Quentin Hsu

CS200 Section 30237

9/29/2012

Design Document: Homework 3

\*\*\*ALUDRA BUG - the revalidate function doesn't work on aludra so panels won't revalidate after jframe is resized. if panels are off (very noticeable), simply drag the window size a bit (make it smaller until panels readjust, then pull back out to approximately normal window size). ALTERNATIVELY - run on eclipse or request a jar file from me for optimal performance\*\*\*

**Purpose/Overview**

The purpose of this program is to update our card game to have networking capability as well as an artificial intelligence player. We are to explore using serversockets and sockets to create a client-server based game. In addition, we will be using multiple threads and will implement at least one runnable class in order to create threads. Only the basic rules of blackjack are implemented so splits and some other moves are still not available.

**Requirements**

This program needs to have all the requirements of homework 1 and 2 and needs to have three main classes. There should be a server class (does not need GUI) that allows for at least 4 players (real or AI) to connect before starting the game. The AI should be a separate program (does not need to have GUI) that acts as if it were a real player. The server shouldn't know whether the player connected to it is a real player or an AI. My game implements various functions that go beyond the requirement and is described in the high level architecture and user interface sections.

**Global Data/Functions**

I have a "global" (in the blackjack package) settings class that contains most of the information needed to run blackjack. This universal class is created as an object once in the application class, then passed as a reference to every panel upon creation.

**High Level Architecture**

My game is split into three packages: client application, server, and ai application

**Client Application:**

The client application contains all of GUI components from my previous games and is changed to be a display engine in which the player can change settings, connect to the server, and see the status of the game on the server. No game calculations are done on the client side except to get input and to adjust some of the player variables in order to display the correct screen. My client is not limited to accessing information of your own player though. It has the ability to watch other players play in a live view, but can only send actions for its own player.

Design: My central class for my client application is the Application class. This Application class creates all my panels and my client class. My panels need to communicate with each other, so everything that the Application creates has a reference to the Application class. This way, if one panel needs to contact the client, it can call the application reference, then call the client class in the application. My client class is a separate thread that is used to communicate with the server protocol class. All my threads are constant, meaning they are never stopped until the client stops the game, disconnects, or runs out of chips (which is same procedures as disconnection). The protocol is described next. All my panel buttons simply change the command string that I send to the server to request a new action/information.

**Server:**

My server is similar to my application, but with far less panels. The server has a GUI interface to allow for coin adjustment, starting the server, and exiting the server. This was more for player convenience when running the server application. My game class that does all the calculations is moved into my server class and is called to run when needed. Serverprotocol is my runnable class that handles all the interaction with its specific client. Although I have hardcoded the amount of players the server can accept to be 4, my code is mostly written to be adjustable for as many players as the server creator desires (just not implemented yet since I was working on disconnections).

Design: My serverprotocol interaction with the client is similar to the TCP/IP threeway handshake except that it is more of a two way handshake. The client contacts the serverprotocol and sends a String. The serverprotocol receives the String, decides what to do with the String, then sends back another string to indicate what further data it is going to send. After sending the first String, my client waits until it receives the second String, determines what the String is, and follows the procedures to read in further data that the serverprotocol will send. Because of the this two way confirmation, I split my run class into multiple while(true) loops for different phases of the game. The commands are constantly sending because of this loop to ensure that a connection exists and that the player can receive input at anytime in the game. The live streaming works because of this constant loop. If the client doesn't have any action, the default command sent is idle. Idle increments an update counter which will change the command to update after 2000 run throughs, allowing for the screen to update.

Disconnections: A very challenging aspect of my game was handling disconnections. I wanted my game to continue even after a player disconnects. Because of this, I designed multiple variables to adjust the game for disconnections. If a player disconnects in the waiting room, a count is simply adjusted and the Treemap that keeps track of available spaces is opened up. The next player that connects to the server will take the spot of the disconnected player. If a player disconnects midgame, the player is set to a temp player and my game skips the temp player for that round. At the end of the round, people are sent to the waiting room and need to wait until the set number of players connect again. When a client stops the game, the coins in the profile are saved and the client simply disconnects from the server. The server handles the rest of the procedures. One case of disconnections that I don't handle is disconnection when the player is currently being requested to take action (bet, hit, stay). Since disconnections is not a requirement, I will program this in a later date.

**AI Application:**

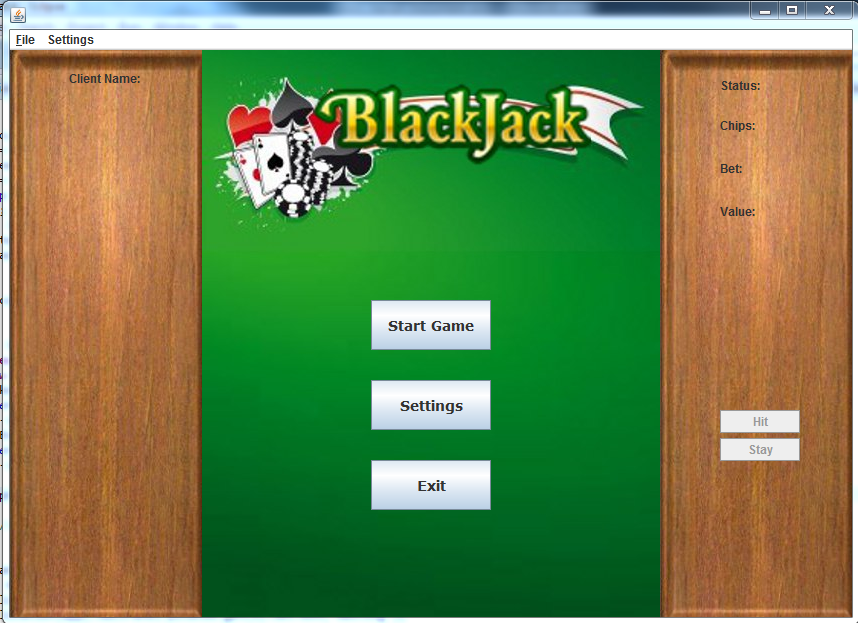
My AI is a cut down version of my player application. Most of the code is similar to the player application. The main differences is that the commands sent by the client are hardcoded before each protocol block so that responses such as waiting for players to get ready or continue are instantaneous. The heart of my AI program is the decision making function that decides what it should do when it is betting and choosing whether to hit or stay. The AI program bets a random number between 1 and half the number of its chips. It's decision as to hit or stay is based off of a basic blackjack strategy that looks that the cards it has, looks at the dealer's shown card, and decides. I have attached the strategy table in my images file. Since my game doesn't have splits or double downs, splits are handled as if they were regular values and double downs are based off another table as to whether or not to hit or not (available on wikipedia). Disconnections are similar to the client application but with fewer prompts.

**User Interface**

**Client Application:**

My interface is similar to HW2, but with minor upgrades. The playerpanel now has a label to indicate what your selected profile name currently is. There menubar is cut down and the create player button is disabled once the game starts. Save game and load game no longer saves the whole game, but rather saves the profiles the client has created and is all client based. Player creation is upgraded into profile management and the player can add the current created player, add new players, select players, and remove players in the manage profiles settings menu. The client also does not connect to a server until the start game button is clicked. In the settings menu, the player can input a server address to connect to so that multiple servers can be running at the same time on different websites and the player can choose which one to connect to. The default address is localhost so if no address is set, the client will attempt to look for a server on the current computer. JOptionPane prompts pop up when needed for user input as well as message notices. Below are several screenshots of the GUI design.

Main Menu:



Settings:



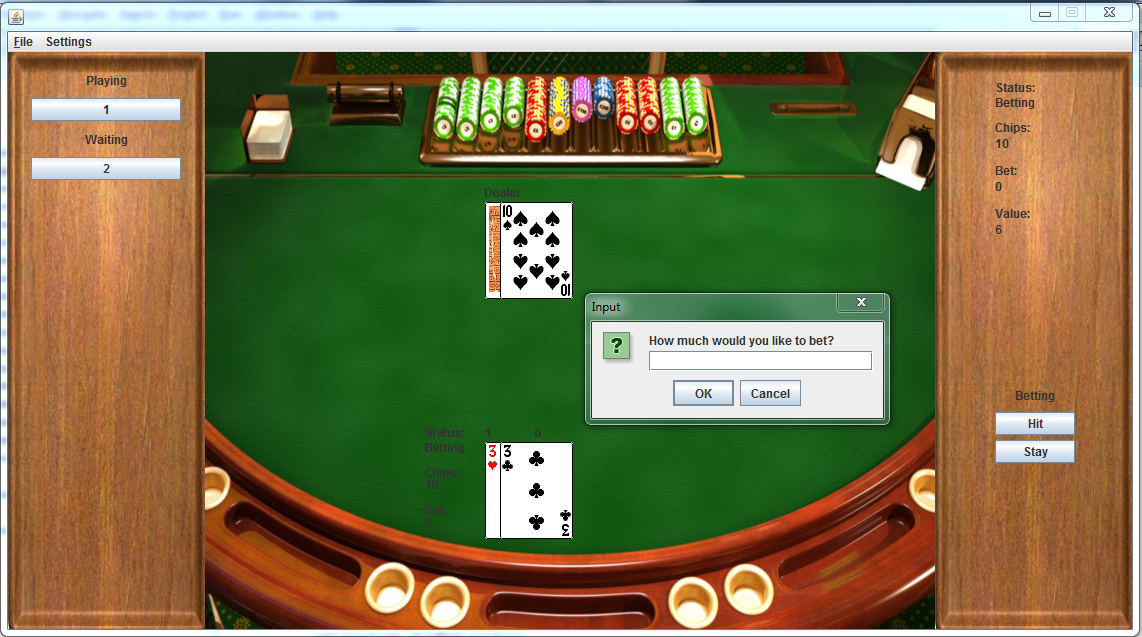
Profiles (Player List autoadjusts to how many profiles you have):



Waiting Room:



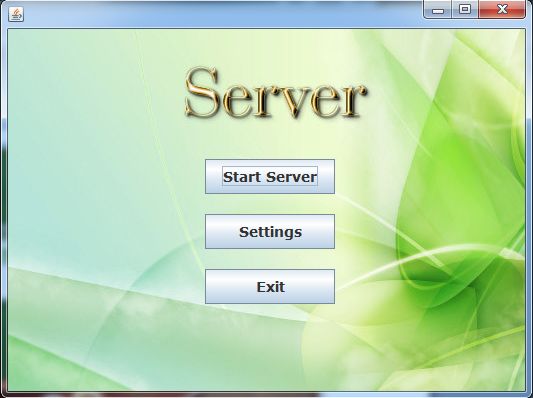
Layout while playing the game:



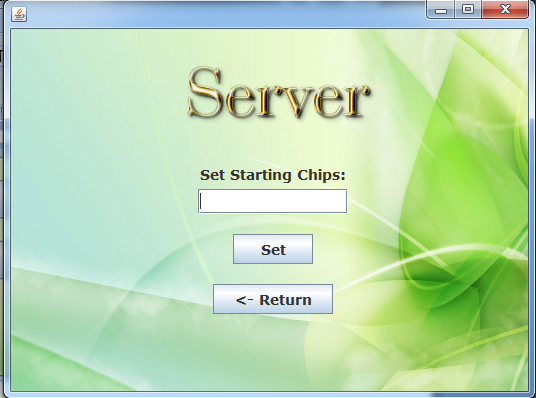
**Server:**

My server has a minimal interface that is more for the convenience of the server runner. It has a settings menu to adjust the number of coins new players will start with and has a start button that will start the server. Stopping the server so that it can be run again with new players became too complicated, so the player can only start the server once, then exit the whole server when desired (will cause clients to do disconnection procedures).

Main Menu:



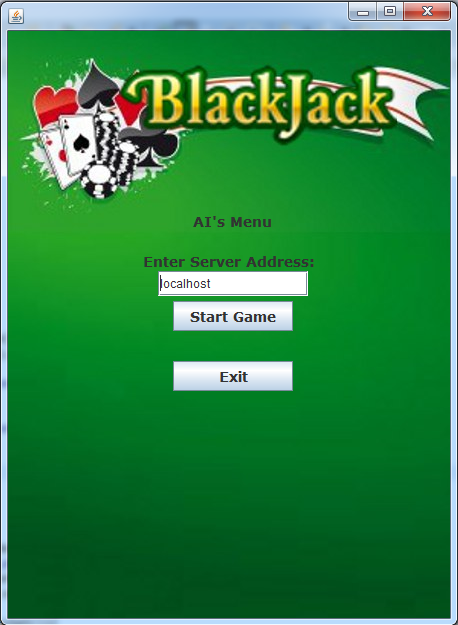
Settings:



**AI Application**

My AI GUI is also simple and is similar to the Client Application but with no side panels and no settings menu. There is simply one textfield to indicate which server to connect to and a start button. The rest of the interface is the same as the Client Application but without any side panels. It is just a screen so the AI runner could watch what's going on with their AI.

Main Menu:

****

**Classes**

New Classes

ProfileList - profile class that contains arraylist of name and player coins.

* Members
  + ArrayList<String> names - profile of names
  + ArrayList<Integer> coins - list of coins
* Methods
  + ProfileList() - constructor
  + copy(ProfileList) - copy function
  + size() - returns the size of the list
  + add(String, Integer) - add profile
  + remove(Integer) - remove a profile at a specific index
  + getters and setters

Client - player's class to communicate with server

* Members
  + JFrame app - reference to app
  + Settings info - reference to info
  + Thread thread - thread to run this client
  + Socket s; //socket to talk to network
  + String serverName - server name stored
  + ObjectOutputStream out - output stream
  + ObjectInputStream in - input stream
  + String command - command received
  + String commandSent - command to sent
  + Integer updateLoop - counter to update the screen
  + ProfileList profiles - loaded list of profiles
  + String name - selected user name
  + Integer coins - coins for current client
  + Integer id - thread ID
  + Boolean check - verify button was pressed
  + PaintMap map - paint the screen
* Methods
  + Client - constructor
  + connect() - connect socket to serverName that is stored
  + updateWait(TreeMap<Integer, Boolean>, TreeMap<Integer, String>) - iterate through treemaps to update the waiting room screen
  + run() - core code to communicate with server...all the protocols
  + save(String path) - save the client's profiles at specified path
  + load(String path) - load client's profiles at specified path

AIApplication - main application class for the AI

* Members
  + Settings info - local data for display use
  + AIClient gamer - runnable function that handles server communication
  + AIMainMenu frontpage - mainmenu
  + AIWaitRoom - waiting room
  + AIGamePanel - graphics panel to paint the game on
  + Integer phase - which screen to show
  + Boolean end - unused
* Methods
  + AIApplication() - constructor
  + reset() - reset the application
  + main(String[]) - run the application
  + paint(Graphics) - repaint correct menus
  + revalidate() - revalidate function for aludra use
  + actionPerformed(ActionEvent e) - repaint()
  + removeCenter() - remove the correct center panel
  + exit() - close the app
  + overridden windowlisteners and getter and setters

AIClient - class to communicate with the server

* Members
  + JFrame app - reference to app
  + Settings info - reference the settings
  + String serverName - server name to connect to
  + Thread t - thread to run this class
  + Socket s - socket to connect to server
  + ObjectOutputStream out - output stream
  + ObjectInputStream in - input stream
  + String command - command received
  + String commandSent - command to send
  + Integer updateLoop - counter to check when to update the screen
  + Integer decisionLoop - counter to slow down ai decision making
  + String name - name of the AI
  + Integer coins - coins AI has
  + Integer id - thread ID of this AI
  + PaintMap map - painter
* Methods
  + AIClient(JFrame \_app, Settings \_info) - constructor
  + connect() - connect socket to serverName that is stored
  + updateWait(TreeMap<Integer, Boolean>, TreeMap<Integer, String>) - iterate through treemaps to update the waiting room screen
  + run() - core code to communicate with server...all the protocols
  + firstDetermine() - strategy during first request phase
  + determine() - strategy based on values
  + getters and setters

AIGamePanel - display game screen

* Members
  + JFrame app
* Methods
  + constructor
  + paintComponent - calls client's paint function

AIMainMenu - main page of AI GUI

* Members
  + JFrame app - reference to app
  + Settings info - reference to data
  + ImageIcon background - reference to background image
  + JLabel title - AI Menu
  + JButton start - start the game
  + JLabel serverLabel - Indicate input for servername
  + JTextField serverField - input for servername
  + JButton serverSet - button to change servername
  + JButton exit - exit app
* Methods
  + AIMainMenu - constructor
  + paintComponent - repaint background
  + actionPerformed - perform appropriate action for buttons clicked
  + create() - prompt for AI username and set client's name

AIWaitRoom - waiting panel

* Members
  + JFrame app - reference to app
  + Settings info - reference to info
  + ArrayList<JLabel> pNames - list of name labels
  + ArrayList<JLabel> pStatus - list of status labels
  + ImageIcon background - background
* Methods
  + AIWaitRoom - constructor to layout everything
  + paintComponent - paint background
  + getters and setters

Server - handles all the serverprotocols, runs the game

* Members
  + ServerPanel screen - main menu
  + ServerSettings settingscreen - settings menu
  + Integer phase - which menu server is displaying
  + Timer timer - timer
  + Thread thread - thread for server
  + Integer numPlayers - total number of players
  + Integer donePlayers - number players done processing
  + Integer contPlayers - players who clicked continue game
  + Integer readyPlayers - # of players who clicked ready, also used to determine disconnection actions
  + Integer tempPlayers - # of temporary players, also used to determine disconnection actions
  + Integer accessPlayer - player count to see how many players passed through a command
  + TreeMap<Integer, Boolean> openSpaces - keep track of which positions are open
  + TreeMap<Integer, Boolean> ready - keep track of which players clicked ready
  + ServerSocket ss - serversocket
  + Socket s = null - socket
  + Game bjack - game class
  + Settings info - game data
  + Integer firstStart - used to control looping of server connection seeking
  + Boolean gameStarted - used to indicate that game round has started
  + Boolean restarting - used to let serverprotocols know when to restart
  + Boolean loop1 - used to pause the server
  + Boolean loop2 - used to pause the server
  + Boolean loop3 - used to pause the server
* Methods
  + constructor
  + reset() - initialize everything as new variables
  + start() - start the server
  + run() - run the server and game
  + paint(Graphics g) - repaint the screen
  + actionPerformed - repaint()
  + main(String[]) - entry point to run server
  + drawSpace() - return integer of next available spot on the server

ServerPanel - main window panel of server

* Members
  + JFrame server - reference to server
  + JButton start - start button
  + JButton set - settings button
  + JButton exit - exit button
  + ImageIcon background - background
* Methods
  + ServerPanel - constructor
  + paintComponent - repaint background
  + actionPerformed - does appropriate action for each button pressed

ServerSettings - settings menu to adjust chips

* Members
  + JFrame app - reference to app
  + Settings info - reference to data
  + JLabel chipsLabel - set chips label
  + JTextField chips - field to enter chips
  + JButton chipSet - set chips
  + JButton back - return to main menu
  + ImageIcon background - background image
* Methods
  + ServerPanel - constructor
  + paintComponent - repaint background
  + actionPerformed - does appropriate action for each button pressed

ServerProtocol - runnable class that handles the client (communications)

* Members
  + Thread t - thread for this class
  + Socket proSocket - reference to socket
  + Server data - reference to server
  + ObjectOutputStream out - output stream
  + ObjectInputStream in - input stream
  + Integer wait - counter to see if screen needs updating
  + Integer threadID - this thread's id number on server
  + String command - command received by client
  + Boolean addedCont - used to determine whether continue was clicked
  + Boolean addedDone - used to determine whether player is done
* Methods
  + ServerProtocol - constructor, setup serverprotocol and initialize data
  + run() - protocols and client communication

PlayerlistPanel - basically the createpanel but without the ability to create players.

* Members and Methods - see below at createpanel. Cut down version in order to have an actual menu button for displaying player list.

Updated classes

CreatePanel - panel to create new players as well as displays list of players

* Members
  + ImageIcon background - background image
  + JFrame app - reference to app
  + Settings info - reference to info
  + JLabel createLabel - "Create New Player:"
  + JLabel createName - "Enter username"
  + JTextField name - enter name
  + JButton create - submit entry
  + JLabel playerLabel - "Players"
  + JScrollPane scroll - JScroll pane to fit player list that is larger than the screen
  + JPanel scrollPanel - panel to put scrollpane in
  + JPanel scrollItemPanel - panel to put in the arraylist of panels that will be added to the scroll
  + ArrayList<JPanel> playerTable - arraylist of jpanels that will hold the row of each player information
  + ArrayList<JLabel> nameLabels - list of "Name"
  + ArrayList<JLabel> playerNames - list of player names
  + ArrayList<JLabel> coinLabels - list of "Coins:"
  + ArrayList<JLabel> playerCoins - list of players coin
  + ArrayList<JButton> select - list of player selection buttons
  + ArrayList<JButton> remove - list of player removal buttons
  + Integer phaseReturn - int to keep track of what window to return to
  + JButton back - button to return
* Methods
  + CreatePanel(JFrame, Settings) - constructor to make panel
  + update() - re-adds components to update player list
  + paintComponent(Graphics) - repaints background and updates
  + actionPerformed(ActionEvent) - checks which button pressed and does appropiate action
  + getters and setters

Barely changed/Unchanged classes:

GamePanel - JPanel for game to be painted on

* Member
  + \*\*\*Changed - no more reference: Game bjack - reference to Application's bjackGame
* Methods
  + GamePanel(Game \_bjack) - constructor that sets panel size
  + paintComponent(Graphics g) - overwritten method to call game's paint method

MainMenu - JPanel with the main menu that is displayed when app is started

* Members
  + JFrame app - reference to app
  + Settings info - reference to data
  + ImageIcon background - background image
  + JButton start - start the game
  + JButton settings - go to settings
  + JButton exit - exit app
* Methods
  + MainMenu(Jframe, Settings) - constructor to set up panel, adjust size and alignment of all components, and attach listeners
  + paintComponent(Graphics) - overwritten function to paint background and revalidate components
  + actionPerformed(ActionEvent e) - performs proper actions for each button pressed
  + create() - run if user starts game without any active players created. prompts player to create players. allows for input cancelling
  + getters and setters

MenuBar - customized JMenuBar

* Members
  + JFrame app - reference to app
  + Settings info - reference to info
  + JMenu file - file menu
  + JMenuItem newGame - reset everything but keep current players
  + JMenuItem saveGame - save game
  + JMenuItem loadGame - load game
  + JMenuItem stop - stops game and resets app
  + JMenuItem exit - exit
  + JMenu Settings - settings menu
  + JMenuItem create - go to create new player screen
  + JMenuItem list - list players
* Methods
  + MenuBar(JFrame, Settings) - constructor to add menu and menuitems
  + actionPerformed(ActionEvent) - handle what each menu item does, turns off createpanel when game starts

PlayerPanel - left panel that allows user to click players to see their cards on the screen

* Members
  + JFrame app - reference to app
  + Settings info - reference to info
  + TreeMap<Integer, JButton> buttons - player buttons arranged in order of display on the panel
  + TreeMap<Integer, JButton> players - player buttons unsorted, in order of info.player
  + TreeMap<Integer, Integer> remap - remaps button position to unsorted players position
  + ImageIcon background - background image
  + JLabel clientName - "Client Name:"
  + JLabel playing - "playing"
  + JLabel waiting - "waiting"
* Methods
  + PlayerPanel(JFrame, Settings) - constructor to set up layout and labels
  + paintComponent(Graphics) - updates panel
  + create(Integer) - make new button for player passed
  + update() - refresh button treemap and remaps player buttons correctly
  + customize() - remakes each button and re-adds to panel. disables buttons as appropriate
  + disableAction() - inactivate all buttons
  + enableAction() - reactive all buttons
  + actionPerformed - check which button clicked and adjust screen to reflect that player. turns on and off stats panel buttons as appropriate
  + setClientName(String) - resets jlabel to display name

Settings - important class that holds all game data and is accessed by all panels

* Members
  + myDeck - game's deck of cards
  + players - Treemap of players
  + compPlayers - stores positions of compPlayers for AI in future
  + id - int of current player being processed
  + round - counter to keep track of how many players processed
  + phase - current state of game
  + screenPlayer - keeps track of what player is painted on screen
  + pausePhase - keep track of what phase game was in before pause
  + paused - see if game is paused
  + chips - default starting chips for all players (can be set in settings)
* Methods
  + Settings() - initialize everything
  + copy(Settings) - copy function, used when loading game
  + getters and setters

SettingsPanel - panel for settings menu to create player, adjust chips, and load game

* Members
  + app - reference to app
  + info - reference to data
  + createLabel - "Manage profiles"
  + create - button to go to create menu
  + loadLabel - "Load Game"
  + load - loads game
  + saveLabel - "Save Game"
  + JButton - save game
  + \*\*\*Removed chips: chipsLabel - "Set Starting Chips"
  + \*\*\*Removed chips: chips - textfield to enter chips
  + \*\*\*Removed chips: chipSet - submit chip reset
  + back - go back to main menu
  + background - background image
* Methods
  + SettingsPanel(Jframe, Settings) - sets up the menu and components
  + actionPerformed - goes to correct menu for each button or does correct action for setting chips
  + paintComponent(Graphics) - paint background

StatsPanel - right hand panel with selected player's stats and action buttons and prompt

* Members
  + app - reference to app
  + info - reference to data
  + background - background image
  + statusTitle - "Status"
  + chipsTitle - "Chips"
  + betTitle - "Bet"
  + valueTitle - "Value"
  + status - status
  + chips - chips
  + bet - bet
  + value - value
  + prompt - game prompt for different phases of game
  + hit - player pushes button to hit
  + stay - player pushes button to stay
  + cont - continue button for when game is paused
  + pause - boolean to track if it is first loop that game has been paused
* Methods
  + StatsPanel(JFrame, Settings) - construct all components and layout
  + actionPerformed(ActionEvent) - \*\*\*Changed to update command sent in the client class
  + setText(String) - set prompt
  + setTextSize(Integer) - set prompt label size
  + disableAction() - turn off action buttons
  + enableAction() -turn on action buttons
  + paintComponent(Graphics) - update stats panel
  + getters and setters

Game - actual game of blackjack that updates its state and does all the calculations for the game

\*\*\*Cut down many functions to adapt to server environment. Functionality still the same

* Members
  + Application app - reference to the app
  + Settings info - reference to app's game information
  + PaintMap map - painter helper
* Methods
  + Game(Application) - constructor
  + void run() - decide what state game is and choose correct function to call
  + void save() - serializes info and the printobject of paintmap class(keep track of what to paint when reloaded)
  + void load() - reload from save file
  + void createDeck() - make a deck
  + void deal() - deal cards to each player
  + void bet(Integer) - prompt the chosen player for their bet
  + void request() - prompt for hit or stay
  + void hit(Integer) - draw card for chosen player
  + void stay() - end request phase, skipping all temp players
  + void setMap(Player) - set which player paintmap will paint
  + void autorun() - computer hits and stay's based of bjack rules
  + void results() - compare results and distribute coins
  + void requestRound() - ask player for permission to continue
  + void newRound() - continue dealing with a new deck
  + void endGame() - close window
  + void pause() - function that keeps game paused until unpaused
  + void paintgame(Graphics2D) - call paintmap's paint function

Application - JFrame to display game

* Members
  + \*\*MOVED TO SERVER -> Game bjackGame - blackjack game
  + Settings info - all the game information on a local basis to display correct things
  + MenuBar topBar - menu bar
  + MainMenu frontpage - main menu
  + SettingsPanel settingsMenu - settings menu
  + CreatePanel - settings menu to manage player profiles
  + GamePanel bjackPanel - JPanel that game is painted on
  + StatsPanel stats - right hand panel with stats and action buttons
  + PlayerPanel players - left hand panel with player buttons to choose which player to see on screen
  + Integer phase - current phase shown by JFrame (which menu is currently shown)
  + Boolean end - controls when exit will prompt for save
* Methods
  + Application() - default constructor, sets up all the panels
  + static void main() - create and setup app window
  + void paint() - repaints all panels
  + void revalidate() - revalidate not available on aludra, so I made this to revalidate all panels (reupdates all the panels since window adjustments mess up button display)
  + void actionPerformed(ActionEvent e) - tell Game to run if necessary and calls repaint
  + void removeCenter() - checks what phase frame is on, then removes the correct middle panel
  + void exit() - close app
  + windowClosing(WindowEvent) - prompt for save if closing during specific times of game
  + multiple window listener events

Card - this class is the base of any card game and was created first. It holds data to keep track of its type, value, suit, and image. Methods are available to construct the card, set the proper card image, and to paint itself onto a JFrame.

* Members:
  + Integer value - this is the numerical value of the card (ie. Two is 2)
  + String type - this is the card type in words (ie. Two, Jack, Queen)
  + String suit- this is the card's suit
  + Boolean faceUp- this is used to determine whether or not the card is faceup
  + Boolean active - this is used to determine if the Ace's value is 11 or 1
  + ImageIcon pic - this is the actual image of the card
* Methods:
  + Card() - default constructor that makes a Black Joker
  + Card(String, String) - constructor to make card based on the type and suit passed
  + Card(Integer, String) - constructor to make card based on numerical type and suit
  + Card(Card) - copy constructor
  + void setPic() - change image filepath to the correct image based off type and suit
  + Integer typeConvert(String) - convert string type to its integer value
  + String typeConvert(Integer) - overloaded function to convert from integer to string type \*\*FIXED BUG - Jack was never displayed in deck since it was returned as an Ace.
  + String suitCheck(String) - check if suit entered is valid
  + void activate() - turn on Ace's 11 value
  + void deactivate() - turn off Ace's 11 value
  + void print() - print out as "type of suit" to console
  + void paintCard(JFrame,Graphics2D,Integer,Integer) - print out card
  + various getters and setters

Deck - a deck of cards that is created using a system of treemaps. Allows direct access to cards by going through 2 treemaps typing suit and type. Also allows access to cards based off position in deck. The system of treemaps is to allow for class reuse for future card games.

* Members
  + TreeMap<Integer, Card> deck - TreeMap of cards with the key as the position of the card in the deck
  + TreeMap<String, TreeMap<Integer,Card>> cards - first level treemap that contains the 4 suit treemaps
  + TreeMap<Integer,Card> clubSuits - second level treemap that contains all the club cards
  + TreeMap<Integer,Card> diamondSuits - second level treemap that contains all the diamond cards
  + TreeMap<Integer,Card> heartSuits - second level treemap that contains all the heart cards
  + TreeMap<Integer,Card> spadeSuits - second level treemap that contains all the spade cards
  + Integer top - int variable to keep track of current position of top card of deck
* Methods
  + Deck() - default constructor
  + void generate() - this creates 52 cards and adds them to both TreeMap organizations.
  + void shuffle() - this shuffles the deck
  + Card draw() - returns the card from the top of the deck
  + void clear() - clear the deck
  + void print() - prints out all 52 cards to the console in order or position

Player - all the player information is stored in this class.

* Members
  + String name - player's name
  + Integer id - player's position for the game. not used in this assignment
  + ArrayList<Card> hand - card that the player has. This member is public since it needs to be accessed often and changed
  + Integer score - the numerical sum of the player's cards
  + Boolean bust - determine is player is busted
  + Boolean bjack - determine if player has blackjack
  + Integer chips - how many chips player has
  + Integer bet - what the player bet
  + String status - current action for player in the game
  + Boolean temp - check if player is temp player
* Methods - all methods changed to synchronized for locking
  + Player() - default constructor
  + void updateScore() - calculate total sum of player's cards (chooses suitable value for the ace)
  + void bjackCheck() - check if player has a blackjack
  + void win() - update chips based off bet for winning (considering blackjack)
  + void lose() - update chips based off bet
  + void push() - clears bet
  + void clear() - reset player's hand
  + boolean contains(String) - checks to see if the hand contains the card
  + various getters and setters

Dealer (previously Computer) - computer player that inherits from Player. Used for the dealer in this game

* Members
  + Boolean playing - keep track of when computer is playing (not used)
* Methods
  + Computer() - default constructor
  + void reveal() - turn facedown card up

PaintMap - helper class to paint everything on the screen at set locations

* Members
  + TreeMap<Integer, Player> printObject - treemap to store players to print out according to priority
  + ImageIcon background - image of background
  + JFrame app - reference to the JFrame the class is painting to
* Method
  + PaintMap(JFrame) - constructor
  + void paint(Graphics2D) - paint everything according to priority locations

**Testing Cases**

I tested my game by first attempting to connect to the server, then adjusting functions so that the game performed like my previous game but with a server. I went through each phase of the game, adjusted the commands and protocols that I had, and added new counters in order to keep the threads in check. Most of my testing was on the thread logic of the game. This was challenging since I would have around 13 threads running each time (2 from each client, 4 clients, 1 for each serverprotocol, 4 serverprotocols, 1 server) and there was a lot of race conditions to check. A majority of the time was spent on perfecting the locking mechanisms so that threads would run according to when they should run. Disconnections also required more logic testing and variable changing in order to disconnect without affecting the game too much.

Nominal test cases were mostly just running the game through multiple times. I first ran the game as I would like to play it, find an error, and reperform my steps multiple times while checking variables in order to adjust the logic properly. Afterwards, I did more specific tests to check different conditions. For example, I tested disconnections by disconnecting the client during a specific phase in the game, then rewriting my code and testing again until the disconnection worked properly. Function documentation below has test cases for the more troubling functions that I had to deal with.

**Function Test Cases**

\*\*unchanged classes and functions and simple functions (GUI would just be look if it's correct, etc) are not documented

Client Class

* void save(String path) - saved profile, then loaded it to see if it was correct
* void load(String path) - tested with the save function
* void updateWait(TreeMap<Integer, Boolean> tempReady, TreeMap<Integer, String> names) - went to waiting room, added more clients to the room to see if names adjusted, clicked ready and checked if correct player's status is changed
* Integer connect() - tried connecting to server
* \*\*\*tested along with server's run\*\*\* run() - ran through game multiple times with specific actions to test different bugs (ran over 100 times)
  + Waiting room - ran multiple clients, tested ready buttons and checked screen updates
  + Game Phase - tested hit and stay buttons, seeing if screen updated properly. Played through game to see if cycling of players is normal as well as if buttons are enabled and disabled at correct times
  + Restart Phase - many bugs here especially when testing for disconnections. Took specific actions to create bugs, then logically traced the bugs using debug printouts
  + Disconnection handling - checked game progressed normally and skips temp players. tested restart phase multiple times to make sure game goes to waiting room at correct times, functioned the same after restarted, etc. Checked stopping the game and rejoining with same client

AI Client Class

* run() - similar testing to client class. ran through multiple times with varying amounts of AI's and real players. Check if AI's actions and decisions simulate a real player's

\*\*previous documentation cases checked to see if works with server

Game Class (only changed functions are documented)

* void run() - test case: test last to see if game progression logic is correct
* void bet(Integer) - run game, see if bet prompt appears, enter different inputs to test error catching. Check if player's bets are updated. press cancel and see if game pauses
* void request() - see if hit and stay works when buttons are pressed. check if busted player causes game to pause and wait for continue input
* void stay() - check if temp players are skipped
* void requestRound() - check error input loop. see if new round is successfully create. check for proper pausing upon pushing cancel
* void newRound() - check game state, player's chips, deck's cards, and if temp players are activated now. continue playing to see if any other settings are off
* void endGame() - see if game closes without prompting for save

Application Class (only changed functions are documented)

* Application() - check if panels are all properly laid out
* void paint() - check if all panels are updating properly
* void revalidate() - test on aludra to see if revalidate works now when frame is adjusted
* removeCenter() - test multiple menu transitions to see if center frame is removed
* exit() - exit app
* windowClosing - check if save is prompted at correct locations

CreatePanel Class

* CreatePanel(JFrame, Settings) - see if panel components are properly laid out
* update() - see if player list updates when entering the create panel
* paintComponent(Graphics) - call repaint and check for proper updates
* actionPerformed - test each button to see if correct action taken

GamePanel Class

* paintComponent(Graphics) - call repaint and see if game paints properly

MainMenu Class

* MainMenu(JFrame, Settings) - make sure buttons are properly aligned
* paintComponent(Graphics) - see if background is painted
* actionPerformed(ActionEvent) - check if each button goes to proper menu. check if start game prompts if there are no players created and if there are players created.
* create() - create users and see if they are correctly set up. test the cancel button

MenuBar Class

* MenuBar(JFrame, Settings) - see if menu is correctly displaying with all buttons added properly
* actionPerformed(ActionEvent) -test each menu item in different phases of the app

PlayerPanel Class

* PlayerPanel(JFrame, Settings) - see if layout is correct
* paintComponent(Graphics) - see if player buttons update correctly. see if they are properly disabled when not playing game or players are temp
* create(Integer) - see if new player button is added to panel after repaint
* update() - see if buttons shift accordingly when next player is playing
* customize( ) - see if buttons are equally sized and aligned
* disableAction() - see if buttons disable
* enableAction() -see if buttons enable
* actionPerformed(ActionEvent) - push each button to see if correct player is being displayed

SettingsPanel Class

* SettingsPanel(JFrame, Settings) - see if components all setup nicely and line up
* actionPerformed(ActionEvent) - see if each button goes to correct menu or does correct action. check for valid input with chips.
* paintComponent(Graphics) - see if background painted

StatsPanel Class

* StatsPanel(JFrame, Settings) - make sure everything laid out correctly
* actionPerformed(ActionEvent) - see if buttons interact correctly with game
* setText(String) - see if prompt changes
* setTextSize(Integer) - see if prompt size changes by checking alignment
* disableAction() - see if hit and stay are disabled
* enableAction () - see if hit and stay are enabled
* paintComponent(Graphics) - see if correct stats are displayed and correct buttons are turned on and off