Software Specification
for
Kings of Chess

Version 2.1

Team 15
Aviraj Mukherjee
Martin Alexander Gomez
Henry Thy Bendiksen
Sarthak Sharma
Rebecca Ko

University of California, Irvine 2020

Table of Contents

1. Client Software Architecture Overview.....3
1.1. Main Data Structure Types.....3
1.2. Major Software Components.....4
1.3. Module Interfaces.....5
1.4. Overall Program Control Flow.....9
2. Server Software Architecture Overview.....10
2.1. Main Data Structure Types.....10
2.2. Major Software Components.....12
2.3. Module Interfaces.....13
2.4. Overall Program Control Flow.....17
3. Installation.....18
3.1. System Requirements, Compatibility.....18
3.2. Setup and Configuration.....18
3.3. Building, Compilation, Installation.....18
4. Documentation of Packages, Module, Interfaces.....19
4.1. Protocols Used and Called.....19
4.2. Data Structures.....19
4.3. Functions and Parameters.....21
4.4. Input and Output Formats.....23
5. Development Plan and Timeline.....24
5.1. Partitioning of Tasks.....24
5.2. Team Member Responsibilities.....24
6. Copyright.....26
7. References.....27
8. Index.....28

Glossary

Algorithm - is a sequence of well-defined instructions that allows a computer to process and produce an output.

Control Flow - is the order in which a program's execution components are connected.

Data Type - is the type of data a variable can be or hold and is usually based on storage capabilities such as integer, floating point, or character.

Directory - is a special type of file in Linux that contains names of other files and corresponding inodes for everything it contains.

Function - a set of statements that perform a task together when the program runs. Usually functions have a return type and parameters when declared.

Graphical User Interface (GUI) - is a type of interface in which a person will interact with that has icons to click on instead of being text-based and having to type out commands.

Inode - is a data structure on Linux that stores information about a file or directory. However, it does not store the actual data and name of the file.

Linked List - is a collection of data elements, instead of being referred to in a program by index there is a collection of nodes that point from one piece of data to the next.

Node - is a basic unit for data points in a larger network or program.

Parameter - is a part of function definition and includes data that is needed for the function to run and typically follows the initial declaration of a function in parentheses.

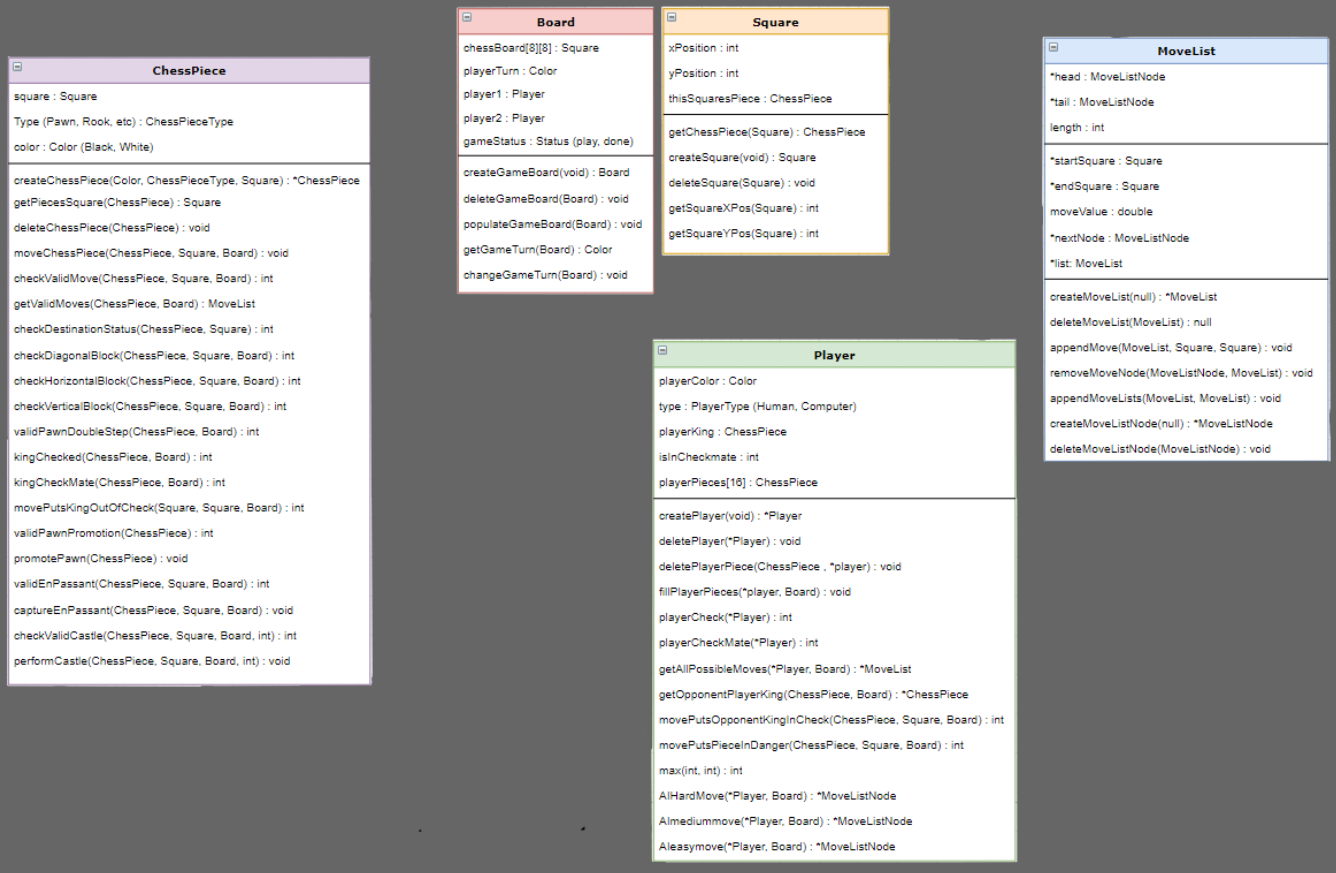
Software Architecture - refers to the fundamental structures of a system and the creation of the structures

Structure - is a user defined data type in C that allows a combination of different types of variables.

1 Client Software Architecture Overview

1.1 Main Data Structure Types

1.1.1 Data Structure Diagram



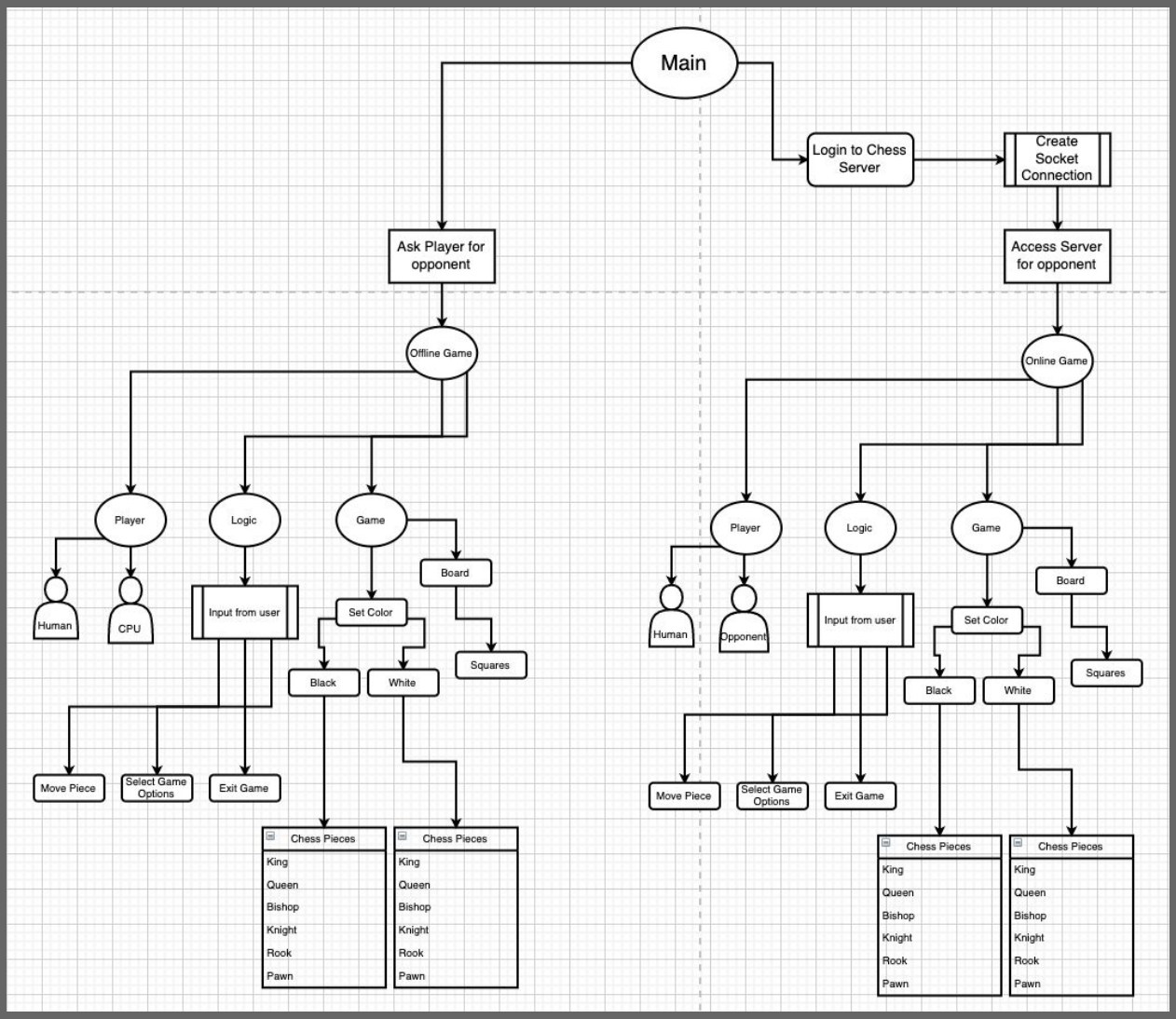
The above diagram shows each of the Data Structures in our program, and descriptors of each of the functions associated with each element. Mainly this program uses Enums, Linked List, and Structs to complete its functionality.

- Enum GRID
- Structs
  - GameBoard
  - MoveList
  - Player
  - ChessPieces
  - Square
  - Message

- User
- UserList

1.2 Major Software Components

1.2.1 Module Hierarchy Diagram



1.3 Module Interfaces

1.3.1 API of Major Modules:

A holistic list of functions allowing for interactions with major modules: PLAYER and PIECES

Player:

Player *createPlayer(Color color, PlayerType type); : Creates a new Player

void deletePlayer(Player *player); : Deletes a player struct

int login(Player *player, char *username, char *password); : Logs In a User

int sendMessage(char *message, Player *sender); : Sends a message to the Opponent

void deletePlayerPiece(ChessPiece *piece, Player *player); : Deletes a Player’s piece from player->playerPieces[]

void fillPlayerPieces(Player *player, Board *gameBoard); : after populating the gameboard with the pieces, assign the pieces to the player that owns them

int playerCheck(Player *player); : returns 1 if player's king is in check, 0 otherwise

int playerCheckMate(Player *player); : returns 1 if player's king is in checkmate, 0 otherwise.

MoveList *getAllPossibleMoves(Player *player); : returns a list of all of a player’s valid moves

ChessPiece *getOpponentPlayerKing(ChessPiece *piece, Board *gameBoard); : returns opponent player’s king

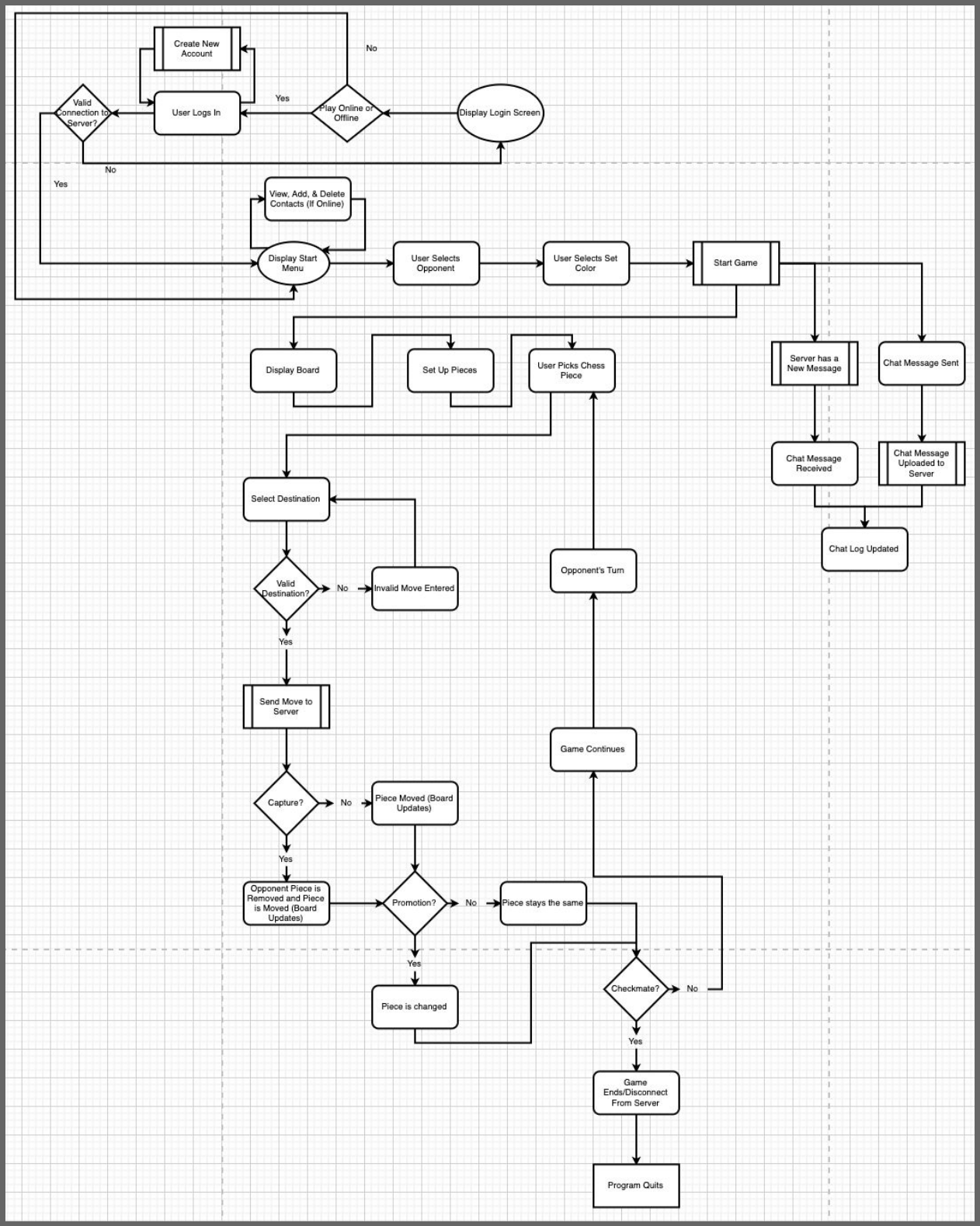
int movePutsOpponentKingInCheck(ChessPiece *piece, Square *endSquare, Board *gameBoard); : for chess AI to check to see if a move will put the players king in check, 1 if true, 0 if false

```
int movePutsPieceInDanger(ChessPiece *piece, Square *endSquare,
      Board *gameBoard); : for chess AI, checks to see if move
      will put piece in unprotected square

int max(int a, int b); : helper function to find max of two
      numbers(used for linked list lengths)

MoveListNode *AIHardMove(Player *player, Board *gameBoard);

MoveListNode *AImediummove(Player *player, Board *gameBoard);

MoveListNode *AIeasymove(Player *player, Board *gameBoard);


Pieces:

ChessPiece *createChessPiece(Color color, ChessPieceType
      pieceType, int xPosition, int yPosition); : allocate
      memory for chess piece

King *createKingPiece(Color color); : allocate memory for a
      King piece, and set the default value for isInCheck to 0

int getPieceXPos(const ChessPiece *piece); : return file
      (column) of the piece

int getPieceYPos(const ChessPiece *piece); : return rank (row)
      of the piece

void deleteChessPiece(ChessPiece *piece); : deallocate the
      memory for a chess piece

void deleteKingPiece(ChessPiece *piece); : deallocate the
      memory for a king piece

void moveChessPiece(ChessPiece *piece, Square *endSquare);

int checkValidMove(const ChessPiece *piece, const Square
      *endSquare); : returns 1 if the move is valid, 0
      otherwise
```

```
MoveList getValidMoves(const ChessPiece *piece);   //return a
      list of all the valid moves for a piece

int checkDestinationStatus(const ChessPiece *piece, const
      Square *destinationSquare); : return 0 if destination is
      unoccupied, 1 if destination has the opposite color piece
      on it, -1 if same color

int checkDiagonalBlock(const ChessPiece *piece, const Square
      *endSquare, const Board *chessBoard); : return 1 if a
      piece blocks diagonal movement, 0 if there is no block

int checkHorizontalBlock(const ChessPiece *piece, const Square
      *endSquare, const Board *chessBoard);

int checkVerticalBlock(const ChessPiece *piece, const Square
      *endSquare, const Board *chessBoard);

int validPawnDoubleStep(const ChessPiece *piece, const Board
      *chessBoard); : return 1 if a pawn can perform the double
      step, 0 if it can not

int kingChecked(ChessPiece *piece, Board *chessBoard); :
      returns 1 if player's king is in check, 0 otherwise

int kingCheckMate(ChessPiece *piece, Board *chessBoard); :
      returns 1 if player's king is in checkmate, 0 otherwise

int movePutsKingOutOfCheck(Square *startSquare, Square
      *endSquare, Board *gameBoard); : returns 1 if move will
      put the king out of check

int validPawnPromotion(const ChessPiece *piece); : return 1 if
      pawn can be promoted, 0 if it can not

void promotePawn(const ChessPiece *piece); : promote pawn if
      validPawnPromotion returns 1

int validEnPassant(ChessPiece *piece, Square *endSquare, Board
      *chessBoard); : checks for valid en passant, returns 1 if
      valid and 0 if invalid
```

```
void captureEnPassant(ChessPiece *piece, Square *endSquare,
      Board *gameBoard); : moves pawn to diagonal position and
      captures pawn that is passed

int checkValidCastle(ChessPiece *piece, Square *endSquare,
      Board *gameBoard); : checks for valid castle, returns 1 if
      valid, 0 if invalid

void performCastle(ChessPiece *piece, Square *endSquare, Board
      *gameBoard); : performs the castling special move
```

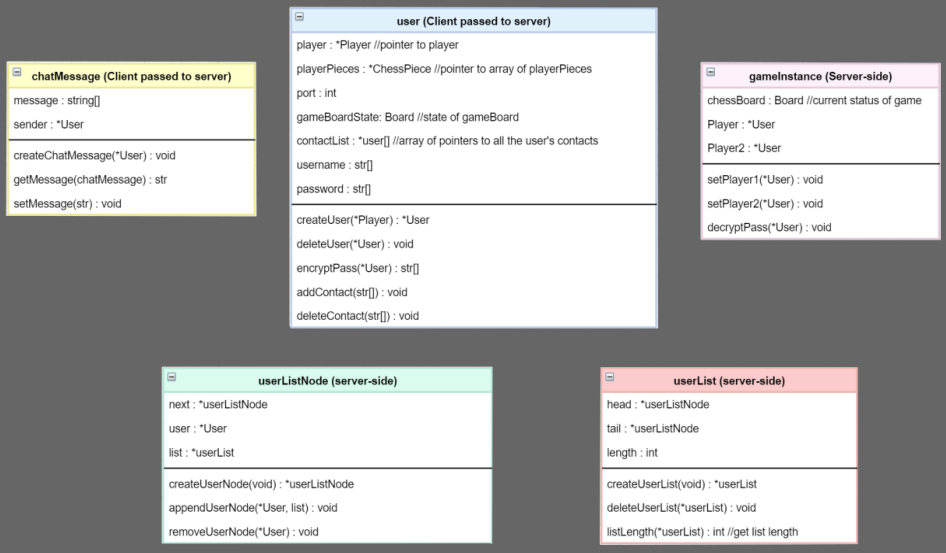
1.4 Overall Program Control Flow



2 Server Software Architecture Overview

2.1 Main Data Structure Types

2.1.1 Data Structure Diagram

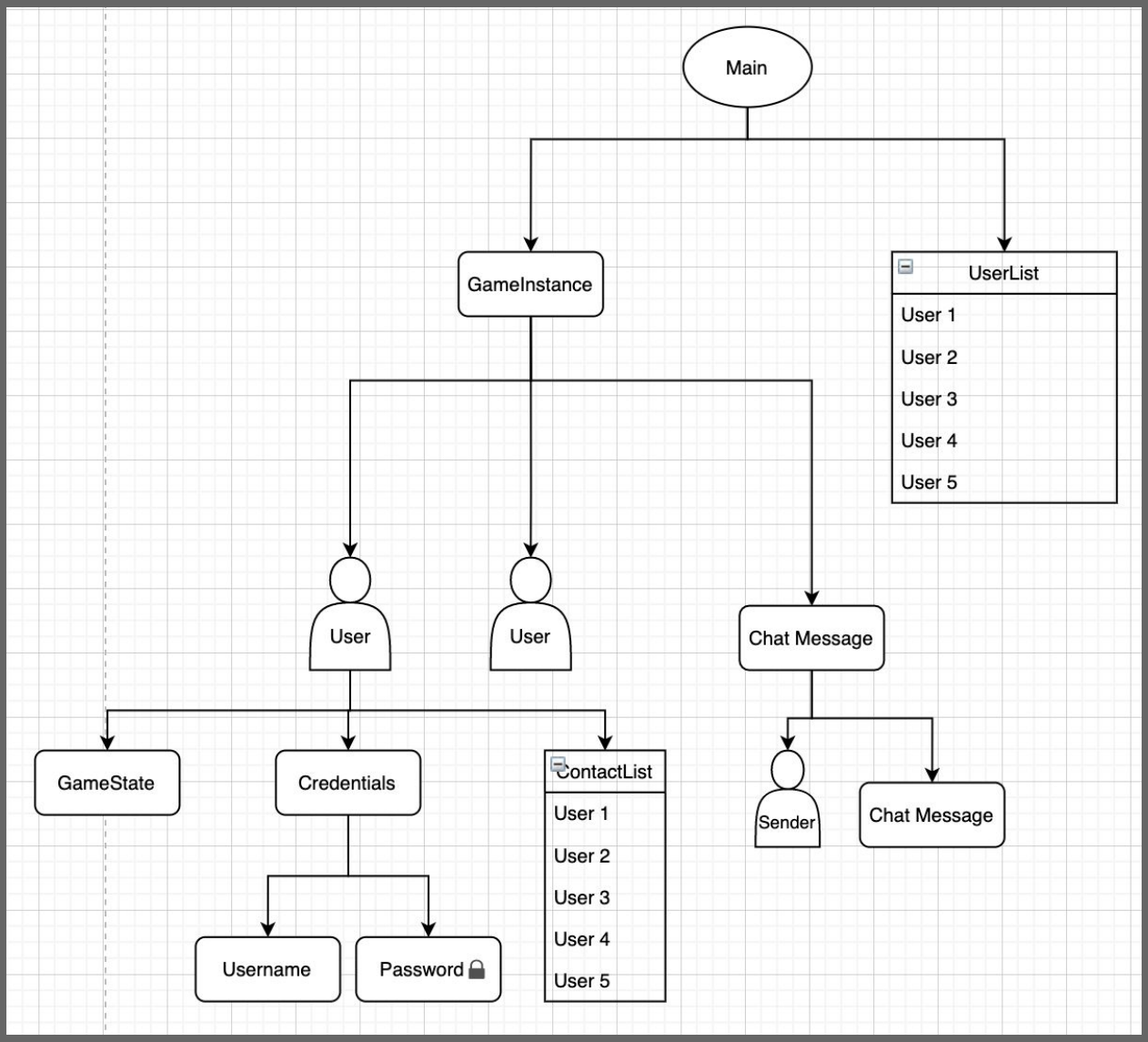


The above diagram shows each of the Data Structures in our program, and descriptors of each of the functions associated with each element. Mainly this program uses Enums, Linked List, and Structs to complete its functionality.

- Structs
  - User
  - ChatMessage
  - UserList
  - gameInstance

2.2 Major Software Components

2.2.1 Module Hierarchy Diagram



2.3 Module Interfaces

2.3.1 API of Major Modules:

A holistic list of functions allowing for interactions with major modules: USER, GAMEINSTANCE, and USERLIST

User:

```
User *createUser(Player *player, int port, char username[], char password[]);

void deleteUser(User *user);

void encryptPass(char unencryptedPassword[]);

int addContact(User *user, char username[], userList *list);
//returns 1 if successful, 0 if the username doesn't exist

int deleteContact(User *user, char username[]); //returns 1 if
successful, 0 if the username doesn't exist
```

gameInstance:

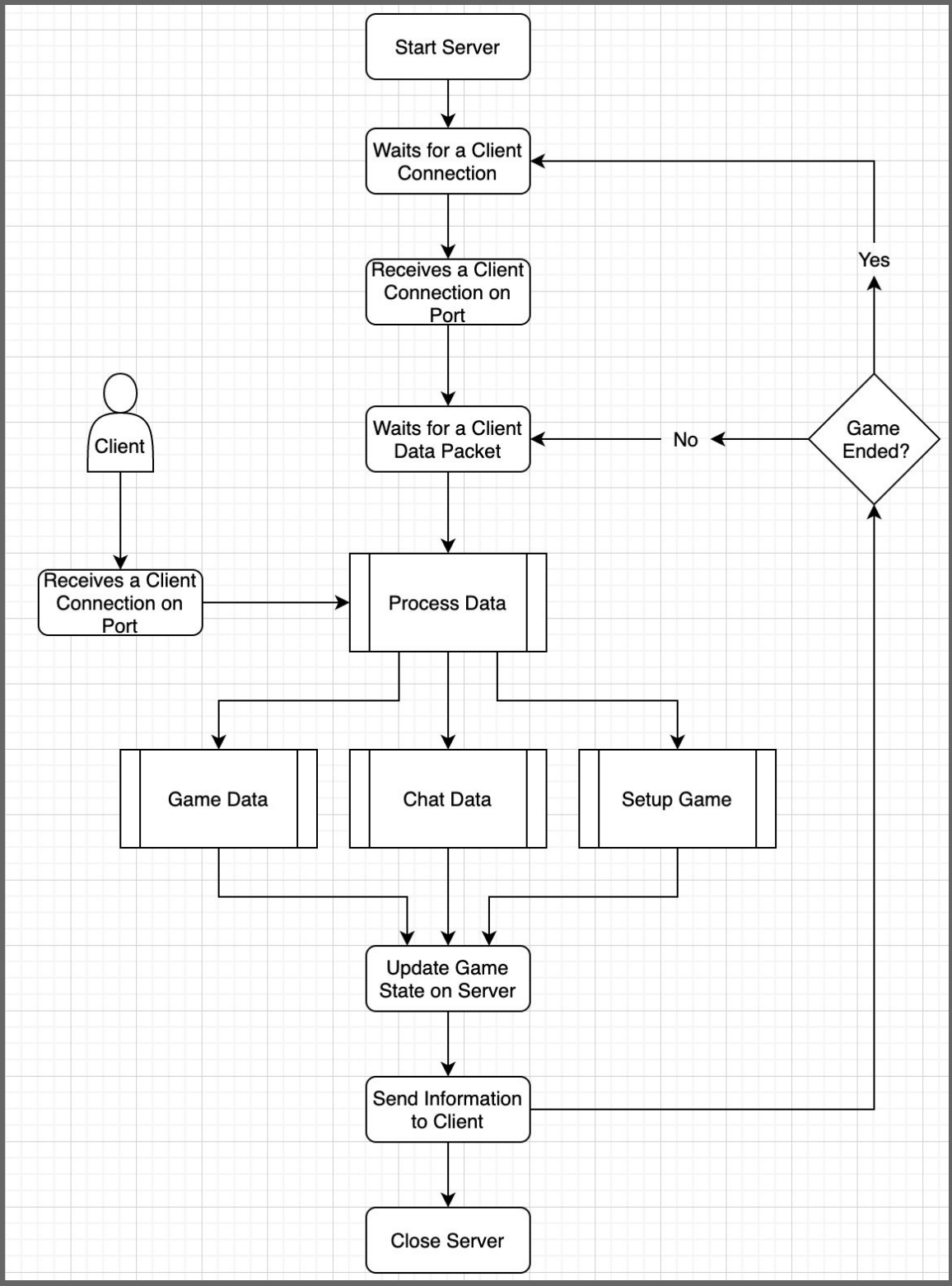
```
gameInstance *createGameInstance(void);

void deleteGameInstance(gameInstance *game);

void setPlayer1(gameInstance *game, User *user);      //set the
     gameInstance’s player1 to a certain user

void setPlayer2(gameInstance *game, User *user);
```

userList:

```
userList *createUserList(void);

void deleteUserList(userList *list);

userListNode *createUserNode(User *user, userList *list);

void deleteUserNode(userListNode *node);

void appendUserNode(User *user, userList *list);
```

```
void removeUserNode(User *user, userList *list); //remove a
user node from the  list
```

2.4 Overall Program Control Flow


Start Server
Waits for a Client Connection
Receives a Client Connection on Port
Waits for a Client Data Packet
Game Ended?
Yes
No
Client
Receives a Client Connection on Port
Process Data
Game Data
Chat Data
Setup Game
Update Game State on Server
Send Information to Client
Close Server

3 Installation

3.1 System Requirements, Compatibility

- Windows, Mac,or Linux OS
- GCC Compiler
- Math Library
- Internet access to download game files .tar package
    - Not required to play the game
- Storage space to store game files

3.2 Setup and Configuration

Load up your GCC Compiling software.
Create a directory to hold the project (if desired).
Download the game .tar package into your desired folder.
Extract all the game files from the .tar package

- gtar -xvzf Chess_Kings.tar.gz

Read the readme if you need additional help

- vi README.md

3.3 Building, Compilation, Installation

Go to the bin folder and locate the makefile

- cd bin

To compile the game you must run the makefile

- Make run

Now you can run the game whenever you want from bin folder by typing

- ./ChessKings

4 Documentation of Packages, Module, Interfaces

4.1 Protocols Used and Called

Protocol Used

Transmission Control Protocol - TCP

- Used for quick processing and secure sending
- Prevents game from sending erroneous input and enables instant validated refreshing for each player
- Enables best quality gameplay

Functions Opening Connection to the Server

Client Function (startClientConnection) created in:

~/network/chessClient.c

Files calling Client Function:

~src/GUI/GUI_Chess.c

- area_click(): waits for input from opponent if it is not the player's turn
- MoveThePiece(): sends the player's move to the opponent
- on_delete_event(): sends a shutdown message to the server when the user quits the application

4.2 Data Structures

ChessPieces contains information about the chess piece such as its position, type and color. ChessKing is a chess piece that has an addition to it for checking whether it is in check or not.
Declaration:

```
typedef struct ChessPieces{
  Square *square;

  ChessPieceType Type;
  Color color;

} ChessPiece;
```

MoveList is a list type structure which holds the start and end move of each piece that has been moved from its original starting position.

Declaration:

```
typedef struct MoveListStruct{
    MoveListNode *head;
    MoveListNode *tail;
    int length;

} MoveList;
```

Player structure contains information about the two players in the game. It keeps track of the color selected as well as whether or not the player is human or the computer. This structure also contains information about how many pieces each player has and whether their king piece is in check.

Declaration:

```
typedef struct ChessPlayer{


  Color playerColor;
  PlayerType type;

  King *playerKing;


  int isInCheckmate;

  ChessPiece *playerPieces[16];

} Player;
```

The Square structure represents each square of the chessboard. It contains data about its x and y position (rank and file), and the piece on the square. If no piece exists on the Square, it should be null. The Board Struct is just an 8x8 two-dimensional array of Squares.

Declaration:

```
typedef struct BoardSquare{
  int xPos;
  int yPos;

  ChessPiece *thisSquaresPiece;
} Square;
```

MoveList is a linked list that contains nodes for a move (start Square and end Square). The MoveList Structure also has data for a pointer to the head and tail of the list, and the length of the list. The MoveListNode contains the data for the move in addition to a pointer to the list it's a part of and a pointer to the next node.

Declaration

```
typedef struct MoveListStruct{
    MoveListNode *head;
    MoveListNode *tail;
    int length;

} MoveList;


typedef struct MoveListNodeStruct{
    Square *startSquare;
    Square *endSquare;

    MoveListNode *nextNode;
    MoveList *list;

} MoveListNode;
```

4.3 Detailed Description of Functions and Parameters

Each structure has a constructor/destructor. The constructor takes in no parameters, allocates the memory for the struct and returns a pointer to the allocated memory. The destructors take in an instance of the struct and frees the allocated memory to it.

4.3.1 ChessPiece

a) moveChessPiece(ChessPiece *piece, Square *endSquare) : Moves the chess piece from the current square to the new square if it is a valid move

b) checkValidMove(ChessPiece *piece, Square *endSquare, Board *gameBoard) : Returns 1 if move is valid, 0 if move is invalid , 2 if the move will result in a capture

c) getValidMoves(ChessPiece *piece, Board *gameBoard) : returns a list of all valid moves for a piece

d) checkDestinationStatus(ChessPiece *piece, Square *destinationSquare) : returns 1 if destination has no piece on it, 0 if it has an opponent's piece, -1 if it has the player's piece

e) int login(Player *player, char *username, char *password); : Logs In a User

f) int sendMessage(char *message, Player *sender); : Sends a message to the Opponent

g) checkDiagonalBlock,checkHorizontalBlock, checkVerticalBlock(ChessPiece *piece, Square *endSquare, Board *chessBoard) : returns 1 if there is a piece blocking the user's desired piece from moving to the desired endSquare , returns 0 if no block

h) validPawnDoubleStep(ChessPiece *piece, Board *chessBoard) : returns 1 if a pawn can perform the double step, 0 if it can not

i) kingChecked(ChessPiece *piece, Board *chessBoard) : returns 1 if player's king is in check, otherwise it returns 0

j) kingCheckMate(ChessPiece *piece, Board *chessBoard) : returns 1 if the player's king is in checkmate, otherwise it returns 0

k) movePutsKingOutOfCheck(Square *startSquare, Square *endSquare, Board *gameBoard) : returns 1 if specific move will put the king out of check

l) validPawnPromotion(ChessPiece *piece) : returns 1 if the pawn can be promoted and 0 if it can not

m) promotePawn(ChessPiece *piece) : promotes pawn if validPawnPromotion returns 1
n) validEnPassant(ChessPiece *piece, Square *endSquare, Board *gameBoard) : returns 1 if a valid en passant and returns 0 otherwise
o) captureEnPassant(ChessPiece *piece, Square *endSquare, Board *gameBoard) : moves pawn to diagonal position and captures pawn that is passed
p) checkValidCastle(ChessPiece *piece, Square *endSquare, Board *gameBoard) : returns 1 if valid castle and returns 0 if not
q) performCastle(ChessPiece *piece, Square *endSquare, Board *gameBoard) : performs the castling special move

4.3.2 Player

a) fillPlayerPieces(Player *player, Board *gameBoard) : after populating the game board with pieces the pieces are set to the player that owns them
b) playerCheck(Player *player) : return 1 if player's king is in check, 0 if not
c) playerCheckMate(Player *player) : return if player is in checkmate, 0 if not
d) getAllPossibleMoves(Player *player, Board *gameBoard) : return a linked list of all the valid options a player has to move his pieces
e) getOpponentPlayerKing(ChessPiece *piece, Board *gameBoard) : returns opponent player's king
f) movePutsOpponentKingInCheck(ChessPiece *piece, Square *endSquare, Board *gameBoard) : returns 1 if a move will put the opponent's king into check , 0 if false
g) movePutsPieceInDanger(CHessPiece *piece, Square *endSquare, Board *gameBoard) : returns 1 if move will put a piece into a dangerous position, 0 if false
h) max(int a, int b); :returns whichever value is greater between the two
i) AIHardMove(Player *player, Board *gameBoard) : returns a move from a list for the AI
j) AImediummove(Player *player, Board *gameBoard) : returns a move from a list for the AI
k) AIeasymove(Player *player, Board *gameBoard) : returns a move from list for the AI

4.3.3 Square

a) getSquareXPos(Square *square) : return square's X Position
b) getSquareYPos(Square *square) : return square's Y Position
c) getChessPiece(Square *square) : return piece on square

4.3.4 Board

a) populateGameBoard(Board *board) : set the initial positions for the game pieces
b) getGameTurn(Board *gameBoard) : returns whose turn it is
c) changeGameTurn(Board *gameBoard) : changes whose turn it is

4.3.5 MoveList/MoveListNode

a) appendMove(Square *start, Square *end, MoveList *list) : add a move in a node to the MoveList
b) removeMoveNode(MoveListNode *node, MoveList *list) : remove a node from the moveList
c) appendMoveList(MoveList *list1, MoveList *list2) : append two move lists

4.3.6 User

a) addContact(User *user, char username[], userList *list) : add a contact to the user's contact list. userList is the server's user list
b) deleteContact(User *user, char username[]) : delete a user from the server's user list
c) encryptPass(char unencryptedPassword[]) : encrypt the user's password
d) decryptPass(char encryptedPassword[]) : decrypt the user's password (used by server)

4.3.7 UserList/UserListNode

a) appendUser(User *user, userList *list) : add a User in a node to the userList
b) removeUserNode(User *user, userList *list) : remove a node from the moveList

4.4 Detailed Description of Input and Output Formats

1. Move Input by user

a. User selects a piece to move first, then chooses a square on the board to move the piece to.
b. gint area_click(GtkWidget *widget, GtkEvent *event, gpointer data) : when a chess_icon is clicked it calls this function if it clicks on a piece it returns to the main function to wait for the click destination.If destination is clicked moves the chess piece.
c. CoordToGrid(int c_x, int c_y, int*g_x, int *g_y): converts coord to grid.
d. DrawBoard(): Draws the updated Board.

2. Format of log file/Recording moves
   a. Gint area_click(GtkWidget *widget, GtkEvent *event, gpointer data): Prints every time the player clicks on a chess icon, and where it moved.
   b. The 1st line for each move is the piece color and type.
      i. Ex. BLACK PAWN, WHITE KNIGHT
   c. The 2nd line is in the format (oldX, oldY)->(newX, newY).
      i. (oldX, oldY) is the piece the user selected to move, (newX, newY) is the square the user decided to move to
      ii. (0, 0) corresponds to the bottom left of the board and (7, 7) is the top right.
      iii. Ex. (2, 3)->(2, 4)

3. Login input
   a. On the main screen, a button to create a profile will appear. Click the button, then enter a username and password and it will create your profile, connect you to the server and allow you to play against another user.

4. Chat input
   a. A text box to the right of the screen when the main game is running will be visible. To send a chat, the user will input their message and hit the send button, this will send the message in a message struct to the server, and the server will then send it to the recipient (the other player in the game).
   b. After the message is sent, the user on the other side will have the message appear in their chat box.

5 Development Plan and Timeline

5.1 Partitioning of Tasks

Week 1:Learn the rules of chess.

Week 2: Completed user manual and started creating outline functions necessary for chess game. Created flowchart for data structures so everyone would be on the same page when starting to code project. Discussed how computer will be a player and decided random selection from a list of all possible moves for basic level.

Week 3: Start working on all components of the program by splitting functions so each team member covers something necessary to the program. Also create software specification manual while working to get ideas for how functions should connect. Start working on GUI through GTK.

Week 4: Finalize. Finish working game. Users can select to play against the program or another human. Users may also select a side to play on. Connect GTK with functions to control the flow of the game. Develop the GUI menu to GTK.

Week 5: Work on logic for special moves like Castling and En Passant. Begin work on Check and Checkmate functions in addition to working on functionality for a computer player.

Week 6: Continue working on AI, improve the AI to make smarter decisions. Implement functionality to play vs different AI difficulties. Fix bugs with end game conditions (check and checkmate) and get rid of general bugs we encountered. For example, there were errors in how the GUI handled Castling and En Passant.

5.2 Team Member Responsibilities

Aviraj Mukherjee

1. Create data structures/methods to represent chess
2. LinkedList implementation
3. Checking valid moves/moving pieces
4. Player methods
5. Function to get all possible moves
6. Work on AI for hard difficulty
7. Check/Checkmate functionality
8. En Passant
9. Castling

10. Debugging
11. Create data structures and methods for server communication

Martin Alexander Gomez:

1. Work on getting/editing GUI chess images and putting them in GTK. (PSCP)
2. Work on partitioning tasks and keeping meeting notes.
3. GTK Board/Rank/File/Widgets
4. GTK Implementation Logic/Implementation
5. GTK piece moving/removing
6. Help connect GUI to backend
7. Helped debug
8. Create GUI for chat/login

Henry Thy Bendiksen:

1. CreateBoard/PopulateGameBoard coding
2. Installation Details
3. Checking special moves En Passant and Castling
4. Checking pawnPromotion eligibility
5. Helped work on updating Software Specification and User Manual
6. Chat functionality

Sarthak Sharma:

1. Start Working on Chat Design/GUI
2. Work on the Server/Client Control Flow and Module Hierarchy Diagrams
3. Work on Setting up Network Communication

Rebecca Ko:

1. Work on Check Valid Move
2. Work on AI for easy and medium difficulty
3. Work on documentation
4. Work on updating Software Specification and User Manual
5. Server coding/setting up control of server

Copyright

References

Domer, Rainer. *EECS 22: Advanced C Programming*. 2018. Powerpoint Presentation

Eeles, Peter. "What Is a Software Architecture?" *IBM*, 15 Feb. 2006, www.ibm.com/developerworks/rational/library/feb06/eeles/index.html.

Index

Algorithm, 2
Building, compilation, installation, 18
Client software architecture overview, 3
Control flow, 2,9
Data structures, 3,10
Data type, 2
Directory, 2,18
Function, 2,3,5,10,21
Functions and parameters, 21
Graphical User Interface, 2
Inode, 2
Input and output formats, 23
Linked list, 2,3,10,20
Main data structure types, 3,10
Major software components, 4,12
Module interfaces, 5,13
Node, 2,12
Overall program control flow, 9,17
Parameter, 2,21
Partitioning of tasks, 24
Server software architecture overview, 10
Setup and configuration, 18
Software architecture, 2
Structure, 2,3,10
System requirements, compatibility, 18
Team member responsibilities, 23