Final Report

Group Messaging System

IEEE Computer Society International Design Competition 2003

Shawn French

Stephen Haber

Matthew Chmiel

I Abstract

This report documents the fourth-year project efforts of Shawn French, Matthew Chmiel and Stephen Haber for Carleton University in 2002-2003. The project discussed in this report is the Group Messaging System and was undertaken in response to the Computer Society International Design Competition. The purpose of the Group Messaging System is to better facilitate intra-group communication, specifically communication within the home.

The following sections of the report discuss the background, design, implementation and testing of the Group Messaging System. It is hoped that by the end of the report, the reader will have a clear understanding of the concept of the Group Messaging System, how it was designed, why certain trade-offs were made, and what future possibilities might exist for its use in everyday life.

II Acknowledgements

The team would like to thank Professor G. Wainer for the time and effort he devoted to mentoring the team throughout the project's lifecycle, and for his continuing support and mentorship as the team competes with students from around the world in the CSIDC 2003 competition.

III Table of Contents

IV List of Figures

V List of Tables

1 Introduction

This fourth-year project report, submitted by Shawn French, Matthew Chmiel, and Stephen Haber, details the design and implementation of the Group Messaging System (GMS). In addition to being the focus of our fourth-year project, GMS was also entered in this year's Computer Society International Design Competition (CSIDC). This document is necessary for the successful completion of the Computer Systems Engineering course SYSC 4907 at Carleton University.

1.1 Background

The annual Computer Society International Design Competition was started in the year 2000 by the IEEE in collaboration with industry professionals, and with the sponsorship of leading organizations. The aim of the CSIDC is to advance excellence in the field of computer science and computer engineering through friendly competition between teams of university students around the world. In general, the goal of the competition is to encourage the design of computer-based solutions to real-world problems that are of benefit to society. In the pursuit of this goal, students will learn about teamwork, product design, and implementation in a creative, and engineering-oriented environment. Past competitions have involved the design of healthcare systems and systems using Bluetooth wireless technology. [1]

Competing teams are expected to submit both an interim and a final report. Based on an evaluation of the final report by CSIDC judges, the top ten teams will be selected as finalists to appear in Washington D.C., in May 2003 to demonstrate their product. The teams will be graded according to a set of criteria available on the CSIDC website, and prizes will be awarded to the top teams.

1.2 Problem Motivation

For the 2003 competition, CSIDC officials have put forward the following project definition: "Added Value: Turning Computers into Systems." Each competing team must "take a PC, laptop, or hand-held computer and turn it into something new by adding an external interface and the appropriate software. The end system should be socially valuable by solving a real-world problem." [1]

It is the goal of this team to create a product that meets this project definition in an exciting way. In particular, this team has emphasized the importance of software engineering principles and multimedia design that will make the final product competitive for both Microsoft awards. To meet this goal the team had to select a project that would be of benefit to society; the concept had to be unique; and the technological requirements could not exceed the scope of the CSIDC.

1.3 Problem Statement

A great deal of time was taken early in the school year selecting an appropriate problem to tackle. It was decided that communication within the home or office could greatly benefit from an easy-to-use computer-based messaging system. In a busy home or office environment there has always been a need to leave brief messages or instructions for the people around you. Although some messages are trivial, many carry vital information that if not read could cause serious problems. To ensure that this type of communication is successful, a system is needed that can provide accurate and understandable messages. These messages must also be accessible to all users and organized in such a way as to allow easy retrieval. Currently, these requirements are not being met.

In most homes, messages are written down on notepads or sticky notes. Unfortunately, quick messages are often written poorly which can lead to misunderstandings. Also, these messages can easily be overlooked or misplaced. These factors greatly reduce the ability of one family member to accurately leave a message for another family member.

1.4 Proposed Solution

It is the purpose of the Group Messaging System to allow better communication between members of a household or an office environment. Since this system is designed to replace messaging with pen and paper, it should be just as easy to use, if not easier.

With these basic requirements in mind, namely, being more accurate, understandable and easier to use than pen and paper, a video/audio-based messaging system was designed. Since messages are based on video and audio capture, they convey the same meaning as if the person was standing right in front of you. This eliminates many of the inaccuracies and misunderstandings caused by scribbling short messages on paper. Furthermore, talking is something just as easy as, or easier for most people than, writing with pen and paper.

If sold as a small consumer electronic device, GMS could be placed at some central location in the house or office. It would appear no larger than a medium-sized picture frame containing an LCD, a small video camera, a fingerprint reader, a single-board computer, and some controlling buttons, all in one small, streamlined form factor. However, the Computer Society's International Design Competition rules state that competing projects must add value to an existing PC. To comply with these rules, the prototype was designed so that the GMS software and peripherals could be bundled and sold together for use on an existing PC. This report will focus on the concept of GMS as a device, whereas the final report for the competition will focus on GMS as software and a set of peripherals. The push towards convergence of appliance and electronic devices, such as integrating a touch screen LCD in a fridge by

LG electronics, seems to indicate a future market for GMS to be used as a standalone or integrated device.

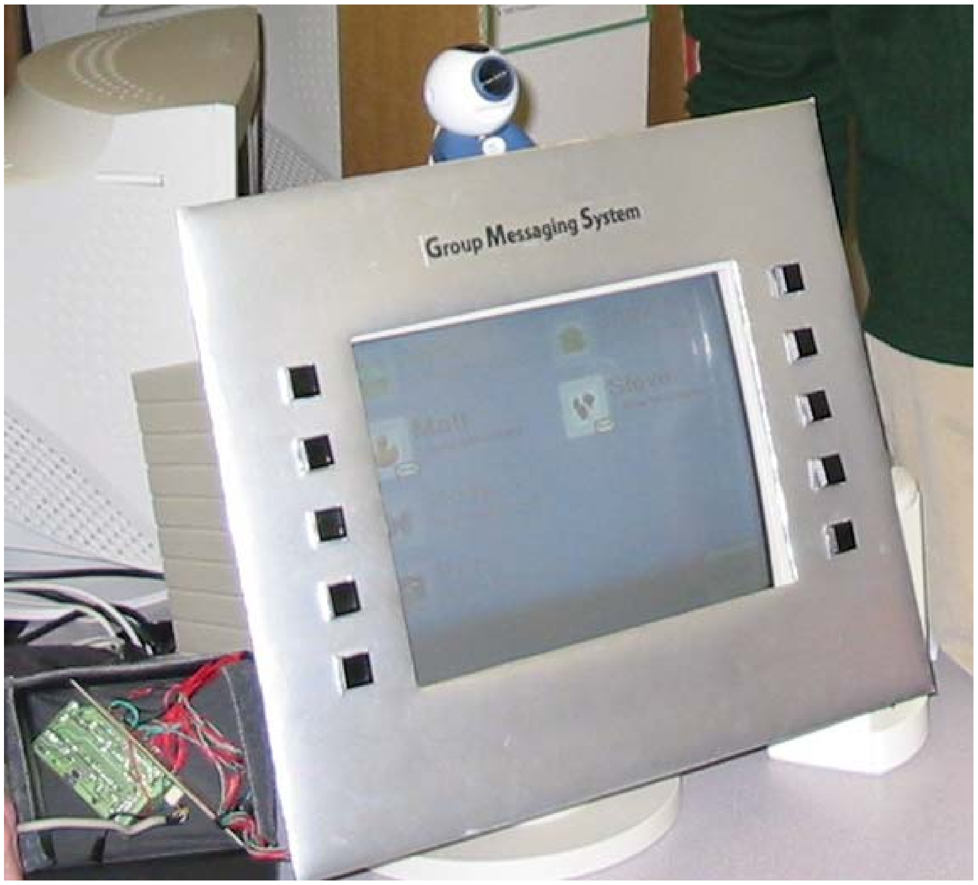


Figure 1 – Photograph of the Group Messaging System Prototype taken on March 14, 2003

The prototype Group Messaging System (shown in Figure 1) runs as an application on a standard Microsoft Windows-based PC using a Logitech web camera with built-in microphone. User input is handled by ten buttons, five running down each side of the monitor, much like an ATM banking machine. Also like an ATM, navigation is handled by dynamically associating buttons on the screen with the physical buttons the users can press. This allows for a custom set of buttons for each screen, rather than only ten dedicated buttons for the entire application. Although a fingerprint reader is part of the conceptual GMS design, this feature was left out of the prototype due to budget and time constraints.

In terms of real world use, GMS functions much like a simplified email application; however, text emails are replaced by video or audio messages and the device is not connected to a network. Since this is a standalone device, all messages are stored internally. Like an email client, GMS displays all messages in a list, showing who sent what message and at what time. From this list the user has the ability to view any message. Once a message has been selected for playback, the user can control standard video features like play, pause, fast forward, and rewind. Users also have the ability to reply to the message. Composing new messages involves selecting one or more users of the system to send the message to, followed by a choice to send a video or audio message. When the type of message is selected, a quick countdown ensues, followed by a live recording window. Users can stop recording at any time, review their message and re-record if necessary before sending it. Individuals using the system for the first time can add themselves by using a simple wizard that requires them to enter their name and select an icon. Throughout the entire system, it was extremely important to present clear, concise and relevant information to the user and to ensure that all tasks would be accomplished with a minimum number of button presses.

By allowing users to access GMS in an easy and intuitive way, we hope to bring home all the power of modern computer video and audio messaging in a device that people can use on a daily basis.

1.5 Accomplishments

The software engineering process followed (as recommended in the Software Engineering course SYSC 4800) called for the creation of a requirements analysis document (RAD), and a system design document. Both of these documents helped focus our design efforts and allowed us to clearly define system requirements and how our system would fulfill these requirements. The RAD presents a high level view of the GMS and provides use cases, sequence diagrams and class diagrams that describe the structure of the system. The system design document provides a more in depth look at how the system fits together at the implementation level. When both of these documents were complete the GMS prototype was implemented using Visual C++ and a Microsoft Access database. In addition to creating software, the team designed a physical user interface using an old keyboard and soldering key connections to ten new buttons placed in a cardboard frame, five buttons to each side of the screen. Once the prototype was completed, the system was tested. This included a systematical approach to testing each feature of the GMS. Also, usability tests were conducted at the fourth-year project poster fair using the actual GMS prototype. These surveys tested the user interface and provided valuable input on what users found easy to use, hard to use and what they would change.

The CD included with this report in Appendix G contains a soft copy of each appendix, as well as a copy of all the software needed to install and run GMS on a PC. The CD also includes a README file that explains how to install the software.

The following table details our team’s specific accomplishments and what each member was responsible for.

Accomplishment	Description	Shawn	Matthew	Stephen
Requirements Analysis Document (RAD)	-requirements elicitation -requirements analysis -user interface design	X	X	X
User Survey	-potential user's reactions to GMS and suggestions	X	X	
System Design Document	-design goals -subsystem decomposition	X		
GUI Design	-screen requirements -graphic prototyping	X	X	X
Boundary Class Implementation	-user interface subsystem implementation	X	X	
Control Class Implementation	-function subsystem implementation	X		
Entity Class Implementation	-model subsystem implementation	X		X
Keyboard User Interface Mockup	-keyboard controller soldered to ten buttons and placed in a cardboard template	X		
Media Player Implementation	-playback and manipulation of video using Microsoft Libraries		X	
Logitech Web Camera Implementation	-recording and playback of live video with audio using Logitech SDK			X
Usability Testing	-test GUI and program flow as well as user reaction to prototype	X		
GMS Software CD	-documents and software on CD -README file to help with installation			X
Final Report	1 INTRODUCTION			X
	2 DESIGN	X		
	3 IMPLEMENTATION	X	X	X
	3.1 HARDWARE IMPLEMENTATION CONSIDERATIONS	X	X	X
	3.2 SOFTWARE IMPLEMENTATION CONSIDERATIONS	X	X	X
	3.2.1 Function (Control Classes)	X		
	3.2.2 User Interface (Boundary Classes)		X	
	3.2.3 Model (Entity Classes)			X
	4 TESTING	X	X	
	4.1 CODE TESTING		X	
	4.2 USABILITY TESTING	X		
	5 CONCLUSION		X	X
	5.1 ACCOMPLISHMENTS			X
	5.2 COMPARISON TO CURRENT SYSTEMS 5.3 NEXT STEPS IN DEVELOPMENT (IDEAS FOR THE FUTURE)		X	
	5.4 WRAP-UP			X

Table 1 – Table of Accomplishments

1.6 Overview of Remainder of Report

The remainder of the report is divided into the following sections: Design, Implementation, Testing and Conclusion. The design section concentrates on our approach to designing the GMS software with discussions on requirements, use cases, state behavior, dynamic models and user interface design. The system design section discusses our design goals and the responsibilities of each GMS subsystem. Following this, the implementation section is an account of the issues, choices and tradeoffs that we faced while implementing the GMS prototype. The testing section discusses our testing procedures for both the GMS prototype and the user interface design. The conclusion will summarize our team's accomplishments and discuss such issues as future ideas for the GMS and potential marketing.

2 Design

2.1 Approach

Software engineering is defined in the IEEE Standard Glossary of Software Engineering Terminology [2] as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1).

As can be discerned from the above definition, success in the design and implementation of software often requires a systematic process. Because of our involvement in the IEEE Computer Society International Design Competition, we wanted to follow such a process since the competition's aim is to have teams "go through an entire 'product design cycle'" as well as have all participants "combine design skills with software engineering skills." [1] A systematic process would help the team go from the requirements to the finished product in an organized and efficient manner, as well as help create the appropriate deliverables along the way.

The IEEE 1074 Standard for Software Lifecycle Processes is one such process, and was recommended in the 4th year Software Engineering course SYSC 4800. The standard is very complete with respect to all of the processes, sub-processes, and activities. For example, Development is a process with three sub-processes: Requirements, Design, and Implementation; Create Source is one of the activities in the Implementation sub-process.

However, as complete as the standard is with respect to the lifecycle of software projects, the standard doesn't specify how the processes can be best managed or documented. To help create the appropriate design documents, the group used the textbook from the 4th year Software Engineering course (SYSC-4800), "Object-Oriented Software Engineering – Conquering Complex and Changing Systems" [3] by Bernd Bruegge and Allen H. Dutoit.

The authors approach is to use the Unified Modeling Language (UML) to model the system throughout the design process. Below is an illustration of the textbook's software lifecycle [9]:

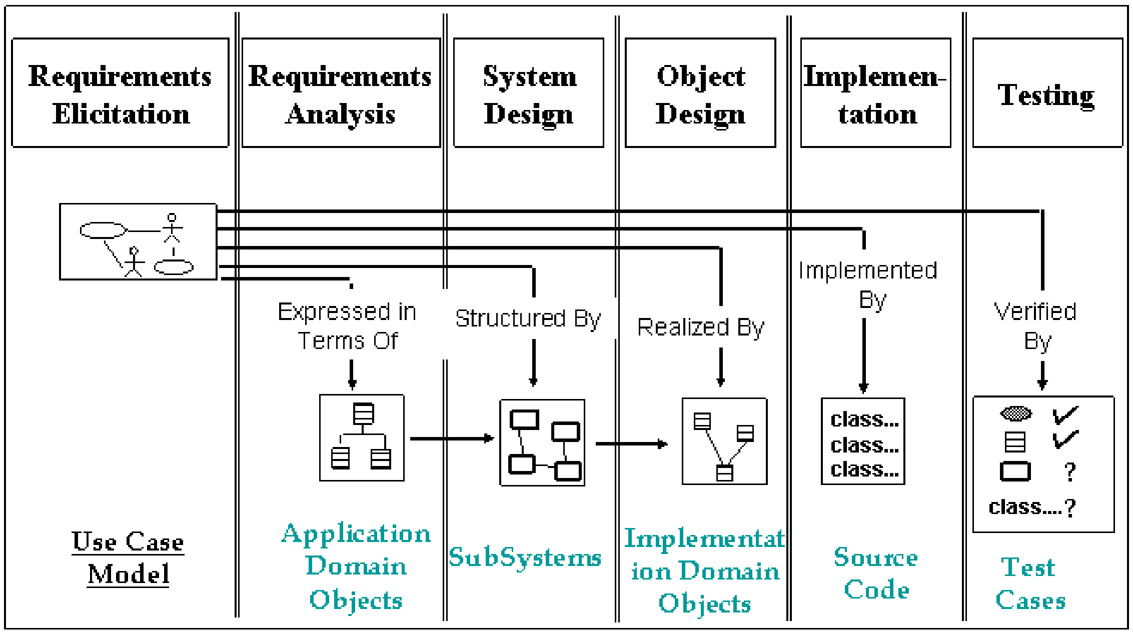


Figure 2 - Software Lifecycle Model Followed for the GMS Project

The remainder of the Design section will detail the specific outputs from Requirements Elicitation to Object Design as we followed the process described in the textbook. Below is a summary of the steps involved in the process, as well as a mapping of design documents to appendices:

Step	Deliverable
Requirements Elicitation	Requirements Analysis Document (sections 3.2, 3.3, 3.4, 3.51, 3.52 in Appendix A)
Requirements Analysis	Requirements Analysis Document (sections 3.53, 3.54, 3.55 in Appendix A)
System Design	System Design Document (Appendix D)
Object Design	Application Programming Interface (Appendix G as "/Documents/Object_Design.doc")
Implementation	Source Code (Appendix G as "/Software/GMS_Source/")
Testing	Usability Testing (Appendix E, with results in Appendix F)

Table 2 - Details of the Output at Each Step in the Software Engineering Process

2.2 Requirements Elicitation

The first step of the lifecycle is very important as it defines the system from the customer's perspective. The problem statement is first described in English so that it can be read and understood by the customer, and is then modeled more formally into UML. The formal version can be used to communicate the problem to developers without any of the uncertainties that a spoken language can sometimes introduce.

2.2.1 System Requirements

Since we did not have a customer who had hired us to undertake this project, eliciting requirements could not be done in the standard fashion through face-to-face meetings with the customer. Instead, as the idea for GMS came from trying to improve communication within our own homes, it was only natural to model the system using ourselves, our families, our friends, and our roommates as the main customers.

Many brainstorming sessions helped narrow down the scope of the system from an all encompassing organizer complete with calendar, address book, and appointment manager, to a more specialized video messaging system. Kim Vicente, a well known Canadian cognitive engineer and professor at the University of Toronto recently made comments that mimicked exactly the group's thoughts at the time. Professor Vicente told an interviewer that "people don't want 500 features, they want about four." He went on to point out that "if you focus on what people really need, you'll find that products will be easier to use and be much more successful." [4] In other words, if users are given too many options and functions, they may feel it is far too complicated and not bother learning to use it (or buy it in the first place). This is very important in a device targeted to the average North American household since users of all ages and backgrounds should be able to use GMS easily without any formal training.

The following sections will describe the system's functional and non-functional requirements. A full discussion can be found in sections 3.2 and 3.3 of our Requirements Analysis Document (RAD) in Appendix A.

2.2.1.1 Functional Requirements

Functional requirements describe the requirements of the system from the user's perspective. From adding new users, to logging in and out, to composing messages, many of the requirements for the system were found within common messaging applications. Using email as a model, it was decided that the ability to save messages in a more permanent fashion would be helpful to the user. Users of the system might want to separate "current" messages (in the inbox) from messages that they would like to keep for long periods of time. If users knew the message was special when they first viewed it, they could save it then and not worry about accidentally deleting it from their inbox at a later date.

The ability to save a message permanently drew the group into a long discussion about the disk requirements for the device. If the GMS device were to be sold to the public, some sort of disk-space management would be needed to ensure that users shared the space in a fair and consistent manner. An agreement was reached that called for a fixed allocation of message space to each user. Although there would be some unused disk-space, especially in the case of only a few active users, the extra development time associated with devising an ultra-efficient management policy was beyond the scope of the project.

The possibility that some people would not like the idea of being recorded on video arose after talking with friends and family about GMS. The solution was to give users the ability to record audio-only messages as well as video messages.

Another piece of functionality considered is the fact that most people are used to typing a subject in email messages they compose. However, as mentioned previously, the design called for a 10-button interface, so it was assumed that users would find it too long to compose a subject, and would not even use the feature. We tested this assumption, and many others with a user survey as described in the next section.

The GMS functional requirements are summarized below in Table 3.

Functional Requirements
Setup New User Account: add new users to the system
Default Screen: a screen where user may login that shows the # of new messages each user has
User Authentication: identify users and log them into their message inbox
Message Composing: allow users to compose and send video messages to multiple recipients
Message Storage: allow users to store received messages to view at a later time
Message Viewing: allow users to view new and stored messages that have been received
User Logout: allow users to logout at any point of their GMS session

Table 3 - Summary of Functional Requirements

2.2.1.2 User Survey

To gauge the response of potential customers, find any missed requirements, and determine their preferences on issues such as composing a subject, a survey was conducted. The survey was filled out by 13 subjects with an average of 3.3 people and 2.2 computers per household. Although each respondent used email, the majority of in-house messages were communicated using handwritten notes or voicemail. Almost all surveyed admitted to forgetting to pass on a message and all surveyed thought that a video messaging system would ensure messages were communicated in a more reliable fashion.

The survey results helped us decide on some functional requirements. For example, 100% of the respondents said they'd like to have the ability to compose audio only messages as well as video (with audio) messages. After being shown a sketch of the intended 10 button interface, only 6 of 13 surveyed said that they would take the time to compose subjects for their messages. Also, 85% (11 of 13) said they would like the ability to save messages in a separate area from the inbox. All of the survey questions can be found in Appendix B, and a summary of the results can be found in Appendix C.

2.2.1.3 Non-Functional Requirements

The survey helped us determine some non-functional requirements, such as hardware considerations (what were users prepared to pay for), and even the physical environment that people would subject the system to (the kitchen environment might entail a waterproof system). We've paid particular attention to the user interface and human factors, because we wanted to ensure even children could use the system. Also, because GMS stores personal messages onto a hard disk, security was considered. In the real device, some sort of encryption would be used to ensure the user's privacy. However, such a feature was not a main concern in the prototype as it could be implemented later on using external libraries. For more functional requirements, please see section 3.3 of our RAD in Appendix A.

2.2.2 Actors

The external entities were modeled as "Actors" in UML, and Table 4 shows all of the actors involved in the Group Messaging System.

Actor	Description
User	The person interacting with GMS
Database	The system responsible for persistent data storage
Camera	Device used to capture video
Microphone	Device used to capture audio
Speaker	Device used to play audio from video and audio messages
Fingerprint Reader	Hardware peripheral used to read users' fingerprint for authentication purposes

Table 4 - The External Entities (Actors) Involved in the Group Messaging System

2.2.3 Scenarios

All of the scenarios that might arise in the operation of the system were identified. Each scenario is "a concrete informal description of a single feature of the system from the viewpoint of a single actor." (p.108) [3] Scenarios were identified by stepping through all of the functional requirements from the perspective of the user and documenting the system's reaction. The complete list of scenarios (see sections 3.5.1 of our RAD in Appendix A) was then formalized into UML use cases.

2.2.4 Use Cases

A scenario (not UML) is an instance of a particular use case (UML). Every piece of functionality of the system and its possible deviations are described in one use case description. Since use cases are much more formal then scenarios, particular attention was paid to describing them using application domain terms. For example, Actor names are used instead of an informal description of roles. Also, an effort was made to use the same nouns (such as Message) across all use case descriptions. This would help later on when trying to identify application objects that will be implemented using classes.

A sample use case is given in Table 5 that describes the case of a user composing a message. All of the GMS use cases are documented in section 3.5.2 of our RAD in Appendix A.

Use Case	**ComposeMessage**
Participating Actors	`User, Display, Buttons`
Entry Condition	1. The `User` has selected the "Compose Message" Button.
Flow of Events	2. The `User` selects recipients using the `SelectRecipients` use case. 3. The `User` selects the message format (audio or video) by executing the `SelectMessageFormat` use case. 4. The `User` records the message using the `RecordMessage` use case. 5. `User` reviews message and selects whether to send it or not.
Exit Condition	6. `Display` shows confirmation that message was sent.
Special Requirements	The `User` can cancel this process at any time by pressing the "Cancel" button using `Buttons`.

Table 5 - The Compose Message Use Case Description

2.2.5 System Use Case Diagram

The final activity in the requirements elicitation process involves taking all of the use case descriptions and explicitly showing their relationships among one another as well as with Actors. The result is a system use case diagram shown below in Figure 3:

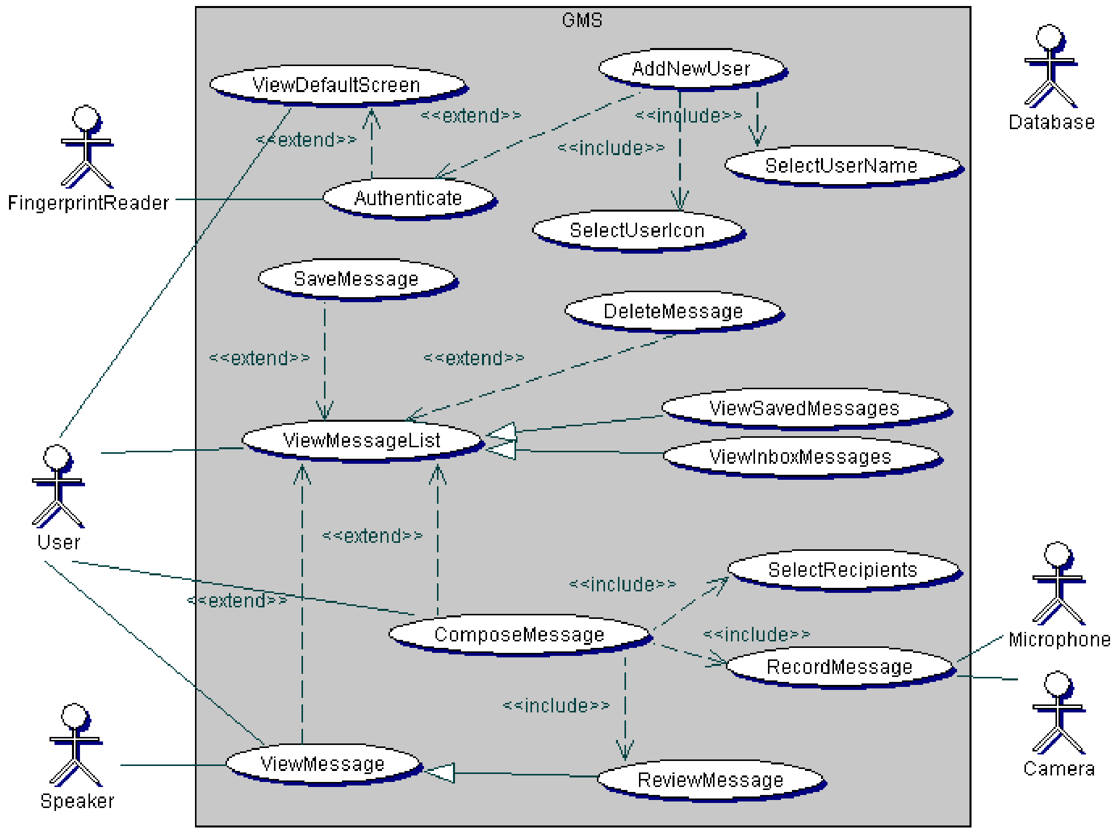


Figure 3 - GMS Use Case Diagram

2.3 Requirements Analysis

The following is a discussion on how and why the objects in GMS were classified into entity, boundary, and control stereotypes. The dynamic models of the system's message passing and state behavior is explored in section 2.3.2.

2.3.1 Object Model

To describe the application in UML, nouns and verbs from the use cases were mapped into objects and messages for the object model. Using these objects and messages, a class diagram was created to model the system's main classes. Those classes were then stereotyped (using UML stereotypes) into roles of Entity, Boundary, or Control.

Having these types of objects leads to models that are more resilient to change. [9] Once all objects were stereotyped, it would be easy to ensure they followed certain rules to ensure minimal impact when major changes must be made. For example, if the production version of the application were to be implemented on a different platform than the development version, the boundary objects would be greatly affected as they deal with low-level OS interfacing. However, as long as the control and entity objects don't use the boundary interface, they could withstand such a change with little to no modification.

2.3.1.1 Control Objects

A control object takes information from the boundaries (that was taken from an external actor) and reacts to it appropriately by creating new boundaries and/or persisting information into entity objects. The Group Messaging System consists of one main state-full object called a GMS_Frame. The "_Frame" is part of the name because the class inherits from a CFrameWnd, which is a Microsoft Foundation Class (MFC) for creating a frame (window) within an application. This control class was modeled using an extended state pattern as described by Alan Shalloway in his design pattern matrix document. [5] The illustration below, taken from Shalloway's matrix, describes the extended state design pattern using UML.

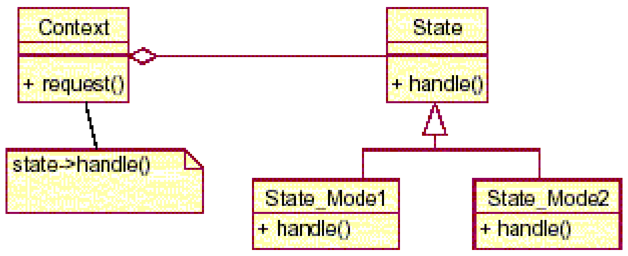


Figure 4 - Extended State Pattern

The main idea for the state pattern is that instead of having the state-full object keep track of what state it's in (i.e., using switch statements everywhere) it simply has a state object that it calls to react to any input. This pattern was used because the GMS application will always be in one state or another, and depending on what state it is in, it reacts to the user input differently. Therefore, the class diagram of the control objects consists of this main "context" object (GMS_Frame), and the many implementations of the generalized state object. The Default_State is an example of an object that inherits from the generalized GMS_State object.

The GMS model called for three "complex" states that were state full themselves. The GMS_Complex_State is really the "context" object of a state pattern that has sub-states (inheriting from GMS_Sub_State) to handle events. One can think of these classes as being a state pattern implementation within the GMS_Frame state pattern. This helped model "wizard-like" states such as ComposeMessage_State which had 3 sub-states: SelectRecipients_State, RecordMessage_State, and ViewComposedMessage_State. See Figure 5 (below) for a class diagram of our control objects.

This model mapped a single boundary object (a GMS_View) to a single control object (a GMS_State). In other words, every state has a view to display data to the user, and every view has access to its state to retrieve the needed data for display. When GMS_Frame tells its current state to paint itself, the state then tells its view to do the painting. Composite states similarly pass the paint events to their current state (as with any other type of state-dependent message), and then the sub-state tells its view to paint itself.

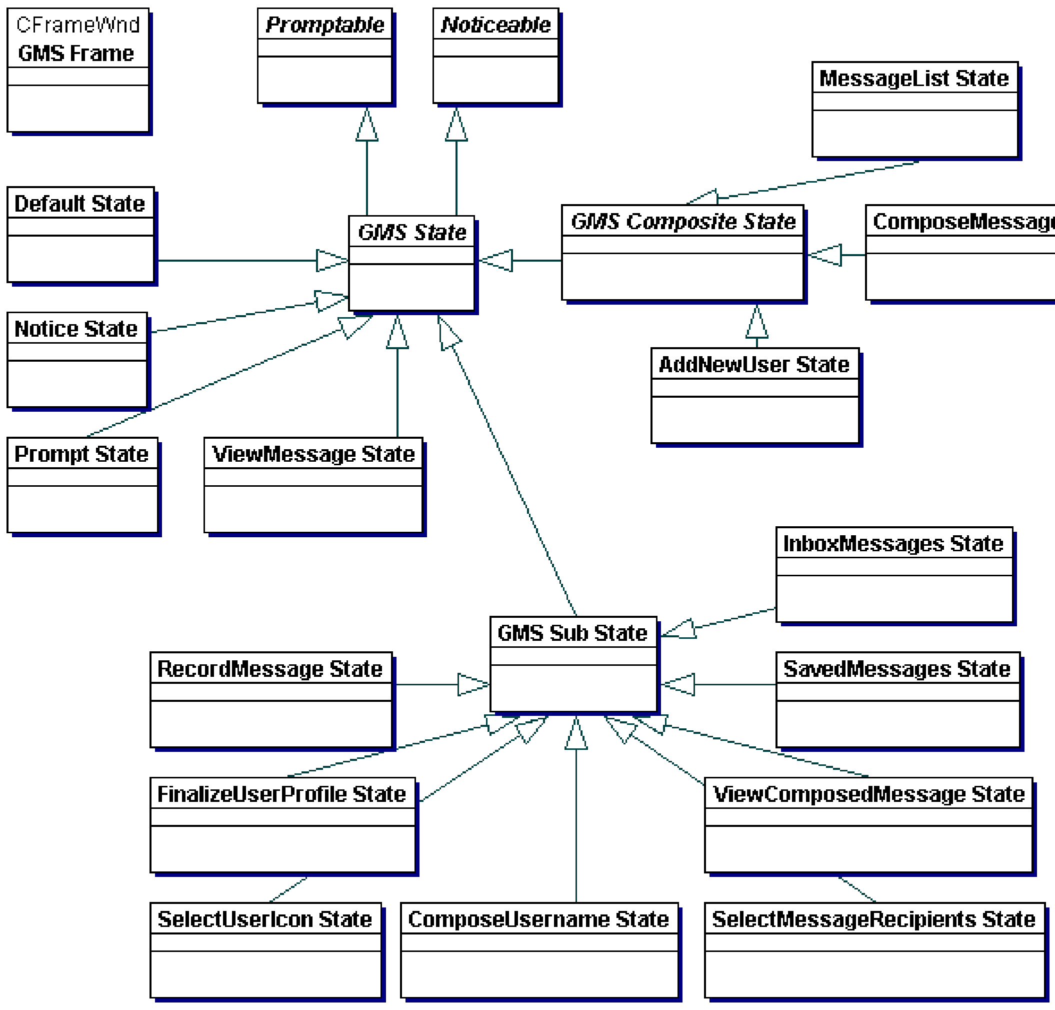


Figure 5 - Control Class Diagram

2.3.1.2 Boundary Objects

Boundary objects are those objects that communicate directly with the Actors of the system. Often they use entity data to display information to the user, and are created and controlled by controller objects. For GMS, only the objects that displayed information to the user, or captured information from the user were stereotyped as system boundaries. Therefore every class responsible for showing a screen (or a "view") to the user was modeled as Boundary objects inheriting from the GMS_View class. See Figure 6 (below) for a class diagram of our boundary objects.

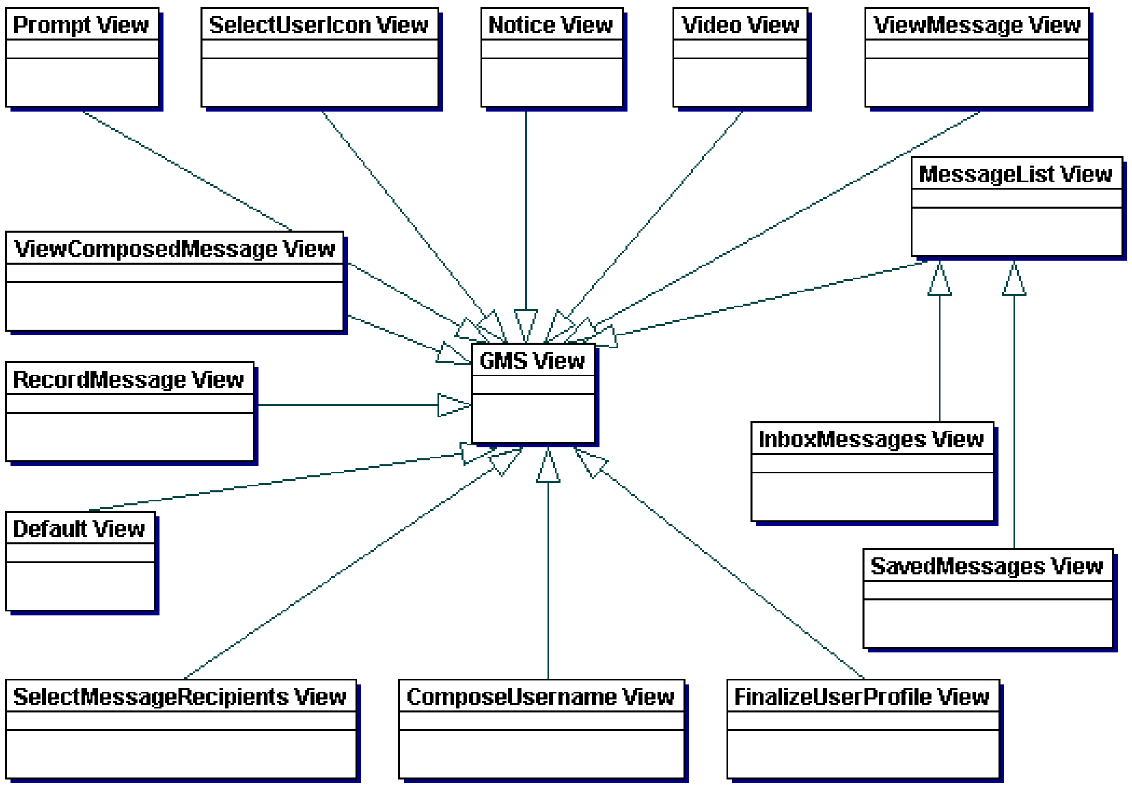


Figure 6 - Boundary Class Diagram

2.3.1.3 Entity Objects

An entity object represents the persistent data of the system. GMS requires persistent records of the user and the messages that they send and receive. The User, Message, and ReceivedMessage objects were created to handle such data storage. Apart from being able to create and store new information, each entity in GMS would also need to retrieve and modify records from the database. More detail on their usage will be provided in the discussion about implementation in section 3.2.3. The figure below shows the basic class diagram of the GMS entity objects.

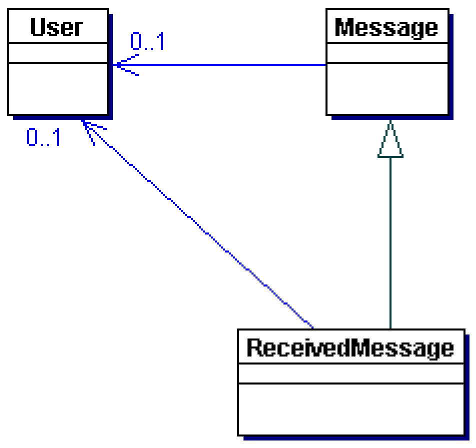


Figure 7 - Entity Class Diagram

2.3.2 Dynamic Model

Another part of analysis is the identification and modeling of interactions between objects. Also, at this point in the development process it's necessary to formalize and explain the non-trivial behavior of state-full objects, namely GMS_Frame. Both of these steps help in identifying the needed attributes for objects, as well as associations and dependencies between objects.

Since the more complicated sequences of events occur when changing states, these transitions involve putting the GMS_Frame object into a new state, and in the case of a prompt, a pseudo-state. The prompt is a temporary state, and reports information back to the invoking state that asked for the prompt. The sequence diagram of entering GMS_Frame into a prompt state is shown below. It shows the pseudo-state change (the current state of GMS_Frame is not affected, but the display and user input handling are done by the prompt state) as well as the prompt handling user input.

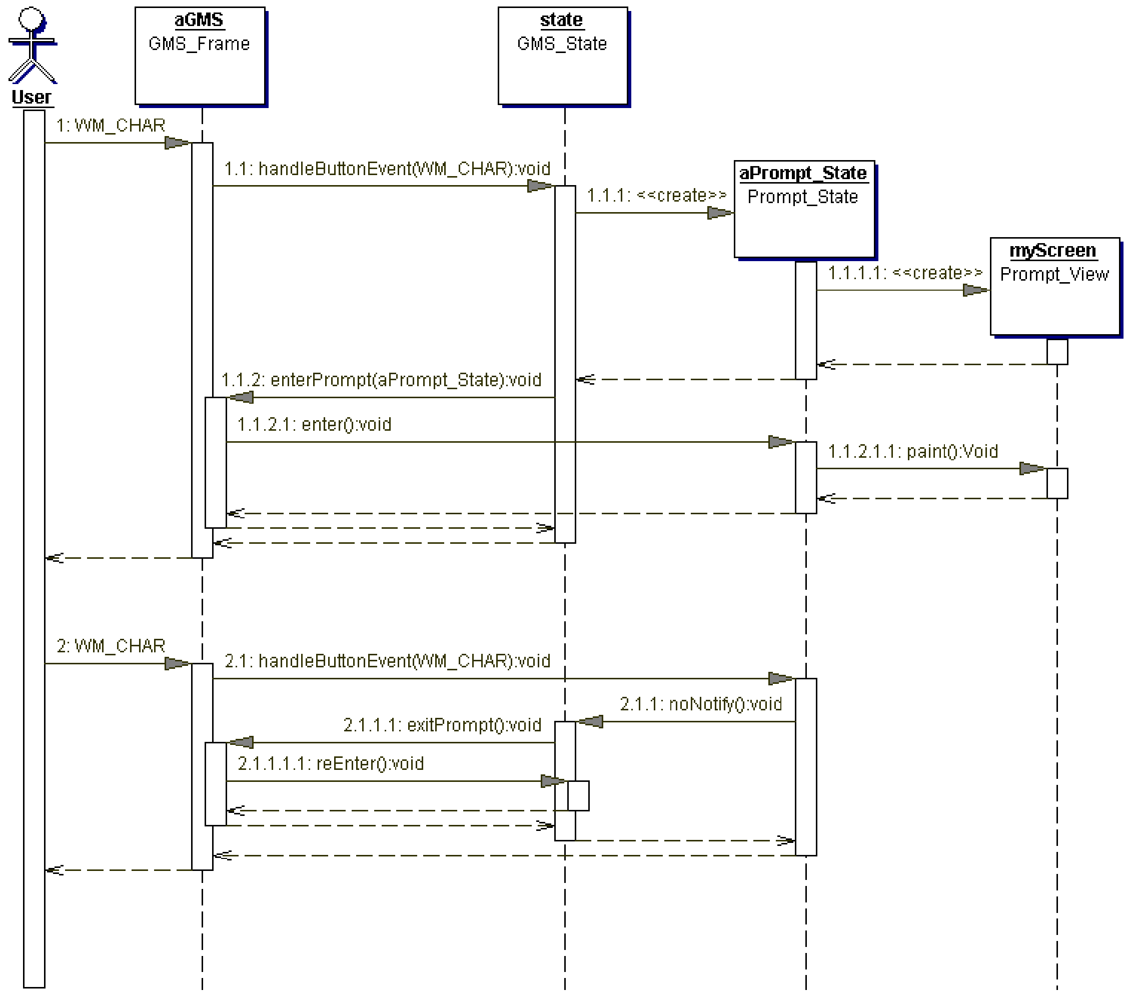


Figure 8 - Sequence Diagram for Entering and Exiting a Prompt_State

The user starts the sequence by pressing a button (triggering a “WM_CHAR” event) that corresponds to some action that requires the current state to use a prompt. The state then creates the new Prompt_State object and tells GMS_Frame to enter that prompt. Once the user responds by pressing a button, the prompt notifies the state of the “Yes” or “No” answer. The state then tells GMS_Frame to exit the prompt and so GMS_Frame will send all future events to the state (as before).

See Figure 22 (on page 44) for the statechart of a GMS_Frame object. It is a UML model of all the different simple states, composite states, and sub states the GMS_Frame can be in at any one time. A detailed explanation of GMS_Frame’s states can be found in section 3.2.1. Please see section 3.5.4 of our RAD in Appendix A for the rest of the system’s dynamic models.

2.4 User Interface Design

The last activity in analysis was modeling the Graphical User Interface (GUI). The system's functional requirements, use cases, class diagrams, and dynamic models were used to determine what screens would need to be modeled and later implemented. It was a fairly straightforward activity as only one screen per GMS_Frame state needed to be modeled. This really amounts to one screen per Display boundary (GMS_View implementations).

Each screen was first drawn by hand, and then a digital version was created using the graphical software Adobe Illustrator 10. Although the software created the graphics based on vectors, the graphics could easily be exported as bitmaps to be later used in the GMS display algorithms. Without the exported bitmaps, all graphical screen components would have to be drawn using only basic shapes. To ensure the GUI had a high degree of usability, and that it provided a pleasant user experience, we constantly requested feedback from friends and family on the visual components and overall look (such as colors and shapes) of the design.

2.4.1 Device Mock-ups

As mentioned in section 1.4, the GMS device would have only 10 buttons for user input. Although many concepts were explored in rough (see Figure 9), a graphical representation of the device was created and is shown in Figure 10.

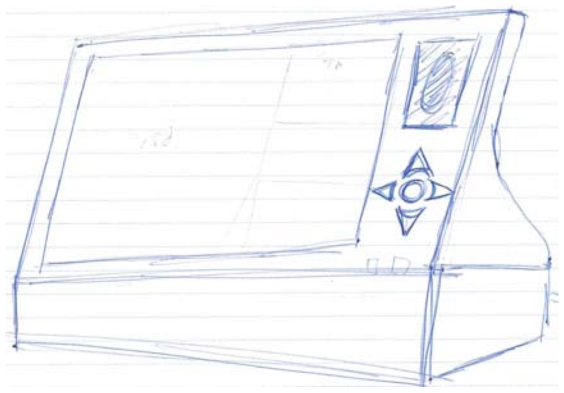


Figure 9 – Original GMS Device Concept

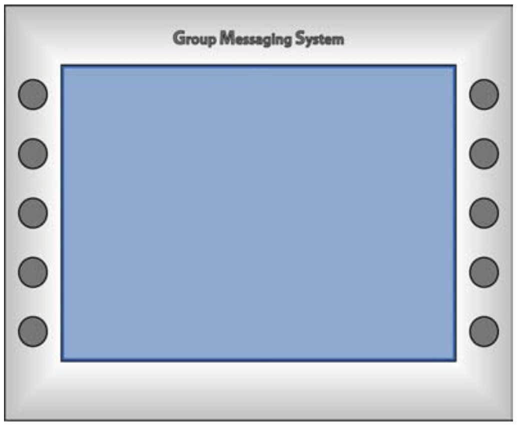


Figure 10 - GMS Frame with Buttons

A graphic of the device concept was created that included a camera, a fingerprint reader (shown in red), a microphone, speakers, 10 buttons (5 on each side), and the display. The graphic was later modified (see Figure 11 below) to include the mock-up of the inbox screen.

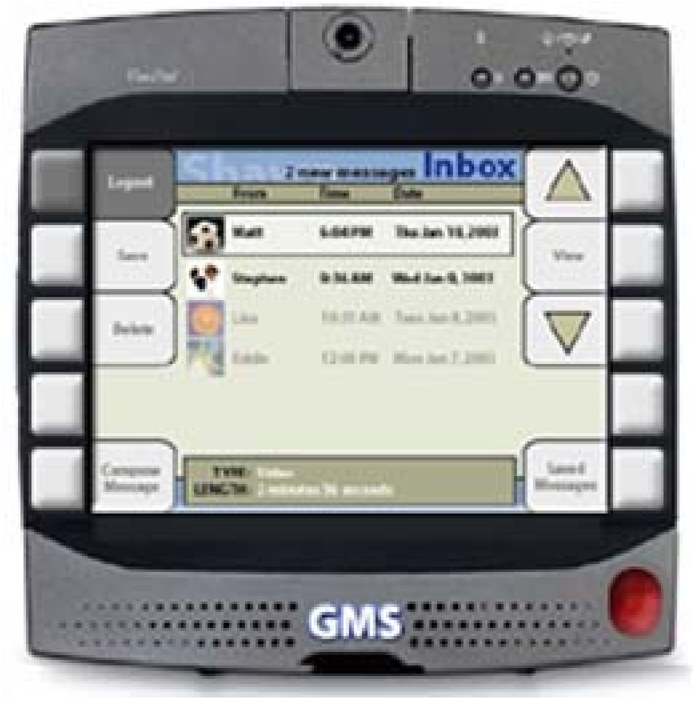


Figure 11 - GMS Device Mock-Up

2.4.2 Screen Mock-ups

Before beginning to draw each screen, the group decided on a standard look and feel to follow when drawing screens. Once colors and a common button format were picked, and the placement of common components (such as the screen title, and the current user's name) was standardized, the group digitized the rough sketches using Illustrator.

From the time that the screens were first sketched out in rough, the notion of an easy-to-use interface was at the forefront of the group's graphical endeavors. As mentioned before in section 2.2.1.3, one of the non-functional requirements states that the system should "be designed as an easy-to-use device that can be used by anyone with basic computer skills." This requirement is arguably the most important when considering selling a system to the general public. A product with an unusable or ugly interface will fail in the marketplace as consumers would rather purchase a more usable and "visually pleasing" product. The standardization of components and color scheme are examples of some of the measures taken to ensure a usable product.

Also, a set of usability goals was created to help guide the design process. The system should be easy for all types of users to learn without any formal training or documentation. Also, users should be able to complete their tasks (i.e., compose a message) in as little time and with as few steps as possible. The speed factor is important since our product aims to replace existing systems such as pen and paper. Finally, a very important feature of the interface is that it would not allow users to perform tasks that caused permanent changes to the system without first being prompted to do so. For example, if a user presses a button asking GMS to delete a message, the system should confirm the action before deleting the message.

The three figures below represent the default, inbox messages, and select message recipients screens respectively:

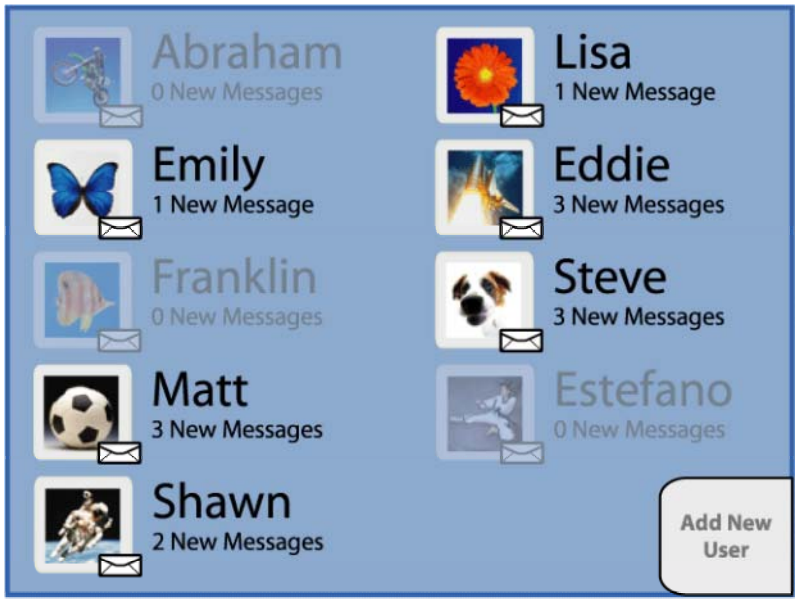
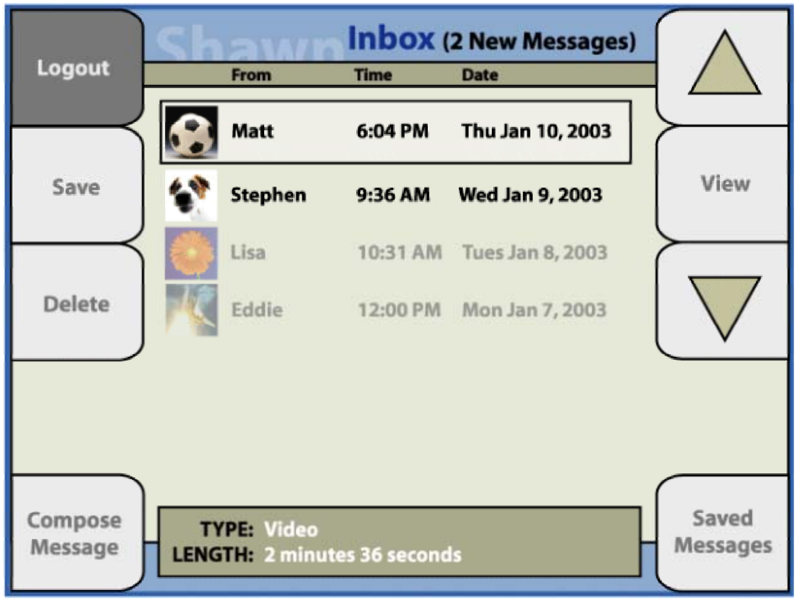


Figure 12 - GMS Default Screen



From	Time	Date
Matt	6:04 PM	Thu Jan 10, 2003
Stephen	9:36 AM	Wed Jan 9, 2003
Lisa	10:31 AM	Tues Jan 8, 2003
Eddie	12:00 PM	Mon Jan 7, 2003

TYPE: Video
LENGTH: 2 minutes 36 seconds

Figure 13 - GMS Inbox Messages Screen

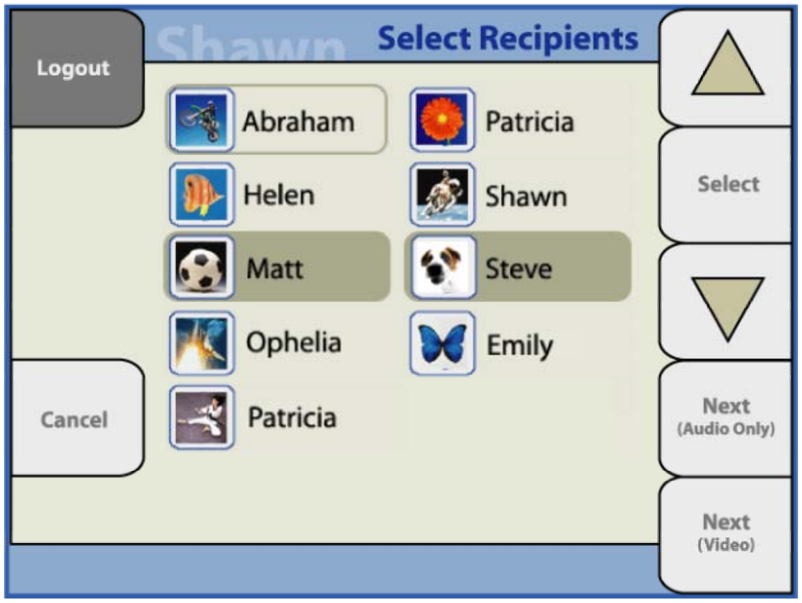


Figure 14 - GMS Select Message Recipients Screen

In total, GMS has 10 different screens (one for each simple state and sub state of GMS_Frame). Along with normal screens, we needed to model prompts and notices. See Figure 15 for a mock-up of the delete message confirmation prompt. Notices are very different from prompts as they are flashed on the screen for a few seconds to show users information about an action that the application has taken on their behalf. See Figure 16 for a mock-up of the message deleted notice. Refer to section 3.5.5 of the RAD document in Appendix A for all of GMS's screen mock-ups.

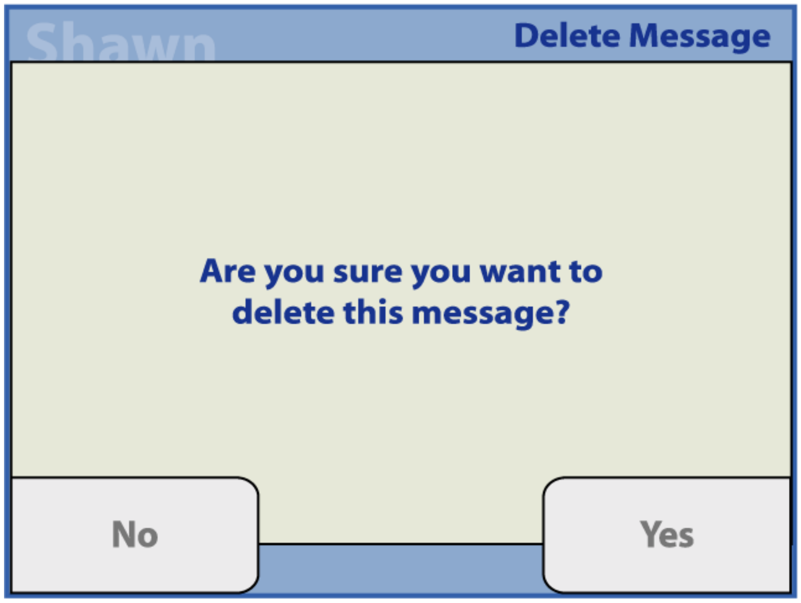


Figure 15 - Delete Message Confirmation Prompt

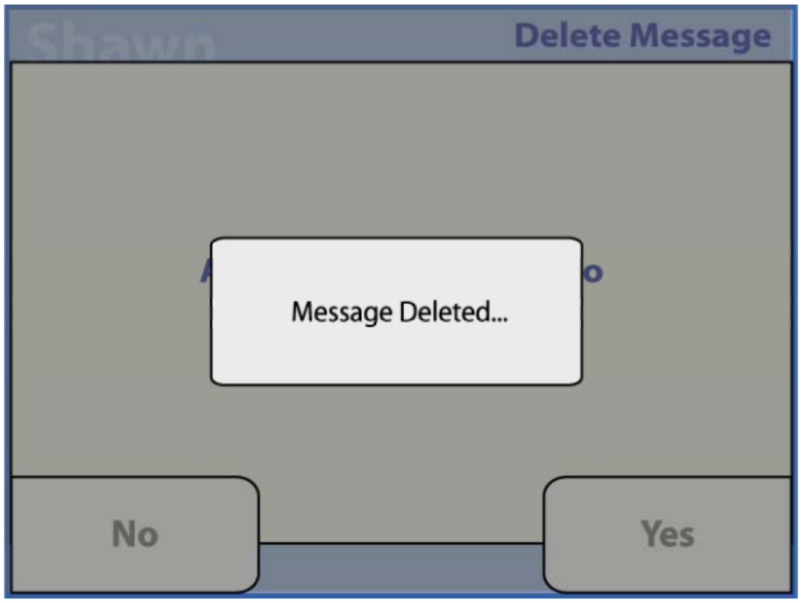


Figure 16 - Message Deleted Notice

2.4.3 Navigational Paths

To clearly show all of the possible routes a user may take while using the application, navigational paths were mapped for each screen. The path a user takes depends on what screen is currently being displayed, and which button the user pushes. Figure 17 shows the paths leading in and out of the View Message screen. The orange arrows coming in and the green arrows going out show transitions into and from the View Message screen respectively. The transitions marked with a "P" indicate that a prompt will be given to the user before executing the transition. It also implies that the user must agree to the prompt (click on the "yes" button) in order to start the transition. For a complete listing of all of GMS's navigational paths, please see section 3.5.5 of the RAD document in Appendix A.

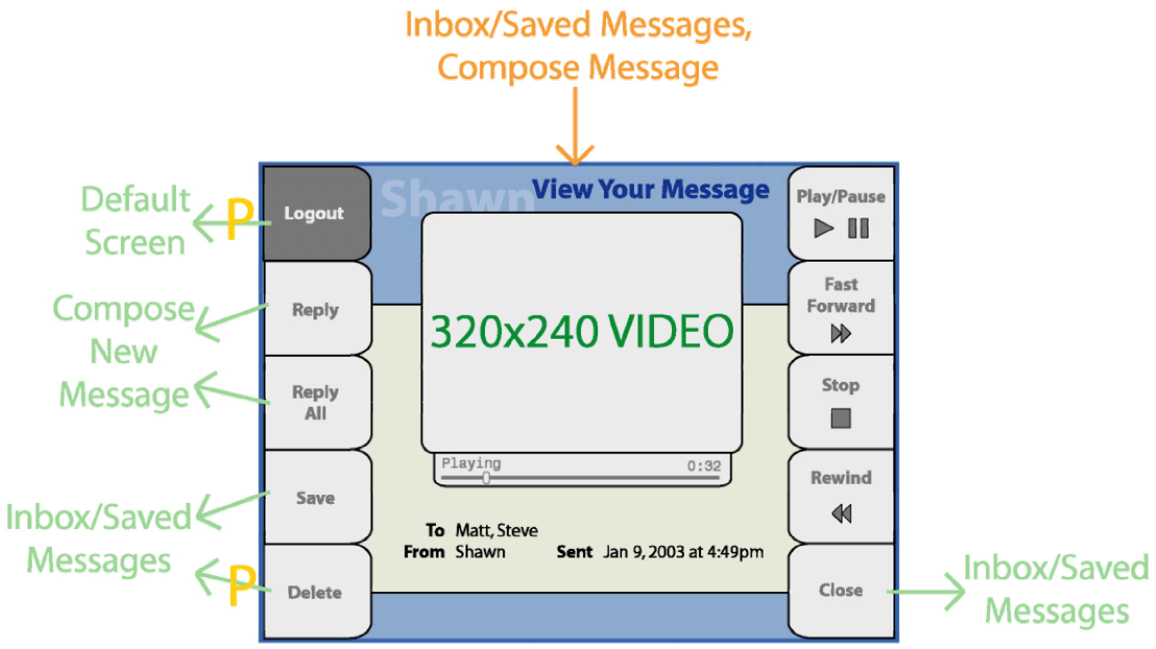


Figure 17 - Navigational Paths To and From the View Message Screen

2.5 System Design

As prescribed by the software engineering process being followed, the next step was System Design. The design of the system was approached with two system implementations in mind: the GMS device that would one day sell in stores, and the GMS prototype that would be built for this project. The main difference between the device and our prototype was the hardware. The device would consist of a small PC with a small form factor (like a tablet PC) running GMS software exclusively, and would contain all of the system's peripherals. Designing such a system would have been above and beyond our capabilities within the school year, and would possibly take our idea out of the scope of CSIDC. However, we knew that we needed to write the software as a proof of concept on a PC. Therefore, we focused our system design on the software rather than on the hardware aspects of the device. The rest of this section will detail some design goals that were established as well as the subsystems that were identified. We will also discuss our strategy towards the management of persistent data, and the handling of certain boundary conditions in software. Please see Appendix D for the complete GMS System Design document.

2.5.1 Design Goals

GMS was designed to be used within a home by the general public. The user should not have to know how to debug the software or recover from faults. As a result, the system must be robust to ensure a long lifetime without any major complications. Also, since consumers demand high quality in their electronic devices, the system must not disappoint them in its performance, presentation, or usability. However, complications do arise and must be dealt with. The following is a summary of the design goals of the Group Messaging System all derived from the Non-Functional requirements listed in section 3.3 of the GMS Requirements Analysis Document (see Appendix A).

2.5.1.1 Performance Criteria

GMS users can measure the performance of the software by observing the response time of the system. Therefore, an important goal of the system design would be to ensure all user input is acknowledged immediately with no observable load times. The foreseeable tasks in which response time may be an issue include user authentication with a fingerprint reader, saving video/audio recordings, and loading video/audio playback. However, these areas of GMS use external drivers and libraries whose implementation and therefore response time are not controllable.

2.5.1.2 Dependability Criteria

GMS is a video messaging system, with no direct consequences from its failure except for the possibility of lost user data. Therefore, the robustness, reliability and fault tolerance of the system will not be addressed fully in the software's design. However, invalid user input should not cause the system to fail, and the system should be reliable to prevent frustration amongst its users. Also, the GMS device would need to comply with all home electronics regulations to ensure that the device is not a safety hazard. Security is a concern that the system's design must address and will be discussed briefly.

The authentication mechanism provides an easy-to-use means of controlling access to each user's received messages, but does not control access to the system's data if GMS software is bypassed. For instance, if data were copied directly from the system's hard drive, the perpetrator could easily view all messages sent unless the data were encrypted. Therefore, a commercial product would need to encrypt the messages before being written to disk; however, the GMS prototype will not implement this feature.

2.5.1.3 Maintenance Criteria

The system was designed to be extendable, modifiable, and portable with little effort. Along with a sound system design, the source code must be heavily commented with explanations of why and how the system was implemented the way it was. Apart from in-line comments, Sun's Javadoc code convention was followed. “Javadoc is the tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code.” [6] Following this convention will allow an Application Programming Interface (API) to be created by software such as Doxygen. The following figure shows the template used to document classes, their methods, and their attributes:

```
/**
 * A brief description of the class. Detailed Description.
 *
 * @author Shawn French
 * @version 1.0 15/02/03
 */
class Test
{
  public:

      /**
       * A constructor.
       * A more elaborate description of the constructor.
       */
      Test();

      /**
       * A brief description of the method. Detailed description.
       *
       * @param a An integer argument.
       * @param s A constant character pointer.
       * @return The test results
       */
       int testMe(int a, const char *s);

      /**
       * A public variable.
       * Details.
       */
       int publicVar;
};
```

Figure 18 – A Template of the Javadoc Code Convention Followed in GMS Source Code

2.5.2 Subsystem Decomposition

The GMS system can be divided into 4 main subsystems:

Interface (User Boundary)

Model (Entity)

Function (Control)

Technical Platform (UI/Database/Camera Libraries)

Each subsystem provides services to enable GMS to function as a whole. The composition of each subsystem, as well as the services that it provides, are detailed in the following sections. The following figure shows all 4 subsystems and their dependencies on one another (dotted lines):

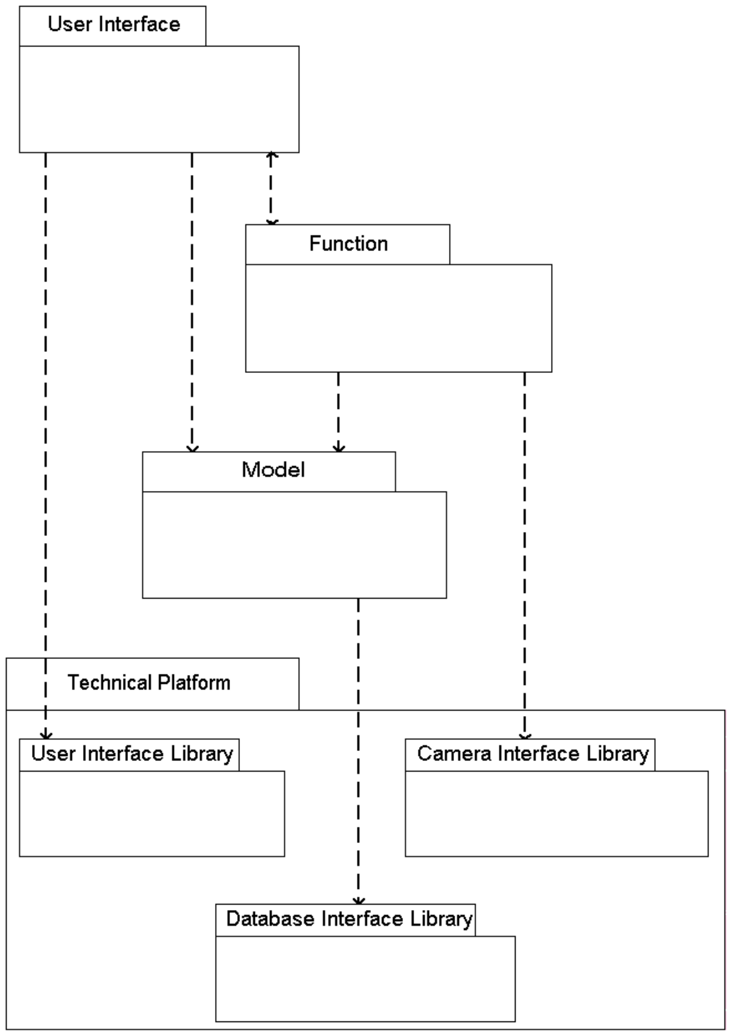


Figure 19 - GMS Subsystems and their Interdependencies

Figure 19, shows how all GMS subsystems (User Interface, Function, and Model) are dependent on the Technical Platform. Also, the figure depicts how the User Interface and Function subsystems are dependent on the public interface of the Model subsystem. Each subsystem's responsibilities, contents and dependencies are explained in the following 4 sections.

2.5.2.1 Function Subsystem

The function subsystem ties together all other subsystems. It's classes are responsible for the control of the entire system by reacting to user input in the appropriate fashion.

The subsystem consists of control classes that decide what boundary class should be shown depending on user input. The function subsystem is also responsible for taking user input and using it to populate objects in the model subsystem (such as a Message) to be saved into the database. For the "View" classes (in the user interface subsystem) to get and display "State" entity information, the function "State" classes will provide public getters. The getters will ensure that data will never duplicated and that all entity information is current when being displayed.

The function subsystem will need to use the video camera library to record video and audio from the peripheral.

2.5.2.2 User Interface Subsystem

The user interface subsystem is responsible for creating and manipulating the graphical user interface.

The subsystem provides a public interface (used by the Function subsystem) that creates and displays all GMS visual components and screens. The classes contained in this subsystem are all Boundary classes which provide a display boundary to the User.

As GMS is being implemented using Visual C++ on a windows platform, the user interface subsystem will use many components and services provided by the Technical Platform subsystem. Specifically, the user interface classes will use the Microsoft Foundation Classes (MFC) and the device drivers it provides to create and manipulate bitmaps, shapes, and windows application components (such as the media player to display video).

2.5.2.3 Model Subsystem

The model subsystem is responsible for providing a means to create, manipulate, and store all of the persistent data used in GMS.

It includes all classes stereotyped as entity classes that represent the system data. The data will consist of user and message information that is created, retrieved from, and saved to the database by classes in the function subsystem. The user interface classes will also depend on the model subsystem's interface to display model data.

Services provided by the Database Management System (DBMS) in the Technical Platform subsystem will allow the model classes to communicate with an ODBC compliant database.

2.5.2.4 Technical Platform

The libraries for the user interface, database, and camera are considered to be part of the Technical Platform subsystem as the team has no control over their implementation.

The user interface library consists of all of the Microsoft Windows platform components, drivers, and utilities that the implementation of the GMS user interface will use. The Windows Media Player component and basic shape and bitmap drawing libraries will be used to display graphics to the user.

The Microsoft Visual C++ ODBC (Open DataBase Connectivity) implementation and its associated libraries make up the database interface library. The model subsystem implementation will use this library to add, edit, search, and delete the persistent data stored in an Microsoft Access database.

Finally, the camera library is the system developer kit (SDK) provided by the Camera manufacturer, Logitech. The SDK allows developers to add the ability to record video and audio files using a Logitech camera into their 32-bit Windows applications.

This subsystem contains only external libraries that are all platform specific. For this reason, the libraries in this subsystem are the most likely to change (or be completely replaced) if the implementation platform were to change. That implies that any code in the model, or user interface subsystems using these libraries would also need to be modified.

2.5.3 Persistent Data Management

The system requires that persistent data be kept to ensure it remains accessible after a system shut-down. The persistent data will describe the system's users, and the messages that have been sent.

The Visual C++ language has a wizard that takes an ODBC compliant database and creates "wrapper" classes that allow for easy database interfacing. Therefore, the system will use an ODBC compliant Microsoft Access database to store all of the persistent information.

Below is the entity relationship diagram that describes the tables that will be kept by the database.

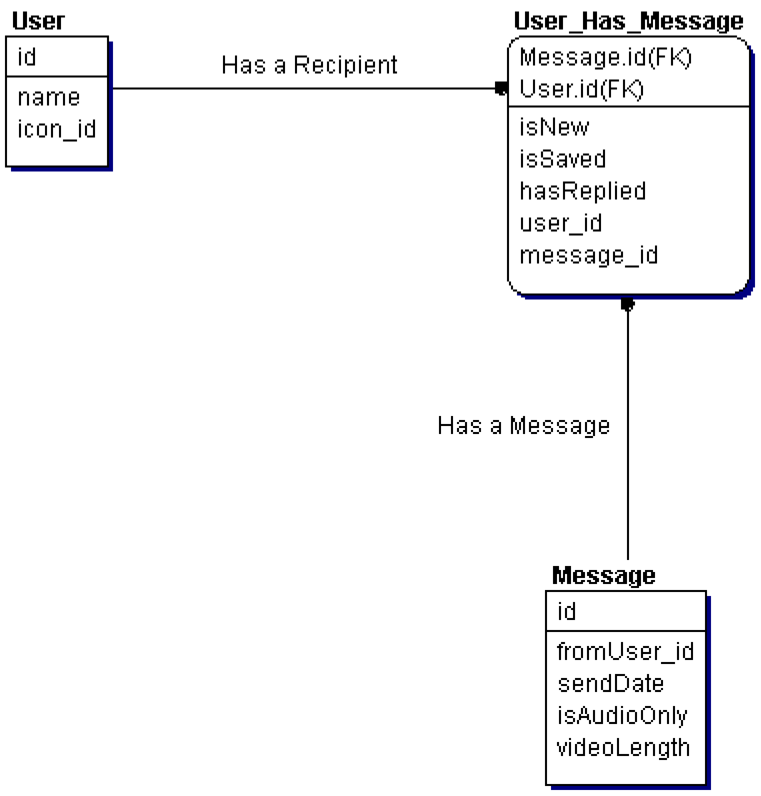


Figure 20 – GMS Entity Relationship Diagram

The User table contains records (sometimes called rows) containing all aspects that differentiate users including a unique ID, name, and the ID of the user’s icon. Every time a new user is added to the system, their unique data will be kept in a row of the User table.

A Message record describes all aspects of a sent message. This includes the unique ID of the message, the ID of the sender (a foreign key to the User table), the date it was sent, whether the message is audio only (isAudioOnly = true) or video with audio (isAudioOnly = false), and the length in seconds of the message. When messages are sent, one new row will be added to the Message table that contains all of the messages data. The actual video files are not stored in the database, but are stored as normal MPEG files on the hard drive. The messages are saved using the message id (from the database) in the format: “ID.mpg” in “c:\GMS_message_files\”. Although the video could have been stored in the database, the group felt that it would cause unnecessary delay as they would need to be taken out of the database and saved as a temporary file on the hard drive before being played.

When messages are sent there will be one or many records of the User_Has_Message table added to the database (as well as the one Message record). Each User_Has_Message record relates a unique Message (specified by the message_id field) to a User (specified by the user_id field) that has been designated as a receiver of the Message. This allows each user to modify their “view” of the message. For instance, the table has an “isNew” column (of type Boolean) that is set to true when the message is first sent, then to false when the receiver first views the message. Similarly there are “isSaved” and “hasReplied” columns that have similar functions. In other words, if a Message is sent to 5 User’s, then 1 Message record and 5 User_Has_Message records will be added to the database.

2.5.4 Access Control and Security

Using a fingerprint reader, the Group Messaging System will only allow recognized users to access their individual inboxes. At the default screen, if an unregistered user presses their finger upon the fingerprint reader, then the Add New User use case is executed. Otherwise, the user is sent to his or her inbox.

Although the authentication scheme mentioned above should prevent unlawful access to messages from the default screen, if a user doesn’t logout, it is possible that a perpetrator could access GMS and could therefore read, delete, and send messages from the registered user’s account. A solution to this problem would involve a logout timer that resets itself after every user input.

A GMS device would have to encrypt video messages before writing them to disk to ensure the user’s data are kept private in the event that the entire device is stolen.

2.5.5 Global Software Control

To ensure a robust, scaleable, and changeable system, there must be certain regulations concerning the communication between subsystems.

The rules for communication for GMS subsystems are based on the standard Entity (Model subsystem), Boundary (User Interface subsystem), and Control (Function subsystem) architecture as described in the Object Model section (2.3.1) of the Design discussion. See Figure 19 for the allowed dependencies between subsystems.

All requests to GMS will originate from actions taken by the user. Such actions include: pressing a button, or pressing on the fingerprint reader. If a button is pushed, the windows operations system will send a “Windows Message” to the GMS Windows Application. Button messages correspond to keyboard events and are handled by implementing the OnKeyUp() and OnKeyDown() message handlers. The fingerprint reader push will also interrupt our application, but at this point the details are not known.

2.5.6 Boundary Conditions

2.5.6.1 Initialization

In a consumer device, the GMS application will be brought up automatically on system initialization assuming the operating system doesn't incur any initialization errors. If GMS has an initialization error (such as not being able to connect to the database) then the program will fail as described in section 2.5.6.3.

2.5.6.2 Termination

If the system is terminated while in normal operation, all data associated with the task (only applies to composing a message and adding a new user) will be lost. Other than that, termination of the program will not affect the system, or its future stability.

2.5.6.3 Failure

Only database errors and driver errors will cause system failure. In the event that the database is not found (through the ODBC interface) the application will tell the user about the problem, using a notice, and ask them to restart the system. The reasoning is that without the database, the users and message data is unavailable, and the system is therefore incapable of even showing the default screen, let alone creating new messages or users. If the camera device driver or the fingerprint device drivers fail or are not found, the same process of notification and suggestion of a reboot will be followed for similar reasons. Without the fingerprint reader, users can't login, and without the camera no messages can be sent.

3 Implementation

After the thorough design of the system, several implementation decisions needed to be made. The first issues that were dealt with were the selections of which external components (hardware & software) were going to be used. On the hardware side, the three main considerations were the camera, the fingerprint reader, and the input buttons, as the monitor and speakers required no special consideration. On the software side, the two main considerations were the implementation language and how to display video playback.

3.1 Hardware Implementation Considerations

3.1.1 Video Camera

The GMS is essentially a video recording system. In order to facilitate video and sound capture, a Logitech web camera is used. This particular type of camera was chosen since one of the group members already had the hardware and in addition, Logitech provides a generalized SDK that is designed to work with any of their cameras. [8] By using this SDK, we were able to connect to the web camera and interface with its functionality.

Although the concept of using the Logitech SDK was sound, there proved to be many problems along the way. This caused several problems as the camera interfacing was done on a Microsoft Windows XP platform. These problems were eventually solved by using an older driver version for the web camera, and by registering several DLLs manually, since this aspect of the SDKs install program did not work under Microsoft Windows XP.

Connection to the camera was handled at the control class level. A camera interface object is created, as part of the objects provided by the SDK, and all commands are funneled through this interface before they are interpreted by the camera. All functionality of the camera is handled by the camera object provided as part of the SDK. This includes displaying a live video feed and recording video/audio using any compression codec installed on the system. For the purposes of this prototype the mpeg 1 video compression codec was used to capture video at a resolution of 320 x 200. Under these conditions the GMS will record video at approximately 1/3 of a megabyte per second. This codec and resolution offer some level of compression while at the same time minimizing the load on the processor to play it back. It is important to minimize system requirements since a lower common hardware denominator is best for marketing purposes. Although all video/audio recording is handled using the Logitech SDK, it is important to note that this does not include video playback. Instead the Windows Media Player API is used for video/audio playback.

3.1.2 Fingerprint Reader

The fingerprint reader is the most futuristic component in the system. Because this device is so new (especially in plug and play PC applications), it was difficult to locate products for sale in non-bulk orders. The most important feature for the fingerprint reader to possess was an API that could be used to integrate the device fully into our system. Unfortunately, all APIs came at a price, and the least expensive one was $1000 USD. This of course would send us far beyond our allotted budget for the competition. Due to this, the implementation of the fingerprint reader has been put on hold and this option will be re-evaluated for inclusion into our system at the competition finals. Because the system would no longer be capable of authenticating users by their fingerprint, several changes were required in the system. As it is still planned to include fingerprint authentication later in development, a logon devoid of security features was implemented. The placement of the user icons on the default screen were lined up with where the buttons would be so that a user simply needs to push the button associated with their icon to log themselves into the system. The lack of fingerprint reader posed another problem: adding new users to the system. To solve this, the number of maximum users was decreased to nine, so that a button could be added in the bottom right corner that would trigger the “add new user wizard” when pressed.

3.1.3 GMS Ten-Button Interface

In order to test the ten-button interface that was designed for GMS, the group decided to create a prototype for use at the poster fair. An old keyboard was taken apart and the controller circuit was removed. As can be seen in Figure 21, the controller has 9 ports on the horizontal axis (labeled as “x” ports), and 16 ports on the vertical axis (labeled as “y” ports).

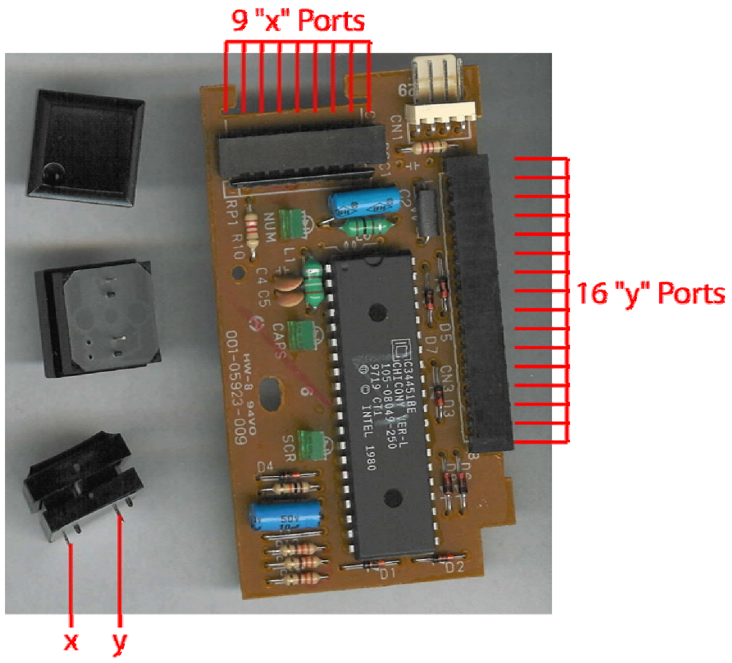


Figure 21 – The Keyboard Controller and 3 Buttons Used for the GMS Ten-Button Prototype

Every key in the keyboard is attached to one “x” port and one “y” port. When the circuit is completed (through a switch) the key’s code is sent through the controller’s PS2 connection to the PC. It was decided to use the 10 number keys of the keyboard for the ten buttons of the GMS prototype. Therefore, the x and y ports for each number key (0-9) was to found and recorded. Ten buttons (3 of which can be seen in Figure 21) were purchased and the “x” and “y” ends were soldered to a wire that connected to the proper “x” and “y” port of the controller circuit. A frame was built using a foam board and silver poster to house the buttons. Once complete, the GMS ten-button prototype was attached to a monitor and the end result was shown in Figure 1.

3.2 Software Implementation Considerations

The major consideration in the software implementation phase was what language to use. The top three languages that were considered were: Java, Visual C++ and Visual Basic. A skeleton of the GMS application was attempted in Visual Basic, but had to be abandoned because of VB’s poor support for object oriented concepts. Java was a strong candidate, but had shortcomings when it came to integrating with our peripherals. Visual C++ is object oriented, and has the ability to interface with all the peripherals that we were planning to use, and would most likely have support for other peripherals in the future, so it was decided to implement the GMS prototype using Visual C++.

The next consideration was how to handle video playback. Windows Media Player was chosen because using Visual C++ meant that Windows Media Player could be used as a DirectX component in our software to handle all video and audio play back.

3.2.1 Function (Control Classes)

The function subsystem includes the actual GMS Windows Application class (GMS_Application) as well as the main frame called GMS_Frame. As described in the Design section of this document, GMS_Frame is the main workhorse class that responds to user input by delegating events to its current state. GMS_State, GMS_Composite_State, GMS_Sub_State and all of their implementations make up the rest of the “Function” classes. Refer back to Figure 5 for the class diagram of the control classes that make up the function subsystem.

The function subsystem and all of its classes were built in 3 stages:

1. Implementation of the GMS_Frame state machine using only the “simple” states
2. Implementation of GMS_Frame transitions
3. Implementation of GMS_Composite_State transitions
4. Implementation of graphical interface and entity manipulation

GMS_Frame inherits from CFrameWnd and is the main frame of the application. It was therefore implemented as the main event handler that captured keyboard and windows “paint” events. Keyboard events are sent when a key is pressed (KeyDown event) and released (KeyUp event). A paint event is sent when the application first displays itself, and whenever the pixels need to be redrawn. Our design would have GMS_Frame catching all of these events but delegating their interpretation to the current state (of type GMS_State). As such, a GMS_State needed to implement handlers for all of the events that GMS_Frame receives. However, GMS_State objects only took care of reacting to user input, and delegated the painting related messages to its GMS_View object. This design worked well as we ended up with many very specialized classes all reacting to events that they could handle.

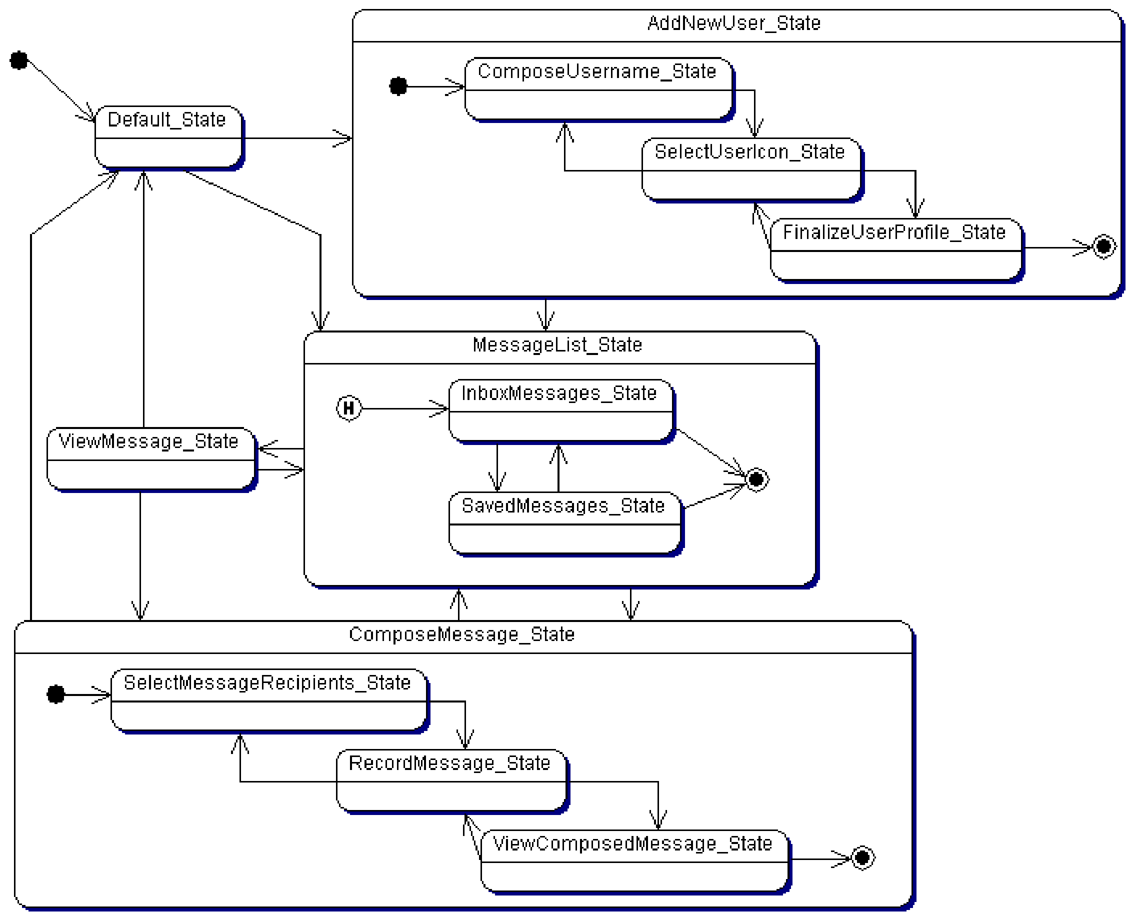


Figure 22 – Statechart Diagram for a GMS_Frame Object

Figure 22 shows all of GMS_Frame's possible states. Although some states contain sub states, all composite states shown in the figure (AddNewUser_State, MessageList_State, and ComposeMessage_State) were first implemented as "simple" states. The idea was to first get the GMS_Frame using the simple states and then move on to setting GMS_Frame to be in composite states containing sub-states. The "setState" method was implemented to take an object of type GMS_State and set it as the current state of the GMS_Frame object. The method saved the current state as it's previous and then called the "enter" method of the new state as prescribed by the state-pattern. Our test simply mapped keyboard keys to a state so that when pressed, an object of the associated state would be created, and then set as GMS_Frame's current state. The team used a logging function to display the current state to the screen for debugging.

Once GMS_Frame objects were shown to be properly setting their states, GMS_Frame's "enterPreviousState" method was implemented. It allowed states to request GMS_Frame re-set the previous state as the current and then re-enter it. This mechanism was needed so that after viewing a message, the ViewMessage_State object could tell GMS_Frame to re-enter the MessageList_State.

Objects of type ComposeMessage_State would also use the method to ensure users were brought back to their MessageList_State object. This method, coupled with the “setState” method allowed GMS_State objects to transition to and from one another. It was therefore time to model the inner transitions in GMS_Composite_States.

As can be seen from the state chart diagram, there was a “wizard-like” behaviour in each composite state as they have only two (next and previous) transitions between their sub states. Therefore, the sub states could ask their composite container to either move to the next or previous sub state at any point during their execution. If the composite was asked to go into the next state while in it’s last sub state, the result would be for the composite to ask GMS_Frame to re-enter it’s previous state. The same goes for if the first sub state asked it’s composite to go back.

Once all of the states and their internal and external transitions were working, the boundary objects (the state’s “View”s) and the entity manipulation were tied in. All that needed to be done in this step was the delegation of paint events to the GMS_View’s, as well as coding the creation/saving/selecting of GMS entity objects to and from the database. For instance, the AddNewUser_State object needed to create and manipulate a User object and eventually (unless the user canceled) save it to the database.

3.2.2 User Interface (Boundary Classes)

The following boundary class subsections (3.2.2.1 – 3.2.2.4) cover the implementation of the user interface, including the control and painting of graphical elements and video.

3.2.2.1 Fonts and Bitmap Images

The success of the GMS software application relies heavily on successfully reproducing the user interface screen mockups that were developed in the design stage. A framework needed to be developed and put in place so that fonts and graphics could be easily used throughout the application. Thus, the Font and MaskedBitmap classes were created (see Figure 23).

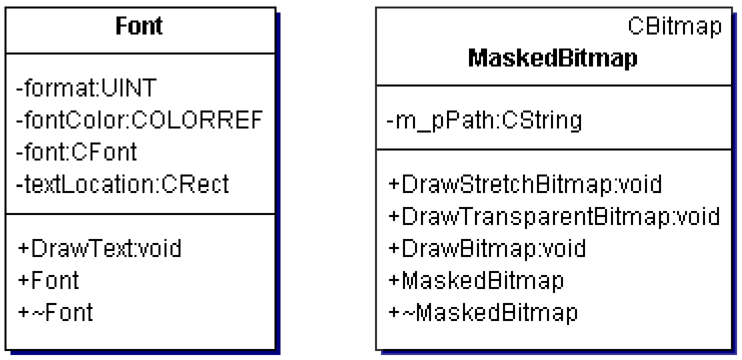


Figure 23 - Font and MaskedBitmap Class Diagrams

All the font types/styles were defined for each font used in the application and assigned an identifier and defined in the Font class header file. Creating an instance of the Font class initializes a CFont object with appropriate font size, font weight, positioning and color corresponding to a font identifier passed in the Font constructor. When a Font object has been instantiated, DrawText() is called with a parameter containing the text to be written to the screen.

The MaskedBitmap class inherits from CBitmap and is passed the location of the bitmap to display in it's constructor. After the MaskedBitmap object has been instantiated, a bitmap can be draw to the window using one of the three draw methods available. The draw methods provide a simple output of the bitmap to the screen in addition to outputting a scaled version of the bitmap. Also, a semi-transparent bitmap can be drawn forcing a color defined by the caller to be transparent when drawn to the screen.

3.2.2.2 DisplayObject and DisplayListWalker

A large part of many of the screens on the user interface require iterating through elements on the screen, such as the letters or icons on the add user screens or the messages in the inbox. Because this 'iterate and display' functionality is used on many screens, two class hierarchies were developed to handle this:

- **DisplayObject**: The DisplayObject class hierarchy is responsible for the formatting and display of data on the screen.
- **DisplayListWalker**: The DisplayListWalker class hierarchy is responsible for iterating through corresponding DisplayObject objects and displaying them in the correct location on the screen.

In the DisplayObject hierarchy (see Figure 24) the leaf classes handle the display of different data. The DisplayObject super class provides common data and functionality for the leaf classes to use and is never instantiated without a leaf class. When these leaf DisplayObject classes are instantited they are passed an entity object containing the data that the DisplayObject is responsible for formatting and displaying. The paint() function of the accesses the entity data and creates the necessary Font and/or MaskedBitmap objects to display the data and draws it to the screen.

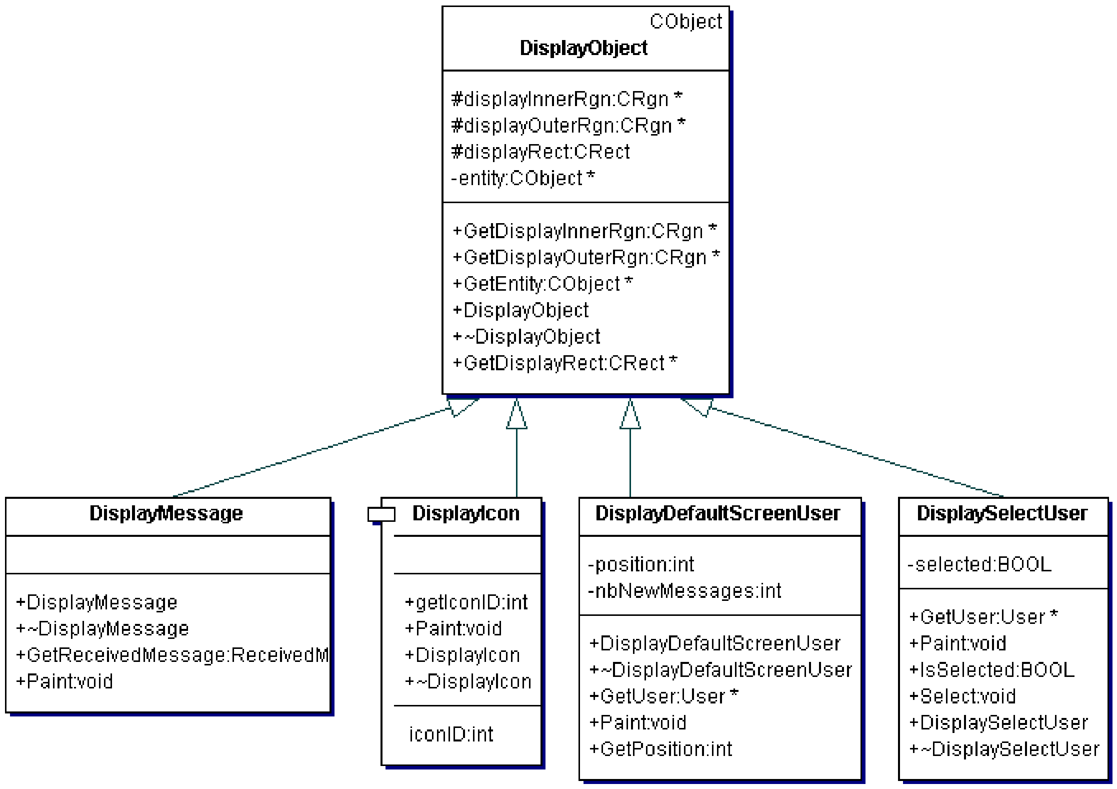


Figure 24 - DisplayObject Class Hirarchy

In the DisplayListWalker hierarchy (see Figure 25) the leaf classes are responsible for iterating through the DisplayObjects and calling the DisplayObjects to display to the screen. The DisplayListWalker super class provides common data and functionality for the leaf classes to use, but is never instantiated without a leaf class. The DisplayListWalker constructor is passed an array of entity objects that in which it iterates through and passes to the constructor of a DisplayObjects class. It then stores all the DisplayObjects that were created in an array as a member variable. The DisplayListWalker provides Next() and Previous() functions for iterating through the DisplayObjects in a circular fashion. Some leaf DisplayListWalker classes override these functions to provide different ways of iterating such as in the DisplayMessagesWalker where the DisplayObject array is not iterated in a circular fashion. The PaintElements() first calls the paint() method of the DisplayObjects that should be displayed onscreen.

Also, it displays any highlights for DisplayObjects that hold the focus onscreen, such as the highlight under the current message in the inbox screen.

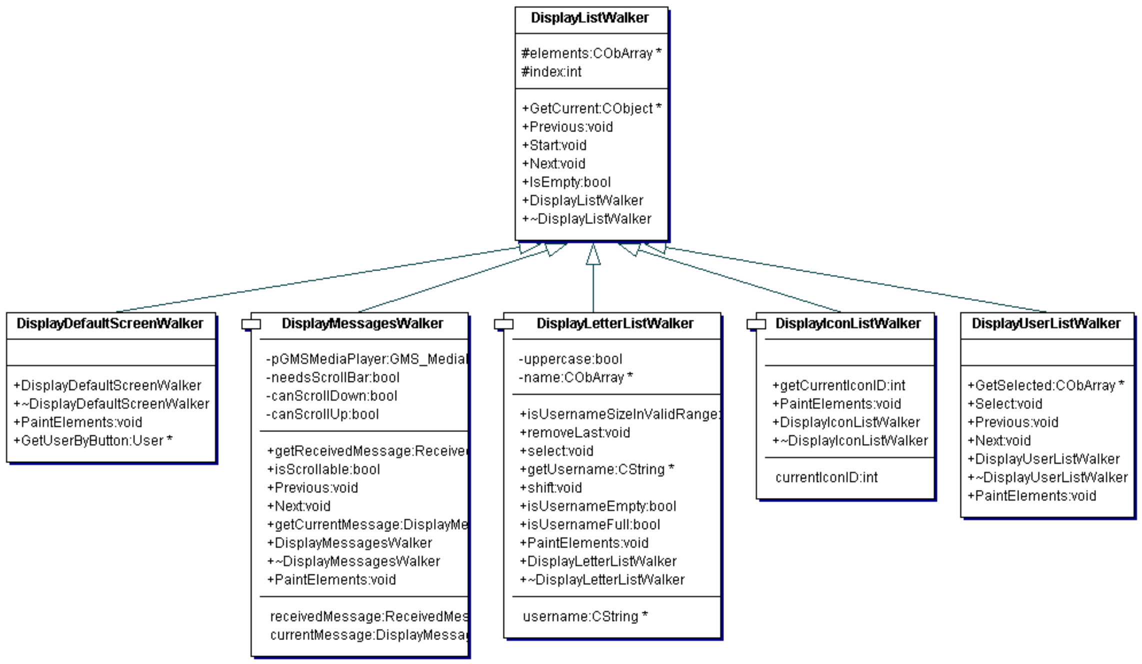


Figure 25 - DisplayWalker Class Hierarchy

3.2.2.3 Overall Flow of Control for Displaying a Screen

Each State class in the system has a corresponding View class that is responsible for displaying the current screen. The view class takes direct responsibility for displaying the background bitmap, the title of the screen and the current user, and calls upon the DisplayListWalker of the current state (if one exists) to paint all other elements. Figure 26 shows the flow of control for instantiating display classes (View, DisplayListWalker, and DisplayObject) in paths 1 & 2, and the flow of control during a paint cycle in path 3. The state in Figure 26 has a DisplayListWalker with two DisplayObjects, thus, if the State was Inbox_State, then the two DisplayObjects would represent two messages in a user's inbox.

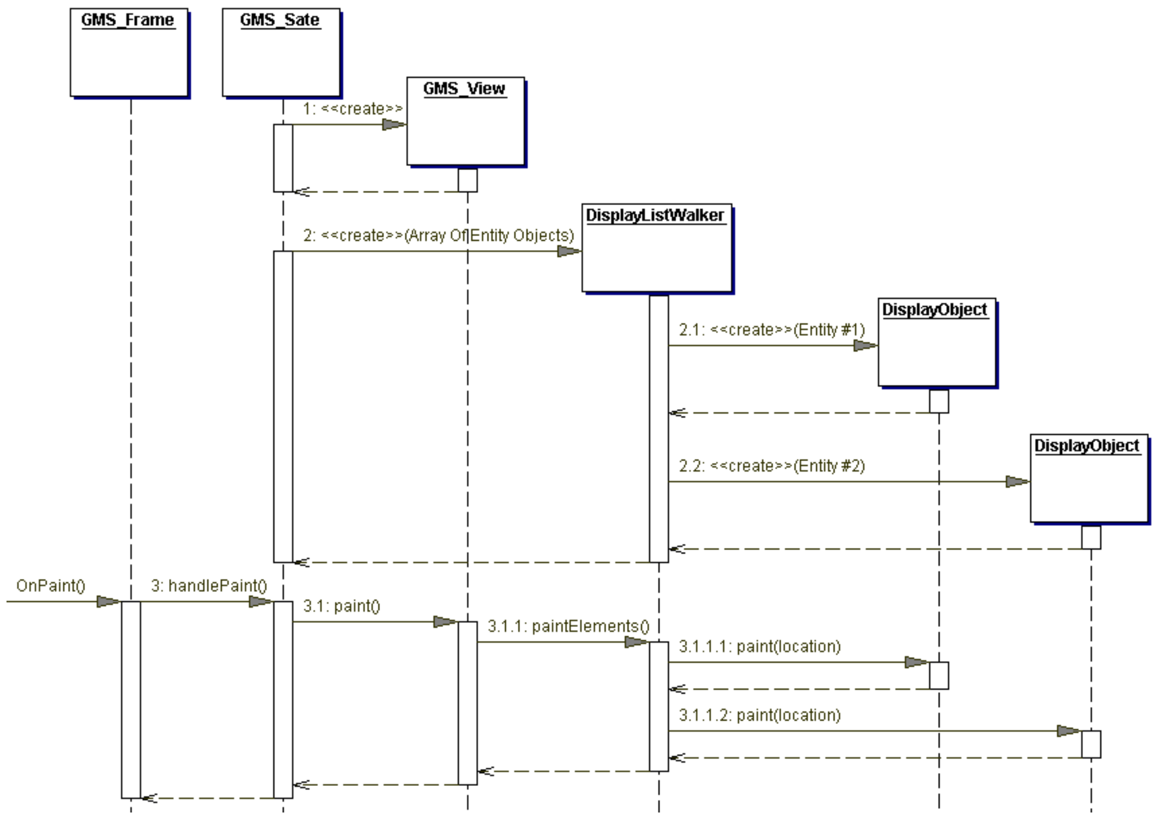


Figure 26 - Sequence Diagram for Instantiating Boundary Objects (1 & 2) and Painting a Screen (3)

3.2.2.3.1 Control Flow Path 1 & 2

When When a new State object is created or entered into it is responsible for creating all boundary objects to deal with the state's display. A State object will first instantiate a View object and, if required, will also instantiate a DisplayListWalker object, passing along an array of entity objects to be wrapped in a 1:1 ratio to DisplayObjects.

3.2.2.3.2 Control Flow Path 3

Painting a screen begins with an ON_WM_PAINT message sent by the OS that triggers the OnPaint() function of the current window. The OnPaint() method must be overridden by the class that inherits from the Microsoft Foundation Class CFrameWnd if anything is to be displayed to the window. In this system, GMS_Frame inherits from CFrameWnd, and when OnPaint() is called the flow of control is passed to the current State class' handlePaint() function, which in turn calls the paint() function of the View class associated with the current State class. The paint() method in the View class is responsible for painting some elements, then calling the associated State class' DisplayListWalker object to paint the appropriate data to the screen by invoking paintElements(), which iterates through the viewable DiplayObject objects and invokes their paint() method to paint themselves on the screen.

3.2.2.4 Video Playback

As stated in the Software Implementation Considerations section above, video playback is achieved using the Windows Media Player DirectX component. "Microsoft DirectX is an advanced suite of multimedia application programming interfaces (APIs) built into Microsoft Windows operating systems. DirectX provides a standard development platform for Windows-based PCs by enabling software developers to access specialized hardware features without having to write hardware-specific code." [7]

The Media Player DirectX component was included in the application, and provided an API for interfacing with Media Player. Thus, a State object that requires video to be displayed instantiates a Media Player object and changes the Media Player's state (current video file, playing, paused, etc.) though calls to its member functions.

The Media Player DirectX component was easily inserted into the application and was handled by the view classes that displayed the video as an object. Controlling the video playback simply required making the appropriate function calls to the media player object.

3.2.3 Model (Entity Classes)

The GMS design calls for three main entity classes: Message, User and ReceivedMessage. At their most basic level, these entity objects contain attributes that mirror the fields found in their respective tables in the database. The advantage of separating these data objects and database interfacing away from the higher up control classes is that internal changes can be made to entity (message or user) structures, and to how they are stored (in a database or otherwise), without changing the way these objects are used. For instance, we could choose to remove the database and store persistent data in our own proprietary format, or we could communicate with a database on a remote server. This decreases the coupling between each layer of the system and allows for easier changes down the road.

While entity classes themselves have no control logic, they do possess basic methods that simplify their use and access to their data structures. Entity objects provide getters and setters to their attributes as well as static methods that allow additional tasks to be completed. For example, the static method of User called getAllUsers() will return a list of objects representing a list of all users in the system. This type of static method can be used as an interface between the control classes and the persistent data stored in the database. By using these static methods, the entity object can gather or set specific types of information in the database on behalf of the caller. Often these static methods will perform the arduous task of assembling arrays of entity objects based on set queries to the database. For instance, the static method getAllInboxMessages(), contained in the ReceivedMessage class, will loop through all Messages that are contained in the inbox of the given user and create an array of ReceivedMessage objects to return to the caller. In order to create the ReceivedMessage objects that are returned in the array, User objects must be created for the sender and recipients of the message. The process of creating a User object involves many additional database queries. Additionally, the creation of a User object requires the creation of a UserIcon object associated with that user. The creation of this UserIcon object is based on another database query. In this way, you can easily see how a static method like "getAllInboxMessages(User*)" contains a lot of logic dedicated to the handling of Messages, and by employing it at the entity level, saves the control classes any need to know about the inner workings of messages or the database.

In order to implement the entity classes, the UML generation application "Together" was used to create a skeleton structure. Based on this skeleton code, the getters and setters were first filled out. At this stage a testing harness was created that ran as a win32 console app. The testing harness was used from this point on to verify the correct workings of each entity class. After creating the getters and setters of each entity object, an interface to the Microsoft Access database was established using the ODBC compliant classes CDatabase and CRecordset, provided as part of Microsoft Visual C++ 6.0. By using a CDatabase object, a registered ODBC data source could be opened. However, in order to proceed, the GMS Access database had to be registered as an ODBC data source in Windows using the "Datasources (ODBC)" icon found in the control panel. By opening a connection to the database using a CDatabase object, CRecordset could be used to perform queries and return recordsets from the database. The database contains three main tables: user, user_has_message, and message. In order to make access to these tables easier, three additional entity classes were created, one for each database table. Each of these classes inherits from CRecordset, but is customized to make access to a particular table easy. In this way, each static method that needs access to the database would first create a database connection using CDatabase, then pass the open connection to the CRecordset derived object that gives access to the specific table of interest. Based on the attributes of the objects contained in the returned recordset, new Message, User or UserIcon objects can be created and returned to the caller of the static method. The insertion of new users or messages into the database is handled in a similar way to retrieving data. For instance, in order to add a new user to the database, the "dbInsert()" method is called on the User object that is to be added to the database. This method will establish a connection to the database and create a CRecordset object for the user table. Instead of executing an SQL insert query, the "AddNew()" method of CRecordset is used. This provides a blank spot at the end of the user table recordset, in which the attributes of the recordset can be set according to the attributes of the User object. Once a new user has been added to the recordset in this way, the recordset can be committed to the database using the CRecordset "Update()" method.

There was one last hurdle to overcome in the implementation stage of the entity classes. The inbox contains many message objects, however, it also needs to access many aspects of these message objects that require additional database queries. These include: checking if a message is new to the current user, and checking if a message has been replied to by the current user. Using the simple

Message object to gather this information required unnecessary database access that would result in a noticeable slow down on older machines. To correct this, a new ReceivedMessage class was created that inherits from Message. ReceivedMessage was specifically designed to address these performance issues by gathering all the necessary data in one shot as soon as the object is created and storing the values in local variables. This focuses all database access and since the local variables containing the query results are maintained throughout the life of the object, the database doesn't have to be accessed again. This particular methodology of focusing all database access seems to speed up the system since the slow down appears to be the result of opening and closing a database connection, not the actual queries. Using the old Message object to gather the required data caused a database connection to be opened and closed each time a piece of data was required, where as the ReceivedMessage object executes all queries using a single connection.

In terms of system integration, the entity classes were added to the system after the getters and setters were tested. By integrating the entity classes with the rest of the system at such an early stage of development, before any of the more complicated or database driven methods were finished, the rest of the team was able to continue developing their own parts of the system using basic data objects. As the database functionality was added and the entity classes grew to completion, the rest of the team was able to incorporate these features in the system. By implementing and testing the entities in this piece-wise fashion, the team as a whole was able to develop at their own rate.

4 Testing

4.1 Code Testing

Extensive testing was completed on the system during and after the implementation stage. Throughout the implementation phase each group member was responsible for unit testing their respective functions and classes. When the system was complete, it was thoroughly tested by executing all use cases and then rolled out to be tested by users at the fourth-year project poster fair. Several bugs arose in this stage, but were quickly fixed. The system ended up proving very robust in the real world user testing stage in part because the application tightly controls the input of users and therefore there are not many opportunities to crash the application.

4.2 Usability Testing

On March 14, 2003, Carleton University's Systems and Computer Engineering department held its annual Poster Fair. The group took the opportunity to hold usability tests to ensure people (other than its developers) thought the GMS user interface was pleasant and usable. Our usability test asked participants to complete tasks that tested all of the functionality GMS offers. As we knew some people would not have time for a long test, we had a long and a short usability test and asked passers-by if they would participate in this very important step of the development process.

Although only five tests were conducted, the results gathered offered the group more feedback than imagined. Each participant agreed that the interface was very pleasing to the eye and praised our color selection and the overall look and feel. Also, many people commented on how they would like to see a similar device in their homes one day! However, the purpose of the tests was not only to elicit praise from possible users (although we were hoping for some), but to identify any problems that users had while using GMS. The tests pointed out that some of our on-screen instructions were too vague, and some were confusing. Also, the placement of some buttons was questioned as some users pressed others thinking "it should have been there!"

The group is very satisfied with the results, and plans on making changes to the interface based on user errors and feedback. Please see Appendix E for the tasks and questions that participants were asked to complete, and Appendix F for a summary of the results.

5 Conclusion

5.1 Accomplishments

The process of bringing the GMS to the current level of completion has involved a lot of planning, hard work and late nights. We began by brainstorming appropriate ideas for the competition. After arriving at the idea of the GMS, we developed the requirements analysis document (RAD), followed by the system design document. From these solid foundations, we broke up the work of implementation into the boundary, control and entity classes and assigned each to a separate group member. Additional implementation issues were also handled at this time, including interfacing with Windows Media Player, the Logitech web camera and creating the ten-button user interface that attaches to a monitor. By designing the system in a modular fashion, we were able to divide the work up evenly and work in parallel to complete the system as quickly as possible.

5.2 Comparison to Current Systems

GMS is targeting improving the current way that non-real-time messages are communicated. The system is not competing against technologies that provide immediate, real-time communication. In most homes messages are written on paper or post-it notes and left in a central location. This approach has several shortcomings:

- The note can get lost or might never be found by the person to whom it's directed.
- Someone else might find it and read it, thus giving away information the sender wanted to keep private.
- A message could be intended for multiple recipients but could be discarded by the first person who reads it.
- Usually the note is thrown away after the fact and there is no record to relate back to in the future.
- It is extremely difficult to convey long explanations in the tiny space most notepads provide.
- Handwriting can be messy and misunderstood by the reader, leading to a wrong or confusing interpretation of the message.
- Tone cannot be conveyed well on paper.

GMS aims to provide solutions to all these problems with this current system. GMS provides message persistence with each intended recipient getting a copy of the message. This also provides privacy, as the message cannot be accessed by anyone other than the intended recipient(s). Video and audio messages are much clearer and can convey tone much better than text messages. With the ease of creating an audio or video message using GMS, the system is able to provide a better method to send and receive messages.

5.3 Next Steps in Development (Ideas for the future)

Several enhancements can be made to the current system to improve the functionality and value of the system in the future, such as including a touch screen and implementing a security model and adding an administration mode. GMS has already been designed with a UI that could be easily ported to function on a touch screen. This would simplify the interaction with the user, as the user would no longer need to focus on how the input buttons on the side of the current system align with the graphical representation of the buttons onscreen. This would be an obvious improvement, as several users tried to touch the screen of the prototype before realizing it was not a touch screen. This functionality would be easily implemented as touch screens work by emulating the mouse, thus only support for mouse clicks over the onscreen buttons would need to be implemented. This feature was not implemented in the current design because of the spending limit imposed by the competition.

Although a partial security model was planned in the design phase, in the form of user authentication via a fingerprint reader, it was never implemented due to the prohibitively high cost of the SDK for the fingerprint reader devices from several manufacturers. This, like the touch screen did not fit the budget set out by the competition. Using a fingerprint reader for user authentication would provide a fast and accurate way of logging a user into the system. The authentication process would also prompt the user to be added to the system should the authentication fail to recognize the user. A second piece of the security puzzle would be to include encryption of all messages on the system. The media files would be encrypted and would incorporate the fingerprint identification information in some way. This level of security may not be necessary in home or similar applications, but it could become a requirement for adoption by businesses and governmental agencies.

The third enhancement, which would be required for a commercial release of the system, is an administration mode. The administration mode is needed to accomplish system level tasks that, depending on the application, should not be made available to every user in the system. These tasks could include: deleting a user; setting system preferences and setting the system date & time. The administration mode could be set to be accessed by only one of the users on the system, or if security was less of a concern, could be left open to all users. The administration mode would be accessible via the default screen, a button in the administration users' inbox screen (or all users' inbox screens) or it could be made to be activated by a button on the device. The administration mode was left out of the design and implementation of the prototype, because it was not crucial at this stage of development.

A final modification that would need to be made, which would have no effect on the system's functionality from a user's point of view, would be to implement the design on a non-proprietary platform and link with non-proprietary components. Because the prototype's target platform is Microsoft Windows, should the device be sold as a standalone consumer electronics device, the target platform would need to change to either a Windows Embedded OS, Linux or another available embedded OS. Changing the platform would also require changing the other Microsoft proprietary components, such as the MFC used throughout the code, the Windows Media Player DirectX component that plays messages and the Microsoft Access database used to hold all persistent data.

5.3.1 Marketing and Manufacturing

This device has been developed with several target audiences in mind. First and foremost the device's primary audience would be families. This device is ideal for this market because it can be marketed as a time-saving device as well as a device that could help bring families closer together. The user surveys that were conducted during the development of this device show that this would be beneficial. The device could also be marketed towards small business as a more personal form of communication. This would also be beneficial in any busy environment where messages are left, from hospitals and reception desks to university residences.

There are also several options available for distributing GMS. Besides the option to distribute the system as a standalone consumer electronics device, other options include: distributing the software only and requiring the user to have a PC and compatible peripherals; distributing the software and a device (composed of a screen and a video camera) that can be connected wirelessly to a PC in the home; or as a standalone dumb terminal device where the software is offered through the internet from an applications service provider. Although the original design is for a standalone device with software included, the distribution mentioned above could help reduce prices and help penetrate other markets. If the system was sold as software and peripherals only the cost of the system would be greatly reduced as there would be no need to include a computer and screen. Only the buttons would need to be included, with the possibility of also having a package with camera and speakers.

To remove the complications of having to install and configure these third party peripherals, the system could be manufactured as a wireless monitor device that would include speakers, a web cam and button if the monitor was not a touch screen. Although this option would cost more than the previous option because of the inclusion of the monitor, it still does not require a new PC, and would make the installation phase and maintenance much easier on the end user.

A final level of abstraction for manufacturing the device would be to sell the device as an internet appliance, where the software would reside on an application service provider's servers. Using this method of distribution, the hardware cost could be subsidized by a services contract between the end user and the application service provider.

Also, if the functionality that GMS provides did not provide a compelling enough value to be purchased as a separate product, then the GMS software could be bundled with other devices or more features could be bundled with GMS. A possible application of this would be to include the GMS in a refrigerator that has a touch screen computer device. GMS would simply serve as one of the applications available on the device.

5.4 Wrap-Up

The improvement of communication within groups of individuals has always been a desirable goal. It is the hope of this group that the Group Messaging System will help meet this goal by facilitating better communication, particularly in the home environment. Although the concept of GMS fulfills the basic requirements of the CSIDC competition, it was felt that the application of strong software engineering concepts in the design phase were necessary to stand out amongst other competitors. To this end, an extensive phase of system design and planning was conducted prior to implementation. Not only does this put our group in a strong position according to CSIDC marking guidelines, but it greatly aided us during the implementation phase. Since GMS was designed in a modular way, each member of the group was able to implement different components of the system in parallel. This factor along with the use of standard Microsoft Windows APIs and the Logitech camera SDK, contributed to a relatively short implementation time. After testing GMS, it was displayed at the fouth-year project poster fair, where it received a warm reception. Displaying GMS at the poster fair also gave us the opportunity to follow up with usability surveys and allowed us to discuss potential marketing possibilities.

Although this report is the conclusion of our fourth-year project responsibilities, it is not the end of the group's work on GMS. As of this writing, the group is waiting on confirmation to proceed to the next stage in the CSIDC competition and write the final CSIDC report. If successful, we will have the opportunity to compete against other finalists and demonstrate GMS to CSIDC judges.

6 References

[1] "CSIDC 2003," *IEEE Computer Society,* http://www.computer.org/csidc/ (current March 2003).

[2] IEEE-STD-610 ANSI/IEEE Std 610.12-1990, "IEEE Standard Glossary of Software Engineering Terminology," February 1991.

[3] B. Bruegge, and A.H. Dutoit, *Object-Oriented Software Engineering – Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, N.J., 2000.

[4] Kristy Pryma, "The Man Behind the Machine," *Transcontinental Media Inc*, http://www.itbusiness.ca/index.asp?theaction=61&lid=1&sid=51747 (current March 2003).

[5] A. Shalloway, "Design Patterns From Analysis to Design," *NetObjectives*, http://www.netobjectives.com/dpexplained/download/dpmatrix.pdf (current March 2003).

[6] "Javadoc Tool Homepage," *Sun Microsystems*, http://java.sun.com/j2se/javadoc/ (current March 2003).

[7] "Technology Overview - Microsoft DirectX," *Microsoft Corporation*, http://www.microsoft.com/windows/directx/productinfo/overview/default.asp (current March 2003).

[8] "Logitech Developers Network," *Logitech Inc.,* http://developer.logitech.com (current March 2003).

[9] L. C. Briand, "Software Engineering Course Notes," *SYSC 4800*, http://www.sce.carleton.ca/faculty/briand/teaching/94480/teaching94480.html (current March 2003).

Appendix A - Requirements Analysis Document

Appendix B - User Survey Questions

Appendix C - User Survey Results Summary

Appendix D - System Design Document

Appendix E - Usability Test Questions

Appendix F - Usability Test Results Summary

Appendix G - GMS Software and Documentation CD