flexible, and robust user interfaces that are scalable, manageable, separable, and as much as possible allow maintenance of all code in Java.

Keywords: ODI, user interfaces, GWT, XML, Javascript, AJAX, JBOSS, Java

1. INTRODUCTION

The One Degree Imager (ODI) instrument provides a gigapixel focal plane of 1 square degree of sky at the WIYN 3.5 meter observatory at Kitt Peak. In designing a user interface for this instrument, one must analyze what ODI is intended to accomplish and how best to capture imaging settings from scientists, display status to operators, and display results to operators, scientists, and other end users. A series of meetings with scientists and technical staff resulted in preliminary designs of the user interfaces, and a database design to record science planning and data products. Specifications on user interactivity and the limitations on browser capability and speed necessitate that some parts of the design will be accomplished in Java Swing applications, some parts in native Java on the server side, and some parts in Google Web Toolkit (GWT) on the client (browser) side. Using AJAX (Asynchronous Javascript and XML) allows the application to remain responsive to user input while processing client input and displaying other output from other parts of the application. GWT is new technology that compiles Java to Javascript which facilitates the production of browser specific Javascript code without the necessity to know the vagaries of Javascript programming in the various browser architectures. GWT also provides facilities for event processing, validation, screen design through XML descriptions and internationalization. The back end server for processing requests is JBOSS, a server that hosts Java application objects. Finally, authentication and authorization are provided via a combination of Spring Security, Lightweight Directory Access Protocol (LDAP), and Java Server Pages (JSP).

* Further author information: (Send correspondence to J. Ivens.)

John Ivens.: E-mail: jjvens@noao.edu

Andrey Yeatts.: E-mail: ayeatts@noao.edu,

Daniel Harbeck.: E-mail: धारbeck@noao.edu

Pierre Martin.: E-mail: pmartin@noao.edu

1.1 ODI Instrument Operation Modes and Database Design Issues

ODI has a complex menu of tasks that an observer will be able to choose from. ODI will take data in 2 different modes: calibration data and science data. There are several types of calibration including bias, dark, twilight flats, and dome flats, three fundamental types of science observations: static, local guiding, and coherent guiding [see Harbeck for details], and several miscellaneous technical tasks, which may be specified or may be injected by the operator/system as need. These tasks include wave front analysis, which determines the parameters for adjusting the shape of the primary mirror via adaptive optics, and focus sequences, which bring the telescope into focus. The instrument itself contains a filter mechanism, an atmospheric dispersion compensator (ADC), and a focal plane that can be read out in its entirety or subset as desired. The number of modes and the variety of variables that can be adjusted for every observation requires a flexible system to specify all of the parameters. To this end we have chosen XML as a tool for describing observations because XML is flexible, easy to pass to other processes, and easily alterable¹. XML messaging is described in more detail when we discuss the flow of control.

Users log in, create proposals, create programs, create observations and then schedule the observations. Sometimes the observer will wish to create dither patterns to cover an area of sky and assign these to an observation. As observations are run, events are logged, and products are created. Each of these separate entities (users, proposals, programs, filters, ADC, products, exposure sequences and dither patterns) is stored as a separate entity in a database table, with primary and foreign keys describing the relationships between them. In order to store the different modes of observation, a table with all possible fields of all observation types is created with a discriminator column that specifies the type of observation. In this way, one table captures all modes of the instrument. Hibernate (a tool for mapping database tables to java classes) is used as an intermediary between Java and the database to store the data. Java classes are annotated, or equivalently, Hibernate XML descriptions are specified in order to map data between Java classes and the back end database. In this way, no database specific code needs to be written in order to store data, and moving between a MySQL and an Oracle database becomes easy, for example. This will be described in more detail later.

1.2 ODI Flow of Control

Scientists create exposure sequences using a GWT web interface. In order to actually execute the sequence, several different systems must be contacted in order to furnish enough information to run the exposure. For example, pre-images must be taken and guide stars determined from those images. ADC angles must be calculated for the time at which the instrument is to expose on the sky. This must happen after the planning stage, while the operator runs the exposure. In order to facilitate the real-time exposure, a desired exposure sequence is formed into an XML document. This document is passed between the chained services, each of which alters the document with enough detail in order to hand it to the next step. The document is then consumed by an exposure service that performs the exposure. The diagram below illustrates this graphically.

ODI Exposure Sequence Flow of Control

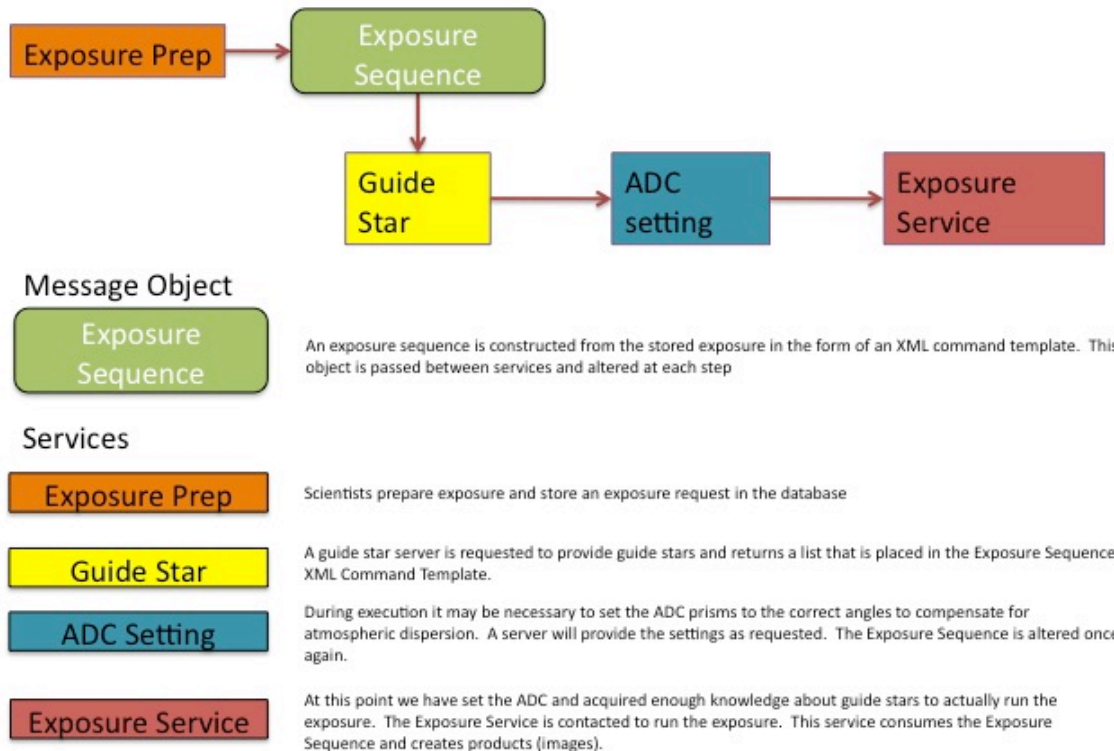


Figure 1. ODI Exposure Sequence Flow of Control

2. INTERFACE DESIGN

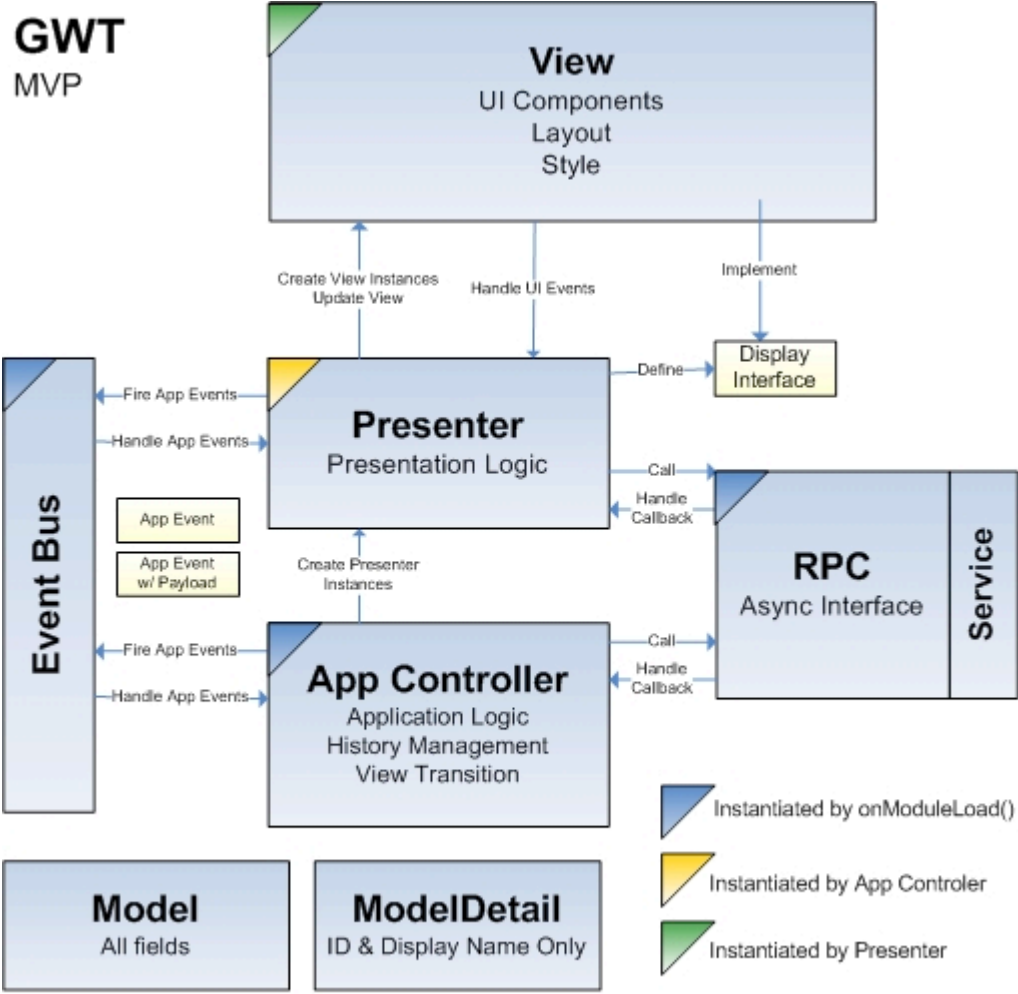
2.1 GWT (Google Web Toolkit) design

GWT is at the heart of the interface design. The main reason for using GWT is that it facilitates multi-browser Javascript coding without needing to know the vagaries of browsers (and, in fact, without needing to know Javascript at all). Javascript is the interpreted language present inside modern browsers that makes interactive web pages possible. GWT supports asynchronous RPC (Remote Procedure Call) calls easily, and it has some libraries for visualization and validation, although these are still in their infancy. In GWT, user interfaces are constructed by programming in Java. GWT has created Java classes that produce text boxes, radio buttons, and the normal tools of the trade for user interface programmers. These classes must be run through a compilation step that produces Javascript code suitable for browsers. This Javascript code is placed in the HTML page and the initial objects are linked into the DOM (Document Object Model) in the `OnModuleLoad()` method of the application. The DOM is a tree structure containing all the elements of a web page displayed in a browser. Server side code is also written in Java, and is hosted on an application server such as JBOSS. This code is compiled as you would compile traditional Java code and produces classes that are loaded into a WAR (Web Archive) file. The WAR file contains the HTML, CSS, Javascript, and Java classes that are necessary for the application to run. The WAR file, when placed in the appropriate directory of an application server, is all the application needs to run.

In order for GWT to distinguish what runs on the server side and the client side, a naming convention is used at the package level: `application.client.*` runs on the client, while `application.server.*` runs on the server.

Packages are ways of separating code libraries in Java, and look similar to URL (Uniform Resource Locator) syntax. Server code can do anything that Java can do, including reflection (referencing Java from within itself) and loading traditional Java classes such as XML parsers, image manipulation, etc. Client side code has no facilities for reflection and can only load classes that are developed using common libraries that have been ported to the client side, such as HashTable, Set, String, etc. This limits what can be accomplished easily on the client side.

A commonly agreed upon design pattern for programming user interfaces in GWT is the Model-View-Presenter architecture. A description of this architecture is graphically presented below.



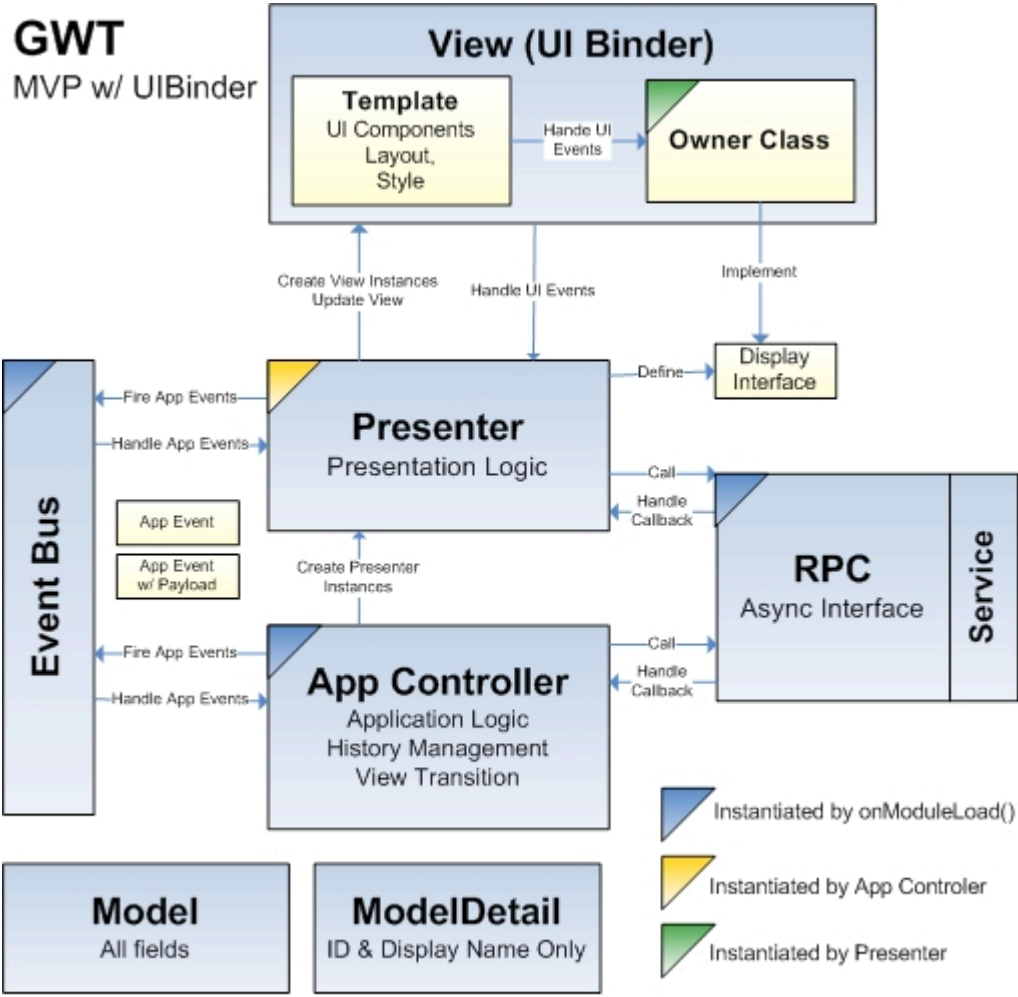
Source: nieleyde.org

Figure 1. Model-View-Presenter paradigm in GWT⁴.

The idea behind the MVP architecture is to isolate the details of the view from the rest of the application. The Presenter handles the events that are fired in the View in order to update the View. The Application Controller handles the details of the application logic, history management, and view transition. GWT programs are essentially one page of Javascript commands that alter objects in the DOM as necessary, which means that transitions are not driven by changing the URL and loading new pages. The App Controller tracks the application state and saves it whenever a transition is deemed important enough to warrant tracking. The Model consists of the basic objects (Filters, Users, Observations, etc) that make up the important elements of the application. An event bus is used as a common communication bridge between

the elements of the application. Model Detail is a useful extension to the model that can facilitate quick client-server loading of basic information, such as loading list boxes with a list of filters or users, without needing to retrieve the entire details of each filter or user. In order to communicate with the server to, for example, return user interface results, an RPC facility is set up. Data is sent to the server in XML format, a callback is registered, and when the server responds the callback returns either success or failure and whatever information was requested via XML.

To facilitate easier user interface construction, GWT 2.0 introduced the ability to build interfaces in a less programmatic way, by introducing UIBinder. Below is a description of the GWT MVP architecture using UIBinder.



Source: nieleyde.org

Figure 2. GWT MVP using UIBinder⁴.

UIBinder describes the user interface in a series of XML documents, which are loaded at run time and linked with an associated class. The class is annotated so that fields of the class are linked with field descriptions in the UI XML. In the figures below, you can see that `targetListBox` (among others) is annotated with the `@UiField` designation in the class definition for Targeting. The XML description for Targeting (which is associated to the class by the name Targeting) links the `targetListBox` field in the class to the user interface list box by `ui:field='targetListBox'`. This follows similarly for the other fields: `raTextBox`, `decTextBox`, and `equinoxTextBox`.

An example of the class definition and XML used to describe a portion of the interface:

```
public class Targeting extends Composite {

    private static InstrumentStatusUiBinder uiBinder = GWT
        .create(InstrumentStatusUiBinder.class);

    interface InstrumentStatusUiBinder extends
        UiBinder<Widget, Targeting> {
    }

    @UiField
    ListBox targetListBox;
    TextBox raTextBox;
    TextBox decTextBox;
    TextBox equinoxTextBox;

    public Targeting() {
        initWidget(uiBinder.createAndBindUi(this));
    }

}
```

Figure 3. Targeting class fields linked to UI Binder XML description

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'
    xmlns:c='urn:import:edu.wiyn.odi.client'>

    <ui:style>
        .pretty { background-color: Skyblue; }
    </ui:style>

    <g:VerticalPanel>
        <g:HTMLPanel>
            <table rows='2' cols='6'>
                <tr>
                    <td align='center'>Target: </td>
                    <td align='center'><g:ListBox ui:field='targetListBox'></g:ListBox></td>
                    <td colspan='4'></td>
                </tr>
                <tr>
                    <td align='center'>RA: </td>
                    <td align='center' valign='middle'><g:TextBox ui:field='raTextBox'>00:00:00</g:TextBox></td>
                    <td align='center'>DEC: </td>
                    <td align='center' valign='middle'><g:TextBox ui:field='decTextBox'>00:00:00</g:TextBox></td>
                    <td align='center'>Equinox: </td>
                    <td align='center' valign='middle'><g:TextBox ui:field='equinoxTextBox'>2000.0</g:TextBox></td>
                </tr>
            </table>
        </g:HTMLPanel>
    </g:VerticalPanel>
</ui:UiBinder>
```

Figure 4. UI Binder XML description naming fields that will reference to the class fields

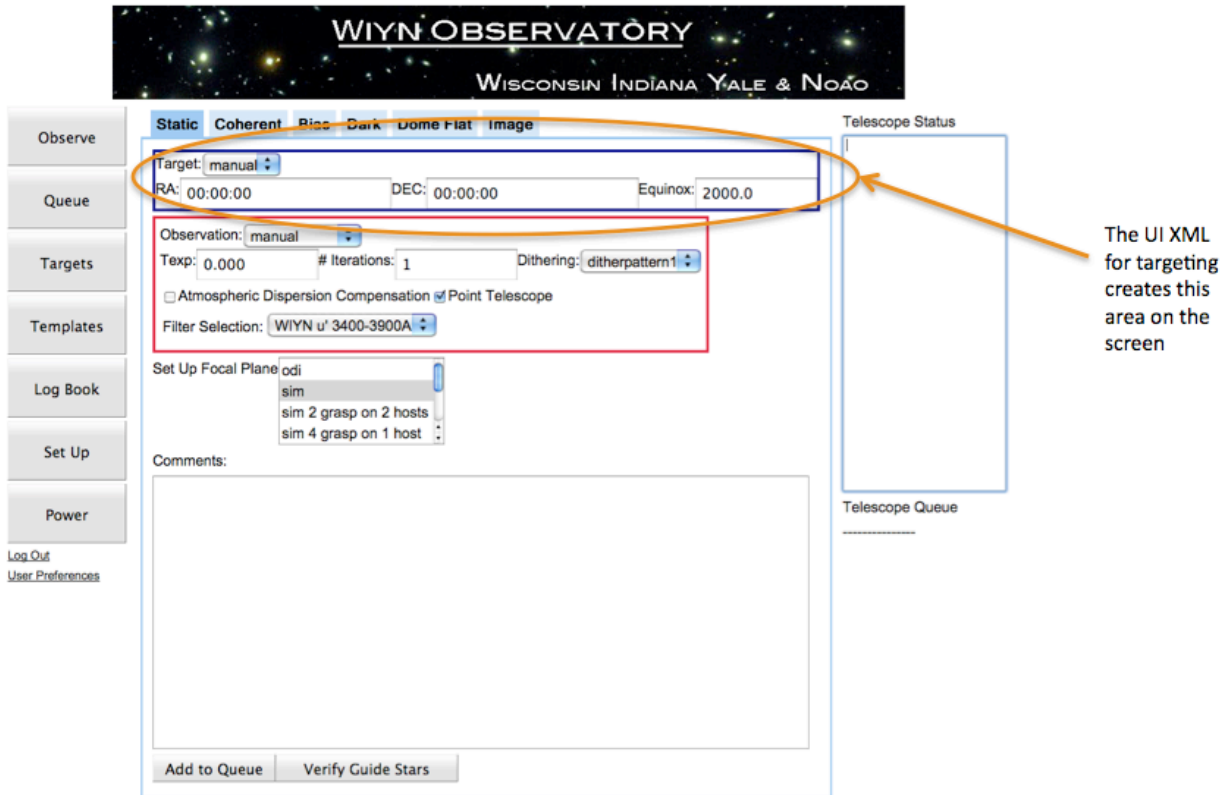


Figure 5. Graphical UI resulting from UI Binder description and class code

2.2 Validation

GWT has a library for doing validation which is JSR303 compliant³. Classes are annotated with validation statements, and upon receiving an event, such as tabbing out of a text box, a validation object validates the field against the constraints. The class must implement the `IValidatable` interface. An example of class annotation that validates the length of `configname` to be between 8 and 25 characters long and that `binningX` is between 1 and 8 are:

```

Public class ReadoutDTO implements Serializable, IValidatable {
    Long id;

    @NotEmpty
    @Length(minimum=8,maximum=25, message="The configname must be between 3 and 25 characters long")
    String configname;

    @Min(minimum=1, maximum=8)
    int binningX;
    ...etc.
}

```

The validation is written as:

```

Set<InvalidConstraint<ReadoutDTO>> readoutDTOInvalidConstraints = readoutDTOValidator.validate(readoutDTO);
... iterate over the constraints...

```

Figure 6. Validation using the GWT validation library

To iterate over the constraints one constructs a constraint iterator and checks each invalid constraint, informing the user of each error and possibly altering the user interface (for example, highlighting a field that needs to be filled in or corrected).

2.3 Persistence

Persistence (saving application data between user sessions) is achieved via four elements: Data Transfer Objects, GWT RPC XML, Hibernate and a back end database. Data Transfer Objects (DTOs) are objects that are lightweight clones of classes that are easy to serialize across the network. The RPC XML mechanism is the means of serialization of the objects across the network between the browser and the application server. Hibernate is a tool that allows one to describe the database architecture in a database neutral way, which allows one to write code that is independent of the particular back end database one is using. The database server (MySQL, Oracle, etc.) contains the tables and databases necessary to capture the information.

Interface data flow is illustrated in the following diagram:

Saving Data Flow

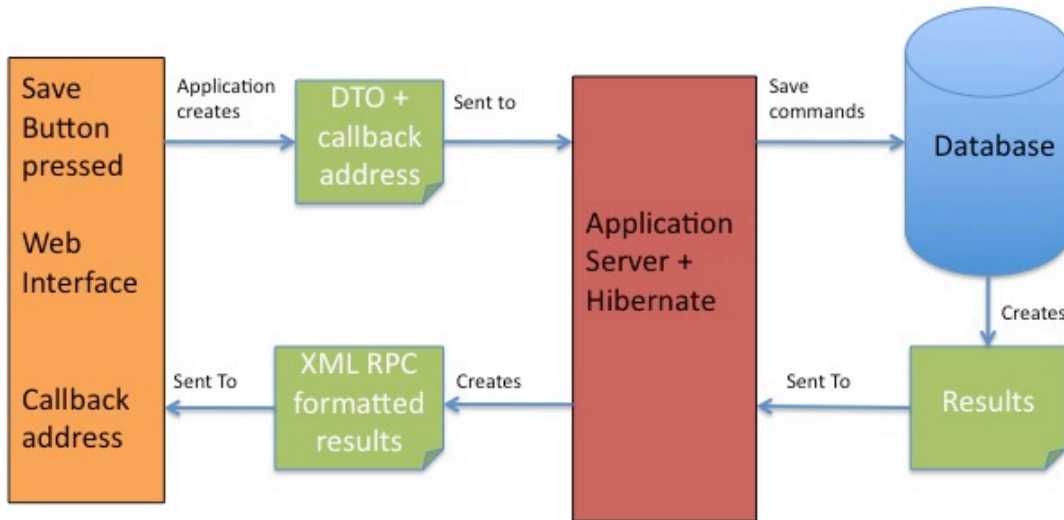


Figure 7. Processing of a save request

When a user does something that involves the necessity of contacting the server, such as pressing a save button in the user interface, the following flow takes place. The application creates a DTO to send to the application server. Along with the DTO the web application sends a callback address so that the server can give it whatever data the client requested along with a success or failure status. The callback address enables the client to behave asynchronously, i.e. the client can process other user input without waiting for the server to respond. The DTO and callback address are sent (via XML RPC format) to the server, which then determines what to do with them. In this case, the command is to save the data, so the application server asks Hibernate to do the dirty work. Hibernate uses its mapping descriptions to create specific database commands that will save the data in the DTO to the database. The database then responds with (in this case) either success or failure. If it were a command to, for example, list out all the activities in a program then the results would contain not only a response code but also a list of activities. The application server takes the results and

packages them in XML RPC format and sends them back to the callback address that the client gave earlier. At this point, the web application now has the status (and the data, if requested). At this point, the user interface communicates to the user that the command failed or succeeded and displays the requested data if necessary.

An example of the XML necessary to describe a simple table looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://
hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="edu.wiyn.odi.client.model.Readout" table="readout">
    <id column="id" name="id" type="long">
      <generator class="native">
        </generator>
      </id>
    <property name="configName" column="ConfigName"/>
    <property name="binningX" column="BinningX" type="int"/>
    <property name="binningY" column="BinningY" type="int"></property>
    <property name="overscanX" column="OverscanX" type="int"></property>
    <property name="overscanY" column="OverscanY" type="int"></property>
    <property name="prescanX" column="PrescanX" type="int"></property>
    <property name="prescanY" column="PrescanY" type="int"></property>
    <property name="addDate" column="AddDate" type="date"></property>
    <property name="editDate" column="EditDate" type="date"></property>
  </class>
</hibernate-mapping>
```

Figure 7. Hibernate XML description linking classes to the database

In this way, fields of classes are mapped to columns in database tables. Once the mapping is complete, the data is easily saved by creating a hibernate session and then calling `readout.save()` on the server. Hibernate creates the database specific SQL calls which save or retrieve the data as necessary. This allows you to easily swap out the back end database as necessary by only changing the initial session creation for one's specific database.

2.4 Authentication and authorization

Authentication is handled via LDAP (Lightweight Directory Access Protocol)⁵. LDAP is a protocol that specifies how to request and grant authorization credentials. Authorization is handled by Spring Security⁶, formerly known as ACEGI. When a user attempts to access a particular method of a class, his or her authorization is checked via Spring Security and access is either granted or denied. Roles are assigned in LDAP such as engineer, operator, scientist, etc. These roles are used by Spring Security to deny or grant access to specific functionality in the web interface.

3. INTERFACES TO ODI

3.1 Dither Patterns

Dither Patterns are overlapping patterns of coverage on the sky that remove gores from images caused by cell and chip spacing across the focal plane. For more details see [2].

3.2 Guide Stars

Guide stars are retrieved through a service (written in Java) that identifies candidate stars for guiding using a pre-image taken with the instrument and provides them to the execution process via XML. The Guide Star user interface looks like this:

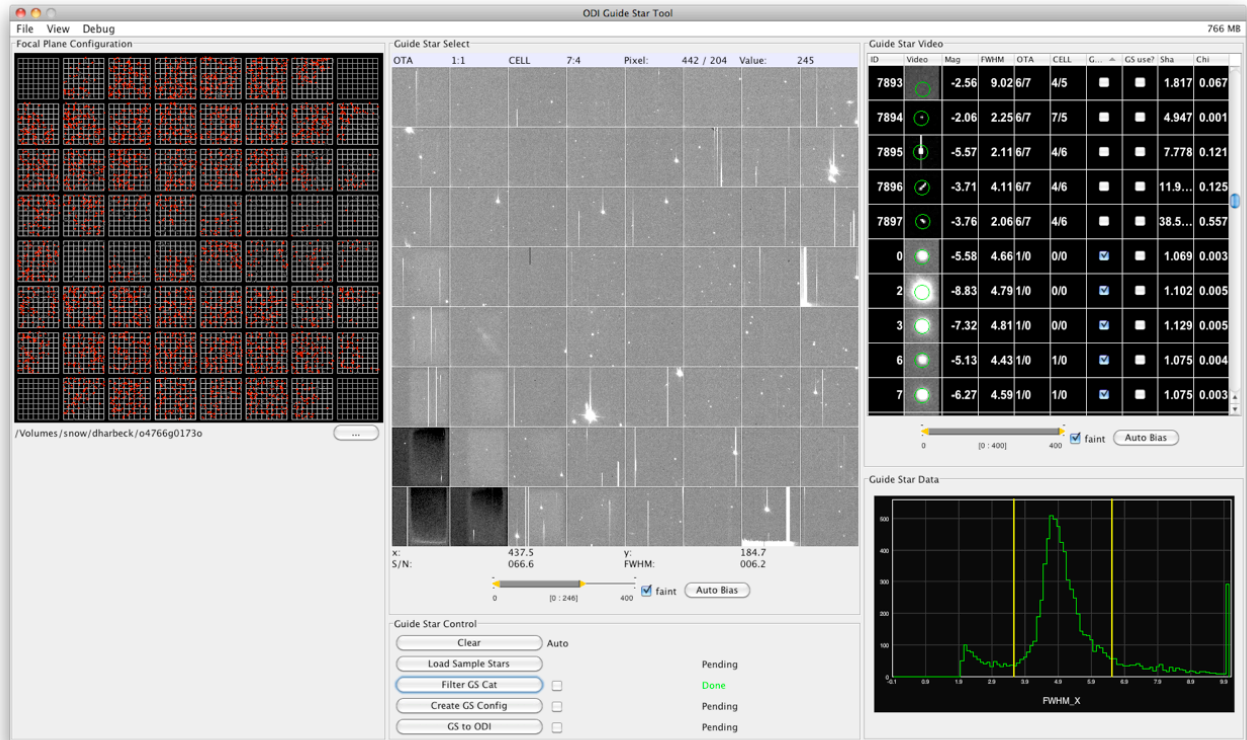


Figure 8. Guide Star Selection Interface

Viable Guide Stars colored in red appear in cells on the ODI Focal Plane in the leftmost window. A subset of the focal plane (one of the 64 OTA detectors) is shown in the center window, where guide stars can be manually be added to the list of guide stars. A sortable table of guide star candidates together with magnitude, cell number, FWHM, etc. appear in the upper right, while further guide star statistics is shown in the lower right. During an integration real-time videos of the guide stars are displayed in the same table. Because of the real-time video display requirement, this application was created using Java/Swing. Identification of guide stars is further discussed in [2].

3.3 Instrument status

ODI has a filter mechanism, an atmospheric dispersion compensator, dewar temperatures and pressure, network connectivity and power status that must be monitored. The dome environment at WIYN provides a publish/subscribe message system that is called GWC. The messages are implemented via Sun RPC methods to serialize them over the network. We wrote a Java interface to translate the RPC interface into an XML stream that contains the information in the RPC stream. This XML standard interface is also used for other instruments that are not coded via the RPC router, thus we are able to talk to all devices over a common protocol. These XML messages are received and translated into a graphical interface for displaying device status for operators (designed for a simple overview of the instrument, and not as an engineering display). This interface was easier to implement in Java Swing than GWT because of the more graphical nature of the displays. It is important to note that the interface can be implemented as an applet, which can then be signed and launched either by GWT or via Java Web Start. The interface is shown below:

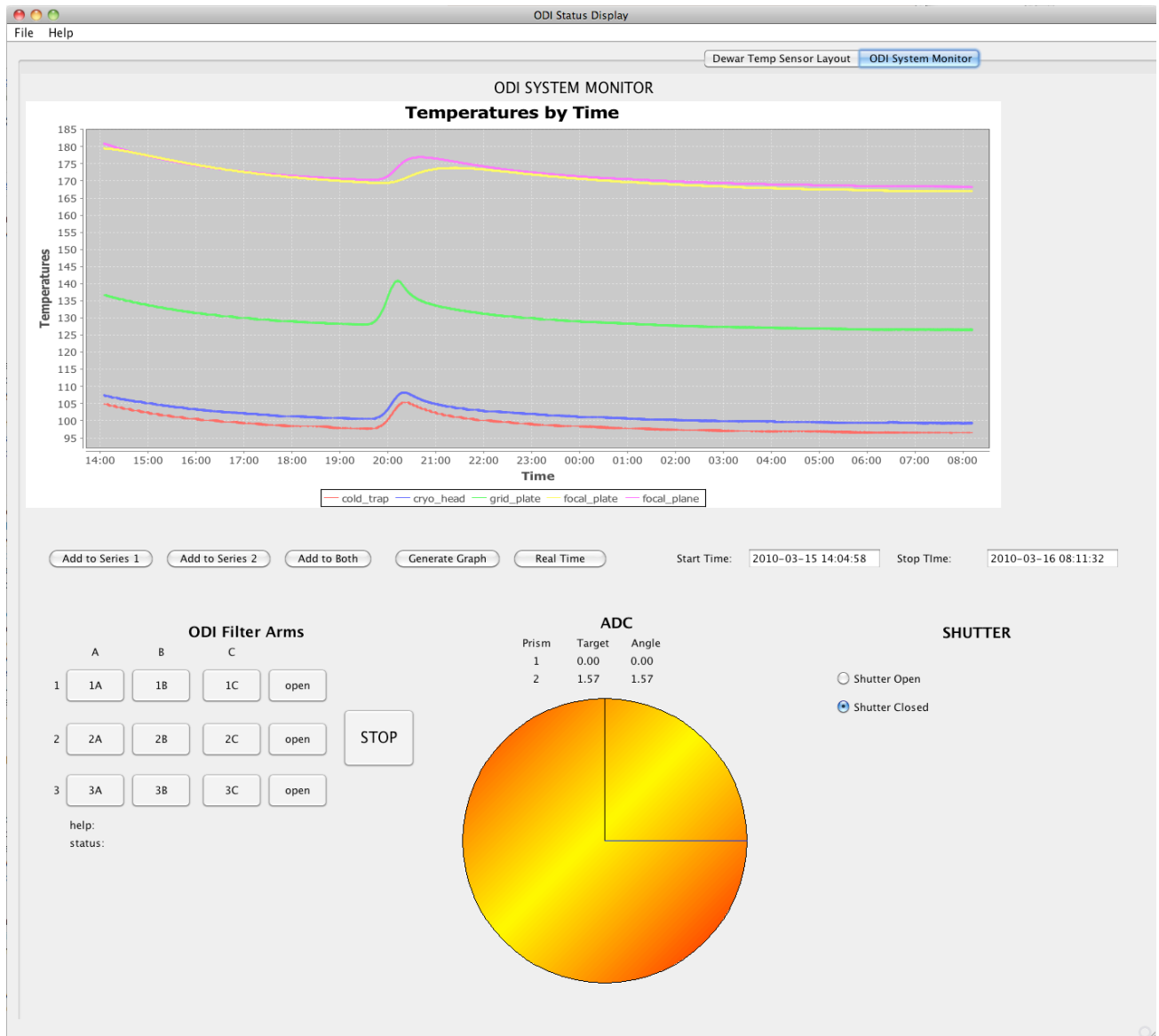


Figure 9. ODI System Monitor Interface

ACKNOWLEDGEMENTS

The authors would like to thank the WIYN Board for supporting the ODI project. ODI is supported, in part, by NSF under the TSIP contract C10482T.

REFERENCES

- [1] Yeatts, A., Ivens, J. and Harbeck, D., “ODI Software Configuration and Scripting”, Proc. SPIE, 7740-24, in press (2010).
- [2] Harbeck, D., Martin, P., Cavin, J., et. al., “The WIYN One Degree Imager: Project Update 2010”, Proc. SPIE, 7735-15, in press (2010).
- [3] Emmanuel Bernard et. al., “JSR 303: Bean Validation”, “<http://jcp.org/en/jsr/detail?id=303>”, online (2010)
- [4] Niel Eyde, “GWT MVP Architecture”, <http://www.nielejde.org/SkywayBlog/post.htm?postid=37782056-c4e1-4dfb-9caa-40ab9552ca3b>, online (2010)
- [5] Sun Microsystems, “LDAP Authentication”, <http://java.sun.com/products/jndi/tutorial/ldap/security/ldap.html>, online (2010)
- [6] Spring Source, “Spring Security”, <http://static.springsource.org/spring-security/site/index.html>, online (2010)