April 2008

# SolarFlare Collaborative Development Environment

Ian M. Bennett
*Worcester Polytechnic Institute*

Nolan Dupree Colter
*Worcester Polytechnic Institute*

Project Number. GFP0703

# SolarFlare Collaborative Development Environment

A Major Qualifying Project Report:

submitted to the faculty of the

**WORCESTER POLYTECHNIC INSTITUTE**

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Ian Bennett

Nolan Colter

Date: April 22, 2008

Approved:

Professor Gary F. Pollice, Major Advisor

1. Webfoot

2. Collaborative Development

3. Eclipse Plug-in

4. Software Development

## Abstract

The SolarFlare Collaborative Environment is a project intended to facilitate a real-time collaborative solution for the Webfoot Project.  Communication is an integral part of the development life cycle, whether it is task based or open-ended.  Project members need a central location to brainstorm, discuss, and debug a software solution as workgroups and the marketplace diversify and become globally distributed.  The goal of this project is to allow developers to communicate and collaborate in a manner that will increase productivity by placing a messaging client tool within a common environment, the Eclipse IDE.  The SolarFlare vision is to develop a flexible and extensible framework for multi-purpose media communication and collaboration, comprised of separate Eclipse Plug-Ins, facilitating the client and server communications.

## Acknowledgements

We would like to thank and acknowledge all of those who helped or were involved with the project.  We want to thank the Webfoot group meetings and meeting participants for brain storming and sharing helpful information.

Our professors who have imbued us with the knowledge to embark upon and complete this project.

Gary Pollice: For advice, knowledge, and encouragement throughout the project.

The writers of "Eclipse: Building Commercial-Quality Plug-Ins" (Second Edition) for writing their book.  We would have been lost for much of the project without it.

Our friends and family for dealing with us when under the stress and pressure of working on this project and its many tiny snags along the way.

Whoever adds to or extends our works in the future.  We hope that we have laid the grounds for some interesting projects and potential collaborative tools in the future.

## Contents

# 1 Introduction

The rapid increase in popularity of collaborative development software in the software engineering industry has raised the need for effective, efficient, and convenient communication amongst programming teams. These collaborative software tools have become a key issue in producing quality project work. At present, projects are typically distributed not just amongst team members, but team members in varying geographical locations (cities, states, countries). The vastness of projects today has grown with the demands of consumers, who need robust solutions in a shorter amount of time. These shortening deadlines place a need for tools that will increase team productivity and alleviate the difficulties created by varying team distributions.

Currently, there are a number of tools designed to meet many of the collaborative needs of software engineering teams; however, in many cases it becomes necessary to utilize multiple tools to meet the collaboration requirements of the project. There are several projects currently underway aiming to increase the functionality of current collaborative tools or create new tools that provide more services than their predecessors.

WPI's Environment Built for Object-Oriented Teams (Webfoot) is one such project. It is an open source project that has been under development for a number of years, with the goal of integrating the Eclipse software development environment with already-existing collaborative tools such as SourceForge. Eclipse is a well-established program that facilitates the creation of software in a number of programming languages, and is designed to be easily customizable – making it ideal for the Webfoot project.

While Eclipse and SourceForge already have a degree of interaction, a great deal of the functionality is completely separate. For example, if one member of a project team wishes to share a document with the rest of the project team using SourceForge they must switch from Eclipse to a separate SourceForge interface and upload the document there. This kind of context shift is not only somewhat frustrating for the user, it is also inefficient. As it continues to develop, Webfoot will increasingly allow software engineering teams to access more and more of their collaborative tools' functionality directly from the Eclipse environment. The goals of the Webfoot project not only include integrating high-end collaborative tools such as SourceForge, but also encompass addressing the importance of smaller utilities such as chat clients.

This project aims to create a functioning and usable collaborative tool that will allow users to work together in a virtual meeting environment from any physical location. To accomplish this, the tool will be composed primarily of separate server and client customizations to Eclipse that will allow users to join "meetings" with multiple other users. In addition to the ability to converse with multiple project team members, our tool also provides the capability to hold separate private conversations between two users.

By adding this tool to the services that the Webfoot project already provides, we hope to increase both the quality and the convenience of communication for software engineering teams.

# 2 Background

The purpose of this chapter is to introduce topics to assist the reader with the conceptualization of the project plan, as well as the design creation of our collaborative software development tool. The history of software tools begins alongside the creation of the first computers in the 1950's. These early computers made use of tools such as linkers and loaders to assist in the development process; as technology and software continued to evolve, tools began to expand in scope and function as well (Kernighan & Plauger, 1976).

The majority of the original tools used for software engineering were light and simple, such as text editors and compilers.  These tools required many parameters to be set manually and/or input in a repetitive manor.  However, the concepts and functionality featured within these tools has been maintained, continuously improved, and integrated into complex and powerful integrated development environments (IDEs). These IDEs provide a consolidated setting, allowing the combination of multiple tools to encapsulate their functionality in one environment, to further assist in software creation.

## 2.1 Eclipse

The Eclipse Platform is a development platform that was donated to the open-source community (which is a community made up of software programmers that publicly share their source code) by International Business Machines (IBM). (This is especially notable not only because of the fact that IBM is a commercial company that could have retained private rights to the IDE, but also because of the expense that went into its design and creation – $40 million (Developing Eclipse Plug-Ins, 2002).) Currently, Eclipse is one of the most popular and effective IDEs available, and is a focus of the Webfoot project, of which this MQP is a part. Written primarily in Java (an Object-Oriented programming language), the default program is designed to assist Java developers in software creation. Eclipse users have the option of further extending its capabilities by installing plug-ins designed for the software framework. For example, the C/C++ Development Tooling plug-in provides users with the ability to use the advantages of the Eclipse IDE to write programs in C or C++ (Eclipse.org, 2008).

### 2.1.1 Extensibility

This ability to expand the capabilities of a program is referred to as “extensibility”, and has become one of the marks of well-designed software. It is essentially a measurement of both a software system’s capacity for expansion and the level of effort required to create an extension. In software engineering terminology, an extension is the addition of functionality, or the modification of existing functionality. The principle behind extensibility is to allow for updates, improvement, and modification of existing software with minimal impact to existing system functions (Software Engineering Terminology, 2006).

### 2.1.2 Eclipse Plug-Ins

The key to the success of the Eclipse Platform is that, beyond providing well-designed open-source architecture, it not only allows users to create and use plug-ins to extend its functionality, but it actually encourages this sort of customization and improvement. Eclipse actually goes so far as to provide a Plug-In Project creation wizard, ensuring that the plug-in has all of the necessary components as soon as it enters development. In fact, with the exception of a small amount of core code, nearly everything in Eclipse is a plug-in; meaning that the majority of the IDE is a set of smaller interlocking pieces of software designed to interact in specific ways (Developing Eclipse Plug-Ins, 2002).

The two primary plug-ins of interest to Eclipse developers are the Workbench and the Workspace – these provide the “extension points” that the majority of third-party plug-ins use to integrate with the platform. There are, of course, other extension points available to developers, such as the Debug or Team components, but these are generally used for very specific purposes.

## 2.2 Collaborative Software Development

With the advent of the Internet and widely-accessible global communication, it is now possible for software development teams to have their members scattered throughout the world. In response to the needs of these software teams, a number of collaborative development tools have been created. In the same way that many software tools have been combined into IDEs, many of these collaborative tools have been integrated into Collaborative Development Environments (CDEs).

### 2.2.1 Collaborative Development Environments

As the software development industry continues to grow at a rapid pace, the use of CDEs has become a near-necessity to manage the project work of a distributed team.

One of the best examples of a collaborative tool that encompasses nearly all of the needs of a software development team is Microsoft’s Visual Studio Team System, which interacts with the Visual Studio IDE via plug-in. This CDE provides the necessary functionality to manage software development, collaboration, metrics, and reporting to ensure that a team has a consistent and reliable standard for managing their project. (Visual Studio Team System 2008 Product Information, 2008).

Unfortunately, Visual Studio Team System does not support Java development, though there are a large number of other options available. This project in particular focuses on using the Eclipse IDE to interface with the SourceForge CDE.

### 2.2.2 SourceForge

SourceForge Enterprise Edition is a set of software development tools for development and collaboration distributed by CollabNet. It provides software and document revision control and project plan management from a web-based interface. There is another version of the original SourceForge CDE provided at SourceForge.net, which provides a free code repository for software developers. SourceForge allows software development teams to use a centralized location to create, manage, and store project tasks, tracker items, documents, source code, and file releases. It also provides a wiki that project team members can update as needed.

The commercial form of SourceForge has a number of additional features that are not available with the free version. For example, it has a much tighter set of security standards – because this platform is intended for use by businesses, security is one of the most important requirements. In the scope of the Webfoot project, the most relevant feature of SourceForge Enterprise Edition is the rich API provided by CollabNet that allows third-party developers to access SourceForge functions inside their programs.

### 2.2.3 Webfoot

Webfoot is an open-source project that seeks to build on the Eclipse framework to provide Java developers with the ability to interface with CDEs from inside the IDE – removing unnecessary, and often annoying, context switches. At present, the project focuses on integrating functionality of SourceForge into Eclipse using the robust SourceForge API. Webfoot continues to provide developers with the capability to access more and more of the SourceForge CDE without needing to open it themselves.

In most cases, a plug-in developed in the Webfoot project provides a "view" inside of Eclipse displaying data from the SourceForge server and any data that can be manipulated inside of the view will be synchronized with the server information. Webfoot follows the principle of extensibility, and future developments can be easily integrated into the existing project framework without causing problems.

### 2.2.4 Addressing Communication

Today there exist many tools for communication over various protocols, used for different purposes. Many of these tools exist to serve as a point of communication, however are not integrated into the development process, such as AIM or XMPP. Typically, the process becomes disjointed by paradigm shifts between classic communication and the development process (i.e. talk on the phone for a meeting, open an instant messaging client, switch back and forth between windows, etc). Not only does this create a gap in communication and focus, but it deters the development process.

Traditionally, internet communication is done through text, whether real-time via instant messaging or an expected short response time such as e-mail, bulletin board services, blogs, etc. Typically, audio functionality is hidden within an interface or confusing to use. The ability to do work with your hands and speak hands free can greatly increase productivity. This project intends to ease and encourage the use of streaming voice content over the network.

## 2.3 Object-Oriented Analysis and Design

Object-Oriented analysis and design (OOAD) is a software engineering technique that approaches the concept of a system by visualizing it as a set of interacting objects. Each individual object in the model represents a specific area of interest that must be addressed by the system, and is characterized by its class, its state (or data), and its behavior. These aspects for each component can be used as needed to generate diagrams displaying the structure of the program, such as an architectural model or a dynamic behavior model.

The OOAD approach to software engineering can be separated into its analysis and design aspects, each of which is instrumental to utilizing the technique properly. The process of object-oriented analysis applies the concepts of object-modeling to analyze the functional requirements of the system as a whole, while object-oriented design further elaborates the models created by the analysis to produce the specifications needed for implementation. Simplified, object-oriented analysis focuses on determining what exactly the system does, where object-oriented design focuses on determining exactly how it does it.

### 2.3.1 Objects

In software engineering terminology, an object is "a representation of a real-life entity or abstraction. For example, objects in a flight reservation system might include: an airplane, an airline flight, an icon on a screen, or even a full screen with which a travel agent interacts" (Bray, 1997). Utilizing the idea of objects, it is possible to convey relevant data to a program that would otherwise be abstract – a computer does not truly know what an airplane is, but it can understand the values relevant to something *called* an airplane as long as it has been described appropriately.

### 2.3.2 Object-Oriented Systems

An object-oriented system is composed of modules viewed abstractly as objects, and the behavior of the system results from the interaction and collaboration between these objects. System objects communicate with each other by sending messages; any message sent from one object to another must be composed in such a way that the receiver will be able to understand it properly. This means that it must use some format specific to the receiving object, and must contain the minimum pieces of information required for the receiver to operate on the data.

Sending a message is different from having one object call a function in another; instead, sending a message means that one object sends information, and the receiver independently decides what to do with it.

### 2.3.3 Object-Oriented Analysis

Object-oriented analysis produces a conceptual model of an object-oriented system by looking at the problem domain and breaking it down into its components based on the information available in the area. This analysis is preliminary to any sort of actual implementation considerations, such as distribution or persistence – those issues are dealt with as needed during the design. The problem domain can be defined through a number of methods, including a list of written requirements or interviews. It should be noted that using a formalized written statement of some kind is often the best option, as it reduces the likelihood of miscommunication between software developers and their clients.

The conceptual model that results from object-oriented analysis should describe the system's functional requirements, and is often composed of a set of use cases (examples of how the software will react in expected scenarios, from the point of view of the user, not the programmer), one or more class diagrams (usually utilizing Unified Modeling Language notation), and interaction diagrams. If necessary, it may also include a visual model of what the user interface may look like.

### 2.3.4 Object-Oriented Design

Object-oriented design utilizes the conceptual model produced by the analysis process to create an implementation model, taking into account any possible environmental factors or technical constraints.

The product of this step is usually a set of detailed class diagrams, showing each class, its attributes, and how it interacts with other classes. This step may also produce a more detailed version of the high-level interaction diagram that the analysis step can produce.

### 2.3.5 The Entire Process – Object-Oriented Development

There are five major steps in OOAD development process, specifically: gathering requirements, analysis, design, implementation and testing, and distribution. Each step of this process is important for the creation of a quality product that is easy to both maintain and update (McLaughlin et al, 2007).

In Phase 1, referred to as the "gathering requirements" phase, the developer combines all of the technical issues, such as hardware restrictions, and the needs of the client to form a set of specifications (or guidelines) that the software must meet. These requirements must be laid out in the beginning of this entire design process, to prevent confusion later. The requirements are subject to change, within reason, at any time, which meant that the design should be capable of shifting to accommodate any new specifications.

Phase 2 of this process is the "analysis" phase which, despite the name, does not actually utilize the object-oriented analysis techniques (they are used in Phase 3). When applied to software development, this means the prediction and prevention of potential problems before the application is placed into a real-world context. For the most part, this step involves identifying any possible issues that may arise in the real world, simplifying them, planning out a reliable solution, and adding that solution to the software requirements or updating use cases as necessary.

Next is the “design” phase (Phase 3), which involved the architectural planning and design of the software. This phase is where object-oriented analysis is applied to the requirements produced by phases 1 and 2 to produce the conceptual model; object-oriented design is then applied to the results of this analysis, to produce class diagrams. The previous two phases are very important and help to streamline the creation process. Instead of piecing together a program based on a vague idea of what it is supposed to do and formulating solutions to problems as they appear, it is now possible to design the software based on a very clear set of guidelines. While this step sometimes involves a small amount of coding, it usually revolves around creating the preliminary skeleton for the program.

Phase 4, “implementation and testing,” is fairly self-explanatory. This is the step in which the actual software code is written, based firmly on the architecture set forth during the design phase. One of the most important items for programmers to keep in mind during this step is the clarity of their code; it should be possible for any programmer with background knowledge of the requirements to look at another programmer’s code and understand it with minimal effort. Testing the code during this phase provides assurances that each individual module successfully performed the required functions.

The final phase (Phase 5), “distribution,” involves planning and executing the distribution of the application to the users. Because the application that was designed through this project was a plug-in for the Webfoot project, this step involved little more than adding it to the Webfoot project itself.

Figure 2.1 displays the layout of the phases for the Object-Oriented Design Process – note that the Testing and Implementation phases occurred at the same time, and that moving between those two phases is allowed (and oftentimes expected).

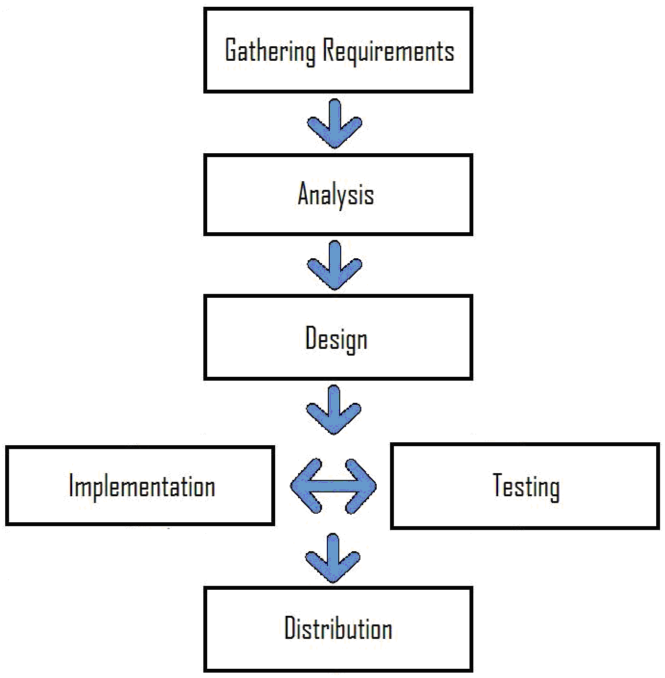

*Figure 2.1* – Object-Oriented Design Process

# 3 Methodology

This section explains the steps that we have taken in the development of our Eclipse plugin, including research and the process we followed to design, create, and implement the software code. The goal was to create a usable collaborative tool that could be used inside of Eclipse to communicate with other users through the use of text and audio chat.

## 3.1 Phase 1: Gathering Requirements

Because the initial scope of our project was not fully-defined, our tentative plan was to focus on creating a text and audio chat client in Eclipse that would allow for meetings and private one-on-one communication. Our initial research into the topic convinced us that it would be a fairly simple and straightforward project if those were the only requirements. As such, we determined a number of possible additions that could be made for the plugin that would improve both usability and expand functionality. For example, we originally planned to add the ability to translate the contents of a chat session, or “meeting”, into a SourceForge object (such as a task or a document). The following list of user stories (Table 3.1) reflects much of our original planning:

**Table 3.1 – User Stories**

- A user must be able to open a voice chat from within eclipse.
- A user must be able to record a live chat.
- A user must be able to send and receive audio.
- A user must be able to archive an audio chat.
- A user must be either a host or a participant for an audio conference chat.
- A host must be able to invite participant(s) to an audio chat.
- A participant must be able to accept or decline an invitation before or during an audio conference chat's scheduled time.
- A user must be able to mute their line in an audio chat.
- A user must be able to disconnect during a conference or one-on-one chat.
- An invitation will be comprised of a user, a chat place, a host, a valid time period, and a chat pass code.
- A user with an invitation can connect or re-connect at any time during the chat's valid time period.
- A user must be able to edit a recorded meeting into sub-segment clips that can be associated in SourceForge.
- A user must be able to record an audio clip without needing a scheduled meeting (personal note or audio comment).
- A user will have a username, avatar, text, audio, and webcam accessibility (which may be enabled or disabled depending on availability).
- A user will have a group collaborators (buddy) list.
- A user can open a one-on-one text/audio/video chat with another user on their group collaborator (buddy) list.

This list denotes the primary requirements we had set forth for the project; requirements for many of the secondary additions we had brainstormed were not included, and would be determined later if time allowed. Instead, we intended to keep the possible add-ons in mind while developing the main portion of the plug-in. These user stories were looked then converted into use cases, an example of which can be seen in Figure 3.1.

| | |
|---|---|
| **ID:** | GFP0703-01 |
| **Name:** | Opening a Voice Chat |
| **Actors:** | User |

**Basic Flow:**

A user is engaged in a private or multiple chat session. The user enables audio sending and reception, and is able to converse with other members of the chat session using audio messages. The use case ends.

**Revision History:**

18-Sep-2007, Nolan Colter, Version 1.0

*Figure 3.1* – Use Case Example

## 3.2 Phase 2: Analysis

Due to the nature of this project, many of the real-life context issues that may have affected the modeling and design were already addressed by the initial set of requirements/user stories set forth in the Gathering Requirements phase. The majority of these issues were centered on user interaction with the software, and requirements such as “A user must be able to mute their line in audio chat” met many of the real-world needs that our project group brainstormed.

Though we were able to determine some issues that we had not already address, such as bandwidth restrictions and ensuring the minimum hardware/software requirements for Eclipse were met, the majority of them were problems that were the responsibility of the user. Possible problems with audio lag or other streamlining concerns were considered as secondary objectives.

## 3.3 Phase 3: Design

Applying object-oriented analysis to the information gathered in the previous two phases allowed us to create a conceptual diagram (seen below in Figure 3.2) modeling the object interaction needed to meet our requirements.

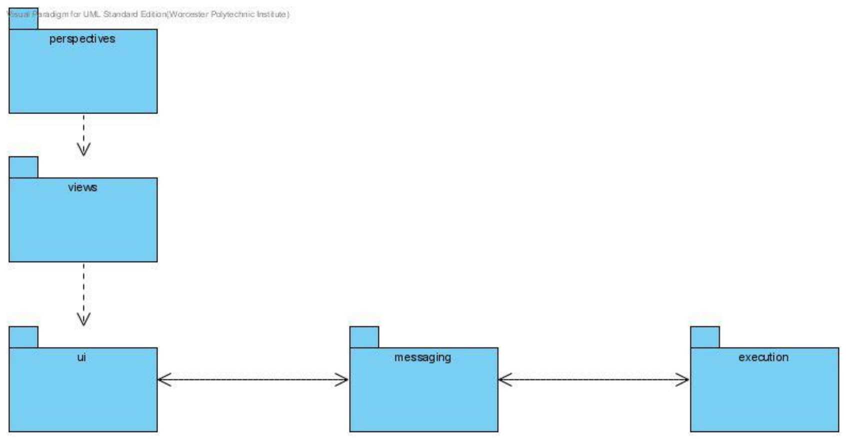
Visual Paradigm for UML Standard Edition(Worcester Polytechnic Institute)
perspectives
views
ui
messaging
execution

*Figure 3.2* – Conceptual Model

From this diagram, we were able to determine which areas of interest we would need to research to correctly create the internal design for the networking, text chat, audio chat, and even the plug-in itself. One of the primary decisions we needed to make was whether or not to utilize the Eclipse Communication Framework to build our application.

### 3.3.1 The Eclipse Communication Framework

The Eclipse Communication Framework is designed to support development of third-party communications applications on the Eclipse platform. The ECF project is under constant development and improvement, which has proven to cause issues with some developers who require a more stable API to create their applications. In fact, this was an issue that greatly hampered the work of a computer science MQP group in 2007, whose project aim was to create a usable text chat client plug-in for Eclipse. Despite having had more than a year to settle after causing so many problems for that project group, the framework still has yet to fully stabilize into a reliable tool.

The ECF framework's usability is further degraded by its poor documentation. These important drawbacks forced us to look into creating our plug-in without utilizing ECF to avoid any potential (and likely) pitfalls. Its' implementation made excessive use of the adapter pattern, however upon analysis of current protocol implementations, the bulk of the code was in one package. This completely defeats the purpose of designing a framework, thus made it hard to extend or understand the existing example code base.

### 3.3.2 Relevant Existing Open-Source Projects

After deciding that we would not use ECF as the basis for our application, we began searching for open-source projects dealing with the same material to gain a better idea of how our goals could be accomplished. Finding relevant projects with helpful information proved to be very difficult, particularly for the audio chat aspect of this project. For the most part many communications projects did not use Java Audio or had some sort of monetary cost attached to them – a number of the projects actually used SIP and Skype to handle the audio messaging. These protocols do not support a built in functionality of message storage, nor do they lend themselves to extension and integration with other program functionality.

### 3.3.3 Designing Our Own Framework

With a lack of current tools and projects to use as a base for our application, we decided to create our own basic and extensible communication framework. Unfortunately, the additional work time this would require meant that we had to scale back on our original goals for the project, and we decided to make video and multiple-user chats secondary objectives.

Designing our own communications framework meant that we would need to develop a client and server network structure that we could extend later on to interact with our chat application. Because we wanted to leave the final result of our project extensible, we would also need to make sure that the server structure was capable of handling both one-to-one communication and one-to-many.

### 3.3.4 Project Framework and Extensibility

The design for the messaging framework would need to be both extensible and support multiple forms of messages between the client and the server. To simplify development, we came to the decision that using the same basic framework for both sides of the connection would be ideal; and then we would only need minor additions and modifications to specify functionality.

Our initial expectations had prompted us to plan the inclusion of several forms of messages: text messages, audio messages, video messages, user updates, and other plug-in updates (such as updating a synchronized view shared by all users in the same chat session). To ensure that the framework could be modified to meet any of our needs, or the possible needs of future development, we included specific code points that would allow for modification and the addition of new features.  Our goal is to minimize the points of modification to extend or edit the existing framework, yet still be flexible to support whatever protocol or feature someone would look to add to the project.

## 3.3.5 Logical Design

By combining our conceptual model and the implementation considerations relevant to the context of this project, we created this design diagram (Figure 3.3) which divides the application into its primary layers and displays the package interactions.

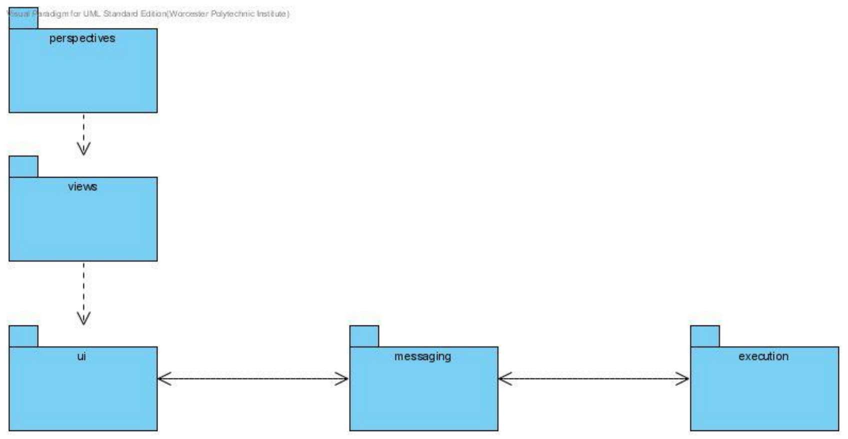
Visual Paradigm for UML Standard Edition(Worcester Polytechnic Institute)
perspectives
views
ui
messaging
execution

*Figure 3.3* – Package Diagram

*CLIENT*

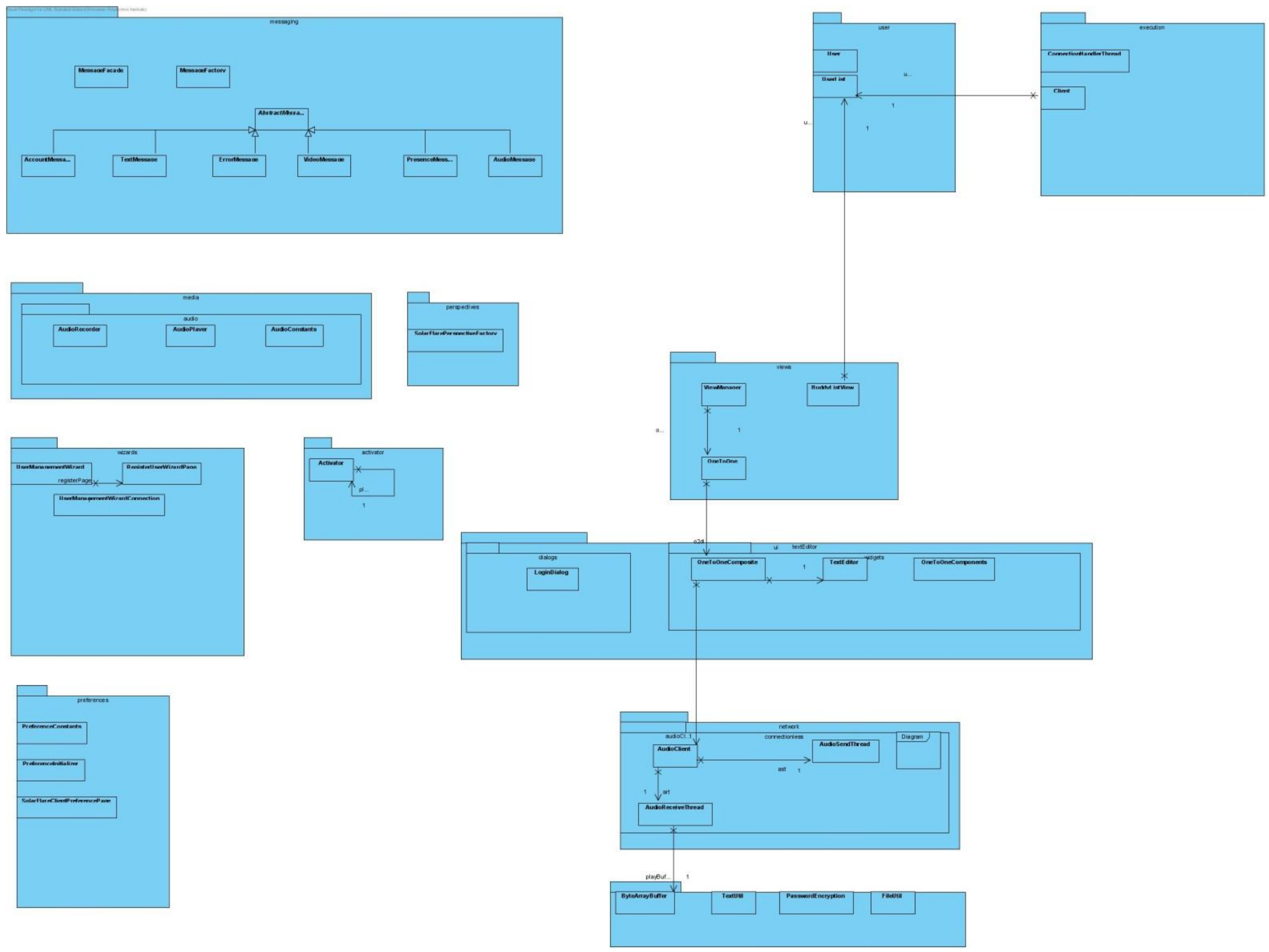

*Figure 3.4* – Package View (Client)

*SERVER*

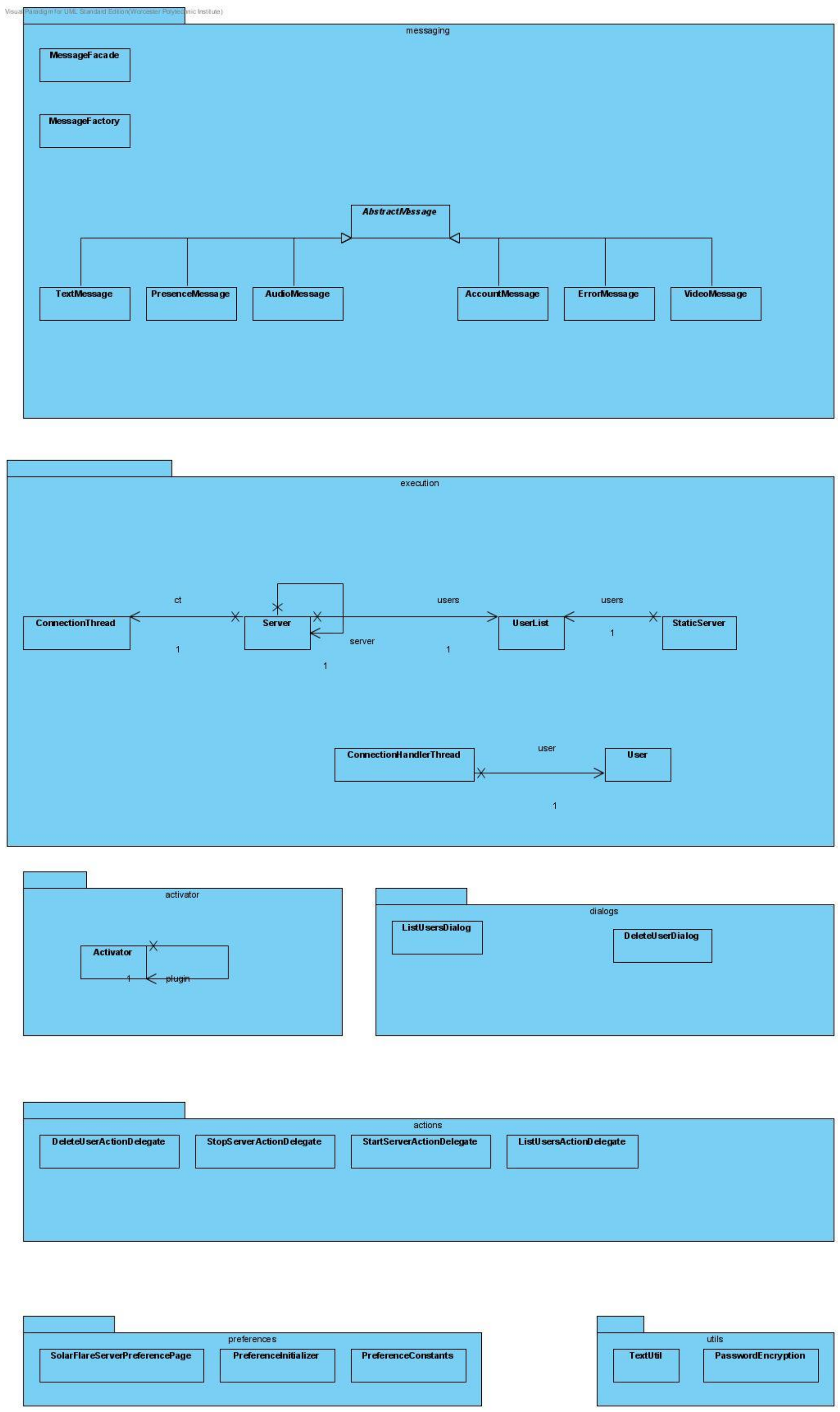

*Figure 3.5* – Package View (Server)

As shown above, Figures 3.4 and 3.5 display the Client and Server Class Diagram breakdowns, which include inner and outer package dependencies. The idea behind the process is to take incoming messages from a connection, learn the message type, and perform the required action. If a response message is necessary, it serves the response through the network connection and this loop is continued until the client or server is terminated.

### 3.3.6 Execution Package Communication

For this project we place the back-end or behind the scenes work within the execution package. Here we make use of our existing messaging structure and client/server communication. The client and server can read and write to or from each other after a connection is established. They each receive an XML based string message and ask the MessageFactory to return the correct type. This then calls a process message function on that message and puts the back end to work. This will do whatever needs to be done for that message type and serve a response or be reflected in the view.

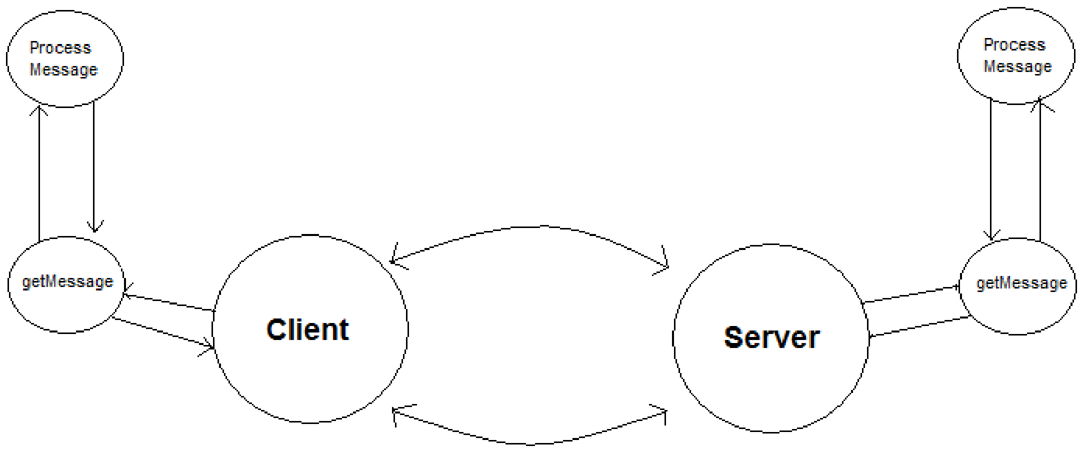

*Figure 3.6* – Execution Communication Diagram

## 3.4 Phase 4: Implementation & Testing

Utilizing the design models from Phase 3, we were able to implement our application, creating the predetermined packages and ensuring that each communicated properly with the others. Writing software code and testing that code concurrently ensured that each individual piece of code worked as intended when it was created, so that the final versions of each class – and ultimately, the package – worked appropriately in the system.

The following sub-sections provide general information about the internal layout of each package, a description of the package's operations, and explanations of the significant classes in that package.

### 3.4.1 Messaging Package Design

The Messaging package is designed to allow for extensibility by extending the AbstractMessage class. The AbstractMessage encapsulates common functionality for all messages that go to or from the client and server. In order for the framework to seamlessly reference different message types, each adds themselves to the MessageFactory. For easy reference in outer packages, the MessageFacade is used.

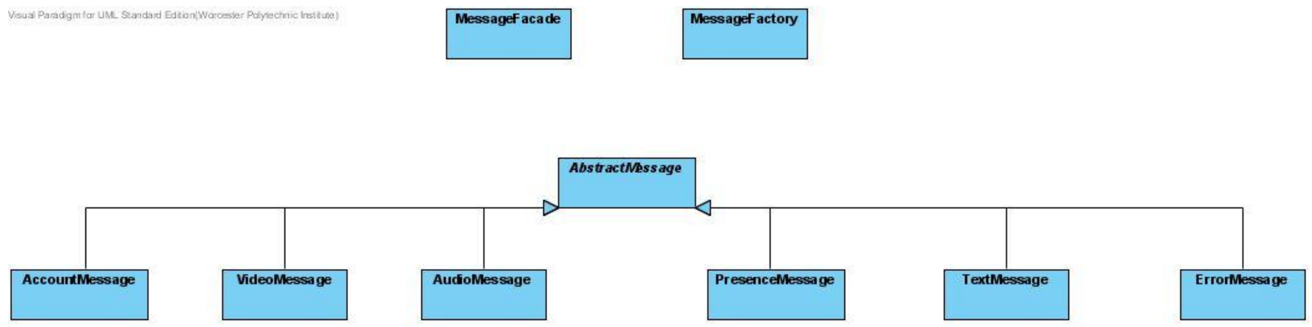

*Figure 3.7* – Class Diagram (Messaging)

#### Significant Classes

#### MessageFacade

This class is used as a façade for outgoing messages, and provides the methods used to construct a message to be sent for any purpose.

#### MessageFactory

This class is used to return the correct message type from an incoming AbstractMessage object.

#### AbstractMessage

Standard Message object format for all Message classes that ensures basic functionality with other packages in the system. Each of these message types is very basic, and designed to be easily modifiable for additional features.

### AbstractMessage

Standard Message object format for all Message classes that ensures basic functionality with other packages in the system. Each of these message types is very basic, and designed to be easily modifiable for additional features.

### AccountMessage

A Message type that carries account registration information. In the current version of this application, this serves only one purpose: registering new users with the server, and contains all of the required data fields for doing so.

### AudioMessage

This type of Message carries audio information for an audio chat session.

### ErrorMessage

This class is used for generating error message within the backend structure of the network.

### PresenceMessage

This type of message is used to inform chat session participants of who is currently logged into the server.

### TextMessage

This type of Message carries text information for a chat session.

## VideoMessage

This type of Message carries video information for a video chat session – because video chat was never fully implemented, this is not currently used.

### 3.4.2 View Package Design (Client Only)

Within the View package exists all of the visual representation frames for the eclipse workspace and a ViewManager as a point of communication for the backend message relay to each corresponding view. Here, updates to visual representations are called and necessary messages are sent to and from the ViewManager and out through the backend for whatever network or messaging communication is necessary based on those actions.

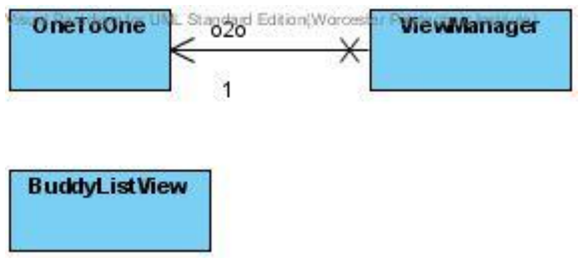

*Figure 3.8* – Class Diagram (View)

**Significant Classes**

#### ViewManager

This class is used to transmit information between the backend of the client and the views, to be used as needed.

#### OneToOne

The one-to-one chat view for the client; this class holds all of the UI visual and processing functionality for the view.

#### BuddyListView

This view is used to display the user's "buddy list", which consists of the users currently registered on the server. It also provides information detailing whether or not a user on the buddy list is currently logged in.

### 3.4.3 Plug-In Design

The SolarFlare Client and Server plug-ins are made up of multi-threaded design that integrates into the Eclipse workbench. There are UI elements that exist in both the Client and Server plug-ins containing mirrored communication layers that allow for coordinated communication of all users. The Server plug-in allows for the setup of a SolarFlare server, as well as starting, stopping, and managing existing users within the server. The Client plug-in features a wizard for creating a new user on an existing SolarFlare server, opening the SolarFlare Client Perspective, connecting to the server with login credentials, and communicating with a list of active participants on the SolarFlare server. The design rolls all of the previously covered elements into one, based on extension of the Eclipse workbench.

Although both plug-ins extend the *org.eclipse.ui.preferencePages* and *org.eclipse.core.runtime.preferences* points of the Eclipse platform to allow for the inclusion of preference settings in each, they share no other similarities. To provide the user interface views, as well as the perspective in which those views are encased, the *org.eclipse.ui.views*, *org.eclipse.ui.perspectives*, and *org.eclipse.ui.perspectiveExtensions* extension points are associated with the Client plug-in. Also associated with the Client is the *org.eclipse.ui.newWizards* plug-in, which is necessary for the addition of the Client User Management Wizard. The only additional extension point associated with the Server plug-in is *org.eclipse.ui.actionSets*, used to allow the server to perform actions such as “List Users”, “Start Server”, and “Stop Server”.

Both plug-ins have a set of user interface tools included. For the Client, this consists of the one-to-one chat view, the “buddy list” view, and a perspective to display those views. The one-to-one view provides a display showing the entirety of the text in the current chat session as well as control buttons and a dialog box for outgoing messages. The “buddy list” view displays a list of registered users and which of those users is currently logged into the server. The decision to encase these views in their own perspective was based on the communication needs of the views, and on the idea that an entire meeting perspective could be further extended to include other elements (such as shared view space). The server's user interface additions are limited to menu elements, primarily used for accessing the main functions of the server (e.g. "Start", "Stop").

## 3.5 Phase 5: Distribution

Upon the completion of the SolarFlare project, the client and server are distributed to the open source community for open beta testing. Here, help documents containing instruction on typical user tasks were integrated with the existing WEBFOOT help documentation. The existing code base was formatted and standardized according to WEBFOOT development guidelines. This was then merged into the WEBFOOT main branch for distribution at the WEBFOOT developer's website. At the time of this writing, the SolarFlare Client and Server projects are openly available for beta testing through WEBFOOT's developer website on SourceForge.

# 4 Results & Conclusions

This chapter analyzes the final product of our work, details issues that occurred during the process, and provides information about possible future work related to the plug-in.

## 4.1 Accomplishments

Initially, we determined a set of basic requirements that we wanted our application to meet. To create an effective application, we decided to prioritize these requirements, separating them into a set of primary and secondary objectives. Table 4.1 shows a list of the primary requirements for the project, with the accomplished objectives marked as completed.

**Table 4.1 – Accomplished Requirements**

| **Requirement** | **Completion** |
|---|---|
| A user must be able to open a voice chat from within eclipse. | Yes |
| A user must be able to record a live chat. | No |
| A user must be able to send and receive audio. | Yes |
| A user must be able to archive an audio chat. | No |
| A user must be either a host or a participant for an audio conference chat. | No |
| A host must be able to invite participant(s) to an audio chat | No |
| A participant must be able to accept or decline an invitation before or during an audio conference chat's scheduled time | No |
| A user must be able to mute their line in an audio chat. | No |
| A user must be able to disconnect during a conference or one-on-one chat | Yes |
| An invitation will be comprised of a user, a chat place, a host, a valid time period, and a chat pass code. | No |
| A user with an invitation can connect or re-connect at any time during the chat's valid time period | No |

| | |
|---|---|
| A user must be able to edit a recorded meeting into sub-segment clips that can be associated in SourceForge | No |
| A user must be able to record an audio clip without needing a scheduled meeting (personal note or audio comment) | No |
| A user will have a username | Yes |
| A user will have an avatar | No |
| A user will have a text accessibility | Yes |
| A user will have audio accessibility | Yes |
| A user will have webcam (which may be enabled or disabled depending on availability) | No |
| A user will have a group collaborators (buddy) list | Yes |
| A user can open a one-on-one text/audio/video chat with another user on their group collaborator (buddy) list | Yes (text, audio), No (video) |

Although we were somewhat satisfied with the final product of our work, we would have preferred to complete our primary list of requirements. Unfortunately, due to numerous issues that arose during the course of the project, we were only able to partially accomplish some of our primary goals, and we were forced to completely ignore the secondary ones.  However, new objectives were created and the project became a much larger undertaking than originally anticipated.

## 4.2 Issues

When designing a plug-in that is intended to provide communication alternatives for users familiar with other tools available (such as Ventrilo or AOL Instant Messenger), certain functions become expected, and can lead to increased complications for a small project team. For example, all but the most rudimentary text chat clients currently offer options for formatting outgoing messages (such as including links for external browsers). But much of this functionality is difficult to put into effective practice, and implementing some of these basic additions consumed a great deal of time and effort.

Our original hope was to extend and contribute to the Eclipse Foundation through the use of the Eclipse Communication Framework (ECF). Had this been a viable and stable option it would have been more realistic to reach our intended requirements. However, the project is in a state of constant flux and the current existing protocol implementations make poor use of the frameworks design and nullify the design intentions.

Existing open source project typically only focus on one of the features that the SolarFlare project is looking to tackle, whether it be VoIP, Video, Text, Whiteboard, etc. In short, this meant that what was out there to work with was not designed with the intent of extension or modification. Other protocol options or solutions were either not open source or not a free alternative. Therefore, we were faced with the task of creating an extensible framework of our own and creating a protocol of our own to show the framework in action.

After designing our framework, implementation was met with many challenges. As is typical with software development, you can expect most of your problems in the places that you least expect it. When starting the project, we were hoping to work from a test driven approach, based on our requirements. However, this became increasingly difficult as time went on, as most of this project is driven by network communication, which itself creates issues as information travels across varying networks of the World Wide Web. Much of the testing became manual and was intended to insure reproducibility in certain aspects of the project, such as audio transmission by comparing bits streamed across the network. If more time was allotted for the project, JUnit tests would have been used much more extensively than they currently are.

There would be times throughout the project where a potential solution would come up for a problem and we'd run with it as far as we could, but ultimately have to start from scratch as soon as it was deemed either too complex or not feasible. One such situation was with the text chat editor. Our initial implementation featured a key listener that would be updated whenever a key was pressed and would keep track of when a user was adding or turning off style features or creating a link. This, however, was not a robust solution. There came a point where it became too difficult and too complex to handle modifications to the existing text and convert this to HTML to be sent to the receiver. Instead, we opted to modify an existing solution that made use of SWT's StyledText editor and a reliable solution was created by parsing the style information after the send button was hit and converting that to an HTML message to be sent out to the user. This is, of course part of the iterative development process, and starting from scratch most likely saved us countless hours of attempting to extend a base that would have performed poorly and been a nightmare to refactor in future iterations.

Initially there was a learning curve for many of the features to be included in the SolarFlare project. This was our first endeavor making use of SWT, Java Sound, the Java Media Framework (which as of this release there are no features making use of it), ECF, Eclipse Plug-In Development, and extensive multithreaded programming. The undertaking and research behind the design decisions was much greater than any course that we have taken as Undergraduate students. Our lack of knowledge and experience caused some issues, where mistakes were made at first. These problems were rectified as greater understanding of the existing tools was gained. There were also times where example code was given as a solution or certain tools required the use of certain methods, but were dated and made use of deprecated methods. This made our task larger, where we'd have to fill in the holes of something intended to be used as starter material and gain a basic understanding of these new API's.

Natural problems inherent in any project existed as well. We are a small group comprised of two senior students, but even then we had two vastly differing schedules that made collaboration with each other difficult. We tried our best to communicate through existing collaborative tools, such as SourceForge and AIM, but that can only take you so far. Towards the end of the project, we weren't even in the same country, where being in different time zones made it difficult to communicate with each other in real-time. However, these setbacks were probably the most minor of the project.

## 4.3 Overall Functionality

Currently, the SolarFlare project allows the basic and necessary functionality to host a server and manage users, as well as client side interaction. Users can connect to a server and initiate communication supported by the server. In order to run and maintain a server, the plug-in only needs to be installed and is launched and seized at the click of a button. A user list can be viewed by the server administrator where they can remove inactive or unnecessary accounts. Server preferences can be set as to where the user data is stored. All passwords are encrypted and saved on the server with the user information. Each user's information is contained within a serialized java object that is loaded upon server startup. One of these objects is stored upon user creation or modification.

The client side interaction is where the majority of the project's functionality comes in, as it features user interfaces for rich text chat, audio capabilities, and preference management. The client allows for users to register and connect to a server and store preferred information to cut down on unnecessary repetitive input. For text chat, the user is allowed to modify the formatting of the text to Bold, Italic, and Underline font styles. Audio communication is initiated and disabled at the click of a button.

Beyond the realm of typical user interaction is the extensibility of the SolarFlare project. The project allows for integration of various protocols in the messaging structure, so long as the client and server both understand the message type.  This is much like speaking a language to someone, if you both don’t speak the same language, you can’t communicate properly.  There are four major points of modification in order to insert a new feature in the SolarFlare project:

- Extend the AbstractMessage in the Client & Server
- Insert reference to new AbstractMessage type in the MessageFactory
- If visual, create a view (modify ViewManager if necessary)
- Insert the new view into the SolarFlare Perspective

These extension points allow for communication within the SolarFlare framework as well as the eclipse workbench.

## 4.4 Future Works/Additions

The SolarFlare project is a project that has high hopes of many additions and extensions to the existing framework.  To say that not meeting the intended requirements makes this project a failure would be an egregious error.  Creating the overall framework was a great undertaking and has paved the way for future projects and developers to add what would be considered the “true vision” of SolarFlare.  Some of these features are not terribly difficult to implement, but when working under a tight schedule and short time frame, a project cannot always deliver everything it intends, no matter how minute.

First and foremost, the ability to store the data to various locations for any form of media chat within SolarFlare should be implemented.  The ability for teams to reference notes and documents concerning a myriad of topics is paramount and a key feature intended for the initial release of SolarFlare.  Some of this functionality can easily integrate with other existing WEBFOOT projects, such as the Document Manager plug-in, Task Manager, and User Stories plug-in.  Our originally visions contained a concept of not only storing one-to-one or one-to-many meeting transcripts or recordings, but potential “note-to-self” style captures that will allow a user to tag an artifact with any form of media note they see fit. This feature could be used for future reference by the creator or others within the team.

A white-board or shared canvas feature should be included in the future. This is something that becomes very useful in real life meetings, as no matter how well a person may try to explain something with words, a visual representation is sometimes necessary and usually aids whatever has been explained. This is another form of collaboration that is sorely missing from many of the standard chat clients that exist today.

Video is another feature that should integrate nicely with the existing audio and text functionality. Sometimes it helps create the illusion of truly being in the same time and place and working with someone if you can associate a face to the voice or text you’ve been working with on a project. The hope is to break the borders that exist in the physical world and bring people together to collaborate in a unified and productive manner.

## 4.5 Conclusions

In closing, the project was a success. Though expectations were high from an initial requirements standpoint, the vast undertaking that the project ultimately became was an incredible task. As a team, the fact that we started with nothing besides eclipse and a book in front of us intended as a beginners guide to developing eclipse plug-ins and ended up with a working, extensible, professional quality product is something to be proud of. Something now exists as a stable foundation for extensive development of collaborative tools. The necessary functionality to communicate with distributed partners is already in place as of the beta release of SolarFlare. With the project in a state of open beta as of the completion of this paper, suggestions and problems to solve will appear as people use the product.  Only be obtaining this feedback will we know what direction the project will ultimately take in the future or what external developers will contribute, as the project itself is intended to be open source.

The project itself encapsulated many of the topics covered throughout courses at WPI.  The framework integrated design patterns studied in Object Oriented Analysis and Design.  There is an idea of cross network communication, where these messages are passed along, as would be covered in Networks.  The collaboration of the SolarFlare developers in an iterative development environment is something studied and practiced in Software Engineering.  Of course the necessary knowledge and understanding of tools that encompass the project are picked up throughout various classes, too many to name.  Essentially the SolarFlare project employed all of our acquired knowledge and then our acquired skills of fast assimilation of new materials to overcome the obstacles presented in the development process.

Though we conclude the project is a success, the body of work must speak for itself.  Thorough use and testing will justify our claims.  Time will tell if our hope of presenting a platform that will be useful and ease the process of design and development is a reality.  The software must live on as its own entity in the open source and educational community, hopefully extending beyond our vision.  We leave this as our last and greatest endeavor as undergraduate students at WPI and believe it to be a proud summation of our time here.

## Bibliography

Software Tools (Paperback)
by Brian W. Kernighan (Author), P. J. Plauger (Author)
1976

Eclipse.org

Software Engineering Terminology
http://w3.umh.ac.be/genlog/SE/SE-contents.html
2006

Developing Eclipse plug-ins
http://www.ibm.com/developerworks/java/library/os-ecplug/
2002

Visual Studio Team System 2008 – Product Information
http://msdn2.microsoft.com/da-dk/vsts2008/products/bb933734(en-us).aspx
2008

SourceForge Enterprise Edition Product Overview
http://www.collab.net/products/sfee/
2008

Free On-Line Dictionary of Computing
Foldoc.org
2008

Webfoot Task Manager
by Brandon M.Greenwood (author), Michael E. Hlasyszyn (author), Gary Pollice (advisor)
2007

User Stories through Eclipse in SourceForge
by Jonathan Bolt (author), Christopher Fontana (author, Gary Pollice (advisor)
2007

Improving Collaboration with On-Line Meetings
by Kerri Edlund (author), Sarah Pickett (author), Gary Pollice (advisor)
2007

SIP programming for the Java developer
http://www.javaworld.com/javaworld/jw-06-2006/jw-0619-sip.html
by Wei Chen
2006

SIP Communicator (Java Open Source Project)
https://sip-communicator.dev.java.net/

Eclipse Communication Framework API Documentation
http://www.eclipse.org/ecf/org.eclipse.ecf.docs/api/

Java Sound Demo
http://java.sun.com/products/java-media/sound/samples/JavaSoundDemo/

Java Speech API
http://72.5.124.55/products/java-media/speech/

Java Media Framework
http://72.5.124.55/products/java-media/jmf/

Eclipse Scribble Share (ECF)
http://www.eclipse.org/ecf/scribbleshare.html
2006

Eclipse ECF Open Source Skype
http://ecf1.osuosl.org/

ECF VoIP Research from Google Summer of Code
http://wiki.eclipse.org/VoIP_in_ECF%2C_GSoC07_DevLog

Java Media Framework (JMF) Programmers Guide
http://java.sun.com/products/java-media/jmf/1.0/guide/index.html

Java "Time Turner"
http://www.jsresources.org/apps/9159_TimeTurner.pdf

Password encryption: rationale and Java example
http://www.devbistro.com/articles/Java/Password-Encryption
By J. Shvarts

## Appendix

All Appendix documents are bundled with the electronic submission of this project.  They include in depth structural overviews of the client and server side code.