Open-source Chess and Instant Messaging Program
Software Specification, Beta Release v2.0

Developed by 22-Move L: Aammarah Idris, Darlena Nguyen, Kiefer Daniel, Kyle Mach, & Sofia Bernstein

EECS 22L, Spring 2020
Department of Electrical Engineering and Computer Science
University of California, Irvine

Table of Contents

Note to .pdf readers: All entries in the table are linked and clickable.

Glossary

Software Terminology

Control flow: The order in which functions are called and executed.

Data type: A data item defined by the value it can contain (ie: int, float, double).

Executable: After compiling the program, the executable is automatically generated and can be used to run the game. Running the executable file is how the user interacts with the main interface of the game.

Function: Functions take in parameters, which may be manipulated or help perform a computation that produces an output.

Header files: These files contain the function declarations and macro definitions that can be shared amongst source files.

Makefile: This file sets targets and dependencies from source files to their object files. Its primary use is to generate the object files and executable to run the program by compiling the intermediary source code.

Module: The entire MephistoSoul program is broken up into organized modules, as visualized in the module hierarchy. See *1.2 Major Software Components*.

Source files: These files store the program code, data definitions, and function definitions.

Structure: A user defined data type meant to group items, possibly of different data types, with elements that are easily accessible by the struct name.

Messaging Terminology

Account: A personal profile tied to a unique identifying username that remembers a user's game records and friends

Guest: A user who does not have a permanent account so is given a temporary account. Guest users are unable to add friends.

Friend: A user that, once added to an account's friend list, can be directly invited to play a game with or chat with

Instant Messages: A type of chat that offers real-time text transmission over a network connection.

Login : Authenticating oneself to the server with a username and password.

Request: A solicitation from one user to another to be mutually added to each other's friends lists

1. Client Software Architecture Overview

1.1 Main Data Types and Structures

Client-side structures

```
struct game {
        int board[8][8];
        int count;              /* move count */
        int checkFlag;          /* 0 by default, 1 if in check, 2 if checkmate */
        int enPassFlag;         /* indicates that an enpass is possible */
        int enPassLoc[2];       /* indicates location that enpass would occur */
        int enPassCapt;         /* flag indicating enpass capture has been made */
        int castleFlag;         /* 0 for no castle, 1 for kingside castle */
        int lastMove[2];        /* holds dest of last move to highlight on printed game board */
        int pawnPromoFlag;      /* indicates to logfile that pawn has been promo */
        int AITurnFlag;         /* indicates AI turn */
        int AIThinkingFlag;     /* indicates AI not making move but calculating possible moves */
        int lateGame;           /* indicates late-game conditions have been met */
                                /* AI king now behaves differently */
        int captureFlag;        /* incidates to logfile that a capture has been made */
        char oppUsername[MAX_NAME_LEN];
        CAPTURES * captures;
};
```

```
struct captures {
        int whiteCount;
        int blackCount;
        char whiteCapt[17];
        char blackCapt[17];
};
```

GAME is the primary structure in MephistoSoul. It is initialized when a new game begins, and stores the current location of pieces on an 8x8 game board, the current move count, the locations of both kings, and a handful of flags that are used throughout the program. The lastMove[] array is used when highlighting the most recent move on the printed game board.

The GAME structure itself holds a CAPTURES structure, which contains a count of captured pieces for each color, and an character array of those pieces to display to the user. This structure is used to display captured pieces on the printed game board.

The ENTRY structure contains the x and y coordinates for the origin and destination, value of the piece, pointer to the List, and pointers to the previous and next entry.

```
typedef struct user USER;
```

```
struct user {
        char username[MAX_NAME_LEN];
        int FD;
        GAME * game;
        FRIENDLIST * friendList;
        int guestFlag; /* indicates whether or not the user is logged in */
};
```

This struct links together a user's username, the FD that the server has set for the user, a game struct if the user is in a game, the user's friend list, and a flag that indicates if the user has logged in or not.

```
typedef struct friendList FRIENDLIST;
```

```
struct FriendList {
        FENTRY *first;
        FENTRY *last;
        int length;
};
```

This struct is a linked list header for the FRIEND struct.

```
typedef struct frnd FRIEND;
```

```
struct frnd {
        char username[MAX_NAME_LEN];
        int count; // # of messages
        FRIENDLIST * list;
        FRIEND * next;
        FRIEND * prev;
        char message[300][MAX_MSG_LEN + MAX_NAME_LEN + 2];
};
```

This struct links together a user's friend's username, the number of messages they've exchanged with each other, and the messages that they've exchanged with each other.

```
typedef struct {
        int quit;
        char username[MAX_NAME_LEN];
        int FD;
} STARTFLAGS;
```

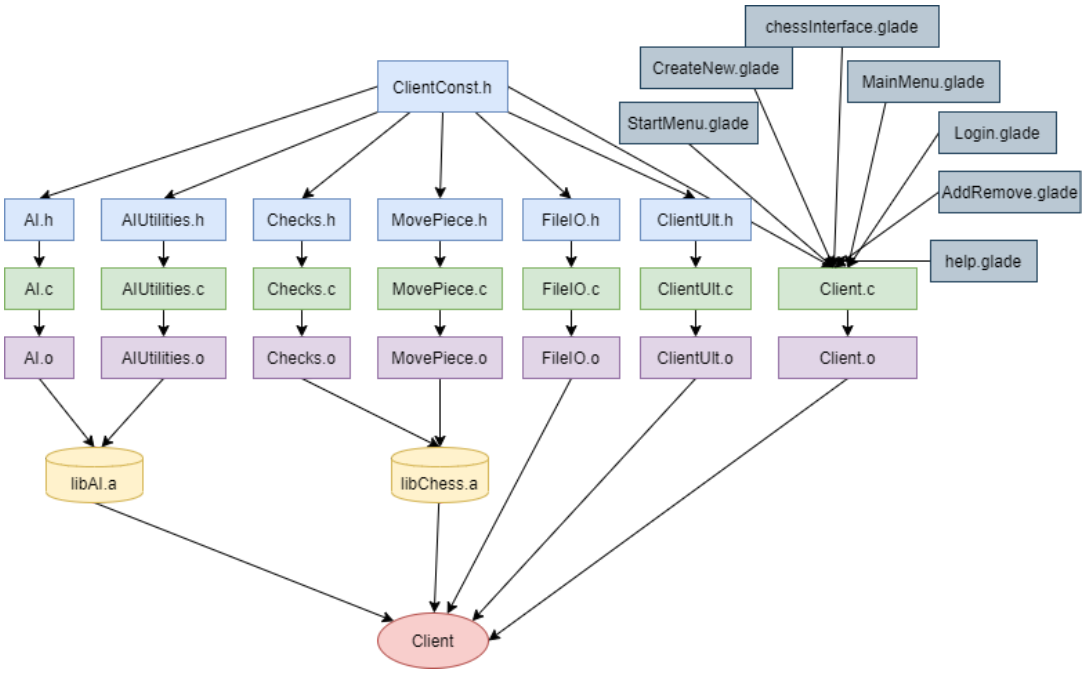
This struct passes in 3 flags for the creation of a new guest user.

```
typedef struct {
        int quit;
        int create;
        int undo;
        int returnToLogin;
        USER * user;
} FLAGPASS;
```

This struct passes 4 flags that handles the creation and quitting as a guest or as a logged in user and the user struct into the gtk functions.

1.2 Major Software Components

Diagram of Module Hierarchy



1.3 Client Module Interfaces

Checks.h

```
int CheckValid( int move[4], GAME * game);
int CheckDanger(int pieceX, int pieceY, int whiteFlag, int boardArr[8][8]);
int EndGame(GAME *game);
void PrintEndGame(int code, GAME *game);
int CheckCheck( GAME * game);
```

ClientUtil.h

```
USER * CreateUser(char * username);
void DeleteUser(USER * user);
FLAGPASS * CreateFlagpass();

void CreateFriendList(USER * user);
void DeleteFriendList(USER * user);
void AppendFriend (char * username, USER * user);
int RemoveFriend( char * username, USER * user);
void ListenForServer( USER * user);
void HandleResponse(USER * user, char * RBuff);
int CheckForErrors(char * buff);
```

GameFunctions.h

```
GAME * CreateGame(void);
void DeleteGame(GAME * game);
void InitializeNewGame(GAME * game);
void TwoPlayerGame(GAME * chessGame );
void OnePlayerPreGame(GAME* chessGame, int * color, int * diff);
void OnePlayerGame(GAME * chessGame, int color, int diff);
void AIvAIGame(GAME * chessGame);
```

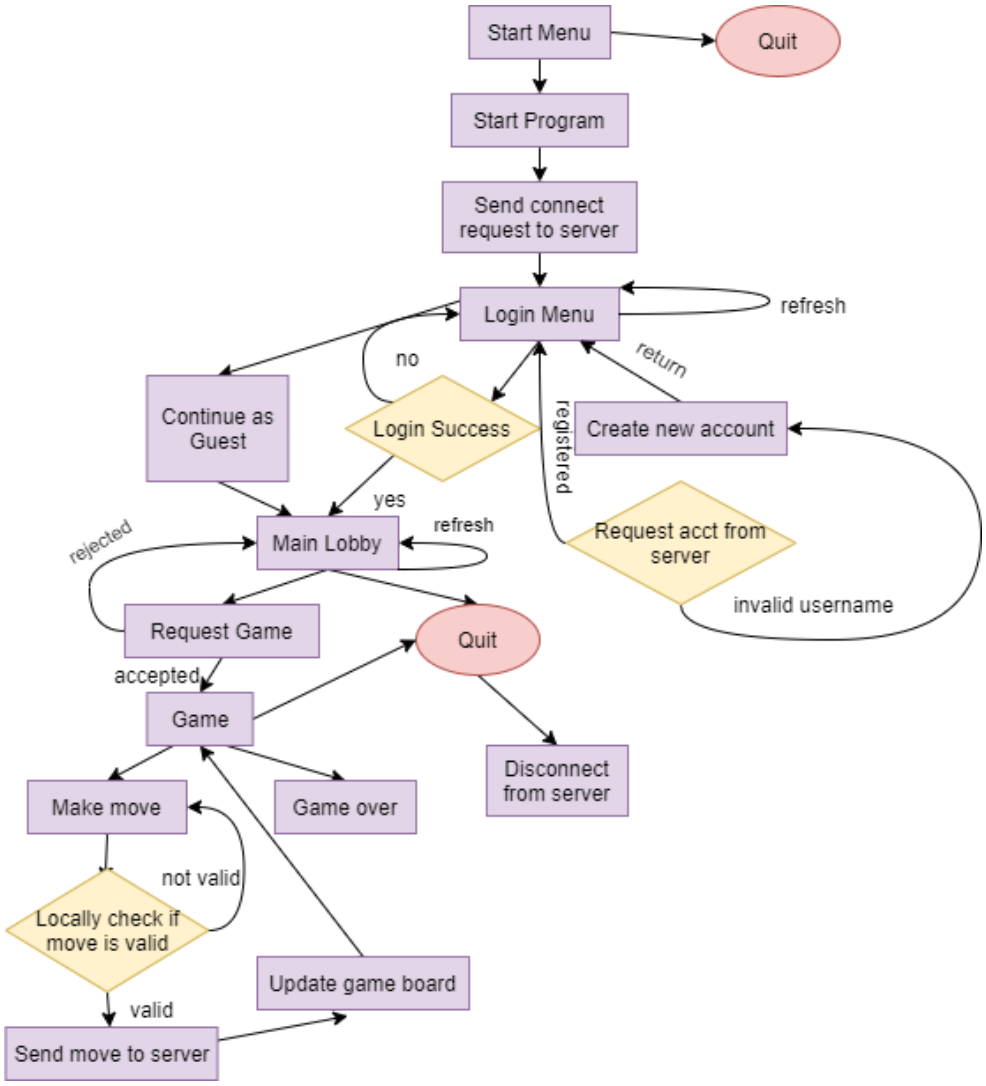
FileIO.h

```
void WriteToLog(GAME * game, char move[], int locMove[], FILE * svFile);
void ArrayToSmith( char smith[], int move[]);
void UndoMove(GAME *game);
int PieceToValue(char piece);
void ValueToPiece(int value, char piece[8]);
int FindSavedGame( char saveName[] );
void DeleteSavedGame( char saveName[]);
int PrintSavedGames(void);
int SaveGame( void );
GAME * LoadGame(char name[], int * AIcolor, int * diff);
```

MovePiece.h

```
int GetUserEntry( int locMove[], char userMove[], GAME * game);
int SmithToArray(char moveEntry[], int locMove[]);
void MovePiece( int move[], GAME * game, char promoFromLoad[8]);
void PrintErrorCode(int illegalMove, int orig, int count);
```

1.4 Overall program control flow



2. Server Software Architecture Overview

2.1 Main Data types and structures

```
typedef struct Account UENTRY;
```

```
struct Account {
        ULIST *list;
        UENTRY *prev;
        UENTRY *next;
        char username[USERLENGTH];
        char password[PASSLENGTH];
        char buffer[BUFFERSIZE];
        FLIST *friendlist;
};
```

The UENTRY struct holds the information about registered users, including username, password, and friends list.

```
typedef struct UserList ULIST;
```

```
struct UserList {
        UENTRY *first;
        UENTRY *last;
        int length;
};
```

The ULIST struct is the linked list header that holds all registered users.

```
typedef struct online ONLINE;
```

```
struct online {
        char * name;
        int FD;
        ONLINELIST * list;
        ONLINE * prev;
        ONLINE * next;
} ;
```

The ONLINE struct makes up the entries of the ONLINELIST linked list, and holds the file descriptors and corresponding username for connected clients.

```
typedef struct onlineList ONLINELIST;
```

```
struct onlineList {
        int length;
        ONLINE * first;
        ONLINE * last;
};
```

ONLINELIST is the linked list header for ONLINE structs.

```
typedef struct FriendList FLIST;
```

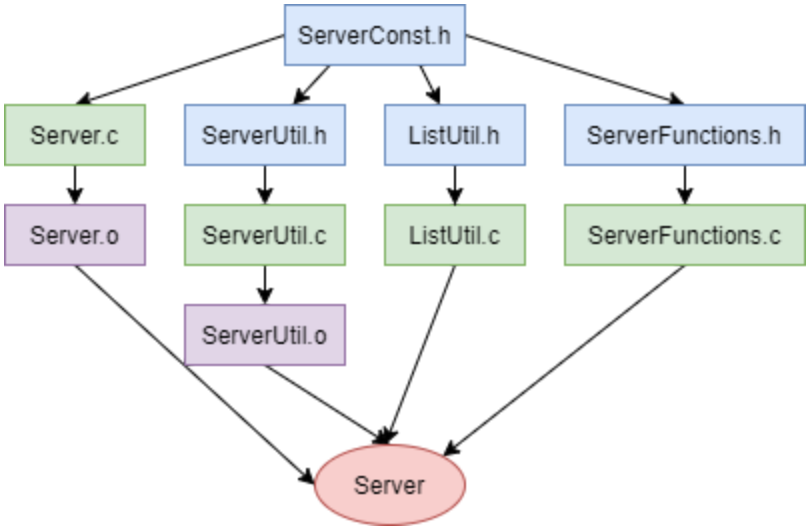
```
struct FriendList {
        FENTRY *first;
        FENTRY *last;
        int length;
};
```

FLIST holds a list of a user's friends.

```
struct Friend {
        FLIST *list;
        FENTRY *prev;
        FENTRY *next;
        char username[USERLENGTH];
};
```

FENTRY struct is the structure holding a user's friends. An FLIST contains all the user's registered friend usernames.

2.2 Major Software Components



2.3 Server Module Interfaces

ListUtil.h

```
ONLINE * CreateOnline(char username[], int FD);
int DeleteOnline(char username[], ONLINELIST * onlinelist);
ONLINELIST * CreateOnlinelist(void);
void DeleteOnlinelist(ONLINELIST * list);

UENTRY * CreateUEntry(char username[], char password[]);
int DeleteUEntry(char username[], ULIST * userlist);
ULIST * CreateUList(void);
void DeleteUList(ULIST * accountlist);

FENTRY * CreateFEntry(char username[]);
int DeleteFEntry(char username[], UENTRY * user);
FLIST * CreateFList(UENTRY * user);
void DeleteFList(UENTRY * user);

void AppendOnline(ONLINELIST * onlinelist, char username[], int FD);
void AppendUEntry(ULIST * accountlist, char username[], char password[]);
void AppendFEntry(char username[], UENTRY * user);
```

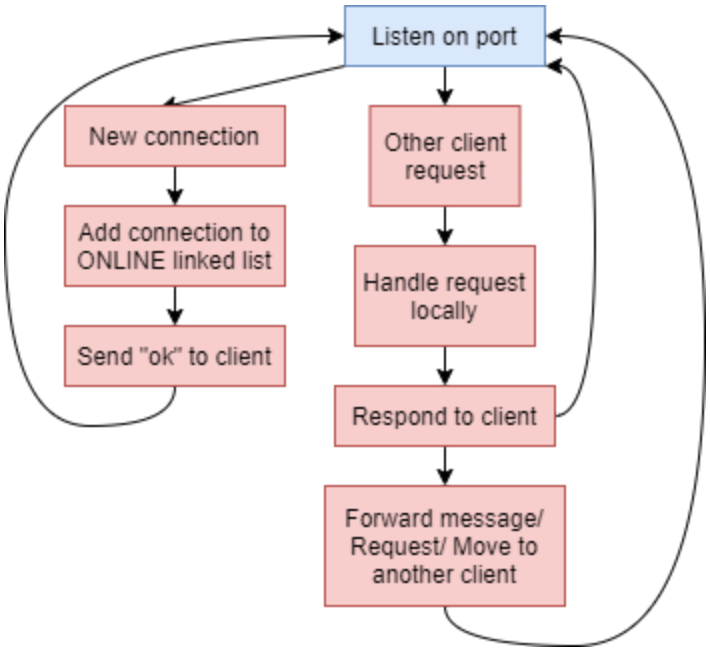
ServerUtil.h

```
int RegisterUser(ULIST *userlist, char *username, char *password, char *outbuffer);
int UsernameToFD(ONLINELIST *onlinelist, char *username);
int LoginUser(ULIST *userlist, char *username, char *password, char *outbuffer, ONLINELIST
* onlineList, int FD) ;

int ProcessFriend(ULIST *userlist, ONLINELIST *onlinelist, char *username, char *dstusernam
e, char *outbuffer, char *dstoutbuffer, int flag);

int ProccessGame(ULIST *userlist, char *username, char *dstusername, char *replyBuff, char
*forwardBuff, int flag);

int SendMessage(ULIST *userlist, char *username, char *dstusername, char *message, char *re
plyBuff, char *forwardBuff);

int ProcessStream(char *receive, ULIST *userlist, ONLINELIST * onlineList, int replyFD);

void HandleTimeout(void);
```

ServerFunctions.h

```
int MakeServerSocket( uint16_t PortNo );
void ProcessRequest( int DataSocketFD, ULIST *userlist, ONLINELIST * onlineList, fd_set * A
ctiveFDs);
void ServerMainLoop( ULIST *userList, ONLINELIST * onlineList, int Timeout);
```

2.4 Overall Server Program Control Flow



3. Installation

3.1 System Requirements

- 4 MB free disk space, minimal hardware requirements
- Operating system: Access to a Linux server via Windows or macOS

3.2 Setup and Configuration

- Binary release:
  - Download ‘Chat_Beta.tar.gz’ and XMing (Windows) or XQuartz (macOS)
  - Enable X11 forwarding on the terminal
  - Unpack with the command ‘tar -xvzf Chat_Beta.tar.gz’
  - Run the server executable with the command ‘bin/Server <portno>’
    - Ex: bin/Server 14222
    - Port number must be greater than 2000
    - It is recommended to choose a port number between 14000 and 14999
  - Run the client executable with the command ‘bin/Client <serveraddress> <portno>
    - ex: bin/Client bondi.eecs.uci.edu 14222
    - Port number must be the same as used for server port number
- Source code release:
  - Download ‘Chat_Beta_src.tar.gz’ and XMing (Windows) or XQuartz (macOS)
  - Enable X11 forwarding on the terminal
  - Unpack with the command ‘tar -xvzf Chat_Beta_src.tar.gz’
  - Make the executables with command ‘make’ or ‘make all’
  - Run the server executable with the command ‘bin/Server <portno>’
    - Ex: bin/Server 14222
    - Port number must be greater than 2000
    - It is recommended to choose a port number between 14000 and 14999
  - Run the client executable with the command ‘bin/Client <serveraddress> <portno>
    - ex: bin/Client bondi.eecs.uci.edu 14222

        - Port number must be the same as used for server port number

3.3 Building, compilation, installation

- Building
    - A combination of header files, source files, object files, a Makefile, and two executables were built to have MephistoSoul organized and run efficiently. We also separated it into /bin/, /src/, /doc/, /gui/, Makefile, INSTALL, README, and COPYRIGHT.
        - /src/ directory includes (present only in MephistoSoul source release)
            - Header files: Checks.h, ClientConst.h, ClientUtil.h, GameFunctions.h, FileIO.h, ListUtil.h, MovePiece.h, ServerConst.h, ServerFunctions.h, ServerUtil.h
            - Source files: Checks.c, Client.c, ClientMephisto.c, ClientUtil.c, FileIO.c, GameFunctions.c, ListUtil.c, MovePiece.c, Server.c, ServerFunctions.c, ServerUtil.c
        - /bin/ directory include (present only in MephistoSoul end user release)
            - Server
            - ClientMephisto
        - /doc/ directory includes
            - P2_UserManual.pdf
            - P2_SoftwareSpec.pdf
        - /gui/ directory includes
            - All GTK related .glade files
            - All .png and image files used in Mephisto Soul
        - Makefile
        - INSTALL: Directions on how to install MephistoSoul
        - README: Directions on how to run MephistoSoul
        - COPYRIGHT: Copyright license
- Compilation
    - In the Makefile, a “make” target complies the executable programs for the server: Server and ClientMephisto.
- Installation
    - To install MephistoSoul, download the “Char_Beta_src.tar.gz” and unpack, and follow “INSTALL”. Then, simply run the program on PuTTY or any Linux terminal with the gcc compiler.

4. Documentation of packages, modules, interfaces

4.1 Detailed description of data structures

Client-side structures

```
struct game {
        int board[8][8];
        int count;              /* move count */
        int checkFlag;          /* 0 by default, 1 if in check, 2 if checkmate */
        int enPassFlag;         /* indicates that an enpass is possible */
        int enPassLoc[2];       /* indicates location that enpass would occur */
        int enPassCapt;         /* flag indicating enpass capture has been made */
        int castleFlag;         /* 0 for no castle, 1 for kingside castle */
        int lastMove[2];        /* holds dest of last move to highlight on printed game board */
        int pawnPromoFlag;      /* indicates to logfile that pawn has been promo */
        int AITurnFlag;         /* indicates AI turn */
        int AIThinkingFlag;     /* indicates AI not making move but calculating possible moves */
        int lateGame;           /* indicates late-game conditions have been met */
                                        /* AI king now behaves differently */
        int captureFlag;        /* incidates to logfile that a capture has been made */
        char oppUsername[MAX_NAME_LEN];
        CAPTURES * captures;
};
```

```
struct captures {
        int whiteCount;
        int blackCount;
        char whiteCapt[17];
        char blackCapt[17];
};
```

GAME is the primary structure in MephistoSoul. It is initialized when a new game begins, and stores the current location of pieces on an 8x8 game board, the current move count, the locations of both kings, and a handful of flags that are used throughout the program. The lastMove[] array is used when highlighting the most recent move on the printed game board.

The GAME structure itself holds a CAPTURES structure, which contains a count of captured pieces for each color, and an character array of those pieces to display to the user. This structure is used to display captured pieces on the printed game board.

```
typedef struct user USER;
```

```
struct user {
        char username[MAX_NAME_LEN];
        int FD;
        GAME * game;
        FRIENDLIST * friendList;
        int guestFlag; /* indicates whether or not the user is logged in */
};
```

This struct links together a user's username, the FD that the server has set for the user, a game struct if the user is in a game, the user's friend list, and a flag that indicates if the user has logged in or not.

```
typedef struct friendList FRIENDLIST;
```

```
struct FriendList {
        FENTRY *first;
        FENTRY *last;
        int length;
};
```

This struct is a linked list header for the FRIEND struct.

```
typedef struct frnd FRIEND;
```

```
struct frnd {
        char username[MAX_NAME_LEN];
        int count; // # of messages
        FRIENDLIST * list;
        FRIEND * next;
        FRIEND * prev;
        char message[300][MAX_MSG_LEN + MAX_NAME_LEN + 2];
};
```

This struct links together a user's friend's username, the number of messages they've exchanged with each other, and the messages that they've exchanged with each other.

```
typedef struct {
        int quit;
        char username[MAX_NAME_LEN];
        int FD;
} STARTFLAGS;
```

This struct passes in 3 flags for the creation of a new guest user.

```
typedef struct {
        int quit;
        int create;
        int undo;
        int returnToLogin;
        USER * user;
} FLAGPASS;
```

This struct passes 4 flags that handles the creation and quitting as a guest or as a logged in user and the user struct into the gtk functions.

Chess Pieces:

Integers in an 8x8 array are used to represent pieces on the game board.
White pieces are denoted with a positive integer, while black pieces are negative integers.

White		Black	
Pawn (unmoved)	11	Pawn (unmoved)	-11
Pawn	12	Pawn	-12
Rook (unmoved)	50	Rook (unmoved)	-50
Rook	51	Rook	-51
Knight	30	Knight	-30
Bishop	35	Bishop	-35
Queen	100	Queen	-100
King (unmoved)	900	King (unmoved)	-900
King	901	King	-901

The following diagram shows the game board as it is printed to the terminal (right) and the corresponding contents of the GAME->board 8x8 array.

Server-side structures

```
typedef struct online ONLINE;
```

```
struct online {
        char name[21];
        int FD;
        ONLINELIST * list;
        ONLINE * prev;
        ONLINE * next;
} ;
```

The ONLINE struct makes up the entries of the ONLINELIST linked list, and holds the file descriptors and corresponding username for connected clients.

```
typedef struct onlineList ONLINELIST;
```

```
struct onlineList {
        int length;
        ONLINE * first;
        ONLINE * last;
} ;
```

ONLINELIST is the linked list header for ONLINE structs.

```
typedef struct UserList ULIST;
```

```
struct UserList {
        UENTRY *first;
        UENTRY *last;
        int length;
};
```

ULIST is the linked list header for UENTRY structs.

```
typedef struct Account UENTRY;
```

```
struct Account {
        ULIST *list;
        UENTRY *prev;
        UENTRY *next;
        char username[USERLENGTH];
        char password[PASSLENGTH];
        char buffer[BUFFERSIZE];
        FLIST *friendlist;
};
```

This struct links together a client's username, password, friend's list, and message buffer.

```
typedef struct FriendList FLIST;
```

```
struct FriendList {
        FENTRY *first;
        FENTRY *last;
        int length;
};
```

FLIST is the linked list header for FENTRY structs.

```
typedef struct Friend FENTRY;
```

```
struct Friend {
        FLIST *list;
        FENTRY *prev;
        FENTRY *next;
        char username[USERLENGTH];
};
```

This struct links together a user's friends and their usernames.

4.2 Detailed description of functions and parameters

Function prototypes and brief explanation

Checks.c

`int CheckValid( int move[4], GAME * game)`

Takes in a move array indicating origin and destination, and the GAME structure. Given the current board array, the function finds the piece at the origin, and determines if the piece is moving to a valid destination. It returns 0 for valid moves or an integer responding to an error code if move is found to be illegal.

`int CheckDanger(int pieceX, int pieceY, int whiteFlag, int boardArr[8][8])`

Takes in a location on the board (x and y), a flag indicating turn color, and an 8x8 board array. CheckDanger runs through squares radially from the piece's position, returning 1 if there is an enemy piece attacking the square. It returns 0 if the board position is not under immediate attack.

`int EndGame(GAME * game)`

EndGame takes in a pointer to the current GAME struct and checks for end-game conditions: checkmate, stalemate, and draw. If no end-game condition is found, EndGame returns 0, otherwise a positive integer is returned (1 if checkmate, 2 if stalemate, 3 if draw).

`void PrintEndGame(int code, GAME * game)`

PrintEndGame takes in the integer returned from EndGame and a pointer to the GAME structure, printing out the end-game condition along with a winner (if checkmate condition is met).

`int CheckCheck(GAME * game)`

Checks if the king is in check or not.

ClientMephisto.c

`int main( int argc, char * argv[])`
The main control function of the Client-side MephistoSoul program. From within main, the Client initialized the program, listens for GTK signals, and calls appropriate functions depending on signals.

`All GUI files will go here:`
`GSourceFunc Timeout (gpointer user_data )`
Prints out when the client is timed out from the server.
`void EndProgram( USER * user )`
Closes the FD associated with a user and deallocates the users structs associated with the user.
`void on_ProgStart_clicked (GtkButton *b)`
Hides the Start Window and sends CONNECT to the server to get a guest username from the server.
`void on_ProgQuit_clicked (GtkButton *b)`
Hides the Start Window and sets quitLogin to 1 so the program can end.
`void on_loginButton1_clicked( GtkButton *b)`
Closes the Login window so the user can be prompted to create an account
`void on_loginButton2_clicked( GtkButton *b)`
Closes the Login window and sets create and returnToLogin to 0 so the user can play as guest
`void on_loginButton3_clicked( GtkButton *b)`
Reads the username and password entries to login in with the server. If the login is failed, then “Incorrect username or password. Please try again.” message is shown. Else, it is successful and then the Login window closes and sets guestFlag, returnToLogin, create to 0. Their username is shown in the Main Menu window.
`void on_newButton1_clicked( GtkButton *b)`
When create an account is clicked, the new username and password entries are read. If the passwords don't match, the user is prompted again. Else, the client sends the username and password to the server to REGISTER. The server responds with NOUSR if an account under the same username is already found or OKUSR to close CreateNewWindow and set returnToLogin to 1.
`void on_newButton2_clicked( GtkButton *b)`
Closes create new window and sets returnToLogin to 1 so the the user can return to main menu without logging in.
`void on_mainButton1_clicked( GtkButton *b)`
Shows login window.
`void on_mainButton2_clicked( GtkButton *b)`
The user can logout once the client sends LOGOUT and the username to the server. Then, the user is set as a guest username from the server.

`void on_mainButton3_clicked( GtkButton *b)`
The play game window is displayed.

`void on_sendIMButton_clicked( GtkButton *b)`
Sends a message to the server by reading the user's message entry and sending it to the server. The server responds with the other user's message.

`void on_playGameButton_clicked (GtkButton *button)`
When new game is clicked, the user is prompted for their opponent's user name. Then, the client sends a game request from the user to the opponent. If the game request responds "NOGAMEREQ", the username enter is not found or the game request was denied. If the server responds with "OKGAMEREQ", the chess interface window pops up to play a game with their requested opponent.

`void on_AddRemoveButton_clicked (GtkButton *button)`
When the add/remove button is clicked, another window prompting for the friend's username to be added or removed.

`void on_addButton_clicked (GtkButton *button)`
Reads the user's entry for username and sends a FRREQ with the user's and friend's username to the server. If the server responds with OKFREQ then the friend is added to their friend's list and returned to the main menu window. Else, a "no username not found. Please try again." message is displayed.

`void on_removeButton_clicked (GtkButton *button)`
Reads the user's entry for username and searches through the user's friend's list. If the friend is found, the friend is removed. Else, a "Friend not found. Please try again." message is displayed.

`void on_returnToGameButton_clicked (GtkButton *button)`
Closes the help window and opens the chess window.

`void on_refreshButton_clicked (GtkButton *button)`
When refresh is clicked, REFRESH is sent to the server for an update. If the server's response is one of the protocols (FRREQ, SHUTDOWN, etc), the client responds according. Else, the main menu and chess window remains the same.

`void on_QuitButton_clicked (GtkButton *button)`
When the quit button is clicked, all windows are closed and the client disconnects from the server to shutdown.

`void on_HelpButton_clicked (GtkButton *button)`
When the help button is clicked, another window will pop up detailing how to use the chess/instant messaging interface.

`void on_acceptGameButton_clicked(GtkButton *button)`
The user accepts the game request so GAMEACCPT is sent to the server. Once the server responds with OKGM, the chessFlag is set to 1.

`void on_rejectGameButton_clicked(GtkButton *button)`
Closes the game request window.

`void on_acceptButton_clicked (GtkButton *button, gpointer user)`

Closes the friend request window and sends FRACPT to the server. The friend's username is added to their friend's list.

`void on_rejectButton_clicked(GtkButton *button)`

Closes the friend request window.

`void on_returnToLobbyButton_clicked (GtkButton *button)`

When the return to lobby button is clicked, it returns to the lobby menu.

`void on_a1button_clicked (GtkButton *button)`

When the a1 position on the board is clicked, it adds the board position to the position array.

For each position (64) on the board, there is a corresponding button function that returns its position.

ClientUtil.c

`USER * CreateUser()`

Allocates memory for the USER structure, returning an initialized empty USER *.

`void DeleteUser(USER * user)`

Frees memory for the USER struct (and the FRIENDLIST struct within).

`FLAGPASS * CreateFlagpass()`

Creates FLAGPASS struct

`void CreateFriendList(USER * user)`

Allocates memory for the FRIENDLIST structure, pointing the friendList member of the user argument to the allocated memory.

`void DeleteFriendList(USER * user)`

Frees memory for the FRIENDLIST struct.

`void AppendFriend(char * username, USER * user)`

Adds a friend to the list of friend's for a user.

`int RemoveFriend(char * username, USER * user)`

Frees memory for when you remove a friend from your friend's list.

`void AddMessage(char * username, USER * user, char * message)`

Adds message sent to both user's message history.

```
void ListenForServer(USER * user)
```
Listens for the server's response by checking what command is in the recv buffer.

```
void HandleResponse(USER * user, char * RBuff)
```
This function responds to what message is sent from the server to the client. For example, if LOGINSUCESS is sent, then the client will populate it's account struct with the username that the server sent along with the message.

```
int CheckForError(char * buff)
```
This function checks the buffer to see if the client sent a valid command to the server, if not then it will prompt the client to send a valid command.

GameFunctions.c

```
GAME * CreateGame(void)
```
Allocates memory for the GAME structure, returning an initialized empty GAME *.

```
void DeleteGame(GAME * game)
```
Frees memory for the GAME struct (and the CAPTURES struct within).

```
void InitializeNewGame(GAME * game)
```
Resets all flags in the GAME struct and initializes the board element of GAME struct with a new chess board.

ListUtil.c

```
ONLINE * CreateOnline(char username[], int FD)
```
Allocates memory for when a user comes online and creates a struct that links the user's name with the FD that the server provides it.

```
int DeleteOnline(char username[], ONLINELIST * onlinelist)
```
Frees memory for ONLINE struct when a user goes offline.

```
ONLINELIST * CreateOnlinelist(void)
```
Allocates memory for a linked list header to hold the list of online users.

```
void DeleteOnlinelist(ONLINE * list)
```
Frees up memory for ONLINE structs when the server is shutdown.

```
UENTRY * CreateUEntry(char username[], char password[])
```
Allocates memory for when a user creates an account or logs in as a guest and creates a struct that holds username and password for a user.

```
int DeleteUEntry(char username[], ULIST * userlist)
```
Frees up memory for UENTRY when a guest logs off, user is deleted, or when the server shuts down.

```
ULIST * CreateUList(void)
```
Allocates memory for a linked list header to hold the list of all users: on or offline.

```
void DeleteUList(ULIST * accountlist)
```
Frees up memory for ULIST and all the UENTRY's in the list.

```
FENTRY * CreateFEntry(char username[])
```
Allocates memory for when a user accepts a friend request.

```
int DeleteFEntry(char username[], UENTRY * user)
```
Frees up memory for when a user deletes a friend from their friend's list or when the server shuts down.

```
FLIST * CreateFList(UENTRY * user)
```
Allocates memory for a user's friendlist, FLIST.

```
void DeleteFList(UENTRY * user)
```
Frees up memory that is taken by FLIST and the FENTRYs that are in the list.

```
void AppendOnline(ONLINE * onlinelist, char username[], int
FD)
```
When a user comes online, this function calls CreateOnline and adds the user to the ONLINELIST.

```
void AppendUEntry(ULIST * accountlist, char username[],
char password[])
```
When a guest comes online or when a user account is created, this function adds the guest/user to the list of all users, ULIST.

```
void AppendFEntry(char username[], UENTRY * user)
```
When a user adds a friend, this function calls CreateFEntry and adds the friend to the list of friends for both users in their friend list.

MovePiece.c

```
void PrintErrorCode(int illegalMove, int orig, int count)
```
PrintErrorCode takes in the return value of CheckValid (in CheckValidMove.c) and prints the error code corresponding to the error of the attempted move.

```
void SmithToArray(char moveEntry[], int locMove[])
```
SmithToArray takes in a user-entered move string in Smith notation and translates it into an integer array of length 4 in the form (origin x, origin y, destination x, destination y).

```
void MovePiece( int move[], GAME * game)
```
MovePiece takes in the translated move array and GAME struct pointer and updates the board. If the move involves a capture (including en passant), the game->captures elements are updated, incrementing the capture count and adding the captured piece to an array of captured pieces. If the move leads to a pawn promotion, the user is prompted to provide the desired promotion and the board is updated accordingly. MovePiece() handles en passant captures and castling as well.

Server.c

```
int main( int argc, char * argv[])
```
The main control function of the Server-side MephistoSoul program. From within main, the Server initializes the program, listens on a user-specified port, and handles received client requests accordingly.

ServerFunctions.c

```
int MakeServerSocket( uint16_t PortNo)
```
A function adapted from Professor Doemer's ClockServer.c. Takes in the user-given port number and returns a corresponding file descriptor.

```
void ProcessRequest( int DataSocketFD, ULIST * userlist,
ONLINELIST * onlineList, fd_set * ActiveFDs)
```
A function adapted from Professor Doemer's ClockServer.c. Takes in the communicating file descriptor, the server's lists of ONLINE connections and registered users (ULIST), and the set of active file descriptors. Depending on the information read from the DataSocketFD, ProcessRequest() calls ProcessStream() to handle the client request or closes the connection with the client.

```
void ServerMainLoop( ULIST * userList, ONLINELIST *
onlineList, int Timeout)
```
A function adapted from Professor Doemer's ClockServer.c. Takes in the server's list of ONLINE connections (ONLINELIST), registered users (ULIST), and a timeout value (in microseconds). From within this function, the Server program loops in execution, ending only when it hears a SHUTDOWN command from a client.

ServerUtil.c

```
int UsernameToFD(ONLINELIST *onlinelist, char *username)
```

Takes in a username, searches the ONLINELIST for the username and returns the corresponding file descriptor

```
int RegisterUser(ULIST *userlist, char *username, char
*password, char *replyBuff)
```

Searches every UENTRY in the ULIST to see if there is a matching username

```
int LoginUser(ULIST *userlist, char *username, char
*password, char *replyBuff, ONLINELIST * onlineList, int
FD)
```

Searches the ULIST for the username and checks if the password matches. Sends LOGINSUCCESS or LOGINFAILED to the client by writign to the outbuffer.

```
int Logout(ONLINELIST * online, char * UN, char *replyBuff)
```

This function replaces ONLINE username with a guest username.

```
int ProcessFriend(ULIST *userlist, ONLINELIST *onlinelist,
char *fromUN, char *toUN, char *replyBuff, char
*forwardBuff, int flag, int * fwdFD);
```

Handles the commands FRREQ, FRACPT, FRREJ depending on the flag. Updates the friends list for the two users. Writes a response to both clients in two buffers depending on the command.

```
int ProccessGame(ONLINELIST * onlinelist, ULIST * userlist,
char * fromUN, char * toUN, char * replyBuff, char *
forwardBuff, int flag);
```

Handles the commands GAMEREQ, GAMEACPT, and GAMEREJ depending on the flag. Writes a response to both buffers to indicate the clients should begin a game on their end or to notify the user that a game request has been sent or received.

```
int SendMessage(ULIST * userlist, char * fromUN, char *
toUN, char * message, char * replyBuff, char *
forwardBuff);
```

Sends client 1 ‘OK’ to their reply buffer so they know the message has been received and sends client 2 the message from client 1.

```
int ProcessStream(char * receive, ULIST * userlist,
ONLINELIST * onlineList, int replyFD, fd_set * ActiveFDs);
```

Handles the commands sent to the server from the client. For example, if the command LOGIN is sent, then the server will change the guest

username to the login username and allocate memory for the user's friendlist.

```
void HandleTimeout(void);
```

Prints out a spinning wheel on the server side.

4.3 Detailed description of the communication protocol

```
- Client Request

        <request> ::= CONNECT
                    |   REFRESH
                    |   REFRESH_FR <fromUN>
                    |   REGISTER <fromUN> AND <password>
                    |   LOGIN <fromUN> AND <password>
                    |   LOGOUT <fromUN>
                    |   MOVE <fromUN> AND <toUN> AND<move>
                    |   IM <fromUN> AND <toUN> AND <message>
                    |   FRREQ <fromUN> AND <toUN>
                    |   FRACPT <fromUN> AND <toUN>
                    |   FRREJ <fromUN> AND <toUN>
                    |   FRRM <fromUN> AND <toUN>
                    |   GAMEREQ <fromUN> and <toUN>
                    |   GAMEACPT <fromUN> AND <toUN>
                    |   GAMEREJ <fromUN> AND <toUN>
                    |   QUIT_GAME <fromUN> AND <toUN>
                    |   DISCONNECT <fromUN>
                    |   QUIT_SERVER

- Server Response

        <response>::= OKCN <fromUN>
                    |   OK_REFRESH <onlineuser1> AND <onlineuser2>
                    |   OK_REFRESH_FR <friend1> AND <friend2>
                    |   OKUSR <fromUN>
                    |   NOUSR <fromUN>
                    |   LOGINSUCCESS <fromUN>
                    |   LOGINFAILED
                    |   OK_LOGOUT
                    |   MSG <fromUN> <message>
                    |   OKFRREQ <fromUN>
                    |   OKFR <fromUN>
                    |   OKRM <fromUN>
                    |   OKGM <fromUN>
                    |   NOGM <fromUN>
                    |   OK_QUIT
                    |   OK_DISCONNECT
                    |   SHUTDOWN
```

<fromUN> and <toUN> mean from Username and to Username. Most commands that the client sends include a unique username to indicate where the request is coming from. Commands that are client to client communication indicate the destination of the message as the second parameter so the server can properly forward the message.

Initially, client requests to CONNECT and the server responds OKCN with a temporary guest number to indicate that it has accepted the connection. REGISTER or LOGIN are sent and the server will either create an entry for the new client, or look up if it has the user already stored in its database. Clients can send friend requests, accept or reject requests and the protocol specifies that the client must indicate the source and destination as well as the message or move that is passed to the client they are chatting with or playing a game of chess with. Game requests work similar to friend requests.

On the client side with server responses, OKUSR, NOUSR, and DUPUSR indicate the username is either okay, not processed (some other error occurred) or a duplicate and the user needs to try again. The ONLINE response followed by friends is so the client side can populate its information about a stored user that the server holds such as the friends list (these must be filled again every time the client logs in and out). OKMSG followed by who sent the message and what message they sent is parsed by the client to display to the recipient. Finally, ERROR <message> prompts the user to try their action again as the server did not properly receive it or did not know how to handle the request.

CONNECT
Before communication with the server, it is necessary for the user to use the command: CONNECT to establish a connection with the server and retrieve a temporary guest username. The username will be returned so that, if the user desires to use the program without logging in, they are still able to communicate with other users.

REGISTER <username> <password>
To register a username and password with the server, use the REGISTER command along with the desired username (20 or fewer characters, no spaces) and password (20 or fewer characters, no spaces). This must be done before logging in with the username. If successful, the server will respond OKUSR.

LOGIN <username> <password>
To login to a registered account, use the LOGIN command along with username and corresponding password. If login is successful, the server will respond LOGINSUCCESS <username>. Otherwise, the server will respond LOGINFAILURE.

LOGOUT <username>
To logout from an account, use the LOGOUT <username> command.

REFRESH
In order to capture unsolicited messages and requests from other users, use the REFRESH command to check any waiting messages.

IM <fromUsername> <toUsername> <message>
To send a message, use this format.

Development plan and timeline

5.1 Partitioning of tasks

- Week 8: user manual, understanding the client/server, GTK, and Glade
- Week 9: converting the TUI chess interface to a GUI chess interface, client/server communication
- Week 10: client GUI communicating with the server, chess GUI implementation

5.2 Team member responsibilities

Aammarah Idris:
- Client/Server Communication

Darlena Nguyen:
- Converting the TUI chess interface to a GUI chess interface
- Add Remove, Help, Friend Request, Game Result GUI

Kyle Mach:
- Client/Server Communication

Kiefer Daniel:
- Client/Server Communication

Sofia Bernstein:
- Start, Main Menu, Login Menu, & Create New User GUI
- Client/Server Communication

Copyright

References

“GTK+ 2 Reference Manual.” *GTK+ 2 Reference Manual: GTK+ 2 Reference Manual*, developer.gnome.org/gtk2/stable/.

“A User Interface Designer.” *Glade*, glade.gnome.org/.

Index