Configuration and Control of a Real-Time Mission Data Processor

by

Michael Molignano

Timothy Navien

Steven Shidlovsky

A Major Qualifying Project
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
in
Computer Science
by

_____

_____

_____

October 2009

APPROVED:

_____

Professor George T. Heineman, WPI Advisor

_____

Mr. Scot DeDeo, MIT Lincoln Laboratory Advisor

_____

Dr. David Holl, MIT Lincoln Laboratory Advisor

Abstract

Massachusetts Institute of Technology Lincoln Laboratory Group 39, Air Defense Techniques, supports national security through various efforts. One such effort is the Lincoln Laboratory Visualization Interface and Scalable API (*LLVISTA*). Although *LLVISTA* provides the necessary analytic functionality, it currently requires too great a degree of domain-specific knowledge for wide-spread adoption. This Major Qualifying Project developed a visual design tool named Slique for the real-time mission control and data analysis provided by *LLVISTA*. Developed within the Model-View-Controller design paradigm and adhering to software design principles, Slique is a modular, scalable, and extensible application to enhance productivity and increase usage of *LLVISTA*.

Acknowledgements

We would like to thank the following people for their guidance and assistance through our work on this project.

Mr. Scot DeDeo - MIT Lincoln Laboratory

Professor George T. Heineman - Worcester Polytechnic Institute

Dr. David Holl - MIT Lincoln Laboratory

Mr. David Oostdyk - MIT Lincoln Laboratory

Professor Ted Clancy - Worcester Polytechnic Institute

Mr. Vivek Varshney - MIT Lincoln Laboratory

Contents

List of Figures

Chapter 1

Introduction

The Major Qualifying Project (MQP) described in this report was completed at MIT Lincoln Laboratory under the supervision of Professor George T. Heineman (Worcester Polytechnic Institute), Mr. Scot DeDeo (Lincoln Laboratory), and Dr. David Holl (Lincoln Laboratory). The primary goal of this project was to develop a visual design tool that enables greater access to and adoption of an in-house software system known as Lincoln Laboratory Visualization Interface and Scalable Transport API (*LLVISTA*).

Described below is an overview of *LLVISTA* and how it was designed, deployed, and used within Lincoln Laboratory. Scot DeDeo, the current maintainer of this system, envisioned the need for a visual design tool that would enhance the effective use of *LLVISTA* and increase adoption by more users. The goal of this project was to develop a system that would meet Mr. DeDeo's specific requirements.

The project was scheduled for completion within nine weeks. Given this short timeframe, the team developed a rigorous schedule at the onset of the project. Adhering to this schedule ensured delivery of the required project deliverables. Full details on the schedule and

methodology are located in chapter 2. To fulfill the project requirements, the team designed a system architecture, described in chapter 3, capable of interfacing with *LLVISTA*. The team also developed a stand-alone application, based on the design architecture, for modeling and executing the computational processes used by *LLVISTA* analyses.

Chapter 5 describes the evaluation process used to verify the delivered system against the success criteria, defined at the onset of the project. This report is concluded by describing, in Chapter 6, areas of future work that can be performed to increase the capabilities of the system.

1.1 Background

MIT Lincoln Laboratory, a federally funded research and development center (FFRDC), develops new technologies in support of national security. To determine if these new technologies perform correctly, testing and validation must be performed to gather raw data and verify performance objectives. Some technologies may be particularly expensive to evaluate due to involvement of air, sea, and land assets. In these cases, it is critical to conduct some analyses in real time to ensure correct test mission execution. Upon test completion, detailed raw data recordings are run through more detailed analyses often requiring elevated compute power such as that offered by compute clusters which are often unavailable in the test environment. The coordination of real time and post mission analyses is labor intensive, but may be greatly accelerated by automated analysis and control systems.

One such system used for analysis is *LLVISTA*. This framework provides various types of components, including: data input components, data manipulation components, and graph-

ical user interface components. These components may be assembled into data processing pipelines and linked through the use of Groovy [7] scripts, a Java-like scripting language. Processing pipelines may be executed directly on live test data or on recorded data from past tests, and data export to tools such as MATLAB [11] is available for forensic analysis.

The main goal of *LLVISTA* is to allow dynamic and robust interconnections between data sources, data modification and manipulation stages, and visualization components. *LLVISTA* is built primarily in the Java [8] programming language with the addition of open source libraries. To facilitate the display of data, *LLVISTA* provides graphical user interface components which produce an output for each of the data streams registered. These graphical components present information on-screen as raw numbers, in a plot, on a chart, or through a map. This allows the *LLVISTA* framework to create user-customized Graphical User Interfaces (GUIs) allowing the creation of a visual display for analysts and test control operators. From these GUIs, the analysts can ensure correct test execution and diagnose any faults as they occur in real time.

LLVISTA is comprised of four tiers, each constructed against a set of interfaces allowing extensible communication between other tiers. The incoming data flows primarily down the tiers in a linear fashion starting at the highest tier, data sources. The data used by the application can come from almost any source and be of a variety of data types supported by *LLVISTA*. *LLVISTA* can currently process and distribute data such as: radar tracking information, missile path, and position data. The data source tier ensures all incoming data will conform to Oasis Data Stream Service [2] specifications. The data is then passed on to the data distribution tier through its respective interfaces. The data distribution tier is responsible for routing the data within the registered components and sources. Flexibility is

a key element to this tier, allowing many-to-many connections among the data sources and receiver components.

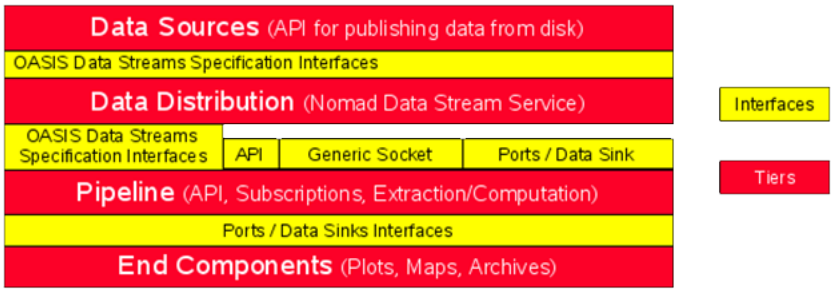

Figure 1.1: *LLVISTA* Tiers separated by their Interfaces

The last two tiers consist of the pipeline and end-component tiers. The pipeline tier is constructed from data manipulation components that alter their incoming data and forward it to the next component in the chain. Data is retrieved from the data distribution layer using various methods. The raw data is then transformed, processed, and converted into a viewable form. The user-created configuration file determines how the display (GUI) will look and which conversions will occur. This framework allows the analysts the most freedom and flexibility to modify and test the incoming information in a short amount of time.

The major advantages of this framework are its flexibility and extensibility. New components are able to be added easily, provided they conform to the interfaces for the tier they will extend. This flexibility comes at the price of usability. Using *LLVISTA* requires in-depth knowledge of software architecture and of the architecture of *LLVISTA* itself. This has thus far prevented wide spread adoption of the framework.

Each use of *LLVISTA* must be tailored for a specific scenario. Currently this process involves manually creating a groovy script to configure and run processing pipelines for

each test scenario. However, creating the scenario script can be time consuming, sometimes requiring upwards of two weeks per scenario. Interested parties must contact the only current developer, Scot DeDeo, for each configuration and modification required. Also, hand-creation of these scripts increases the likelihood of inadvertent user errors, and the hand-generated script must be tested to identify and eliminate any defects. Within each processing pipeline, all components and connections must be validated upon rendering of the script. As a scenario becomes increasingly complex by having more components and connections, the amount of time required to create and validate these scenario scripts also increases.

Chapter 2

Methodology

2.1 Schedule

The team was given nine weeks to meet the requirements set out by the sponsor. At the onset of the project a schedule was constructed to ensure that the development of the system proceeded at the correct pace. It also allowed the team to set project goals throughout the term so that all aspects of the project were completed on time and in an orderly fashion. To view the team's original proposed schedule, see Appendix A.

To provide for adequate time to completely test and validate the application, the team instituted two important dates. The first of these dates was the feature freeze. This was the date on which no more additional features would be added to the application. This prevents the final system from having half finished components that the team did not have the time to adequately finish. The feature freeze provided a week in which all in-development features could be finished and properly tested before the next major step in the process.

The second date that was set was the code freeze. At this point, no new code was to be

added to the system. This date forced the team to look away from development and finalize the project. The only code development that was to be done after this date was fixing defects found within the system. This gave the team time to finish the required documentation for the system and prepare all the working materials for hand off to the sponsor and maintainers.

2.2 Software Engineering Principles

During the course of the project, various software engineering principles were utilized. The usage of these principles provided the team with methods to ensure the proper execution of the project. The principles that were used are iterative development, source code control, documentation, and testing.

2.2.1 Iterative Development

The use of iterative development is an important aspect of any software engineering project. By utilizing this principle, the team was able to provide the sponsor a working product every week. This style of development benefited the team in multiple ways. The weekly releases provided a good indicator of the current progress of the project. With each iteration, the team came closer to fulfilling the requirements of the sponsor. Using this development style allowed the team to drop features that could not be completed in time so that more vital features could be included. Another benefit is that the sponsor was able to view a working system at the end of each week. It allowed them to provide feedback and opinions on the current direction of the project. This feedback was vital in molding the final system to fit the sponsor's needs as much as possible.

2.2.2 Source Code Control

Source code control is vital to any software project. Without it, work can be easily lost and no backups would exist in the event an issue arises in the current working copy of the code. For this project, source code control was maintained by a Subversion [4] server provided by Worcester Polytechnic Institute. Subversion assisted by allowing development of the code to occur disjointly but still be easily available to other members of the team. Since the Subversion repository was located at Worcester Polytechnic Institute it was accessible outside of Lincoln Laboratory. This allowed team members to work on the project while not at the Lab. Also, the repository handled merging files that had been modified by multiple times. Finally, Subversion provided the team with the ability to restore and review older versions of code. This ability allows team members to examine what occurred between commits and revert to older versions if necessary.

2.2.3 Documentation

On any large software project, documentation is crucial. Documentation makes it easier for others to understand the source code and allows the developer to remember what they were thinking when they wrote it. To ensure that the project had adequate documentation, all source code files were fully commented according to the JavaDoc industry standard. Doxygen [12], a tool that generates a report based on JavaDoc, was used to generate an illustrated report of the system.

2.2.4 Testing

To ensure system stability and correctness, testing must be performed. For this project, the JUnit [1] testing protocol was utilized along with the Eclipse plugin Eclemma [6]. JUnit requires that test methods be written that will test specific aspects of the code. The tests rigorously evaluated the system to ensure that the methods and classes performed as desired despite any invalid input and when other error conditions arouse. The Eclemma plugin will use these test cases to generate a code coverage report determining how much of the code is covered. The goal for the team was to achieve the industry standard of 80% coverage. The standard is set for 80% since it is not practical to achieve 100% code coverage on the application. There are many aspects of the system that can not be easily tested through unit testing, such as actions performed on the graphical user interface. Eclemma will color the source code based on if the code was executed in a test case or not. This allowed the team members to go back over the remaining code that was not tested through JUnit tests and perform code reviews. These code reviews ensured the code was functioning accurately and as expected. Rigorous testing helps to lower the possibility that defects will make it into the final release of the product.

Figure 2.2 shows how the test coverage of the team's code progressed over the course of the project. At the beginning of the project, test coverage was low until the foundation of the system was developed. Dips in code coverage occurred when major development was taking place but was quickly brought back up when testing occurred on the new code. As the end of the project is reached, code coverage jumps as the final tests are written to ensure that the test coverage meets industry standards.

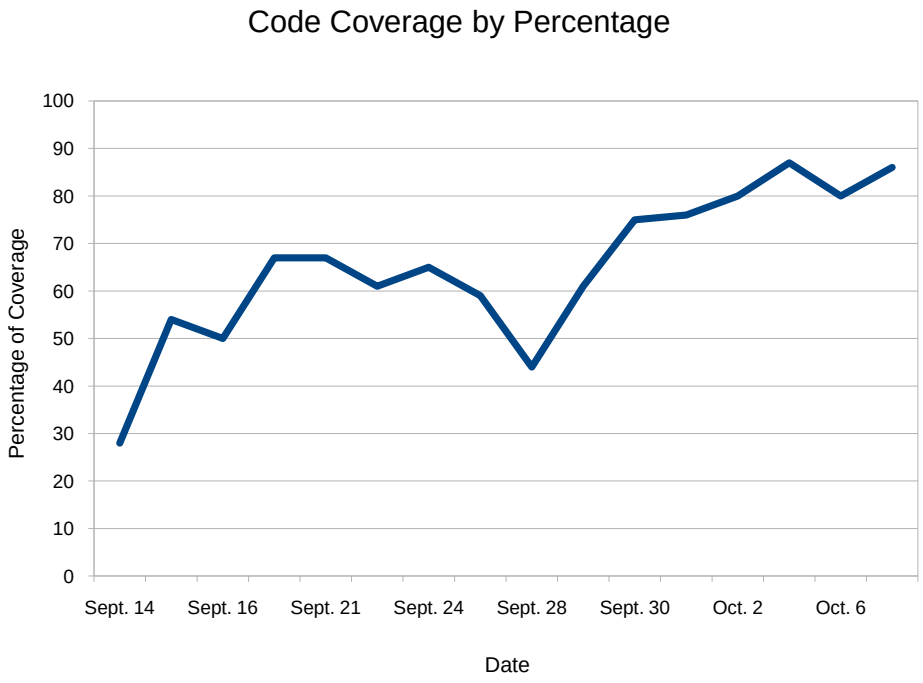

Figure 2.1: Daily code coverage by percentage over the lifetime of the project

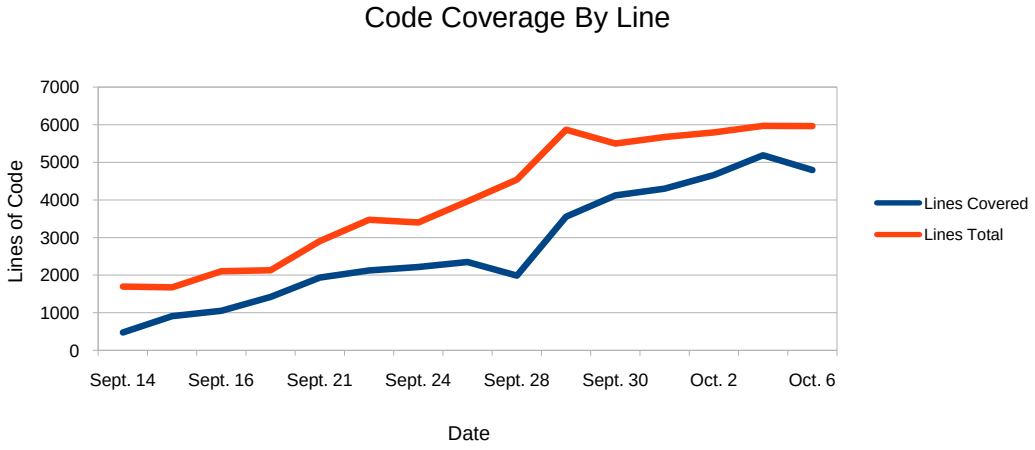

Figure 2.2: Daily code coverage by number of lines over the lifetime of the project

2.3 Designing the System

As with any project, the first step in uncovering a solution is understanding the problem. The team learned about *LLVISTA* by reviewing reports provided by the sponsor and analyzing examples of the hand-written Groovy scripts. With a basic understanding of what the sponsor desired for the project, brainstorming began on the use cases for the system. The use cases would form the foundation of all future design decisions as they provide the basic operations that the user is able to perform within the system. This provides a focus of what the most important aspects of the system will be and allow the design to fully cover these areas. To ensure that the requisite functionality was met, the use cases were reviewed with sponsors early. This prevented the team from adding in unnecessary functionality at the expense of missing functionality that the sponsor desired. After the use cases were finalized and approved, analysis began on the use cases.

To assist with the creation of the system architecture, different strategies were employed. The first was to utilize United Modelling Language [3] (UML) in constructing the layout of the system. The language allowed ideas to be quickly constructed and reviewed by peers to check the feasibility of an idea. Revisions to the ideas could be done quickly and then combined with other sections until the entire system had been created. The final UML diagram provided a foundation that all team members understood and could develop against.

Another strategy that was utilized was rapid prototyping. Some ideas, particularly ideas about the graphical user interface, could not be expressed in UML. Instead, it was more practical to turn these ideas into diagrams or short segments of code. During the team's rapid prototyping, different components were quickly created until a satisfactory piece had

been achieved. These pieces were constructed into a graphical interface that was shown to the sponsors for approval. After implementing suggestions from the sponsor, this graphical interface was later integrated into the system.

At the end of the design process, the team had finalized an architecture. The system would be structured around the Model View Controller [3] paradigm. Within this structure, several key design patterns, such as Singleton [3] and Observer [3], were incorporated. There are several immediate benefits to structuring the system using the Model View Controller. One benefit is that the structure of the system would be easily expandable. Additional features can be added in as singletons and connected to the rest of the system via controllers. Flexibility is gained from this structure as controllers are the primary method of communication between the disjoint views. There are no tight dependencies contained within the system. Separation of the graphical interface and the underlying code is another advantage of this paradigm. Graphical interface code is difficult to test, which would mean if the underlying code was attached to it tightly it would also be difficult to test. The separation between the two entities allows testing to be performed on the underlying code easily and gives the graphical interface the ability to call controllers that have been tested thoroughly.

As work continued on the system, inevitable changes occurred. These changes were often bits of information about the underlying framework that the team had not been informed about when the project was started, or were additional features that the sponsor wished to have added. Each change requested required inspection for feasibility within the time constraints and the current system process. The goal was to deliver a working system at the end of allotted time. As specified in the schedule (See Appendix A), a feature freeze was instituted on September 29, 2009. No additional features would be added after this date.

October 5, 2009 was the date that all functional code was completed by. After this date, the team restricted its programming to cleaning up errors in the code and improving the quality of the test cases.

2.4 Challenges

As the project progressed, various challenges were encountered. Each of these arising issues had to be dealt with in order to continue progress on the application.

2.4.1 Restricted Development Environment

At the onset of the project the team was informed that certain portions of the system were not accessible outside of Lincoln laboratory. One such aspect was the *LLVISTA* framework and its dependencies, this prevented the team from developing much of the application off-site. However, this constraint had a positive outcome. The team was able to focus more on modelling the system. Therefore the resulting application is loosely coupled with the *LLVISTA* framework, allowing the *LLVISTA* framework to be independent of the Slique application.

2.4.2 Java Swing Development

Although the team was aware of the graphical aspect of this project, only one member of the team had experience in developing graphical applications with Java's Swing framework. Throughout the project the team encountered many challenges in syntax, call-ordering, position, and event propagation while working within Swing. However, each challenge provided

greater insight into the workings of Swing and enhanced the resulting application.

2.4.3 Emerging Information About *LLVISTA*

When the project started, the team sought as much information about *LLVISTA* as possible. As the initial designs were created, the team believed that they had a good understanding of the framework. However, new information about more subtle aspects of *LLVISTA* were discovered. These new aspects primarily focused on the creation of the Groovy scripts.

Originally the team was under the impression all pipelines contained graphical components and, as such, the pipeline could inherit properties and information from a single source. Once the team discovered the possibility of non-graphical pipelines, the groovy generation process was modified to only inherit the relevant properties.

Another misguided assumption the team encountered was that these properties were stagnant. Each pipeline may have various properties, most revolving around the graphical display, which apply globally to many components or affect the model directly. In order to account for these properties being variable, a new class was added to expose them within the visual design tool. The team decided to create an additional tab within the properties window which displays a table of the properties and their associated values. This approached solved the problem of applying properties uniquely to the focused model.

Similar to the model properties it was also uncovered that the model could have non-default scales and units. Thus the current approach of initializing the default values for scales and units needed to be modified. Time constraints prevented the team from implementing these changes, however, the team developed the system with the required hooks to allow for

the functionality to added in the future.

Nearing the conclusion of the project the team discovered that the underlying *DataStreamService* (*LLVISTA* external data input component) and *Advertisements* (another type of input component) also had to be exposed to certain components. Working with the sponsor, the team developed an introspective test for components and modified base level components to create advertisements. While this is not ideal, this approach achieved the necessary functionality and greatly enhanced the functionality of the system.

2.4.4 *LLVISTA* Component Variation

The initial components the team worked with all conformed to a single standard. Believing that all other components would also meet these standards, the system was designed around this standard. However, through the development and testing process many non-conforming components were uncovered. Addressing this issue became a primary concern for the team. Discussions with the sponsor revealed that all components were supposed to follow the standards, but that the team could modify non-conforming components to adhere to the standards. Working with the advisors, the team was able to create a set of criteria and develop a conformance reporting utility using Java Reflection and Introspection. This allowed the team to identify the non-conforming components and modify them, without changing their functionality, so they could be used within the visual design tool. This process worked well, however there were components that did not fit the standard template which had to be accounted for due to their complexity. For these specific components a new marker interface was created and the conformance utility modified. The marker interface required that the

programmer provide a method that would inform Slique how to make the component adhere to the standards.

2.4.5 Groovy Rendering

Once the team advanced to the point where groovy scripts could be generated and executed a major issue was uncovered. The *groovy class loader* was maintaining a cached copy of the compiled groovy scripts thus preventing Java's internal garbage collection from cleaning up the groovy script and freeing the resources the script acquired. Without cleaning up these loose resources, the groovy scripts became non-responsive while taking up valuable process cycles and memory usually resulting in an application crash. At first the team tried to remove the cached copy, however this proved unsuccessful as the groovy class loader had cached a reference into the non-modifiable Java class loader. Next the team tried rendering each groovy script in a separate execution thread and then manually killing the thread when the script completed or the user requested a cleanup. Again, this method was faulty as the Java class loader still contained a reference to the cached script and run-away resource usage persisted. Finally, the team developed a small stand-alone application that only rendered groovy scripts from files or standard input and launched new processes, following the same model as for the threaded approach, for each groovy script. The team still encountered challenges with this approach, including consistent Java classpath and error propagation, but this approach prevented resource utilization from crashing our application and ultimately enhanced functionality of the application.

Chapter 3

System Architecture

3.1 *LLVISTA* Architecture

LLVISTA is comprised of a large number of different components. These components are combined together to form a mission data processing pipeline. Incoming data, from source components, is transformed, via pipeline components, before being exported, to receiver components. Source components may include data from a file, socket, or other external source while destination components may export data to a file, socket, graphical display, or other external destination. Pipeline components accept data from source components or other pipeline components, modify the incoming data and then pass the data along to receiver components or other pipeline components. Maintaining consistency and extensibility among *LLVISTA* components is achieved through well-defined interfaces. These interfaces specify the type of data sent or received by the implementing component. Below is a UML diagram of the basic interfaces that a *LLVISTA* component must implement.

Figure 3.1 is for a component that will receive a set of double data points and then send a

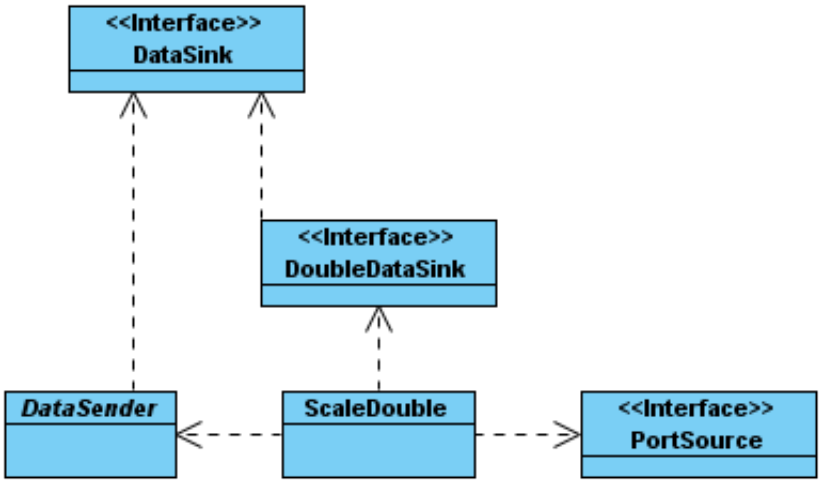

Figure 3.1: A basic *LLVISTA* Component, ScaleDouble

set of double data points. The component must extend the DataSender class to be considered a valid source component. This class contains the methods that any sender of data is expected to have and supports the generic DataSink. DataSink is a marker interface class that is used as a superclass to all the types of data that exist within the system. The DataSender must have the correct generic so receiver components may correctly register themselves to obtain data. In this case, the DataSink would be the DoubleDataSink representing double data points.

Receiving components must implement the PortSource interface to access the port system within *LLVISTA*. These ports enable registration among components by specifying the data type to be received as well as additional parameters for special components. Within the interface required getPort method, validation of incoming data types must be performed. Otherwise, source components could send data types that the receiver component is not expecting and is unable to handle. Without this interface, the component will not be able

to connect to source components.

The final interface (or interfaces if the component can accept more than one type of data) that must be implemented is the DoubleDataSink interface. The DoubleDataSink interface extends the DataSink interface by specifying the type of data this data sink will provide. By implementing this interface, the component will be required to add the appropriate methods allowing other components to place data into this component. Within the required methods, the logic to handle the incoming data must be expressed. The component is now able to accept double data points.

In the event that the component does not need to be a sender, such as in the case of a graphical interface component, it would no longer be necessary to extend the DataSender class. However, the component will still need to have the PortSource and DataSink interfaces implemented if it wants to accept any data.

3.2 Integration of *LLVISTA*

Development and extension of *LLVISTA* will continue as the need for additional components and functionality arises. To prevent our application from becoming outdated, it must dynamically adapt with *LLVISTA*. The application was enhanced to search through the *LLVISTA* packages and find all of the conforming components. Thus the Slique application is a stand-alone product capable of updating along with *LLVISTA*.

The primary means of discovery for the *LLVISTA* components is through Java reflection and introspection. Reflection allows the application to look into the classes contained within *LLVISTA* and extract information about them. Introspection provides the system the ability

to look inside of itself during run-time. Both of these technologies were necessary in the creation of the visual design tool.

Reflection is utilized to integrate the design tool into *LLVISTA* by checking for the abstract classes and interfaces a component might be extending or implementing. As described in Section 3.1, the framework uses a multitude of different abstract classes and interfaces in order to manage the different types of data supported by the system. Reflection allows the system to acquire classes and use them to determine how the component will interact with other components. This type of checking forms the foundation on which each component is checked for compatibility with other components. Components that do not conform to *LLVISTA* standards will also be detected and eliminated from inclusion in the set of usable components. This ensures that all components available inside of the visual design tool will be compatible with other components and result in a valid assembly.

Conformance with *LLVISTA* standards is required in order for the visual design tool to be able to use the component. Java reflection aids the team in allowing us to check each component for its adherence to the standard. With the list of interfaces and the ability to check all parent classes of a component, the conformance check is able to ensure that the correct interfaces and superclasses are present. If they are not, the system will be unable to use the proper methods to integrate the component into the framework. This second check will eliminate more components from that set of valid components that would not be accurately generated by the system.

To further prevent a dependency between the system and *LLVISTA*, introspection is used to set various properties on a component. Introspection allows the system to locate the bean methods [9] that exist in each of the components. With these beans, it is possible to get and

set the properties of an individual component.

3.3 *LLVISTA* Conformance

In order for a *LLVISTA* component to used in the Slique application, it must follow certain standards and meet specific criteria from *LLVISTA*. The Slique application is designed around these standards and maintains the assumption that a conforming component is valid. To assist future developers and maintainers, a separate conformance checking program has been created. This simple program uses Java reflection and introspection to ensure that all the components in the provided package are compliant with the *LLVISTA* standards. The components that do not meet the necessary criteria are displayed in the generated report along with each failed check and the associated reason. This report allows users to quickly determine the required modifications to a component so that the component may be available within the system.

In order to become a Slique certified component, the component must adhere to the *LLVISTA* standards. Components must have a valid sender abstract class or a valid receiver interface. Without one of these, the component must either be a Java Swing component or modify the underlying data stream service. If none of these requirements are met, the component can not be used within the application. The collection of valid receiver interfaces are listed within a file in the configuration folder of the project.

Each classification of component requires specific additional criteria. However, there are global criteria that all components must satisfy. All components must have a null constructor. Without this constructor, the component will not be instantiated correctly and will not

produce the desired results. Also, any property within a component that must be set or configured needs to be bean-compliant. This is vital for the component's properties to be identified by the application and exposed to the user.

3.3.1 Source Components

LLVISTA components that may only send data are considered source components. These include file and socket reading components as well as components that generate data themselves. To be considered valid, the component must be derived from the DataSender class. This superclass defines the necessary methods to publish data to other components.

3.3.2 Destination Components

LLVISTA components that may only receive data are considered destination components. These components include plots, files, and output sockets. Valid destination components must implement at least one of the standard DataSink interfaces. These interfaces mandate methods and options that allow the implementing component to receive a certain type of data.

3.3.3 Pipeline Components

LLVISTA components that are both valid senders and receivers are considered pipeline components. These components provide data modification and in-line calculations to the incoming data stream before sending it on to the next component.

3.3.4 Additional Components

While most components fit into at least one of the above categories, there are components that exist outside the scope of the categories. These components modify the assembly in a non-standard method and are usually considered non-conforming. However a few special components, such as the driver components, are essential to the assembly and must be accessible within the application. Driver components exist in a specific package within the *LLVISTA* hierarchy and implement the *setDataStreamService* method to be pseudo-conforming. The issues with these components have been discussed with the sponsor who is now aware of how they have been implemented into the application.

3.4 External Dependencies

To aid in the development of the project, the team utilized a set of external frameworks. These frameworks added additional functionality to the system that the team did not need to develop themselves. The use of these frameworks allowed us to instead focus on the project's primary objective instead of re-developing existing functionality.

3.4.1 JRE 1.6

This project requires the use of the Java Runtime Environment 1.6. All user interfaces were created and tested using the Java Swing framework from version 1.6. The ability to save and load models inside of the application is dependent on the JRE 1.6 XML classes.

3.4.2 Log4J 1.2.15

Log4J was an additional dependency to the application. Error message handling and information messages were handled by the Log4J framework. This tool allows for error streams to be output to multiple locations and provides user configuration of those streams.

3.4.3 Groovy 1.5.1

Groovy was an essential part of the project as our goal was to create groovy scripts with identical functionality to the original hand-written scripts. To be able to render these scripts within the application, groovy is a necessary dependency. Without groovy, the model must be exported to an external groovy file and then run separately outside the application.

3.4.4 JUnit 4

The JUnit framework was used to provide a testing suite for the Slique application. A set of test cases was created with the project to test correct functionality of the classes. A user will need JUnit 4 in order to run the provided set of test cases that were written for the Slique.

3.5 System Components

3.5.1 Component Manager

The Component manager is responsible for containing the component hierarchy. This hierarchy is stored in a tree structure that is composed of two types of nodes; one representing

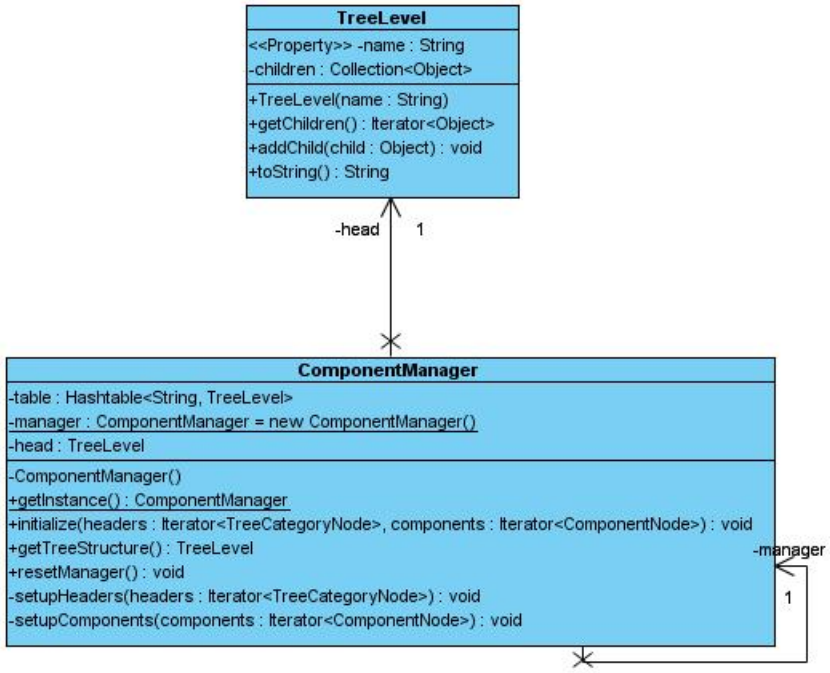

Figure 3.2: UML diagram of the component manager

a component container and the other representing a component. The container node is also responsible for maintaining the collection of its child components. This container structure is then used by the graphical user interface component that will represent the components and containers to determine the different levels within the tree and the components that belong in each level. The second type of node contains additional information that will allow the node to create new *LLVISTA* components within an assembly.

3.5.2 Conformance

Given the dynamic and evolving nature of *LLVISTA*, spanning many years and having many programmers, it was necessary to create a set of criteria to maintain a reliable and extensible conformance checking ability. The developed conformance checker first validates

the type of component that wishes to be certified for use in the Slique visual design tool. Valid types of components must be either an extension of Java *AWT* components or implement specific interfaces within the *LLVISTA* framework. Based on these interfaces, components will be classified into three general categories: data senders, pipeline components, and data receivers. Data sender components only send data and will not receive data from any other component. Pipeline components can both send and receive data from other components. Data receivers may only receive data and will not send data to any other component.

There is a fifth type of component that provides additional functionality without using directly registered connections. These driver components are not currently supported and their integration is planned for future work, and is described in chapter 6.

3.5.3 ErrorManager

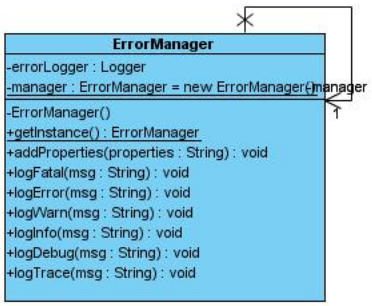

Figure 3.3: UML diagram of the error manager

The error manager is responsible for recording the various types of information that are generated by other elements within the application. This information provides invaluable assistance for errors that may occur and maintaining the current state of the application.

The error manager makes use of the Log4J [10] package to manage the different type of messages that can occur. The Log4J package provides seven different levels of messages, ranging from fatal error to trace information. These fine-grained levels allow for dynamic handling of the message based on the severity of the message.

3.5.4 Model

At the heart of the visual design tool are the assemblies, which represent the mission data processing pipelines being constructed; each managed by a model. This core element of the Model-View-Controller [3] paradigm maintains the data necessary to completely represent the assembly. An assembly is comprised of three parts: components, connections, and properties.

Components in the model represent *LLVISTA* components that have been wrapped with an interface that allows the team to use them easily within the system. Without such an interface, it would make using the component much more difficult and take away from the flexibility of the system. To account for the various properties that a component can have, the wrapper interface contains an object that allows for the various properties on the component to be altered. The ability to change the component properties gives the user much more flexibility in the usage of the underlying component and access to more of the strength of the *LLVISTA* framework.

Connections in the model represent the connections between the various components within the pipeline. The list of connections consists of only valid connections. Any invalid connections have been eliminated through the use of a validation procedure. Similar to the

components, connections also have properties. This is required for some components where connection properties play a part in how the data is rendered.

Pipelines themselves can have properties that affect all pieces of the model. Unlike the properties for each component, these model properties needed to be hard coded into the application. The following is the list of the properties which are set for each model:

- PlotLines
- PlotShapes
- FillShapes
- Maximizable
- ShowLegendWithinPlot
- BatchRedraw
- BatchRedrawRate

The disadvantage of hard coding these properties into each model is overshadowed by the advantage that is provided. Control of the *LLVISTA* framework and each individual model is a major component of this system. We decided this was an acceptable trade off for the loss of flexibility that this coding decision caused.

3.5.5 Output

3.5.5.1 Exporting Images of Assembly

The Slique application contains the functionality to export images of open assemblies. The user is able to choose the format for the exported image. Supported formats are JPEG images and transparent PNG images. However, both of these image formats degrade to an unacceptable quality when they are magnified. To counter this problem, DOT generation was included. DOT is a simple text file that describes how a graph will look that can be used by various programs to generate a graph image in a variety of formats including SVG. This allows for the creation of higher resolution images that maintain their quality upon magnification.

3.5.5.2 Groovy Script Generation

Utilizing the assembly that the user has created, the visual design tool is capable of generating a syntactically correct groovy script file that represents the pipeline as an *LLVISTA* testing configuration. To generate this configuration, construction of the groovy building process is broken down into steps. Combining these steps together into a single file is how the final output is generated. We know that this output is valid since each statement can be proven to be a valid groovy command. As long as the ordering of these valid commands is done properly, the entire configuration will in turn be valid.

3.5.5.3 Groovy Script Engine

Not only does Slique provide the ability to generate groovy scripts, the visual design tool supports the processing and rendering of those groovy scripts. To alleviate issues caused by the groovy scripting engine, each script is run in its own process.

3.5.5.4 XML Generation

Persistent storage of files was implemented in a system and language independent format. The team decided to use an XML representation with a defined schema for this purpose, thus allowing user created assemblies to be saved to disk. This provides the application the advantage of giving models portability and reuse.

3.5.6 *LLVISTA* Component Parsing

Extraction of the *LLVISTA* components was encapsulated within an auto discovery program that scans the *LLVISTA* framework. Through introspection, reflection, and various input files, the program is able to export an XML document that specifies the component categories and the components themselves. In the event of special cases, such as classes that use delegates or require being in special category, the parsing engine identifies the correct categories for each special case. This approach alleviates the startup time of Slique.

3.5.7 Properties

Abstract models, components, and connections all have user configurable properties. These properties include scale factors, plot options, and displayable text; to provide the

user access to these options a properties manager was created. This manager handles the communication between the visual display of the properties and the storage/application of the properties.

3.5.8 Validator

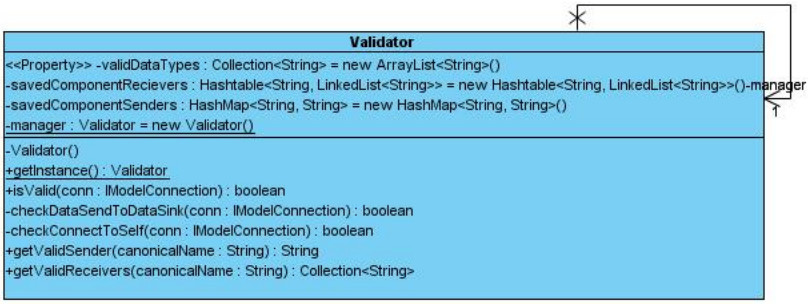

Figure 3.4: UML diagram of the validator

The validator is responsible for verifying connections between components. This process is completed by obtaining information about the component via Java reflection and introspection. When a connection is requested, the validator receives the two components that will be part of that connection. Then the validator performs various verification checks on the components. If all of these methods pass, the validator will assert the connection is valid. In the event that the connection is not valid, the validator will inform the error manager the connection failed. This will include a message containing the sender type of the first component and the list of valid receiver types for the second component. This information will aid the user in understanding their mistake. The validation process is necessary to ensure the consistence of valid assemblies with only valid connections.

Chapter 4

Design Decisions

Over the course of the project, various decisions on the architecture of the application had to be made. This section details some of the design decisions that were made and the reasoning behind the choices.

4.1 Model View Controller Paradigm

After discussing various possibilities for the architecture of the system, the Model-View-Controller [3] was selected to act as the foundation of the system. By selecting this design paradigm, the system was able to acquire the benefits bestowed by it. Flexibility and extendability are qualities that the system needs to have and this design pattern fits perfectly for this requirement.

As a three part system, the Model-View-Controller forces the encapsulation of each section. In addition, due to the usage of this pattern, the graphical user interface was able to be separated from all of the underlying code. Due to this strict encapsulation, it makes

the system easily extensible. New components can be easily added into the system without disturbing the operation of the others. This is possible because none of the components have knowledge that any other components are there. New controllers, data controllers in this pattern, will need to be constructed in order for the new component to be used. However, this method requires much less work than the alternate would have. The alternative would have involved all classes being connected resulting in long dependency chains which would result in an inextensible and rigid system.

Having the graphical user interface disjoint from the underlying code provides a major advantage during testing. Graphical user interface code is difficult to run unit tests on due to the nature of graphical user interface programming. As a goal of the project was to achieve at least 80% testing coverage, the user interface provided a large stumbling block. This pattern's usage of controllers helps to alleviate this issue. The user interface is able to call the controllers which can be tested separately. This will reduce the amount of errors that are present in the final system.

Testing the correct functionality of each component is easier using this design pattern. The pattern calls for all pieces of functionality to be encapsulated within themselves. Due to this, each of these components is able to be tested individually. They have no reliance on any other component of the system. This will provide more control over each of the tests. After each of the views is tested, the controllers can be tested to ensure that they are performing correctly. This allows the system to be rigorously tested to reduce the number of defects in the final system.

In order for the system to function, a large amount of data must be accounted for. By using this pattern, it mandates that all of the data will be located in one location, the

model. As a singleton [3], all of the other objects are able to access and manipulate the data when necessary. This one centralized point for all of the data will help prevent pieces of the system from using out of date data. Controllers, the third part of the pattern, can be used to communicate with the model from other singletons [3]. This prevents dependencies between the data and the managers, allowing for the system to be more flexible.

4.2 Singleton Managers

Each piece of functionality within the application was placed within its own singleton manager following the singleton design pattern. The decision to use the singleton design pattern was heavily influenced by the decision to use the model-view-controller paradigm as the base framework for the system. Despite this, the use of this pattern provides advantages within our system.

Singleton managers allowed each function of the system to be encapsulated and tested individually. By encapsulating each piece of functionality into its own class, it allows the system to be easily expandable. If there is a desire to add new functionality to the system, this new functionality can be placed within a singleton manager. New controllers then will need to be made to allow the rest of the system to communicate with the new piece. In addition, managers can be replaced with updated versions of themselves without disturbing the rest of the application. Testing the managers is also simplified as each manager can be tested on a stand alone basis. The data that is entering the manager can be more controlled so the proper functionality of the manager can be accessed. These two abilities make the application much more flexible and scalable.

They also make it easier to deal with internal data passing. The model is primarily responsible for the data contained with the system. Each manager is able to access, through controllers, this data. The alternate to this could be to pass references of data through multiple classes which would quickly become confusing. This method offers better control over the flow and storage of the data within the application.

4.3 Push Versus Pull Event Mechanism

For this application, the push mechanic of data updating was chosen over the usage of pull. It was chosen for its simplicity and due to the application's usage of a fine grained event system. Data is updated within the application often as users perform actions such as changing properties, altering component positions and adding new components. As this data is altered, various components of the system need to be informed about this change in the data. This could be accomplished in two ways, either pushing the data to all those that needed to know about the update or having those that needed it to pull the data that changed. Pulling would have the disadvantage of each component needing to poll the model to check for changes in the data. This would result in a waste of processor time that could otherwise be used to run the system. The push mechanism had the model push the updated data to those that wanted to know about data updates. This idea follows the Observer pattern [3]. Additional managers can register with the model later if additional functionality is necessary and acquire the updates they need. This continues to keep the system flexible and easily expandable.

The fine grained event system also makes the management of the graphical user display

easier to manage. Instead of trying to do a pull to eliminate various graphical components, the graphical thread can be told which pieces of itself it needs to erase. This speeds up the rendering of the graphical user interface and makes it easier to work with from a programmatical point of view.

4.4 Usage of Swing

The team selected to use the Java Swing GUI toolkit for two reasons. The first reason being that a majority of the components currently used in *LLVISTA* are rendered using Swing. Continuing to use this toolkit will ensure that those who maintain the application are already knowledgeable about its use and would know how to make edits to the graphical section of the application. This would also allow for consistency between *LLVISTA* and Slique. Having multiple graphical toolkits would add extra levels of complexity to the application that would be unnecessary.

The second reason the team selected to use Swing was its compatibility with JFreeChart [5]. JFreeChart is an open source program which provides *LLVISTA* with the ability to render and manipulate graphs. As a large portion of *LLVISTA* is devoted to the creation of graphs and charts, this was a required compatibility. Graphical toolkits that were not compatible with JFreeChart would be unacceptable due to the radical change in *LLVISTA* that would be required. Based on these two points, the team decided that it would be in their best interest to continue usage of Java Swing on the project.

Chapter 5

Evaluation

Evaluating the visual design tool was broken down into three parts: automatic testing and code reviews, user testing, and script generation validation and comparison.

The team used the JUnit [1] testing framework to create test cases; and through conjunction with EclEmma [6] were able to achieve more than 85% code coverage. The test cases that the team developed were also used to provided a metric to compare future modifications against. Therefore, new features and functionality should pass the test cases without additional modification. Since not all of the code could be validated through the automatic testing, the remaining sections were covered through an iteration of code reviews. These reviews were conducted both individually as well as within a group setting to maximize effectiveness. The use of the EclEmma code coverage tool was helpful in providing a means for verifying the code not covered through JUnit test cases. The tool highlights all executable areas of code in green or red allowing an easy visual tool into which sections required attention from the reviews. The metrics determined from testing and review provided a basis for the level of confidence and accuracy maintained by the visual design tool.

The team encountered an intriguing problem trying to validate and verify the generated Groovy scripts as there are no proofs that can be applied to ascertain the generated code will always be correct. However it is possible to assert that the generated code is correct by testing each piece of the script separately. We first began to confirm this assertion by verifying that each statement generated in the script was valid. The team then created a system that would enforce a correct ordering of the statements. Next, the full script was sent to the groovy interpreter which ran and tested the script for any run-time errors. As a final evaluation of the scripts, the team conducted a comparison between manually-written and auto-generated scripts. The team found that the Slique-generated scripts were similar to the hand-written ones and provided identical output when rendered. Provided that the scripts passed all of these tests, the team could confirm the validity of each script and our process to generate them. The actual functionality and usefulness of the script is beyond the scope of this validation.

The team was able to perform user testing throughout the development of the project. The user testing was done by ourselves as well as by *LLVISTA* developers and analysts who would use the system. The testing was performed in an unstructured environment while the team observed and asked questions. The tests were helpful in providing insights to the actual use of the application from a non-developer's perspective. The team was also able to gain information regarding preferred keyboard shortcuts, layout of components, and other visual queues such as color scheme, font, and organization of different areas in the application.

Chapter 6

Future Work

Throughout the course of the project, many ideas and features were proposed. However due to time constraints, not all of these requested could be completed for the final deliverable. Listed in this section are some of these requested that could be implemented to improve upon the functionality of the delivered product.

6.1 Integration of Non-Conforming Components

As mentioned in Section 2.4.4 not all *LLVISTA* components conformed to the set standard. In order to account for this conformance problem, the team was forced to modify some of the *LLVISTA* components to include them into the system without changing their functionality. Future improvements to the integration process of the non-conforming components is required to allow for the full set of *LLVISTA* components to be loaded and accessible in Slique.

6.2 On The Fly Components

Based upon research of the current system usage, a common task that is carried out is the creation of custom 'throw away' components. These types of components are used to perform scenario specific calculations or data flow changes that are not common enough to warrant the addition of a new component to the *LLVISTA* framework. On the fly component creation would seek to emulate this inside of the Slique application. This would provide users the ability to create these special components without having to touch any code while providing them with the flexibility of being able to create scenario specific components.

6.3 Remote Editing and Viewing

This feature would provide users with the ability to remotely edit and view what is occurring inside of the program. This would allow multiple people to work on one assembly and allow a remote person to perform a change if necessary to a model another person is working on. This feature would be primarily about the remote viewing of the executing Groovy script so that many people would be able to see the output of the running scenario.

6.4 Advanced GUI Output and Additional Swing Components

The Slique application currently allows for creating basic GUI outputs for the graphical components in the pipeline. A user can choose from either a flow or grid layout. If more than four graphical components are to output, they are split into separate tabs containing at most

four components. These components can also be ordered to the user's desire. Slique could be extend to allow users to choose from all Swing layouts when outputting their pipeline. Additionally, the application could allow for users to add Swing components to the layout and arranging their components in more advanced ways.

6.5 Component Documentation In-Application

Currently, there is no information about the *LLVISTA* components while running the Slique application. Users looking for information about a specific component currently must consult the source code documentation. In the future, documentation for each component could be included inside the application to allow for easier understanding of the underlying *LLVISTA* framework.

Appendix A

Team Schedule

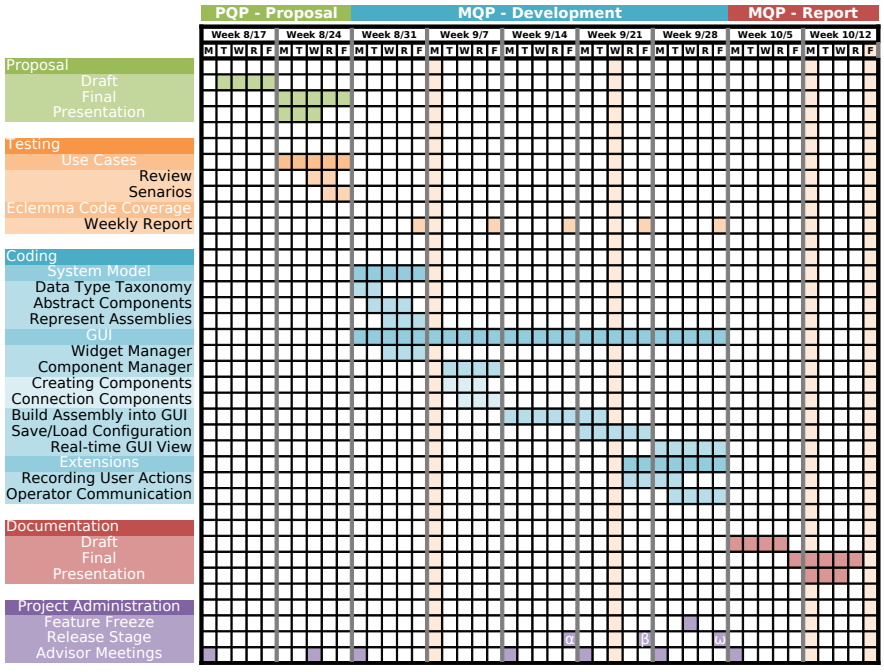

Figure A.1: Schedule used by team in order to complete project on time

Bibliography

[1] Junit.org, August 2009. http://www.junit.org/taxonomy/term/12.

[2] OASIS, October 2009. http://www.oasis-open.org/home/index.php.

[3] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering.* Prentice Hall, Upper Saddle River, NJ, 2004.

[4] Collabnet. Subversion, August 2009. http://subversion.tigris.org/.

[5] Gilbert, D. Jfreechart, April 2009. http://www.jfree.org/jfreechart/.

[6] Mountainminds GmbH & Co. Eclemma, August 2009. http://www.eclemma.org/.

[7] SpringSource. Groovy, Febuary 2009. http://groovy.codehaus.org/.

[8] Sun Microsystems. Java. http://java.com/en/.

[9] Sun Microsystems. JavaBeans. Tech. rep., Sun Microsystems, 1997. http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

[10] The Apache Software Foundation. LOG4J, September 2009. http://logging.apache.org/log4j/index.html.

[11] The MathWorks. MATLAB.

[12] van Heesch, D. Doxygen, August 2009.

http://www.stack.nl/ dimitri/doxygen/.