A Study in the Analysis, Design and Implementation of an Air Traffic Control Simulation System Using UML

Scott Johnson, Kristopher Zarns, Ritu Banerjee, Robert Ellingson, Travis Dazell, Ryan Langseth, and Tyler Mathwich

Department of Computer Science
University of North Dakota
Grand Forks, ND 58202

contact
Dr. Emanuel S. Grant, Ph.D.[1]
grante@cs.und.edu
701.777.4133

Student Submission

Abstract

In today's ever-changing world, *Software Engineering* is a term often used to indicate a focus on the process of designing software, rather than only the actual writing of code. In order to attain time and cost constraints, software design teams tend to focus on learning a *process* of building software. To simulate this process development, a study was done on the design and implementation of an Air Traffic Control Simulation System. The system was modeled in the Unified Modeling Language, with a focus on the design being platform-independent. The software product was implemented using a combination of programming languages. In order to complete the project by the required deadline, several revisions of the design of the project.

[1] Portions of this work was partially funded with a grant from the Rockwell-Collins Corporation

1. INTRODUCTION

The Department of Computer Science at the University of North Dakota offers a senior-level Software Engineering course, designed to teach a structured approach to problem analysis and software design, analysis, and implementation. The course is designed to teach the software development lifecycle, through a *Model-Driven*, *Object-Oriented* approach. The course is project-centric, i.e. software engineering topics are presented to support the development of an Air Traffic Control Simulation system. The goals of the project were the learning of high-level software development techniques, development of management skills when supervising/working on software projects, development of communication skills between members of a team, and the construction of several levels of documentation to accompany the developed software application. After completing the required course of study in the course, it is expected that participants would have developed an understanding of the use of the *Unified Modeling Language* in defining system models, the benefits of modeling in software development, the dynamics of teamwork in software development, and the application of a well-defined methodology in building software in an object-oriented manner.

This paper is a report of the research accomplished in the aforementioned course. Following this introduction, the project is described in detail. This includes the development methodology used to construct the project, as well as insight into what expectations were in place for the final software. The paper then discusses how the project was implemented, from a high-level design through the writing of the computer programming code. Finally, a conclusion summarizing what was learned from this project, as well as areas for future work.

2. PROJECT DESCRIPTION

The course instructor provided an initial set of requirements for the project. Due to the ambiguity of the textual specification, some clarification was required. The requirements focused on a user who was utilizing the program for learning/training or entertainment purposes. Since the software was an air-traffic control simulation, the user's actions would mimic that of an air traffic controller's, but the system did not have the real-time requirements usually inherent in software on which human lives depend.

2.1 Problem Definition

The project required the developers to create a program that simulated an Air Traffic Control system, in which an Air Traffic Controller (ATC) could control planes within a 50-mile radius of the airport. The system graphically represents: the runway, a 50-mile radius airspace, enter/exit gates into/from the airspace, flights within the airspace and flights within 10 miles of entering the airspace. The displayed radar itself had to update every 5 seconds, redrawing flights for their current positions on the screen.

In addition to these requirements, each flight had three constraints: speed, heading, and altitude, that are used to carry out the ATC's instructions. The heading constraint was given in degrees, and represented the current compass direction in which the flight was moving. Speed was a constraint given in knots, and represented a knot, as defined by the Merriam-Webster Online Dictionary [2], as a measurement of speed indicating one nautical mile per hour, which is equivalent to 1.15 miles per hour. Each flight's altitude was a measurement of the height of the plane above sea-level, in feet. In addition to representing these values for each flight, the specifications indicated that the user must be able to change each of the values for flights within the controlled airspace. The heading value would be updated immediately, while the altitude and speed values would be updated over a period of time, effectively simulating the climb or descent of a flight, and the deceleration or acceleration of the aircraft.

Finally, the specifications called for two types of flights: arrivals and departures. Arrivals were created with the intent to land on the runway at this airport. The Flight Logic Sub-system would then remove such flights from the radar screen, after they landed. Departures, on the other hand, were flights created on the runway, bound for some other airport as a destination. The flights created as departures were to be under the control of the ATC until they exited the airspace.

All flights within the airspace were checked routinely for collisions. If two flights approached within three miles of each other, and the two flights' altitudes were within 1000 feet, the ATC had to be warned that a collision could occur between those two flights. If two flights collided, they were to be removed from the screen, and replaced with a different single icon, representing the collision.

2.2 Development Methodology

Most of the major work that went into the project was in designing the overall system. In order to design a system, a development methodology was used. A development methodology prescribed set of activities to be taken in the development of software. It is referred to as a "development process."

2.2.1 Classical Development Methodologies

Typically, the process the developers take when designing and constructing a software system is divided into stages, or phases. Ghezzi, et. al. [1] distinguishes between several different classical methods for defining these phases. One of the most popular methodologies is entitled the "Waterfall Model," [1] in which each successive phase depends on the completion of the previous phase before work can proceed. Other methodologies, such as the "Iterative" and "Spiral" methodologies, are also described by Ghezzi, et. al. [1]. The development for this project conformed to a modified waterfall model, where the standard methodology was applied in such a manner that from any given phase work could flow backward to a previous phase, if revisions were required.

2.2.2 Model-Driven Development

Development of a major software product, especially by teams, requires the coordination of personnel and resources. Object-oriented Design and Programming [3] in software development focuses on specific structures, or objects, within a system to create a structured software product. The use of object-oriented design and programming in software engineering development facilities eases coordination in team development. Commonly involved in this design process are graphical models, depicting the use of the system as a whole, as well as how individual parts of the system work together [3].

2.2.3 UML

To model how the system's components worked together as a whole, the Unified Modeling Language, or UML [4], was used. The UML is the Object Management Group's [4a] standardized notation for the graphical modeling of object-oriented (OO) software systems and applications. It has evolved into the de facto standard for modeling notation in OO software development, and has been applied in a number of areas outside of OO software development. The UML was designed to provide developers with a clear and concise visual representation of how the inner components of a particular software product work [4].

2.2.4 Case Tool (*Rational Rose*)

Representation of the graphical models can be done in a simple drawing program; however, using a specific software design tool has several advantages over these general-purpose drawing tools. One of the major advantages of software design tools have is that once the models have been designed, some programming code can be generated automatically. In addition to this, some software design tools allow the user to generate new models from ones they have already been created in the tool.

For the research done in this project, the tool *Rational Rose*® was used. *Rational Rose* is a tool that has become the de-facto standard for diagramming software using the UML. *Rational Rose* is a complex development tool, and proved to be difficult to learn. Tools similar to *Rational Rose* become essential in software development, once one overcomes the initial learning curve.

2.3 Project Workplan

The first phase of development involved analyzing the requirements of the project, and producing a set of analysis-level models. The goal at this phase of the design process was to identify the users of the system, how these users interacted with the system, and begin to assess the objects involved in the system, at a high level of abstraction.

Upon completion of the analysis-level models, the development moved into the design phase, which involved the creation of more complex and descriptive models. In this phase, the main goal was to take the models derived from the previous stage, and enhance them to be more descriptive of the system as a whole. This phase also introduced more in-depth types of models, as well as construction of several necessary system-level functional algorithms.

After proceeding through the design phase, the work moved into the implementation phase, where the actual code was written and tested. Modules were constructed by individual team members then integrated to form the complete system.

Testing followed after completion of the module development. This stage was augmented in our modified waterfall methodology. There were several instances where modules worked individually, but did not work with the system as a whole. In such an instance, the module was re-implemented, which involved reverting to the previous phases in the software construction cycle.

Finally, the product was delivered. Normally, this phase would correspond to the delivery and maintenance phase of the software lifecycle, but since the system was a semester course project, very little maintenance was actually carried out.

2.3.1 Gantt Chart

In this project, teamwork was essential to getting the job done on time. In order to make sure that the work was clearly managed between the team members, a *Gantt Chart* [6] was used. A *Gantt Chart* is a visual graph depicting each team member, along with the parts of the project they are expected to accomplish. It was correlated with a time axis, to demonstrate the successful completion of the phase of the project.

2.3.2 Team Cooperation

The *Gantt Chart* allowed the team members to break the seemingly massive project into a series of smaller sub-projects, each of which was handled by a single group member. Communication among group members was essential, as many of the modules in the software product were heavily dependant upon each another.

Once a team member had their particular task finished, they may or may not have been able to move forward with the next task on the chart. If they could not move forward (due to a prerequisite that may not have been met yet), that team member would assist in other tasks that were not yet completed.

3. PROJECT IMPLEMENTATION

3.1 Requirements Analysis

Requirements analysis for the project was broken into three parts. First, a document was provided that listed the initial project requirements. Specifics included items such as the size of the airspace, the behavior of planes, and some basic information about the user interface.

The second part of the requirements analysis occurred in the form of a demonstration of an existing product by an air traffic control expert[2]. In some areas, the demonstrated product contradicted the original requirements. This brought about the third and final stage of the requirements analysis, to clarify and refine the original specifications.

At this point, the development team contacted the contractor of the project, which, in this case, was the instructor of the course, to clarify ambiguities between the initial textual requirements and the expert's demonstration. This aspect of requirements analysis and refinement continued throughout the remainder of the project. At each stage where some requirement was found to be ambiguous, the contractor was contacted, and the ambiguity was clarified.

3.2 Analysis Models

The analysis and design stage of the software development process included three design sessions. These three phases of the development were the creation of analysis level, design level, and detailed design level UML models of the system. At each stage in the design process, the team discussed the models developed, problems encountered, and solutions that were implemented in response to the problems. The team also discussed any significant events or activities that were experienced during this stage of the software development process.

3.2.1 Analysis Level Diagrams

The first set of models developed were the analysis level models, which included a use case diagram and an analysis level class diagram. The goals at this stage in the design process were identification of the users of the system, determination of how these users interacted with the system as a whole, and assessment of the system's classes and attributes.

The first model developed was the use case diagram. The purpose of this model was to identify the users of the system and the functionalities that the system would provide for these users. Typically, a use case diagram is composed mainly of actors, which represent

[2] We thank Harsh Mathur of the UND Aviation Department who provided us with invaluable assistance.

the users of the system, depicted by stick figures, and use cases, which represent tasks that the system can perform on behalf of the user, represented by bubbles enclosing a service name. An example of the use case diagram developed is provided in Figure 1.

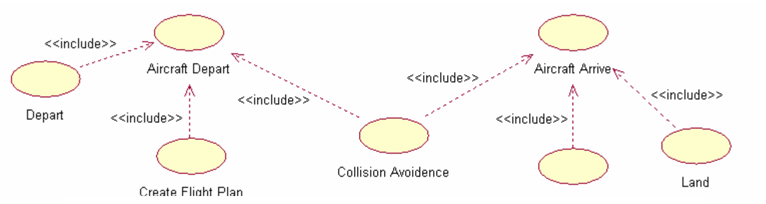

Figure 1: Example of a Use Case Diagram

The team learned the importance of and the contribution to the understanding of the overall functionality of the software system developing the use case diagram provided. Initially, the team struggled with the development of the analysis level diagrams, and had to seek outside resources for information and tutorials on proper use case construction. The *www.objectmentor.com* website provided tutorials on UML design [7] that proved to be very helpful. The author included a walkthrough on developing proper use case diagrams, which aided in the modeling at the analysis level. The most prevalent mistake made was incorporating flow of events into the use case diagram, which is illustrated in later diagrams. The analysis models also included an analysis-level class diagram, an example is illustrated in Figure 2. This diagram illustrates the static objects in the software system, and their relationships. At the analysis level, the classes, their attributes, and relationships between the classes were defined.

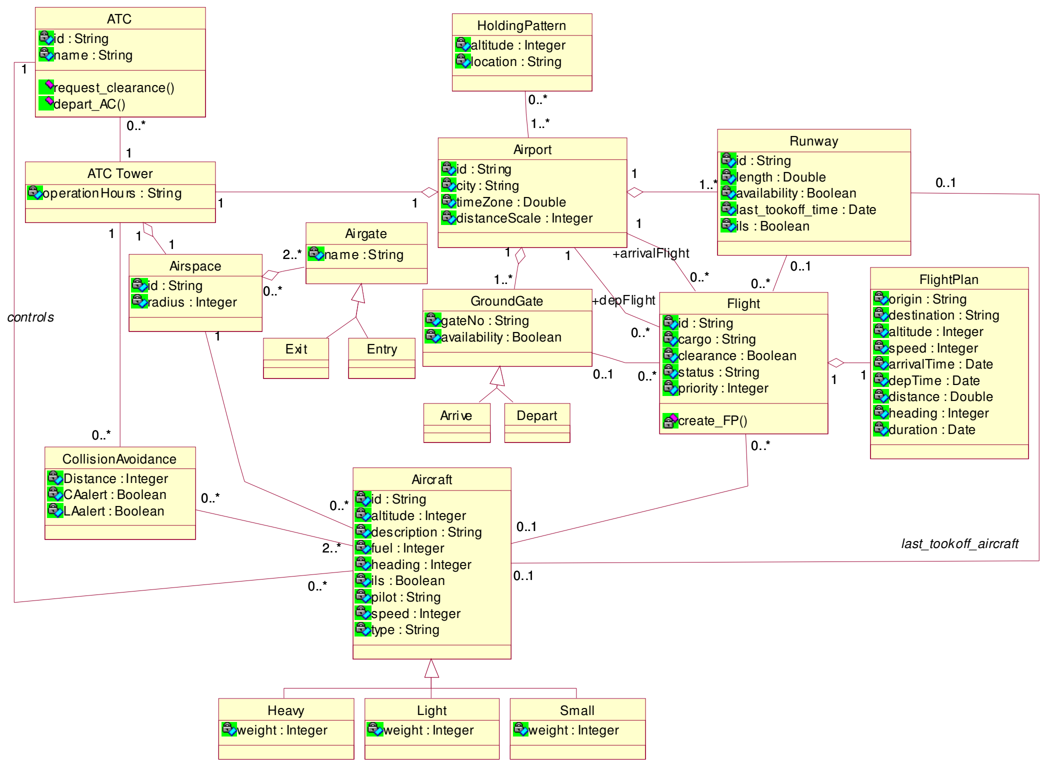

Figure 2: Example of an Analysis Level Class Diagram

The most challenging aspect of developing the class diagrams was enforcing the software engineering principle of “separation of concern” [1]. It's natural to design the class diagram with a particular programming language in mind. Introducing language specific concepts early in the design process is a mistake that can lead to problems later in the process. By separating each step of the design process, the team was able to concentrate on the task at hand and not be influenced by future design issues.

3.3 Design Level Diagrams

The second set of deliverables in the project design was the design level diagrams, which included use cases, a design level class diagram, a static dictionary, and activity diagrams. At this stage of the design process, the details of the static elements are defined and activity diagrams were introduced into our set of UML models.

In the first step of the design phase, the descriptions for each of the use cases in the use case diagram were specified. Each use case included details such as generalizations, specializations, and descriptions. Through the construction of the use cases, the purpose and relationships of our system services became clearer.

A static element dictionary was compiled to provide a clear and concise definition of each of the classes. Writing the dictionary turned out to be more of a challenge than was first imagined. This was due to doubts regarding the importance of a static element dictionary. Most of what was defined was obvious, so there was the need to “*step out of the box*” to realize the need for a well-built dictionary.

The last portion of the second set of deliverables was the construction of UML activity diagrams. For help in designing these diagrams, Martin and Newkirk's “Walking Through a UML Design” was consulted [8]. This helped in relating the activity diagrams to the existing set of UML models. A considerable amount of went into defining activity diagrams, which showed the sequence of events that needed to take place for each use case. It was surprising what was learned about the system during the development of the activity diagrams. Questions arose that had never been considered and a better understanding of the activities and events within the system was gained.

3.4 Detailed Design Level Diagrams

For the third set of project deliverables, a dynamic element dictionary was compiled, UML sequence diagrams were created, and the initial class diagrams were revised. The biggest challenge faced at this level was the anticipation of beginning the implementation phase.

A dynamic element dictionary was easily built, partially because of the team’s previous experience with the static dictionary. While the static dictionary provided definitions of the static concepts in the system, the dynamic element dictionary described the operations and events that exist in the system. This dictionary may prove to be useful for future reference in maintaining and modifying the system.

In addition to activity diagrams, which provide a flowchart-like description of use cases, sequence diagrams were constructed to illustrate the order of operations with respect to time. The sequence diagrams also show the interaction between objects in the system. This proved to be a useful tool for realization of the purpose and importance of class operations. An example of a UML sequence diagram is shown in Figure 3.

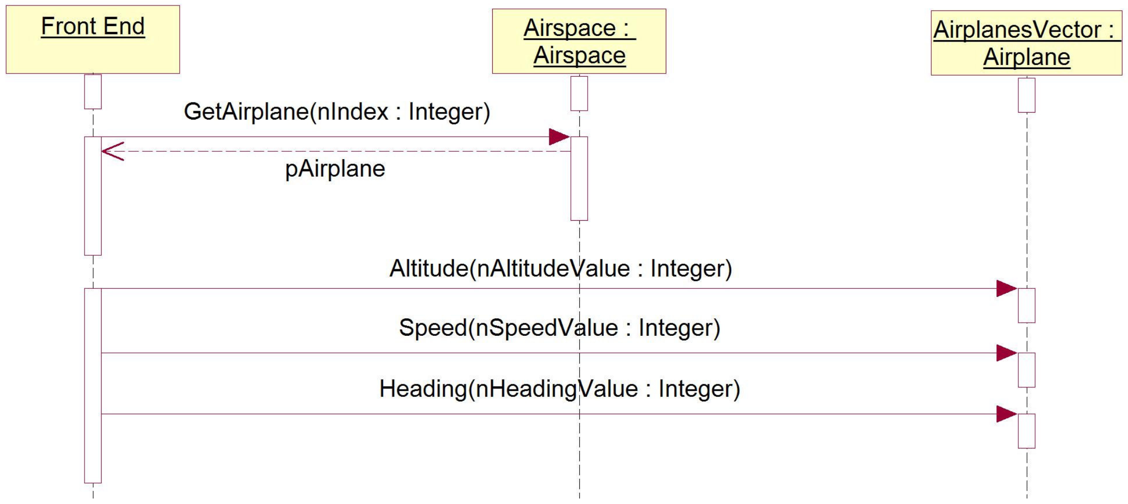
Front End
Airspace : Airspace
AirplanesVector : Airplane
GetAirplane(nIndex : Integer)
pAirplane
Altitude(nAltitudeValue : Integer)
Speed(nSpeedValue : Integer)
Heading(nHeadingValue : Integer)

Figure 3: Example of a Sequence Diagram

After constructing the sequence diagrams, operations were added to the existing class diagrams. This eased the perception of some of the dynamic elements that would exist in our system. Simple *get* and *set* operations for each of the class attributes were the first operations defined. Later, more system-specific operations were added to each class. New attributes were also added that were left out of the initial class diagrams. It was realized that these were necessary attributes as the class operations were developed.

3.5 Implementation

3.5.1 Classes

The software system was developed on the Debian Linux operating system, and the source code was implemented using C++ and the GNU g++ compiler. Loudon's "C++ Pocket Reference" was used to address any C++ questions [9]. The header and source files were separated from the rest of the system application code to ensure modularity. The manner in which the class module was constructed facilitated it being used by any outside module that has reference to the header files.

The container attributes were implemented using vector objects in STL [11]. This allowed the addition and removal of objects when needed. An example of where this is implemented is in the airspace object, which contains airplanes. As airplanes need to be added or removed from the airspace, the vector allows the dynamic allocation of space. To allow outside modules to interact with the class modules, operations were included to return pointers to objects within the vector.

The class diagram proved successful when it came to coupling the classes. The work of writing the classes was divided between two project members. After each class was written, they were put together to complete the associations between classes. By following the plan outlined in the class diagram, it is possible to know exactly how all of the classes would be written and how they would depend upon each other. This made the class construction a much easier task than it would have been without a model to follow.

3.5.2 XML

For graduate credit, additional features were added to the initial requirements for the system, which included save and load functionalities for creating customized airports. The data was written out to an XML file and later retrieved and parsed to restore the saved configuration. A portion of a sample XML scenario file is illustrated in Figure 4.

```
<runways num='4'>
        <runway>
                <runwayID>0</runwayID>
                <heading>92</heading>
                <length>0</length>
                <x>180</x>
                <y>275</y>
        </runway>
```

Figure 4. A section of an XML scenario file

The save and load functionalities were built into a separate module without consideration of any front-end interface, based on the principle of "separation of concern." This makes the module more flexible and modular. After writing the source code to save and load airspace information via an XML file, the module was tested to ensure that all created files were valid XML format, and that the data for loading an airspace would be retrieved successfully.

3.5.3 GUI

The front-end graphical user interface was written in C/C++ using the OpenGL libraries and GLUT, user interface construction tool associated with OpenGL. A prototype GUI was built before the class modules were compiled and ready to be utilized. This provided an indication of what to expect from the final product and gave a head start on the production.

The graphics intense nature of the system combined with the event-driven environment made for a challenging undertaking. The interface consists of a main viewport which displays the airspace. Displayed in the airspace are the entrance and exit gates, runways, and aircraft. To create an arriving aircraft, the user simply right-clicks outside of the airspace boundary. The plane will then enter the aircraft through the nearest entrance gate. By right clicking on a runway, a departing aircraft will be created. The user can then guide the aircraft to the desired exit gate. To change a plane's flight pattern, the user clicks on the plane and enter an altitude, speed, and heading, pressing the ENTER key after each entry. The values entered are displayed in text boxes for the user to see.

In addition to the main viewport, there is a sub-window with buttons that provide the user with additional functionalities. There is a button for optionally displaying circular distance markers at five or ten mile intervals to make it easier to see how far planes are from the center of the airspace. A pause button is included to allow the user to pause the simulator, as well as an exit button to close the program. There are also two buttons, to load and save a scenario respectively.

There is a separate GUI that the user can utilize to create a customized airport. This GUI consists of buttons for displaying concentric circles at five or ten mile intervals, loading a saved airport, saving a created airport, and exiting the utility. There are also buttons available to add runways and gates.

To add a runway to the airspace, the user simply clicks on the “Add Runway” button and then clicks twice on the screen. The first click places the runway on the screen and the second click positions the runway in the direction of the mouse click. Gates can be added to the airspace by clicking on the “Add Gate” button and then clicking on the airspace to place the gate. Clicking inside the airspace boundary will place an entrance gate, while clicking outside the runway will place an exit gate.

Once the airspace is built, the user can click on the “Save” button, type 'save', and hit the ENTER key. This will save the newly created airport into a save.xml file. To load this airport, the user can then open the simulator, click the “Load” button, type 'save', and hit ENTER. This will load the scenario from the save.xml file and display the airport to the user. At any time during the simulation, the user can pause the game and save it. This will save the current positions and settings of all aircraft in the airspace. By loading the file, the user can resume the scenario.

3.4.1 Coding

Coding was performed in a modular fashion. The project was broken down into several modules with each module being developed independently. Modules included: data structures, simulation aspects, xml features, and graphical user interface. After completion, modules were tested individually. First the data structures and simulation aspects were integrated into a simulation system. Following that, the simulation was combined with the interface to allow for further testing. Finally the XML portion was integrated into the system to allow further testing by running various scenarios.

3.5 Testing

The initial testing occurred at the module level. These were white box tests of the modules to check for basic functionality. Each coder was responsible for making sure his module was usable prior to providing it for integration. Further testing occurred at the user level with black box testing as soon as the simulator was integrated with the GUI. This helped to reveal ways in which the GUI was improperly implementing the simulator and problems internal to the simulator. At this point testing for usability, Stress/load, and robustness occurred. Usability tested for two things; whether the user was able to interact with the simulation as stated in the requirements document, and if the system would run properly independent of the development platform. The latter provided valuable information to the GUI developer. This pointed out problems like varying screen resolutions, the usability of the color scheme, and problems with the sound system causing the system to lag. The stress/load tests were performed by running the system in a multitasking environment, testing it on various computers with different hardware specifications, finally testing the responsiveness in worst case scenarios. The reliability testing focused on whether the system would remain stable when the user entered bad input. For example, if invalid values were entered for the speed of a plane; would the plane be destroyed, would the plane be updated to the invalid speed, would it continue with its current speed, or would the system crash.

4. CONCLUSION

The development of the course project gave us hands-on experience in both software engineering and programming. We were able to apply the principles and techniques learned in the classroom to an actual real world application. Given the short time frame for developing the project, it would have been extremely difficult to finish the assignment without developing system models. By implementing the software engineering process, we were able to understand our system much easier and have a plan for the implementation. We feel that the project was very beneficial in helping us tie the classroom education to the development of a software application.

4.1 Benefits of Modeling

Throughout the development process, it was determined that the software development would have been much harder using the "think-code-test" procedure normally used by freshman and sophomore Computer Science students. Modeling the software system allowed the team to conceptualize it before implementation. In addition, it was possible to make assumptions about the software, effectively allowing the developers to rationalize about the system. This created fewer problems, as more problems were discovered (and thus solved) before the actual implementation.

Modeling the software project also allows the work to be split up into logical modules. These modules are separate entities, so it was easier to assign particular modules to be implemented by individual team members.

4.3 Future Work

Unlike most engineering disciplines, when a software product is delivered to the end-user, this work is not finished. Part of the software lifecycle is to maintain and update the software product, even after it is released. Some future work could be done to detect and correct errors that may be present with the software product.

Additionally, modifications to the software could be done to implement features not in the first version. This supports the software engineering principle of "evolvability" [1]. Such features would be highly dependant on user input, and original models would need to be modified before the changes were implemented in the software code.

REFERENCES

1. Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli .*Fundamentals of Software Engineering*, Prentice Hall 2003.
2. http://www.webster.com/cgi-bin/dictionary
3. H. M. Deitel & P. J. Deitel. *C++: How to program, 4/e*, Deitel 2003.
4. http://www.omg.org

4a http://www.omg.org/UML

5. Dan Pilone. *UML: Pocket Reference*, O'Reilly 2003.
6. http://www.ganttchart.com/
7. http://www.objectmentor.com
8. Robert Martin & James Newkirk. *Walking through a UML Design*, http://www.objectmentor.com/publications/Walking through A UML Design.pdf 1998.
9. Kyle Loundon. *C++ Pocket Reference*, O'Reilly 2003.