The Java Analysis and Display Environment (JADE) 
Project Description

A framework for 2D Visualization of Weather Products 

Rob Weingruber 
weingrub@ucar.edu
# Table of Contents

I. Functional Description ........................................................................................................................... 3

II. Project Justification ............................................................................................................................... 4

III. Development Principles ...................................................................................................................... 5

IV. Design Guidelines ............................................................................................................................... 7

V. Design Overview .................................................................................................................................. 9

VI. JADE Applications ........................................................................................................................... 17

VII. The Future of JADE ........................................................................................................................ 18

VIII. Figures / Diagrams ......................................................................................................................... 19

IX. Appendix – Feature List ................................................................................................................... 29

X. Appendix – JViz Tutorial ................................................................................................................... 32
I. Functional Description

The Java Analysis and Display Environment (JADE) is a Java framework that makes it easy to create 2-dimensional visualization applications that display real-time or archival, map-based data. The framework provides an infrastructure and a set of reusable components that provide support for metadata acquisition and representation, data retrieval, rendering and manipulation of data. JADE’s current emphasis of functionality lies in the visualization of 2D maps, SkewT plots and vertical cross section displays, though resources are also being allocated to the development of time series plots and time-height plots (Meteograms). JADE’s visualization products display layered views of several different types of data and/or protocols integrated into a single display, and provide the visualization of both model and observational data.

The framework consists of a set of configurable and modular components that can be leveraged to easily create custom applications. Components and data structures exist that provide functionality with respect to acquiring and representing metadata, as well as retrieving and representing the data itself. Other components exist in JADE to control the data, or to constrain the data in such a way that the application demands. In addition, JADE provides a set of components that contribute to the actual display of the data. The components within JADE are easily combined to create custom, configurable and extensible applications that can be deployed as interactive desktop or browser based applications.
II. Project Justification

JADE is a new infrastructure to replace what is known as the DataCanvas framework, which is the original foundation for the applications created for the Aviation Digital Data Service (ADDS), such as the Flight Path Tool, MDSS and CVIS. Similar to JADE, the DataCanvas library is a set of Java components that are used to create applications, but through its use, it was found to have significant faults with its design:

- **Applet-centric Architecture**

  The DataCanvas was designed to be used primarily within an applet, which is an interactive, web-delivered Java application that runs within a browser. Though Applets were originally considered to be a good mechanism for running applications, they exposed inconsistencies across browsers and resulted in buggy applications.

- **Lack of Modularity**

  The responsibilities of the components within DataCanvas were not clearly defined or divided. This made the components hard to understand since the responsibilities were unclear. In addition, the components were difficult to maintain since the code wasn’t centrally located, and changes in one area of the code would adversely affect other parts of the code.

- **Lack of Extensibility**

  The non-cohesiveness of the code made the DataCanvas framework difficult to extend. Adding new functionality to the core of the framework was difficult since it was unclear as to how the new functionality should fit into the architecture.

- **Lack of Customizability**

  The non-cohesiveness of the code also made the DataCanvas framework difficult to customize in that it was difficult to determine where customizations had to be made. In addition, customizations frequently resulted in “if-then-else” code that was difficult to read, maintain and inherently brittle.

- **Lack of Configurability**

  The DataCanvas applets did not have support for configuration. Therefore, specifying properties for the applications to use, such as data server URLs or rendering styles and color, was done by hard coding the values into the code. Changes in “configuration” required modifications to the code, recompilation, and redeployment of the application. These steps had to be performed on a per application basis.

Each of these factors combined to make the use of the DataCanvas framework difficult for continued development. Extending and customizing the components became increasingly important as 2D visualization demands became more prevalent, but the framework’s design proved to be an obstacle rather than a benefit. Leveraging the DataCanvas for cross project development was especially unrealistic, and as a result, proposals for a new 2D visualization framework presented to RAL management were funded to develop JADE.
III. Development Principles

The main goal of the JADE project was to create a reusable Java framework that could be used across projects within RAL. The intention was to learn from the mistakes in the DataCanvas framework and to develop a solid design for the second generation visualization framework, JADE. The high level development goals that were set for JADE are as follows:

- Start with DataCanvas functionality as the requirement set.
  
  The functionality existing in the ADDS applets is identified as the initial set of requirements, or use cases, for the JADE framework. The extensive codebase from the DataCanvas library was reviewed in order to identify additional system requirements for JADE. Since JADE is to be a second generation framework, additional functional use cases typical to a 2D visualization framework are also added to JADE’s requirements.

- Cross-project development.
  
  The intention of JADE is to provide a framework that can be leveraged by any project within RAL that has 2D visualization requirements. The use of a framework provides a core of functionality that can be immediately leveraged by project specific applications, and allows project software engineers to expend resources on new enhancements. As projects develop new functionality that is applicable to the core, it can be added to the framework for use by other projects. In addition, the use of a framework establishes a software engineering knowledge base across RAL.

- Easy to build applications.
  
  To maximize cross project usage, applications would have to be easy to build from the framework. The framework should not demand a high amount of infrastructure within an application, and should have a reasonable amount of complexity such that the framework is easy to understand and leverage.

- Emphasize Reusability.
  
  To allow the framework to be used across projects, the components must be easily reusable. The design must be done in such a way to maximize reusability, and several design goals have been established (as in the next section) to assure this. Again, simplicity and understanding are deemed as significant goals to strive for.

- Configurable and Pluggable components.
  
  Components must be easily customizable and pluggable into applications. Component attributes must be configurable, and customized components must be easy to plug into applications to ensure maximum reusability.

- Data integration.
  
  The framework should provide a visualization platform that will make it easy to display distinct data formats in one integrated display. JADE will not require any specific physical data formats or protocols and will allow new data formats to be displayed in an integrated
fashion as code is written. This principle will allow projects to leverage the framework and at the same time allow JADE applications to view custom formats integrated with other data sets.

- **Scalability.**

  Projects should be able to keep their applications small by only including parts of the framework that are applicable to their application. As the framework increases in functionality and size, projects should be able to exclude parts of the framework that they do not need.

- **Open Design.**

  Limited resources are available to design the framework from the use cases. Since there is an extensive collection of use cases, design will be performed on the most significant and highest priority use cases first. One of the development principles of JADE is to make sure that the chosen design(s) does/do not exclude, or “design out”, future development of the lower priority use cases as well as unforeseen project specific use cases.
IV. Design Guidelines

In order to achieve the above design principles, the JADE architecture exhibits a number of design idioms. These design attributes are difficult to measure and no attempt has been made to do so, but a concerted effort has been made to design JADE according to the following guidelines:

- **Modularity.**

  One measurement for the quality of a good object-oriented design is its level of modularity. JADE was designed in such a way there are many components that collaborate to achieve some level of functionality. Components are given a “crisply” defined set of responsibilities, not too many and not too few, such that their purpose is well understood and not too complex. JADE was designed in such a way that the components have a high level of cohesion. Each component has a crisply defined set of responsibilities that is appropriate to that component. In addition, components are designed such that they don’t do too little or too much. This modularity allows the design to be broken into smaller, understandable pieces and keeps responsibilities distinct across components. A side effect of this design is that code changes in one component do not (usually) affect others, which increases the maintainability of the framework. In addition, since components are designed with a fine-grained sense of responsibility (but not *too* fine), it becomes easier to plug in new or custom components.

- **Loose Coupling.**

  Most of JADE was designed with an eye towards keeping classes loosely coupled. Dependencies between classes are minimized, and unidirectional associations are preferred over bidirectional. This generalization cannot always be met, but abiding by it guarantees that changes in interfaces and class behavior minimizes side effects and propagating code changes. Again, designing to this idiom emphasizes reusability, extensibility, customizability and pluggability.

- **Flexibility and Extensibility.**

  JADE was designed in such a way that there is a core infrastructure supporting the framework that can be leveraged in applications, but this infrastructure is not imposed on new functionality. Different projects can develop their own modules of functionality without buying into a particular architecture. New designs for unforeseen use cases are easily done in isolation of the JADE framework, but can leverage the framework at the same time, as necessary.

- **Customizability.**

  JADE’s modular design and flexible class hierarchies provide a simple means of customizing parts of the framework. For example, JADE provides classes that acquire metadata, retrieve actual data, render the data, etc. Each of these classes are reusable as is, or can be customized into specialized components specific to a project’s needs.

- **Configurability.**
Most JADE components provide support for XML configuration. This capability allows applications to be configured differently, and deployed without requiring code changes or recompilation. In addition, the framework provides an infrastructure for XML configuration support, and can easily be applied to new functionality as needs arise. The XML configuration also provides the glue for the application, in that several application objects are created and associated to each other via XML.

- Design Patterns.

A design pattern is defined, in the author’s opinion, as “an excellent, well known object-oriented solution to a common problem”. Many proven designs that solve common design problems have been published as design patterns, and as a result, several of them are well known (eg: Gang of Four\(^1\)). JADE leverages several of these solutions in order to avoid reinventing the wheel, and to promote a common vocabulary for communication. Though design patterns are not always applicable, JADE attempts to apply them where appropriate.

- Deployment mechanisms.

JADE is a framework written in Java and has adopted the Model-View-Controller (MVC) design pattern, and as a result allows several mechanisms for application deployment: desktop applications, Webstart applications and browser-based HTML applications. The latter is known as the Remotely Operated Visualization Environment (ROVE) and is covered later in this document. However, each of these deployment options provide a substantial improvement over the previously adopted Applet-based applications.

- Documentation.

Though JADE has not been completely documented, nor will it ever be, there has been a concerted effort to provide a base of documentation. Most of the code contains Javadoc-style comments, and as a result, is published to RAL’s internal website nightly. This provides code level documentation and has obvious benefits. In addition, Unified Modeling Language (UML) documents have been created that diagrammatically illustrate key portions of JADE’s architecture. As development continues, a mechanism is in place to publish new UML diagrams as created, and to integrate them into the generate Javadoc HTML. Lastly, work is currently being done on a JADE website, both one that is internal to RAL and one that has public access, and will contain links to the JADE API documentation as well as documents such as this one.

V. Design Overview

The Model-View-Controller Design Pattern

JADE has been designed using several of the “Gang of Four” design patterns. Design patterns promote good object-oriented design by providing proven architectural solutions. In addition, they improve understandability by allowing the developers to speak a common language. Several areas of the JADE architecture leverage design patterns, where applicable, but the most prevalent pattern adopted by the framework is the Model-View-Controller (MVC). JADE applies the MVC pattern at several layers of abstraction, but this paper will only cover how the pattern applies to the core of the framework.

The MVC design pattern enhances modularity by dividing the components into model, view and controller classes that each have clear and distinct responsibilities. Figure 1 illustrates a typical MVC architecture as adopted by JADE, in the Unified Modeling Language (UML) syntax. The Model classes represent the business objects that exist within the framework. They represent a “model”, in the true sense of the word, for another part of the framework. The Controller classes consist of the components that affect the model and cause the business objects to change. The View classes represent the components within JADE that display the model or the business objects.

The Model, View and Controller classes are loosely coupled as can also be seen in Figure 1. The Model classes exist on their own without dependencies to Controllers or Views. Controller classes exist with associations to Model classes allowing them to affect the Model, but without dependencies on View classes. View classes exist in the context of an associated Model to display and use the Controller classes to affect the model. Each of these relationships is unidirectional which keeps the dependencies to a minimum and provides the advantages of loose coupling. Communication from the contextual Model to the interested View classes is done via a higher level interface, such as a BusObjectListener, that the View implements. Since the View is concerned about Model changes, it registers with its model in order to receive notification events when changes occur, usually due to a Controller.

JADE’s Core Model Classes

Figure 2 depicts the core Model classes that comprise the JADE framework, as well as their relationships. JADE’s Model classes consist of business objects that represent the domain of 2D visualization, and therefore JADE has abstractions such as DataLayer. The DataLayer represents a single layer of data and is what allows for the overlaying of data. DataLayer is a central abstraction in JADE and has several responsibilities that are delegated to other objects, as delegation is a design technique that promotes modularity. These objects are aggregated with DataLayer and help the DataLayer get its job done.

Each DataLayer has a set of DataDesc objects that represent what data sets are available for the DataLayer. These objects are not the actual data, but rather the meta-data that describes what available data sets physically exist for that DataLayer. The acquisition or finding of these DataDescs, as well as the notion of a current DataDesc for a DataLayer, is delegated to an abstraction called a DataDescFinder. The DataDescFinder is the class that is responsible for finding a set of DataDescs (aka: “Blue Dots” since that’s how they typically appear in a JADE application such as JViz) - whether the DataDescFinder hits a database or looks for several files, or even communicates with a server. Regardless of the physical representation of the DataDesc or the mechanism or protocol used to determine the set of available DataDescs, the DataDescFinder
"finds" the available blues dots, returns them to the DataLayer, and has a single “current” DataDesc.

Given that a DataLayer has a current DataDesc, the DataLayer delegates the actual retrieval of its data to an abstraction called a Retriever. The Retriever is responsible for retrieving the data for a given blue dot, or DataDesc. Each DataLayer may or may not have a set of attributes or properties (eg: altitude) that it introduces as constraints for the Retriever to use when retrieving data. When the DataLayer needs to load a new DataDesc/blue dot, it gathers up a set of constraints into a Map, and tells the Retriever to retrieve for a specific blue dot and a given Map of constraints. The Retriever does the retrieving using whatever mechanism or protocol it requires - it may hit a database, hit a server, or read a file. Regardless of the retrieval mechanism, the Retriever will ultimately instantiate and return a ValueObject, which is the actual data for the DataLayer. The ValueObject is simply a marker interface that has no methods or attributes. The ValueObject is whatever the Retriever returns from it’s retrieve() method and the DataLayer keeps this ValueObject as its current data.

The five core Model classes in JADE are DataLayer, DataDesc, DataDescFinder, Retriever and ValueObject. Each of these are abstract superclasses that provide the core infrastructure to the model portion or business objects of the framework. These classes provide a family of classes that are typically subclassed into concrete classes in order to support a specific data format or protocol. A concrete subclass of DataLayer would expose attributes (eg: altitude) that are significant to that kind of DataLayer. A concrete subclass of DataDesc would expose attributes that are significant to that kind of meta data, such as a physical field name like Temperature. The concrete DataDescFinders and Retrievers would contain code specific to the protocol being used.

By delegating the metadata acquisition and representation, the data retrieval and the data itself, JADE provides a simple mechanism for plugging in new data formats or protocols. In order to add a new data type or protocol to JADE, one may create a family of concrete subclasses of the core classes, or one may reuse some of the existing concrete Model classes that are part of the framework. For example, JADE provides support for a data format and protocol called MDV. In order to integrate this data type into the framework, there exists a family of MDV classes as shown in Figure 3. For each of the previously mentioned 5 classes, there is a concrete MDV subclass. By providing this family of classes, the framework is able to support the metadata, data retrieval and the data itself for the MDV format and protocol, and all of this functionality is represented as an abstract DataLayer.

Abstract DataLayers exist within an abstraction called a DLContainer, which contains an ordered list of DataLayers. The DLContainer typically represents a collection of DataLayers that may be overlaid on top of each other and are usually displayed in a single aggregate view using the ordering that the DLContainer maintains.

The Listener interfaces as shown in Figure 2 are the remaining Model abstractions that warrant discussion. As is typical with an MVC architecture, Model classes communicate to registered clients or Views through an abstracted interface. Another Gang of Four design pattern, called Observer, has been adopted in JADE to provide this communication mechanism. This style of communication guarantees that the Model classes do not have tightly coupled external dependencies. The DataLayer objects in JADE communicate to their Views through the loosely coupled DataLayerListener interface. This interface contains notification methods that pertain to changes to the DataLayer’s attributes (such as display name), visibility, data or DataDescs. In addition, DLContainers communicate to their registered listeners through the DLContainerListener interface. This interface contains notification methods that pertain to DataLayers being added, removed or reorganized within the
DLContainer. Through this mechanism the Views are notified of changes and can reflect them in the display(s).

Views are notified when changes occur to the Model, and changes occur on the Model when some sort of Controller affects it, or invokes a method. The sequence of events that occur when a change is made to the Model is best illustrated through an example, as shown in Figure 4. To put the sequence diagram in context, assume that the MdvDataLayer (which is a concrete subclass of DataLayer) has a current blue dot or DataDesc (metadata) describing the data it currently holds, which is its ValueObject. The diagram illustrates what happens when the MdvDataLayer’s altitude is changed by calling the setAltitude( altitude:double ) method. In the MdvDataLayer class, this method sets the layer’s altitude attribute which requires a reloading of data. To trigger the data retrieval, the setAltitude( altitude:double) method invalidates the DataLayer’s ValueObject. This causes the DataLayer to delegate the re-retrieval of data, given the current DataDesc and a Map of constraints, to the Retriever. The MdvDataLayer’s implementation of the getConstraints():Map method adds the altitude constraint to the map, thereby introducing the constraint to its Retriever. Finally, the Retriever is told to retrieve() and is passed the current DataDesc and Map of constraints. Once the Retriever instantiates a new ValueObject and returns, the DataLayer is able to fire an event to all registered DataLayerListener notifying them of the new data.

JADE’s Core View Classes

The View classes within the JADE framework are the objects that play a role in displaying the business objects or the Model objects. View classes all exist within a context of a Model in that they exist solely to display their business object(s). Therefore, View objects each have a reference to a Model context that is unidirectional in order to maintain loose coupling.

The main View classes are the DataLayerView, DataLayerRenderer and DLContainerView, as show in Figure 5. The DataLayerView represents a display of a single DataLayer, and will typically be seen in a 2D visualization product such as a map, SkewT or time series. Since the DataLayerView is a view of its model, the DataLayerView implements DataLayerListener and listens for changes coming from its DataLayer context. When the DataLayerView is notified of changes, it updates its display. Similar to the DataLayer, the DataLayerView delegates much of its responsibilities to other classes. The DataLayerView contains a cached BufferedImage which represents the rendered image of the DataLayer. The DataLayerView delegates the rendering of its DataLayer model to a DataLayerRenderer, which renders the DataLayer’s ValueObject into the DataLayerView’s BufferedImage. The decoupling of the rendering from the DataLayerView abstraction is what allows custom rendering, as custom DataLayerRenderers can be plugged into DataLayerViews per the display needs of a specific project. Lastly, when the DataLayerView’s image is redrawn, all registered DataLayerViewListeners are notified. The DataLayerViewListener interface is a View abstraction that allows clients to register as a listener on a view so that they can be notified when a view changes.

In the same respect that a DataLayerView represents the view of a single DataLayer, a DLContainerView represents the view of a entire DLContainer. The DLContainer has an ordered collection of DataLayers, and likewise, a DLContainerView has an ordered collection of DataLayerViews, where one DataLayerView exists for each DataLayer in the DLContainer. Figure 5 shows that the DLContainerView contains a master BufferedImage. The master image is a composite or stacking of all the DataLayerViews within the DLContainerView into a single integrated image. Note also that the DLContainerView
implements DLContainerListener and DataLayerViewListener. These implementations allow the DLContainerView to listen for DLContainer changes (ordering, layer added/removed), as well as DataLayerView changes (one of its view’s images changed). In both cases, when changes occur and the DLContainerView is notified, it restacks each of its DataLayerView images into the single master image. This sequence of events is shown in the Figure 6 sequence diagram. Lastly, the DLContainerView is a subclass of the javax.swing.JPanel class which allows it to easily be placed into any Java application.

JADE’s Core Controller Classes

The Controller classes within the JADE framework are the objects that affect the model, and therefore, have references to the Model objects that they affect. Again, the references are unidirectional such that Model objects do not have dependencies on the Controllers. Figure 7 illustrates the most significant Controller objects within the JADE framework, and shows that the Controller design within JADE is based on the Gang of Four design pattern called the Visitor.

The Visitor pattern uses a technique called double dispatch, and allows behavior that one would normally expect to be in a target class, such as DataLayer, to be in a separate abstraction called a Visitor object. By putting functionality and behavior into the Visitors, one can keep the interfaces of classes within a hierarchy small and uncluttered. Visitors operate on abstractions called Visitables that typically exist in a tree. When a Visitor is ready to operate on a tree of objects, it is given the opportunity to “visit” each of the Visitables in a tree. Each Visitable in the tree is told to accept the Visitor via the acceptVisitor( v:Visitor ) method, and the Visitable in turn calls the Visitor’s visitXYZ( :XYZ ) method, where XYZ is the concrete implementation of the Visitable. This is what’s known as double dispatch.

The abstract Controller class within JADE has a list of target Visitables that it operates on, and has methods such as addTarget( v:Visitable) as a result. This list of Visitables is effectively a tree of any sort of component that implements the Visitable interface. In JADE, DataLayers and DLContainers implement the Visitable interface, and become the nodes in a Controller’s tree. The Controller operates on the nodes within the tree by delegating its behavior to a Visitor. The Visitor is instantiated by calling an abstract method makeVisitor():Visitor which is implemented by concrete subclasses of the Controller class. This design allows an infrastructure to exist for Controllers, but also allows the behavior of each Controller to be pluggable.

The concrete Controllers within JADE have attributes that are significant to that Controller. For example, Figure 7 depicts a AltitudeController class that has attributes relevant to what you might imagine an altitude controller to have, in the context of a 2D weather display application. The AltitudeController has a min/max altitude range and a current altitude. Since the AltitudeController may need to be represented in a user-interface, it may act as a Model (another application of the MVC design pattern) to some sort of AltitudeControllerView (GUI). The AltitudeControllerView may display the state of the AltitudeController, and may translate user events into method invocations on the AltitudeController object. Figure 8 illustrates exactly this sort of design, and shows what happens when a user changes the current altitude on the AltitudeControllerView. When the AltitudeController’s setCurrentAltitudeRange() method is called, it makes a ChangeAltitudeVisitor (pluggable) and allows it to visit each of the targets within the Controller. The Visitor visits by calling acceptVisitor( :Visitor ) on the specific Visitable target, which then calls, for example, visitDLContainer( :Visitable ) on the Visitor. At this point, the Visitor is given the opportunity to do what it needs to do to change the altitude on the Visitable. In the example
illustrated, it’s clear that not much is done, but there are cases where the Visitor contains significant logic to execute, that is cleanly kept out of the concrete Visitable classes.

**JADE’s XML Support**

It’s clear that there are many components in JADE and that most of them fall into the categories of Model, View or Controller. However, how does one create an application that uses these components? Applications can be built up from these components by simply writing code to instantiate the components, set significant attributes, and then gluing them together by calling various setter/getter methods. This is the standard way to create applications, but results in hard coded values (such as URLs) placed into the code. Changing the values requires editing the code, recompiling the application, and redeploying it. This technique is tedious and not very flexible or customizable.

JADE provides configuration support in the form of XML configuration files. Most of the JADE core classes implement an interface called JadeXMLizable, which indicates that the specific class can be instantiated and configured from XML, and even saved to XML. This design allows most of a JADE application to be specified in a configuration file. For example, an application’s XML file may specify the instances of DataLayers and their attributes, the DLContainer that contains the layers, the DataLayerViews and their associated DataLayerRenderer with attributes, and the TimeController with its attributes (time range, current time) and targets. An overview class diagram depicting JADE’s primary XML classes is shown in Figure 9.

A configuration file for a JADE application will contain several XML Elements and Attributes (“Element” and “Attribute” are XML concepts) that describe the object instances and their properties to be used within the application. Each of the XML Elements represents an instance of a class, and may have child XML Attributes that represent properties of that instance. In JADE, each Element requires a class name and an object id, and an example is shown below.

```xml
<AltitudeController id = "AltitudeController"
   className = "edu.ucar.rap.jade.controller.AltitudeController"
   currentUnits = "pressure"/>
```

The className attribute is used by JADE’s XML loading mechanism to specify the class of the object to instantiate. The id attribute is a unique identifier for the instantiated object that can be used by other objects to refer to the new object.

The JadeXMLizable interface is what indicates that a class in JADE is configurable via XML. Each object created via XML must implement the JadeXMLizable interface. When a configuration file is loaded within a JADE application, a JadeXMLFactory object (Factory is another Gang of Four design pattern) is used to create each of the instances found within the file. The mechanism for creating the JadeXMLizable objects is a two pass operation. First, the JadeXMLFactory takes the className attribute from each XML Element and uses a default, no argument, constructor for instantiating that JadeXMLizable. After the object is created, the XML Element used to create the JadeXMLizable is passed to the new instance by calling its initFromXML( e:Element, bucket:Map) method. This gives the new object the opportunity to pull its attributes and properties from the XML Element that was used to create and configure it in the XML file. Also, during this initialization step, the object must add itself to the passed Map using its id as the key. During the second pass of initialization, each JadeXMLizable that was created from the configuration file is given the opportunity to resolveXMLObjectRefs( e:Element, bucket:Map), which allows each object to resolve object references by pulling external objects from the given bucket/Map by their id.
The sequence of events that occurs when loading an XML file into a JADE application is shown in Figure 10.

This design provides a flexible mechanism for configuring JADE applications. New classes that are written simply implement the **JadeXMLizable** interface and associated methods. The infrastructure provides the mechanism for instantiating the objects from the configuration file and gives them the opportunity to initialize themselves with attribute values and resolve their external object dependencies. As a result, JADE’s XML support provides a simple means for configuring attributes, as well as gluing the pieces of an application together through resolving references.

Though XML support exists for most of the JADE core classes, it does not exist for all of them. Some non-JADE applications exist that allow the entire application to be configured, including entities such as the buttons that appear, window locations, and the overall look-and-feel of the application. For JADE applications, a compromise was made in that application developers are responsible for creating their applications to have a certain set of components (usually user interface components), and a certain layout and look-and-feel. In other words, the developer is responsible for building the application itself, and then using **JadeXMLizable** objects within that application. This division of responsibility from what the developer is required to do and what the core framework provides ensures that the framework is not overly flexible and doesn’t demand that everything is configurable. That being said, there is nothing in the framework that excludes this idea of building the entire application from XML. A good example as to where this rule is broken is in the concept of MenuGenerators. JADE’s core framework provides reusable menu items, along with their associated functionality, as pluggable entities that can be configured into an application via XML. This design is an application of another Gang of Four design pattern called the Command pattern.

Sample Application

The components of the JADE framework and their corresponding support for XML configuration combine to make it easy for the software developer to build 2D visualization applications. JADE is a framework and not an application in itself, and therefore requires the developer to build their own application. In the example that follows, the developer’s intention is to create a configurable application that leverages JADE classes to create a display of a temperature field. The example is not totally complete, but it illustrates how one configures JADE components and then uses them in an application.

The configuration file contains XML Elements that define a number of objects to be used by the application, each of which implements the **JadeXMLizable** interface and therefore has XML support. First, all objects in the configuration file exist within a root Element called the JAM. The JAM, or Jade Application Model, is simply the root of the JADE application tree and contains the master bucket of all the **JadeXMLizable** objects instantiated. Second, the developer specified two time ranges, a min/max time range and a current time range, to be used by a configured TimeController. The TimeController is also given a reference to one of its targets, which is the **DLContainer** defined next in the file and is the target that the TimeController will affect. Next, the developer configured a DLContainer that contains a single DataLayer (of type **MdvDataLayer**) with its associated DataDescFinder and Retriever and their associated attributes. Lastly, the developer specified a **DLContainerView** whose context is the **DLContainer** (id = “container1”) defined previously.

```xml
<JAM id = "jadeApp"
    className = "edu.ucar.rap.jade.appcore.JAM" >
    <JadeTimeRange id = "currentTimeRangeId"
        className = "edu.ucar.rap.jade.util.JadeTimeRange">
        <StartTime beforeNowHours = "0" />
        <EndTime afterNowHours = "0" />
    </JadeTimeRange>
<DLContainer id = "container1"
    className = "edu.ucar.rap.jade.util.DLContainer" >
    <DataLayer id = "temperatureLayer1"
        className = "edu.ucar.rap.jade.util.MdvDataLayer" >
        <DataDescFinder />
        <Retriever />
    </DataLayer>
</DLContainer>
<DLContainerView id = "view1"
    className = "edu.ucar.rap.jade.util.DLContainerView" >
        <Context>"container1"</Context>
    </DLContainerView>
</JAM>
```
The above XML file is nearly all one needs to create a basic JADE visualization application. DLContainerViews create a default DataLayerView for each DataLayer in its contextual DLContainer. In addition, the DLContainerView uses a DataLayerRendererFactory to create a default DataLayerRenderer for DataLayerViews that do not have one associated with them. Both the DataLayerViews used for a DataLayer and the DataLayerRenderer for the associated view can be overridden within the XML, but this was omitted from the example for simplicity. Also for simplicity, a JadeXMLizable ColorScale object was omitted.

Now that the developer has defined the contents of the configuration file, the application itself must be written in the form of a Java class. In the example that follows, SampleApplication is the name of the application that lives in the SampleApplication.java file. The developer has created both the configuration file, called sampleApp.xml, and the sample application, and therefore knows that certain objects are expected within the XML file. The application code simply loads the JadeXMLizable objects from the configuration file using the JAM’s static utility method called loadModelFromXML(URL, boolean). After this call, each of the JadeXMLizable application objects are available from the JAM’s get(id: String):JadeXMLizable method. Since the developer knows that a DLContainerView, whose id is “dlcView”, was defined in the XML, the call is made to retrieve it from the JAM. Lastly, since the DLContainerView extends javax.swing.JPanel, the DLContainerView is placed into a JFrame, and the JFrame is made visible. Additional operations happen behind the scenes, such as the acquisition of the MdvDataLayer’s metadata, the setting of the time range of interest and current time, and the actual data. However, assuming data exists, this application would display a JFrame with a map of a Temperature grid for the time at which it was run.

```java
public class SampleApplication
{
    public static void main( String args[] )
    {
        ClassLoader loader = SampleApplication.class.getClassLoader();
        URL xmlFileUrl = loader.getResource( "sampleApp.xml" );
```
JAM jam = JAM.loadModelFromXML( xmlFileUrl, false );
DLContainerView viewPanel = (DLContainerView) jam.get("dlcView");

JFrame frame = new JFrame( "Sample Application" );
Container contentPane = frame.getContentPane();
contentPane.add( viewPanel, BorderLayout.CENTER );
contentPane.setVisible( true );

JADE Design Principles - Summary

The JADE framework provides a powerful toolkit from which developers can create custom applications. With a working knowledge of the framework’s components and how they collaborate with one another, it’s a simple procedure to create configurable JADE applications. By combining XML support with a wide selection of components available in the framework, actual application code can be kept to a minimum, thereby decreasing development time and increasing maintainability.

JADE’s modular design allows components to be easily customized and extended. Modularity increases the maintainability of the framework by minimizing side effects from code changes. The use of design patterns makes the architecture understandable, and the various class hierarchies make the architecture flexible and extensible. The XML support provides the configurability, and the glue for easily leveraging customized components. All of these qualities of the design provide a very pluggable architecture.

JADE directly allows the integration of new data types and protocols. Project specific applications can leverage the existing JADE components for supported data types/protocols, or code can be written and included for new data types and protocols. The latter approach eliminates the need for pre-converting data to a particular physical structure.

Application scalability is achieved through the modular design and configurability of JADE. In many cases, abstract class hierarchies are kept in the core of the framework, whereas concrete implementations are separated into external packages. This structure eliminates the need to include unused portions of the code into applications. Concrete implementations that are included in an application are glued into the application dynamically via XML configuration files.

The open design of JADE provides substantial infrastructure that can be leveraged by applications, but does not impose any infrastructure outside of the core framework. Project specific applications can be built that use core components of JADE, but also provide their own infrastructure and functionality without buying into a particular architecture.
VI. JADE Applications

Several projects within RAL are now leveraging the JADE framework within project specific applications. Several developers now work on the core JADE framework, both fulltime and part time, and the core of functionality is growing and remaining simple, extensible, flexible and maintainable (in most cases ;-). In addition, other developers are leveraging the framework to create custom applications, and as a result, a common knowledge base is developing which has clear and obvious benefits.

Since JADE is written in Java and was designed with the MVC design pattern, several application deployment options exist – standard desktop applications, Webstart-able applications and browser based web applications. Each of the applications listed below are being developed as Webstart applications, and can therefore also easily be converted to desktop applications.

**JViz** is the JADE application that is being used for the Army Test and Evaluation Command (ATEC) project. It provides real-time and archival visualization of weather products, in the form of maps, SkewT plots and vertical cross-sections, both for model and observational data. It has been successfully deployed at 6 test ranges and is in operational use. Appendix A contains a user tutorial for the application, and illustrates many of the features of the JADE framework.

For the FAA’s Aviation Digital Data Service (ADDS) project, the **Flight Path Tool** (FPT) applet has been converted to JADE and is close to being deployed. Similarly to JViz, the FPT provides real-time visualization with an emphasis on maps and cross sections of weather products, both model and observational data. The FPT is directed for use as a weather visualization tool for pilots, dispatchers and the aviation community, both private sector and military.

The **IcingViewer** is a JADE application that is a scientific visualization tool being used within the FAA’s Icing project. Again, this tool provides 2D visualization with an emphasis on maps and the viewing of columns of weather data. The SkewT functionality of JADE is extended for this application, to show SkewT plots side by side with vertical column displays of derived weather products.

**AthensViz** is a customization of the JViz tool that was used for monitoring the 2004 Athens Olympics. This is a project that was done for the Defense Threat Reduction Agency (DTRA), and was used to monitor the weather and potential impact of hazardous materials during the Olympics.

The Defense Advanced Research Projects Agency (DARPA) has created a JADE application, called **PentagonViz**, to visualize weather and derived products in proximity to the Pentagon. The tool allows users to initiate transport and dispersion models, with corresponding geolocation and material property information, and to view model output results. This product will be used as a decision support system to aid in the management of hazardous materials scenarios.

For the United Arab Emirates (UAE), a JADE application has been developed as an extension of JViz, and is called **UAEViz**. The tool focuses on weather products around the UAE, and provides additional functionality in the form of user initiated data queries. Users are able to easily query a database and view results in JADE’s visualization products (eg: maps, SkewT, cross section) or tables.

Another JADE application is being done for the FAA’s Weather Support to Deicing Decision Making (WSDDM). This product is in Alpha and will provide a distributed version of the C++ tool CIDD.

Additional JADE based applications are being developed for the FAA’s National Ceiling and Visibility project (**CVis**), and the Civil Aviation Authority of Taiwan.
VII. The Future of JADE

The “TODO” list for the JADE framework contains over 400 items. Obviously, work will be done as priorities demand and resources allow, but the high priority areas of functionality that will be developed are as follows: Time Series plots, Time-Height Plots (Meteograms), Tables of data, Climatology support, NetCDF support, New/Cut/Copy/Paste Data Layers, export format capabilities and national scale data sets. In addition, a scientific visualization tool is on the horizon, and will be better targeted to allowing scientists to view their own data than other JADE applications, such as JViz.

Remotely Operated Visualization Environment (ROVE)

The JADE framework is primarily based on the Model-View-Controller design pattern. As a result, components exist for the Model or business objects, Controllers and Views. Most of the JADE applications created to date have been Swing-based desktop or Webstart applications. Swing is Java’s component library that provides a set of interactive user interface components. However, since the Model and Controller classes within JADE are independent of the View classes, it is “easy” to replace the View classes with a different sort of view, such as HTML.

Browser based applications typically deal with HTML for client side visualization. In the J2EE realm, web based applications consist of Servlets and Java Server Pages, also known as JSPs (there are other J2EE technologies that compose web applications, but they are not considered here). ROVE is quickly becoming a Servlet/JSP based component library built on top of JADE, and will provide become a set of actions and custom tags that can be leveraged within web pages. As development progresses, ROVE will provide most JADE functionality within a web browser, rather than as a heavy weight interactive desktop client. Though this technology is not yet scaleable (to several simultaneous users), it will ideally provide another mechanism for JADE-based applications without the overhead of Webstart or installing applications. Ideally, this approach will also create opportunities for hand-held or PDA applications.
VIII. Figures

Figure 1. MVC Design Pattern

for each listener L
L.busObjectChanged(event)
call
fireBusObjectChangedEvent()
Figure 3. Model Classes for the MDV Data Format and Protocol
Figure 5. Core View Classes (shown in green)
Figure 6. Redraw Sequence Diagram
## IX. Appendix – JADE Feature List

### General

<table>
<thead>
<tr>
<th>Feature</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone applications without code, map or configuration file on startup</td>
<td>X</td>
</tr>
<tr>
<td>Standalone applications without dependencies on browser’s JVM</td>
<td>X</td>
</tr>
<tr>
<td>Small, maintainable applications</td>
<td>X</td>
</tr>
<tr>
<td>Save/Load user preference configurations</td>
<td>X</td>
</tr>
<tr>
<td>Export Images to png format (maps, cross sections and SkewT)</td>
<td>X</td>
</tr>
<tr>
<td>Configurable time ranges and time intervals</td>
<td>X</td>
</tr>
<tr>
<td>Real-time mode (updating, real-time data acquisition)</td>
<td>X</td>
</tr>
<tr>
<td>Metadata representation as “Blue Dots”</td>
<td>X</td>
</tr>
<tr>
<td>Dockable toolbar buttons</td>
<td>X</td>
</tr>
<tr>
<td>DataLayer status text</td>
<td>X</td>
</tr>
<tr>
<td>Cut/Copy/Paste/Rename DataLayers</td>
<td>X</td>
</tr>
<tr>
<td>“New” DataLayers – not quite done</td>
<td>X</td>
</tr>
<tr>
<td>Layer Order Manager (up/down)</td>
<td>X</td>
</tr>
<tr>
<td>Support for different data vs view projections</td>
<td>X</td>
</tr>
<tr>
<td>Column Viewer</td>
<td>X</td>
</tr>
<tr>
<td>Altitude Control</td>
<td>X</td>
</tr>
<tr>
<td>Altitude units</td>
<td>X</td>
</tr>
<tr>
<td>Time “lockable” data layers</td>
<td>X</td>
</tr>
</tbody>
</table>

### Data Types and Formats

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mdv</td>
<td>X</td>
</tr>
<tr>
<td>FIDO</td>
<td>X</td>
</tr>
<tr>
<td>SPDB</td>
<td>X</td>
</tr>
<tr>
<td>Shapefiles</td>
<td>X</td>
</tr>
<tr>
<td>GeoTIFFs</td>
<td>X</td>
</tr>
<tr>
<td>ADDS DataServer</td>
<td>X</td>
</tr>
<tr>
<td>Topography</td>
<td>X</td>
</tr>
<tr>
<td>Metars</td>
<td>X</td>
</tr>
<tr>
<td>TAFs</td>
<td>X</td>
</tr>
<tr>
<td>Airmets/Sigmets</td>
<td>X</td>
</tr>
<tr>
<td>VORs</td>
<td>X</td>
</tr>
<tr>
<td>SAMS</td>
<td>X</td>
</tr>
<tr>
<td>Lightning</td>
<td>X</td>
</tr>
<tr>
<td>Feature</td>
<td>Status</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Satellite</td>
<td>X</td>
</tr>
<tr>
<td>Radar</td>
<td>X</td>
</tr>
<tr>
<td>Wind Barbs with density control</td>
<td>X</td>
</tr>
<tr>
<td>Lat/Lon Reference</td>
<td>X</td>
</tr>
<tr>
<td>ETA/GFS</td>
<td>X</td>
</tr>
<tr>
<td>General Rendering</td>
<td></td>
</tr>
<tr>
<td>Pluggable rendering</td>
<td>X</td>
</tr>
<tr>
<td>Semi-transparent data layers</td>
<td>X</td>
</tr>
<tr>
<td>Maps</td>
<td></td>
</tr>
<tr>
<td>Resizing</td>
<td>X</td>
</tr>
<tr>
<td>Zoom/Unzoom</td>
<td>X</td>
</tr>
<tr>
<td>Pan/Unpan</td>
<td>X</td>
</tr>
<tr>
<td>Overview Frame</td>
<td>X</td>
</tr>
<tr>
<td>Preconfigured Areas of Interest (AOIs)</td>
<td>X</td>
</tr>
<tr>
<td>Snap to AOIs</td>
<td>X</td>
</tr>
<tr>
<td>Tooltips</td>
<td>X</td>
</tr>
<tr>
<td>GPS Tool</td>
<td>X</td>
</tr>
<tr>
<td>Configurable contours (min, max, delta)</td>
<td>X</td>
</tr>
<tr>
<td>Color-filled contours</td>
<td>X</td>
</tr>
<tr>
<td>Contour labeling (configurable fonts, colors, density, etc)</td>
<td>X</td>
</tr>
<tr>
<td>Gradient color scales</td>
<td>X</td>
</tr>
<tr>
<td>Configurable color scales</td>
<td>X</td>
</tr>
<tr>
<td>Progressive disclosure or density control</td>
<td>X</td>
</tr>
<tr>
<td>Various projections (Lambert, LatLon, Flat, PolarRadar, Stereographic)</td>
<td>X</td>
</tr>
<tr>
<td>Cross Sections</td>
<td></td>
</tr>
<tr>
<td>Predefined Flight Paths</td>
<td>X</td>
</tr>
<tr>
<td>Flight Path History</td>
<td>X</td>
</tr>
<tr>
<td>Resize</td>
<td>X</td>
</tr>
<tr>
<td>Animation over time</td>
<td>X</td>
</tr>
<tr>
<td>SkewT Plots</td>
<td></td>
</tr>
<tr>
<td>Model pseudo-soundings</td>
<td>X</td>
</tr>
<tr>
<td>Observed soundings</td>
<td>X</td>
</tr>
<tr>
<td>Feature</td>
<td>Status</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Semi-transparent data layers</td>
<td>X</td>
</tr>
<tr>
<td>Zoom/Unzoom</td>
<td>X</td>
</tr>
<tr>
<td>Tooltips</td>
<td>X</td>
</tr>
<tr>
<td>Wind Staffs</td>
<td>X</td>
</tr>
<tr>
<td>Map location references</td>
<td>X</td>
</tr>
<tr>
<td>Color coded soundings</td>
<td>X</td>
</tr>
<tr>
<td>Animation over time</td>
<td>X</td>
</tr>
<tr>
<td>Legends</td>
<td>X</td>
</tr>
<tr>
<td>Hide/Show families of lines</td>
<td>X</td>
</tr>
</tbody>
</table>

**Animation**

<table>
<thead>
<tr>
<th>Animation Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation Forward/Back/Sweep</td>
<td>X</td>
</tr>
<tr>
<td>Real-time Animations</td>
<td>X</td>
</tr>
<tr>
<td>Animation over any data set (time, altitudes, custom – not exposed in GUI yet)</td>
<td>X</td>
</tr>
<tr>
<td>Dwell</td>
<td>X</td>
</tr>
<tr>
<td>Delay</td>
<td>X</td>
</tr>
</tbody>
</table>
X. Appendix – JViz Tutorial
Section II: 4dwx Overviews and Application Tutorials.

L. JViz (JADE Visualization Tool).

This document provides the forecasters and other users with a high level overview of the 2-D data visualization tool called JViz, a description of its application to range meteorological needs, and a tutorial on its use in daily operations.

1. Overview.

The JViz tool is a configurable application that provides an intuitive user interface for viewing two-dimensional integrated data sets. The tool allows for configurable data layers to be displayed in maps for different times and altitudes, and provides standard map operations such as zooming, panning and querying data sets. For 3D data sets, vertical cross sections can be taken to show slices of data over a flight path. In addition, JViz provides the ability to view virtual and observed soundings in a Skew-T plot.

Figure 1 depicts what JViz would typically look like upon startup. The data layers, altitude and time are configurable and hence would affect what actually is displayed in the application.
2. JViz User Interface Overview.

The realestate of the JViz application is divided into functional areas as depicted in Figure 2. The main map for the tool is displayed in the center of the window. As in any application, the menu bar is located at the top of the window, and contains several options that will be covered later in this document. Near the top of the window, and on the left side are two different toolbars that contain shortcuts and tool buttons. The right hand side of the window contains an Altitude Controller that allows the user to look at data at different altitudes. In Figure 2, the "current altitude" is approximately 880 mb. The bottom of the screen contains the Time Controller and a list of the categorized data layers available for display. The Time Controller provides an interface for changing the application's current time, and is currently showing data for 0:00Z. Each layer listed below the Time Controller depicts the available data sets as "dots", lined up vertically to coincide with the Time Controller. Changing the Time Controller's "current time" will affect which data set (or dot) is selected for each layer. In addition, data layers can be turned on/off by clicking on the check box located to the left of each layer.

3. Data Layers and the Time Controller.

JViz is configured with a predetermined set of data layers. The data layers are grouped according to the configuration file (and the specific groups and layers may vary between ranges), and are displayed
at the bottom of the main JViz window. Each group is openable/closeable by clicking on it's respective expand/collapse button. In the example below, "Aberdeen RT-FDDA Layers" is a data layer group that contains several data layers, "Temperature", "Dewpoint", etc. Layers can be turned on or off by clicking on their respective checkboxes, and if the layer has an associated color scale, that color scale will appear in the dropdown as shown in Figure 2.

![Figure 3. Time Controller and Data Layers with Available Data Sets](image)

**Exercise:** Turn on/off one or more data layers. Note that the data for that layer now appears in the main map. If a color scale exists for the layers that you have turned on/off, identify them in the color scale dropdown.

The Time Controller appears above the list of data layers. In Figure 3, the Time Controller has a time range window of approximately 4 hours wide, from slightly before 12:00 Z to slightly after 16:00 Z. To the right of each data layer under the "Available Data Sets" column lies a "dot" indicator for each data set available for that layer. These data set dots are lined up vertically with the Time Controller, and appear gray when the data layer is disabled (off) and blue when enabled (on). For each data layer, the currently selected data set appears bigger than the rest, and if the layer is enabled, the selected data set dot will have a black circle around it. If the data layer does not have any available temporal data sets for the Time Controller's time range, then the words "Static Data" will appear to the right of the data layer in lieu of the dots.

4. **Time Controller Details.**

The Time Controller's "current time" can be changed by sliding it's orange slider bar to a different time. Changing the "current time" will change the current data set for each layer, by selecting the data sets that are on or before the "current time". For example, if one moved the Time Controller's slider to 14:00 Z, the 14:00 Z data set would be selected for the "Temperature" layer, and the approximately 13:52 Z would be selected for the "Infrared Temperature" layer. Changing the Time Controller's current time does not affect layers that have the text "Static Data" in lieu of data set dots.

**Exercise:** Turn on a layer that has data set dots, such as "Temperature". Drag the Time Controller's current time to a different time, and note the effect it has on the various data layers - each data layer's data will potentially change, as well as that layer's "current" data set dot. Identify the selected data set for various layers.

The initial time range window and current time for the Time Controller is determined by the configuration file. However, the Time Controller has a configuration window that allows the user to
Figure 4. Accessing the Time Controller Configuration Window

change it's time range and other attributes. This configuration window is accessible both from the "Configure -> Time Controller..." and the horizontal toolbar, as shown in Figure 4.

The Time Configuration Window, as shown in Figure 5, allows the user to change the start time and end time for the Time Controller's time range. In doing so, each layer will query it's server for available data sets that fall within that time range. This may include data sets in the past as well as data sets potentially in the future (eg: forecasts), as long as they fall within the new time range. In addition, the user can specify the Time Controller's tick interval and whether or not the Time Controller's current time slider will snap to the tick marks.

Exercise: Open the Time Configuration Window. Change the Start Time to be one hour before the original start time. Change the tick interval to "1 Hour", and turn off "Snap To Tick". Click "Apply" (note that this may take several seconds). Notice that the Time Controller in the main window will now reflect your change, and that the data layers will potentially have additional data set dots available. Move the Time Controller's current time slider. Open the Time Configuration Window again, and turn on "Snap To Tick".

Figure 5. Time Configuration Window
5. **Animation.**

Figure 6 depicts the various Animation Controller buttons that are visible within the main JViz window, to the left of the Time Controller. Pressing the "Play Backward", "Sweep" or "Play Forward" button will begin the animation sequence for the data layers that are currently enabled. One animation frame is created for each major tick on the Time Controller, and the frequency of ticks can be changed through the Time Configuration Window as shown in Figure 5. **Tip:** a high frequency of ticks or animation frames runs the risk of running out memory during animation (this bug is on the list of "TODOs").

During the first pass of animation, data is loaded and rendered for all enabled data layers, and is cached for successive iterations thereafter. As a result, the first pass of animation is usually slower than successive passes. When "Stop" is pressed, or when the user does anything (eg: resize) in the user interface, the animation cache is flushed. Animation should normally be stopped when the user wishes to change something in the application, though this is not required.

![Figure 6. Animation Controller Buttons](image)

The Animation Controller can also be configured via the Time Configuration Window's "Animation" tab, as shown in Figure 7. Both the minimum "Delay Between Frames" and the "Dwell at the End of the Loop" can be set in the units of milliseconds.

![Figure 7. Animation Configuration](image)

**Exercise:** Open the Time Configuration Window and verify that "Snap to Tick" is on, and that the tick interval is set to 1 hour. Click on the Animation tab, and change the delay to 100 milliseconds and the dwell to 0 milliseconds. Apply the changes. Experiment with stepping
the animation, and playing the animation forward, backward, and in sweep mode.

6. **Altitude Controller.**

JViz allows the user to change the current altitude of the main map by using the Altitude Controller. The right hand side of Figure 2 depicts the Altitude Controller that has an altitude range of 50 - 1000 mb, and a "current" altitude of approximately 870mb. The actual range and units for the Altitude Controller is determined intiially by the startup configuration file, and may appear differently in your display as a result. The current altitude can be changed by dragging the orange altitude slider to different altitudes, which will cause 3D data sets to reload accordingly. The units of the Altitude Controller are user configurable, and may be changed through the Altitude Configuration Window, accessible from either the horizontal toolbar or the "Configure->Altitude Controller..." menu item (see Figure 8).

![Figure 8. Accessing the Altitude Configuration Window](image)

**Exercise:** Turn on a "Temperature" layer or equivalent 3D data layer. Slide the Altitude Controller's current altitude to different levels, and note it's affect on the map. Access the Altitude Controller's Configuration Window, and change the altitude units to either Feet, Flight Level, Pressure or Meters. Note the affect on the Altitude Controller in the main JViz window.

7. **Different Kinds of Data Layers and Data Layer Configuration**

Although the different data layers are defined within the JViz configuration file, there typically will be certain types of data layers. Each data layer type exists as a category or group of data layers. Figure 9 shows a typical JViz configuration with 7 groups of data layers. Each data layer within JViz provides user configuration options which depends on the specific data layer, and the configuration is accessible from the "Configure" menu. Within the "Configure" menu, there exists submenus for each data layer group, and submenutitems for each data layer (see Figure 10). The data layer types and their configuration options are described below.
The "RT-FDDA" group contains data layers that typically have 2D or 3D model data sets, such as Temperature and Dewpoint. When enabled, these data layers will appear as gridded data sets within the main map, and each will have a color scale associated with it that will appear in the color scales dropdown. In addition, these data layers have configuration options as shown in Figure 11. For all RTFDDA layers one has the options to turn on/off the background grid and contours, and can change the layer's transparency. Most RTFDDA layers will also have color scales associated with them, and will allow the user to change the color scale's minimum and maximum, as shown in Figure 12. Tip: This becomes especially useful when the data being displayed has extreme values that fall off of the data layer's color scale.
The "Wind Speed" RTFDDA data layer will typically have additional configuration options to account for wind barbs. Figure 13 shows a sample configuration window for the "Wind Speed" data layer which allows the user to toggle wind barbs on and off, and increase or decrease the wind barb density.

(b) Satellite Grids
The "Satellite Grids" group contains data layers such as "Water Vapor" and "Infrared Temperature". The only configuration option provided for these data layers is transparency. **Tip:** Transparently comparing the satellite "Water Vapor" to the RTFDDA "Relative Humidity" layer near 400 mb can give a good indication as to how well the model data matches the satellite data.

(c) Surface Observations
The "Surface Observations" group typically contains data layers such as "SAMS", "Mesowest and WMO" and "Lightning". These represent data layers consisting of true observations, and provide the standard transparency configuration option. The "SAMS" and "Mesowest and WMO" data layers provide additional configuration options to toggle on and off the different fields within the observation, such as "Temperature". An example is shown in Figure 14. Note that these layers do not have color scales associated with them, and therefore will have disabled color scale options within the "Color Scale" tab.

In addition, these layers typically provide mouse over information (ie: ToolTips). For
example, if the "SAMS" data layer is enabled and has available data sets, SAMS will appear on the map. If the user "mouses over" the various SAMS in the map, they will see additional summaries of information that pop up on the screen when the mouse is over the observation, and that disappear when the mouse is moved away.

![Figure 14. SAMS Configuration](image)

(d) Soundings

The "Soundings" group contains virtual and observed sounding data layers that are described in the Virtual and Observed Soundings section of this document. These layers provide the standard transparency configuration option.

(e) ATEC Range Boundaries

The "ATEC Range Boundaries" group contains one static data layer for each of the ranges, and each simply consists of an overlay of the range's boundaries. These layers provide the standard transparency configuration option.

(f) Overlays

The "Overlays" groups contains preconfigured overlays of static GIS-like data, such as "Topography", "Rivers" and "Highways". Again, each provides the standard transparency configuration. Tip: These are large data sets and may hinder rendering performance if many of these are turned on at once.

The "Lon/Lat Reference" data layer will provide reference lines of constant latitude and longitude within the map. This layer provides additional configuration options to specify the longitude and latitude increments to be used when rendering, as shown in Figure 15.

Exercise: Experiment with the Configuration Options and ToolTips for various data layers.
8. Data Layer Order Manager

JViz allows multiple data layers to be visible at once, effectively layed on top of one another. If multiple data layers are visible, one may wish to move layers up and down, and can do so through the Layer Order Configuration Window. The Layer Order Configuration Window is accessible through both the horizontal toolbar and the "Configure->Layer Order..." menu item, as shown in Figure 16.

The Layer Order Configuration Window depicted in Figure 17, displays the names of all of the data layers within the main JViz map. Greyed out entries represent layers that are disabled/invisible, and black entries represent enabled/visible layers. The top most layers in the map are shown at the top of the layer list, and the bottom most layers in the map are shown at the bottom of the list. By selecting a specific data layer, one can enable or disable the layer, as well as move that layer to the top or bottom, or up and down one layer at a time.
**Exercise:** Access the Layer Order Configuration Window. Experiment with turning layers on and off, and moving them up and down.

9. **Data Layer Status Window**

Data layer status can be viewed with the Data Layer Status Window. This window shows data loaded and data validity times, which is especially useful for non-static data layers. The Data Layer Status Window is accessible via the button on the horizontal toolbar, as well as through the "View->Layer Status..." menu item, as shown in Figure 18. Figure 19 depicts a sample Data Layer Status Window, and shows that Temperature, Dewpoint and Relative Humidity are valid at 12:00Z on 9/18/03.
10. Lon/Lat Locator (GPS Tool)

The Lon/Lat Locator tool provides the user with longitude and latitude coordinates for the location of the mouse within the main JViz map. To access the Locator Tool, press the "GPS" icon on the horizontal toolbar, as shown in Figure 20. When the tool's window has opened, move the mouse around within the map, and notice that the longitude/latitude location is displayed in the locator tool's window.

![Lon/Lat Locator Tool](Image)

**Figure 20. Lon/Lat Locator Tool**

**Exercise:** Access the Data Layer Status Window. Access the Lon/Lat Locator Tool and identify the lon/lat coordinates of the mouse.

11. Zooming, Recentering and AOIs (Areas of Interest)

JViz provides standard map operations such as zooming within the map, recentering the map and a viewing predefined areas of interest. The zooming and recentering operations are accessible from the vertical toolbar on the left hand side of the map, as shown in Figure 21. Zooming is performed by depressing the Zoom button which remains depressed until a new zoom box is entered into the map, and changes the mouse cursor to a crosshair. To specify a new zoom box, click the the mouse in the map at the upper left of where you wish to zoom, and drag the "rubber-banding" rectangle down to the right until it's in the correct spot. The aspect ratio of the zoom box remains the same as the map, and can be changed by resizing the main window.

JViz performs "smart zooming" and will always attempt to give the user the "best" data possible. When one successively zooms down into the various domains of the ranges, such as zooming from DPG's Domain 1 down into DPG's Domain 3, JViz will return the highest resolution data available for the new zoom area.

The Undo Zoom button reverts back to the last view of the map, and can repeatedly be pressed until the original map view is reached. The Undo Zoom button works for all zoom/pan/AOI operations, and will consistently revert back to the previous map view.

Recentering is performed by depressing the Recenter button from the toolbar, and clicking within the map on the new desired map center. Again, the aspect ratio of the map is retained, and the clicked point will become the center of the new map view.

Areas of Interest (AOIs) are predefined within the JViz configuration file, and can be different for each of the ranges. The AOIs are accessed through the "AOIs" menu, and will cause the map to "snap" to the predefined Area of Interest. Currently, there is a "CONUS" Area of Interest that will revert the map back to the CONUS view.

**Exercise:** Experiment with zooming and the Undo Zoom button. Experiment with recentering the map. Move to various Areas of Interest, and occasionally see what effect the Undo Zoom button has. Experiment with "smart zooming" and identify when it occurs.
12. Vertical Cross Sections

Vertical cross sections can be viewed by specifying a flight path with waypoints within the plan view map. Flight paths are specified by depressing the Vertical Cross Section Button from the toolbar (see Figure 21), and then clicking within the map for each waypoint along the path. Each waypoint along the path initially appears as a number with a white background on the map, and when submitted, the first and last waypoints change to "S" (Start) and "E" (End) respectively. To submit the path and view the cross section, double click on the last waypoint, or right click and select "Submit Cross-Section". To cancel the flight path, right click and select "Cancel Path". Figure 22 shows a sample flight path through the plan view.

The Vertical Cross Sections Window can be viewed by submitting a new flight path, or by clicking on the cross section icon in the horizontal toolbar. A sample cross section is shown in Figure 23. The waypoints from the flight path appear as vertical lines labelled as in the plan view. The data layers that appear within the cross section window directly correspond to the layers that are turned on within the map view. For example, if Temperature is turned on in the plan view, it will appear in the cross section. Tip: With the exception of Topography, you can only see one layer at a time in the cross section (this bug is on the list of "TODos"). Only 3D layers will appear in the cross section, meaning that 2D fields such as metars or ground temperature will not.

Figure 22 also shows that there is an Altitude Controller within the Vertical Cross Sections Window. The current altitude of the plan view is represented as a solid black horizontal line through the cross section, and corresponds to the current altitude of the Altitude Controller within the Vertical Cross Sections Window. The plan view's and cross section's Altitude Controllers will always represent the same altitude range, current altitude and altitude units, and changing either will change the other.

User specified flight paths are stored in memory as a "History", as can be seen in Figure 23. This enables the user to switch back and forth between historical flight paths. In addition, JViz is preconfigured with a set of "Classic" flight paths (which can differ across ranges) and can be viewed at any time.
Figure 22. Sample Flight Path for Vertical Cross Section.

**Exercise:** Create several cross sections by defining flight paths within the plan view. Turn on and off various layers from the plan view, and note which ones appear in the cross section view. Switch back and forth between the flight paths you just created. Select a "Classic" flight path. Change the current altitude and see what effect it has on the cross section and the plan view. Clear the History. Close the window, and reopen it using the icon.
13. Virtual and Observed Soundings - SkewT

JViz provides the ability to view SkewT plots for both virtual and observed soundings. Both virtual and observed soundings exist as separate data layers within the plan view, as shown in Figure 24. These data layers represent the available sounding locations, either virtual or observed, for which actual soundings can be plotted within the SkewT.

The Virtual Soundings data layer contains sounding locations that have previously been specified by the user, and will initially contain zero sounding locations. In order to specify a virtual sounding location, first enable the Virtual Sounding data layer in the plan view. Depress the Virtual Sounding Tool button in the toolbar, and click within the map at the desired sounding location (note that one may have the Lon/Lat Locator window open to help with click accuracy). If the clicked point is at a location that is not covered by the forecast model, a dialog will appear notifying you that the click point is invalid. For valid click points, the SkewT window will then appear as in Figure 25 with a new data layer labelled "Virtual Sounding #", where # designates the first, second, third, etc virtual sounding you may have specified. In addition, the virtual sounding location appears with the same label in the plan view, and is contained within the plan view's Virtual Sounding data layer. Figure 24 shows Virtual Sounding 0, 1 and 2 on the plan view map, and shows the mouse over tooltip that is displayed when the mouse is brought over the Virtual Sounding 0 location.

Exercise: Turn the Virtual Soundings data layer on and off in the plan view. Note it's effect on the Virtual Soundings Tool button in the toolbar. Create a new virtual sounding somewhere in Colorado. Identify the new sounding location in the plan view, and the new sounding data layer in the SkewT.

Figure 23. Vertical Cross Section along a Flight Path
Observed soundings work in a similar fashion to virtual soundings. The plan view contains the Observed Soundings data layer that contains all of the observed sounding locations. The data set dots for this layer represent times for which observed soundings are available, and the particular set of available soundings will typically be different for different times. Figure 24 shows that there are different sets of observed soundings for 12:00 Z, approximately 12:50 Z, 14:00 Z, etc. To select a specific observed sounding, verify that the Observed Soundings data layer is enabled in the plan view. Depress the Observed Sounding Tool button in the toolbar, and then select one of the available observed sounding locations in the map. As with virtual soundings, the SkewT window will then appear with a new sounding data layer whose name and color matches the name and color of the sounding location in the plan view. For example, Figure 24 shows that the sounding location "KDNR" is blue, and Figure 25 shows a new data layer called "KDNR" whose plot is in blue.

**Exercise:** Turn the Observed Soundings data layer on and off in the plan view. Note it's effect on the Observed Soundings Tool button in the toolbar. Create a new observed sounding by selecting an available sounding location in the plan view. Identify this new sounding location in the plan view, and the new sounding data layer in the SkewT.
14. The SkewT Window

The SkewT Window is shown in Figure 25, and is accessed by creating new soundings (virtual or observed) or by clicking on the icon in the plan view's horizontal toolbar. The window looks similar to the main map in that the data layers are categorized and appear at the bottom of the window with their data sets represented as dots lined up with the Time Controller. The SkewT's Time Controller and animation controls are directly tied to those in the main map. Therefore, the SkewT's Time Controller and the main map's Time Controller will always coincide, and changing one will similarly affect the other. The data set dots for each sounding data layer represent available data sets for that layer. For virtual sounding layers, the dots represent times where model data is available and for observed sounding layers, the dots represent the time for which an actual sounding was made at that layer's sounding location. When the Time Controller's current time is changed, each sounding layer will attempt to load a new sounding (at it's specific location) on or before that time.

The rest of the SkewT Window contains the SkewT plots and the wind staves. Data tracings and labelled wind barbs are shown for each enabled sounding data layer. The data plots for each layer consist of both temperature in a solid line style and dewpoint in a dashed line style. Each tracing is color coded to match the color of the sounding data layer's location in the plan view. Mouse over tooltips are provided for each data point in a plot, as can be seen in Figure 25. In addition, each sounding data layer provides
transparency configuration in its respective Configuration Window, accessible through the SkewT window's "Configure" menu.

Two additional data layers exist within the "Overlays" group regardless of the number of sounding data layers. The SkewT Axes data layer contains the "Families of Lines" background which can be toggled on/off and has configuration options to turn on/off each family of lines. The SkewT Legend data layer can be toggled on/off as well, and when on, will display the color legend for each sounding data layer that is enabled. Transparency configuration exists for both the SkewT Axes and the SkewT Legend data layers.

The SkewT Window also provides zooming and unzooming within the main SkewT plot. These buttons are located on the SkewT Window's toolbar, and work in the same manner as the zoom/unzoom buttons on the plan view map.

**Exercise:** Turn the various sounding data layers on and off. Note the effect on the plots, wind staves and the legend. Identify the temperature and dewpoint tracings. Change the transparency for a sounding data layer. Change the Time Controller's current time and note it's effect on the plot(s). Change the transparency of the SkewT Axes and SkewT Legend. Turn on/off the various families of lines. Zoom and unzoom within the SkewT plot. Identify tooltips for the various plots.

### 15. Exporting Images

JViz provides the ability to export images from the main map, vertical cross section and the SkewT. Currently, .png is the only supported format (and other formats are on the list of TODOs). To export an image, locate the "File->Export Image..." menu item in either the main JViz window, vertical cross section window or the SkewT window. Selecting this option will open a file chooser, as shown in Figure 26. Find the directory you wish to save the file in, enter the file name with a ".png" postfix and click on "Save".

![Figure 26. Exporting Images.](image-url)

**Exercise:** Export a png image for the main map, skewT and cross section.