Gonzaga Men's Basketball Yahtzee Game

Final Report

Gonzaga University
CPSC 224-02
Spring 2018
Dr. Yanping Zhang

Benjamin Bladow
Eugene Krug
Brandon Niblock

Part 1

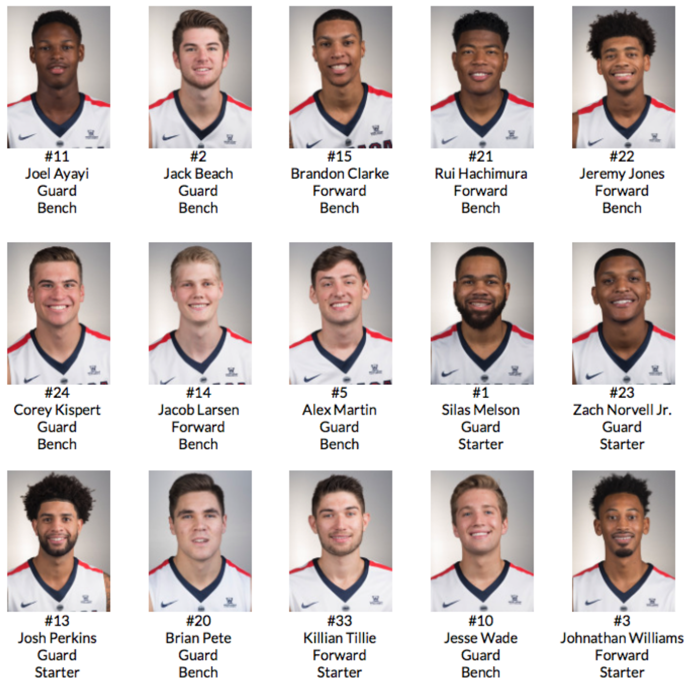
Introduction

1.1 Problem Statement

We were tasked with taking a traditional Yahtzee game, making modifications, and implementing it using a Graphical User Interface (GUI) in the Java programming language. One of the most significant modifications in our game is a theme. Our group elected to use the Gonzaga Men's Basketball team as our theme. We implemented the theme throughout the game. Some highly visible examples of the theme include the fifteen-sided die with each side of the die representing a player on the 2017-2018 team roster. The implementation of the theme will be made more clear throughout this report and in the game. This a multi-user game that tracks users' scores and maintains a leaderboard. For the purposes of clarity, in this final report "user" refers to the people playing the game and "player" refers to a Gonzaga Men's Basketball Player.

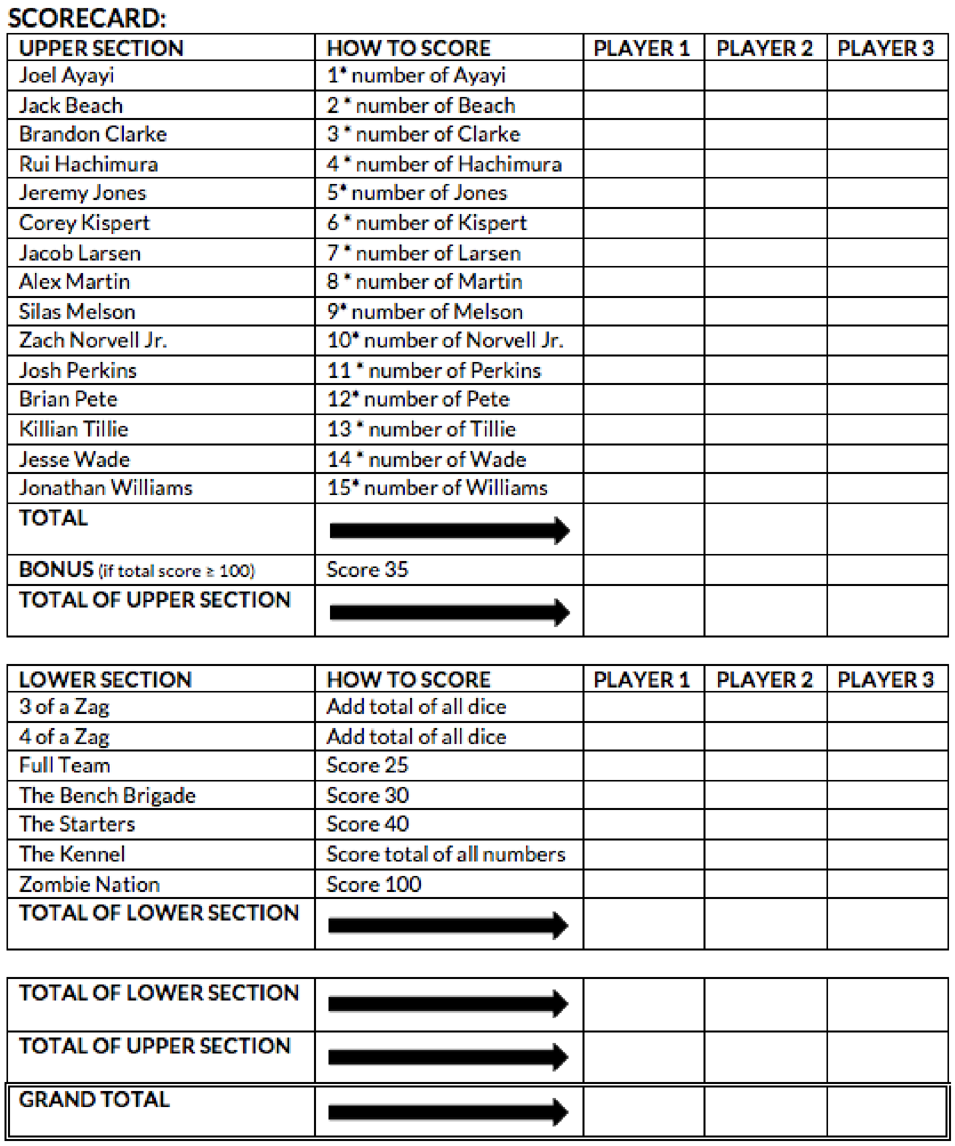
1.2 Game Rules

- **The Dice**
  - The game uses five, fifteen-sided dice.
  - The sides of each die represent the GU Men's Basketball 2017-2018 Roster:


#11 Joel Ayayi Guard Bench
#2 Jack Beach Guard Bench
#15 Brandon Clarke Forward Bench
#21 Rui Hachimura Forward Bench
#22 Jeremy Jones Forward Bench
#24 Corey Kispert Guard Bench
#14 Jacob Larsen Forward Bench
#5 Alex Martin Guard Bench
#1 Silas Melson Guard Starter
#23 Zach Norvell Jr. Guard Starter
#13 Josh Perkins Guard Starter
#20 Brian Pete Guard Bench
#33 Killian Tillie Forward Starter
#10 Jesse Wade Guard Bench
#3 Johnathan Williams Forward Starter

- **How to Win**
  - Roll dice 5 times to get the highest score after all 22 rounds.
- **How to Play**
  - To start, users will roll the dice by clicking the basketball image. After rolling users can either score the current roll on the scorecard by click "Finishing Turn" or reroll any or all of the dice*. Users may only roll the dice a total of five times. After the fifth roll users must choose a line on the scorecard to score their hand in. Users must choose a different line each time (i.e. users CANNOT score their hand in a line they have already scored.)
  - To keep a player, users click their picture. Not clicking a picture will cause that die to be rerolled. For example, if the roll is Jones, Hachimura, Norvell, Tillie, Perkins to keep Jones, Norvell, and Tillie and reroll the others you would: click, not click, click, click, and not click.
  - *Note: For example, after the first roll users can keep 4 dice and reroll the fifth die. Then users can choose to keep all of the die and end that round or keep the first 2 die and roll the next 3 and continue...
- **Scoring**
  - **Upper Section**
    - Each line in the upper section is calculated by the number of the player rolled multiplied by its scaling factor (1-15) as outlined on the scorecard. For example, two Brian Petes in a hand would be of value 24 (2 x 12 = 24).
  - **Lower Section**
    - **3 of a Zag:** A user must have 3 of the same player in their hand. The score is the value of all 5-dice added up.
    - **4 of a Zag:** A user must have 4 of the same player in their hand. The score is the value of all 5-dice added up.
    - **Full Team:** A user must have 3 Guards and 2 Forwards to accomplish this score. The score will be 25 points, regardless of the players number values.
    - **The Bench Brigade:** A user must have 5 players from the Bench to accomplish this score. The score will be 30 points, regardless of players number values.
    - **The Starters:** A user must have all 5 Starters to accomplish this score. The score will be 40 points, regardless of players' number values.
    - **The Kennel:** No combos necessary. The score of this roll will be all the players' numbers totaled together.
    - **Zombie Nation:** A user must have 5 of the same player to accomplish this score. The score will be 100 points, regardless of players' number values.

– **Scorecard**



SCORECARD:

UPPER SECTION	HOW TO SCORE	PLAYER 1	PLAYER 2	PLAYER 3
Joel Ayayi	1* number of Ayayi			
Jack Beach	2 * number of Beach			
Brandon Clarke	3 * number of Clarke			
Rui Hachimura	4 * number of Hachimura			
Jeremy Jones	5* number of Jones			
Corey Kispert	6 * number of Kispert			
Jacob Larsen	7 * number of Larsen			
Alex Martin	8 * number of Martin			
Silas Melson	9* number of Melson			
Zach Norvell Jr.	10* number of Norvell Jr.			
Josh Perkins	11 * number of Perkins			
Brian Pete	12* number of Pete			
Killian Tillie	13 * number of Tillie			
Jesse Wade	14 * number of Wade			
Jonathan Williams	15* number of Williams			
TOTAL	➡			
BONUS (if total score ≥ 100)	Score 35			
TOTAL OF UPPER SECTION	➡			

LOWER SECTION	HOW TO SCORE	PLAYER 1	PLAYER 2	PLAYER 3
3 of a Zag	Add total of all dice			
4 of a Zag	Add total of all dice			
Full Team	Score 25			
The Bench Brigade	Score 30			
The Starters	Score 40			
The Kennel	Score total of all numbers			
Zombie Nation	Score 100			
TOTAL OF LOWER SECTION	➡			

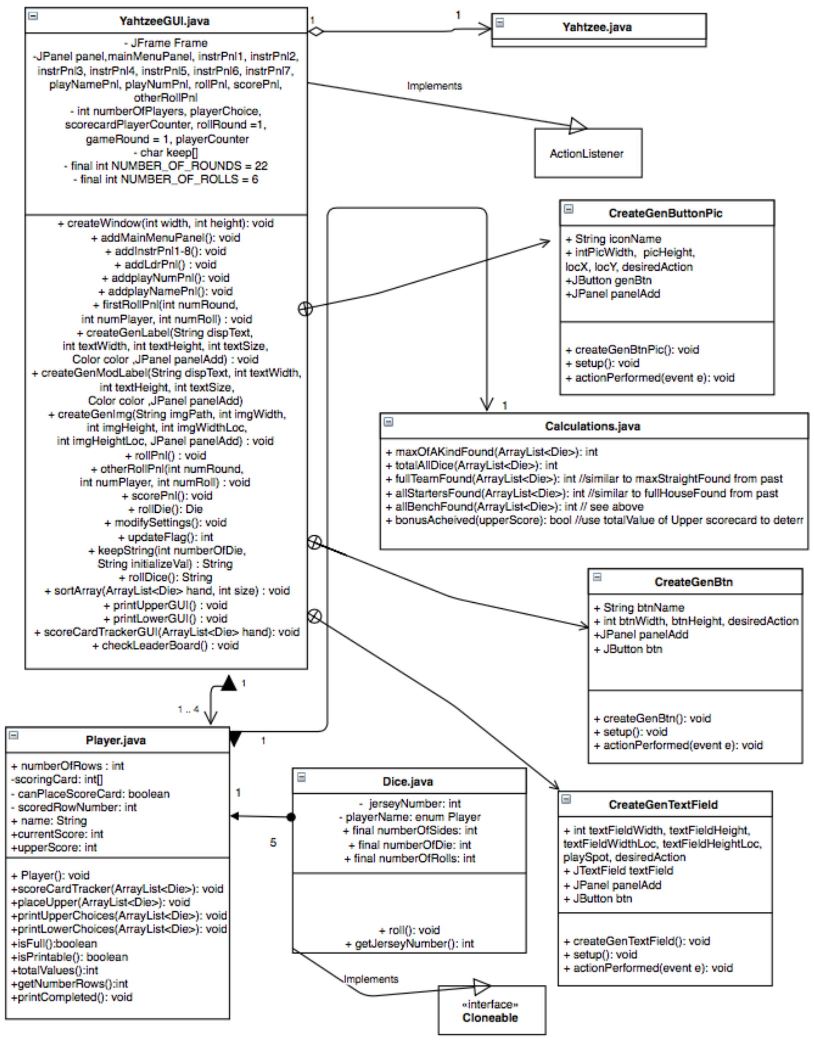
TOTAL OF LOWER SECTION	➡			
TOTAL OF UPPER SECTION	➡			
GRAND TOTAL	➡			

Part 2

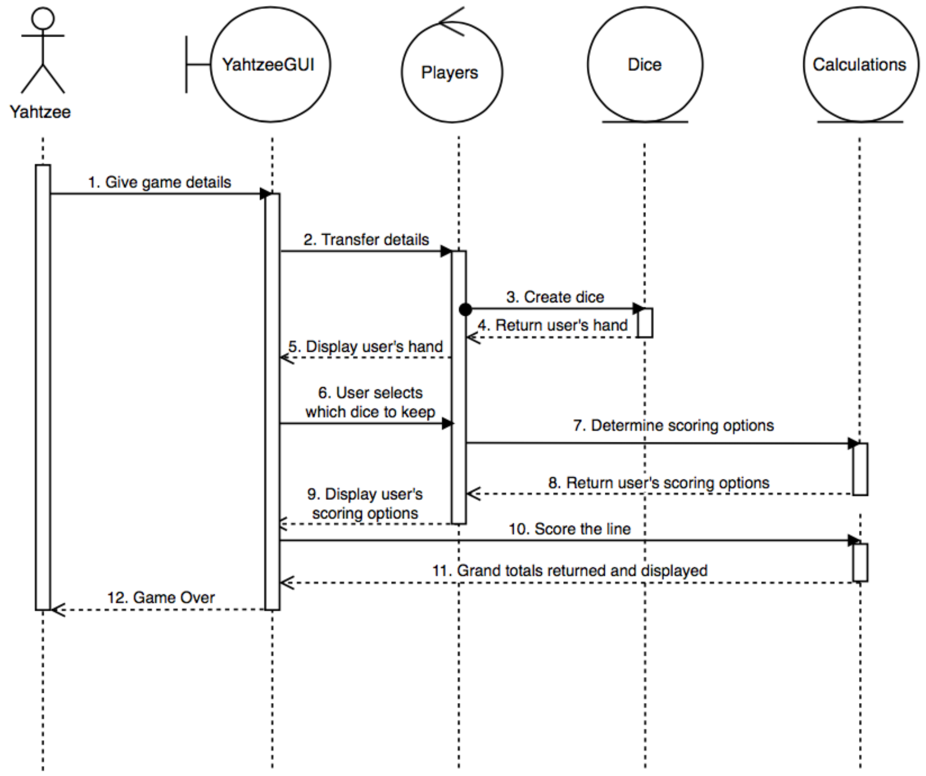
Program Design and Implementation

2.1 UML Class Diagram

There is one exception to our rule of the distinction between "user" and "player". We have a "Player" class that should be named "User" class instead. The exception is in our code and reflected in both of the diagrams. This exception to our rule exists because of legacy code from Programming Assignment 6.


YahtzeeGUI.java
- JFrame Frame
-JPanel panel,mainMenuPanel, instrPnl1, instrPnl2, instrPnl3, instrPnl4, instrPnl5, instrPnl6, instrPnl7, playNamePnl, playNumPnl, rollPnl, scorePnl, otherRollPnl
- int numberOfPlayers, playerChoice, scorecardPlayerCounter, rollRound =1, gameRound = 1, playerCounter
- char keep[]
- final int NUMBER_OF_ROUNDS = 22
- final int NUMBER_OF_ROLLS = 6
+ createWindow(int width, int height): void
+ addMainMenuPanel(): void
+ addInstrPnl1-8(): void
+ addLdrPnl() : void
+ addplayNumPnl(): void
+ addplayNamePnl(): void
+ firstRollPnl(int numRound, int numPlayer, int numRoll) : void
+ createGenLabel(String dispText, int textWidth, int textHeight, int textSize, Color color ,JPanel panelAdd) : void
+ createGenModLabel(String dispText, int textWidth, int textHeight, int textSize, Color color ,JPanel panelAdd)
+ createGenImg(String imgPath, int imgWidth, int imgHeight, int imgWidthLoc, int imgHeightLoc, JPanel panelAdd) : void
+ rollPnl() : void
+ otherRollPnl(int numRound, int numPlayer, int numRoll) : void
+ scorePnl(): void
+ rollDie(): Die
+ modifySettings(): void
+ updateFlag(): int
+ keepString(int numberOfDie, String initializeVal) : String
+ rollDice(): String
+ sortArray(ArrayList<Die> hand, int size) : void
+ printUpperGUI() : void
+ printLowerGUI() : void
+ scoreCardTrackerGUI(ArrayList<Die> hand): void
+ checkLeaderBoard() : void
1
1
Yahtzee.java
Implements
ActionListener
CreateGenButtonPic
+ String iconName
+ intPicWidth, picHeight, locX, locY, desiredAction
+JButton genBtn
+JPanel panelAdd
+ createGenBtnPic(): void
+ setup(): void
+ actionPerformed(event e): void
1
Calculations.java
+ maxOfAKindFound(ArrayList<Die>): int
+ totalAllDice(ArrayList<Die>): int
+ fullTeamFound(ArrayList<Die>): int //similar to maxStraightFound from past
+ allStartersFound(ArrayList<Die>): int //similar to fullHouseFound from past
+ allBenchFound(ArrayList<Die>): int // see above
+ bonusAcheived(upperScore): bool //use totalValue of Upper scorecard to deterr
CreateGenBtn
+ String btnName
+ int btnWidth, btnHeight, desiredAction
+JPanel panelAdd
+ JButton btn
+ createGenBtn(): void
+ setup(): void
+ actionPerformed(event e): void
1
1 .. 4
1
Player.java
+ numberOfRows : int
-scoringCard: int[]
- canPlaceScoreCard: boolean
- scoredRowNumber: int
+ name: String
+currentScore: int
+upperScore: int
+ Player(): void
+scoreCardTracker(ArrayList<Die>): void
+placeUpper(ArrayList<Die>): void
+printUpperChoices(ArrayList<Die>): void
+printLowerChoices(ArrayList<Die>): void
+isFull():boolean
+isPrintable(): boolean
+totalValues():int
+getNumberRows():int
+printCompleted(): void
1
5
Dice.java
- jerseyNumber: int
- playerName: enum Player
+ final numberOfSides: int
+ final numberOfDie: int
+ final numberOfRolls: int
+ roll(): void
+ getJerseyNumber(): int
Implements
«interface»
Cloneable
CreateGenTextField
+ int textFieldWidth, textFieldHeight, textFieldWidthLoc, textFieldHeightLoc, playSpot, desiredAction
+ JTextField textField
+ JPanel panelAdd
+ JButton btn
+ createGenTextField(): void
+ setup(): void
+ actionPerformed(event e): void

2.2 UML Sequence Diagram



2.3 Business Function Model (BFM)

Objective: Play a multiuser game of Gonzaga Men's Basketball Yahtzee in a Graphical User Interface (GUI).

- Starting a game
  - Display the leaderboard (top 3 highest scores in game history)
  - Display game instructions
  - Determine number of users
  - Get the user's name(s)
- Rolling a hand
  - Roll all five dice
  - Determine if rerolls are available
  - Prompt the user to keep dice
  - Reroll all the dice not kept
  - Determine whether to proceed to scoring (all dice have been kept or all available rolls have been used)

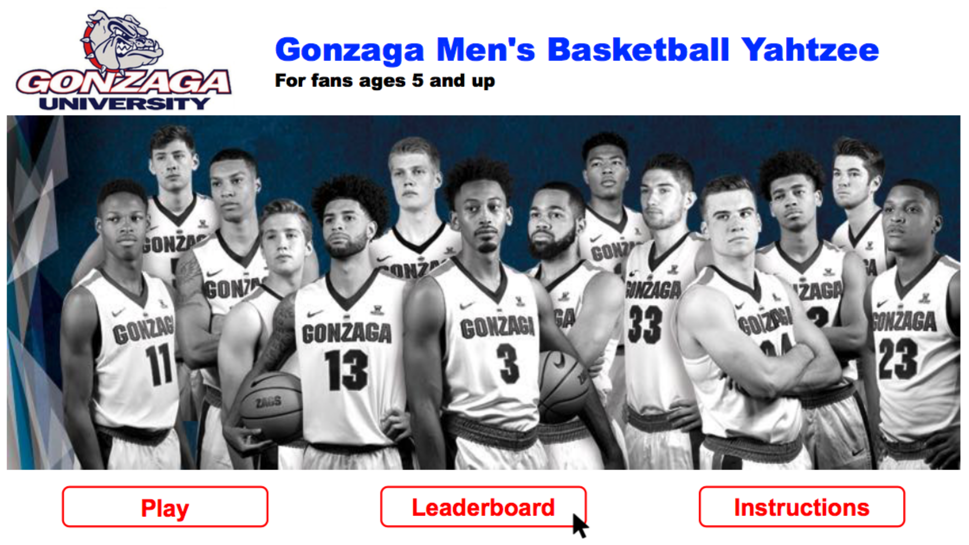
- Scoring a hand
  - Determine the unused scorelines
  - Calculate the score for each unused line
  - Display calculated scores for eligible scoring options
  - Prompt the user to select a scoring option
  - Record the score of the selected option to the user's scorecard
- Finishing a game
  - Determine all users have scored all lines
  - Display the grand total for each user (including the bonus if applicable)
  - Determine if any user's score qualifies them for the leaderboard
  - Update the leaderboard if applciable
  - Thank the user for playing
  - Inform the user to view the leaderboard or to play again they must close the current window and reload the game

2.4 Functional Requirements

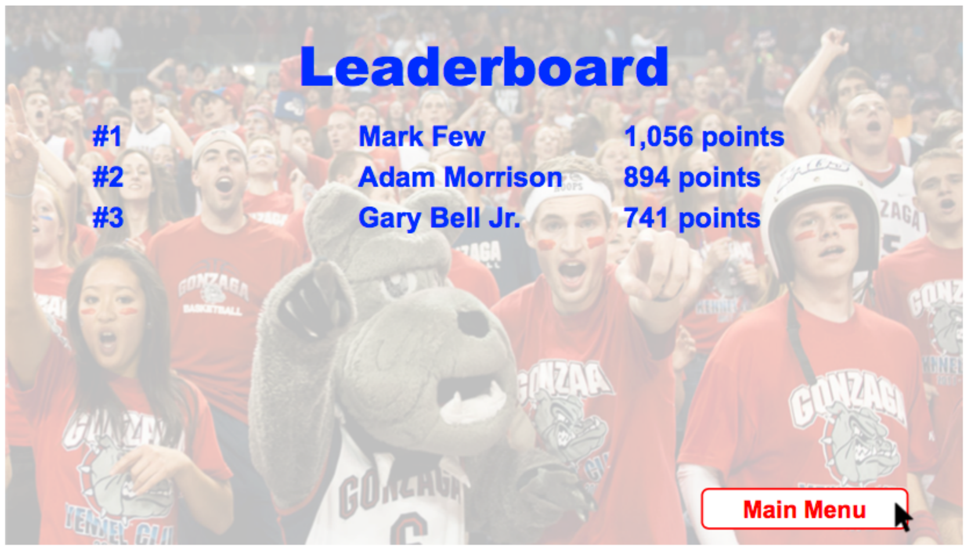
- **Starting a game:** At the start of the game, the user has the option to view the leaderboard (top 3 highest scores in game history). The user also has the option to view the game instructions, as well as each basketball player's name, their number, their position, and if they are a starter or bench, as this is key to scoring for the rest of the game. When starting the game, the user will be prompted to select the number of users for the game and then to enter each user's name. As the users play the game, the game will state who's turn it is.
- **Rolling a hand:** To roll a new hand the first step is to roll all five dice, randomly generating five Gonzaga basketball players. The game will state which roll number the current roll is. The game will prompt the user to select which players to keep in their hand, and which to reroll. The dice that are not kept will be rerolled. At this point, the user's turn will proceed to scoring if one of two conditions is met: either all of the dice have been kept or all available rolls have been used.
- **Scoring a hand:** After each turn, the unused scoreboard lines and the value the user is eligible score on those lines is displayed. The user's current score (before scoring the current hand) is displayed. The user is prompted to input a line number they wish to score. The game only allows the user to choose from unused lines.
- **Finishing a game:** The game is over once all users have scored all lines. At that point the grand total for each user is displayed (including the bonus, if applicable). If any user's score qualifies them for the leaderboard, the leaderboard will be updated. The users will be thanked for playing and informed to view the leaderboard or to play again they must close the current window and reload the game.

2.5 User Experience Mockup

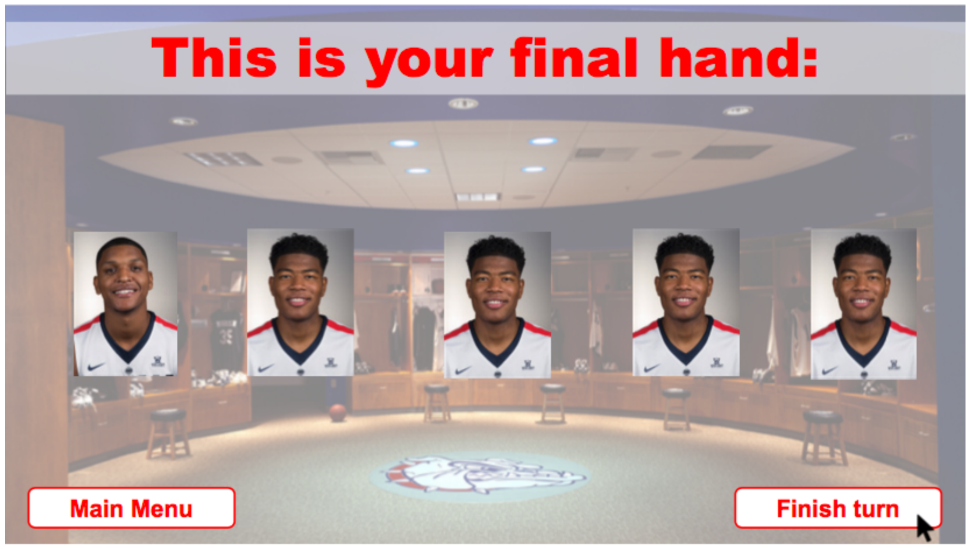
As part of the planning stages, we created a user experience mockup that showed our design for the look and feel of the user experience. Due to the length of the mockup document, we are only including three samples below from the document (screenshots on the next page). The entire document was turned in separately and previously. In general, we followed our mockup design very closely. It was a great resource for us. The main change we made when actually implementing the program is that we did not use background images. Instead, we used a simple color for the background. This decision was made because we were not seeing the results we anticipated.


Gonzaga Men's Basketball Yahtzee

For fans ages 5 and up

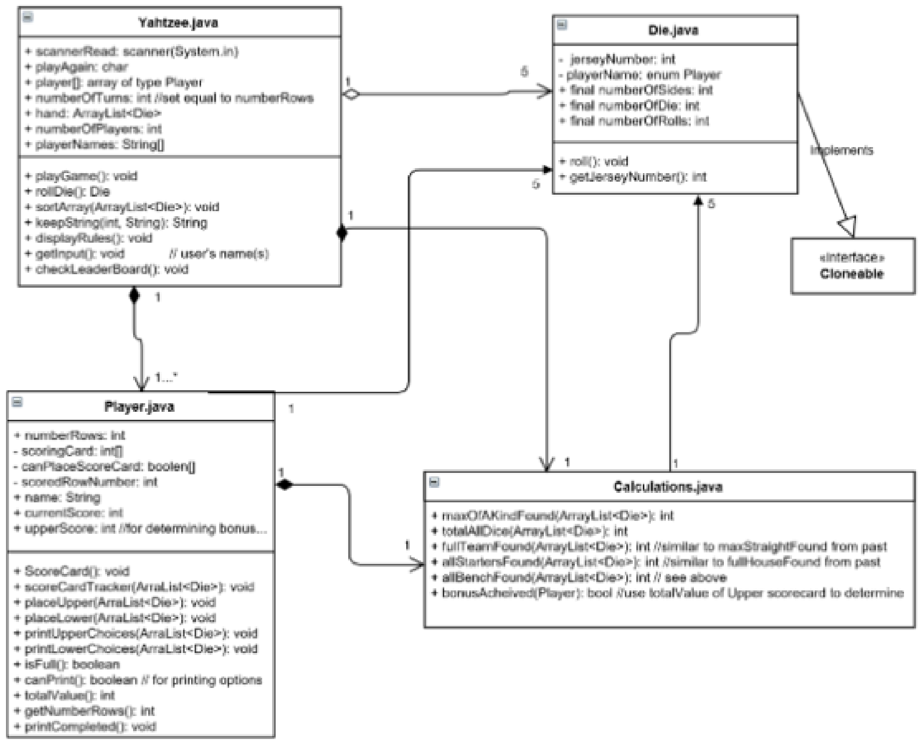

Leaderboard

#1	Mark Few	1,056 points
#2	Adam Morrison	894 points
#3	Gary Bell Jr.	741 points

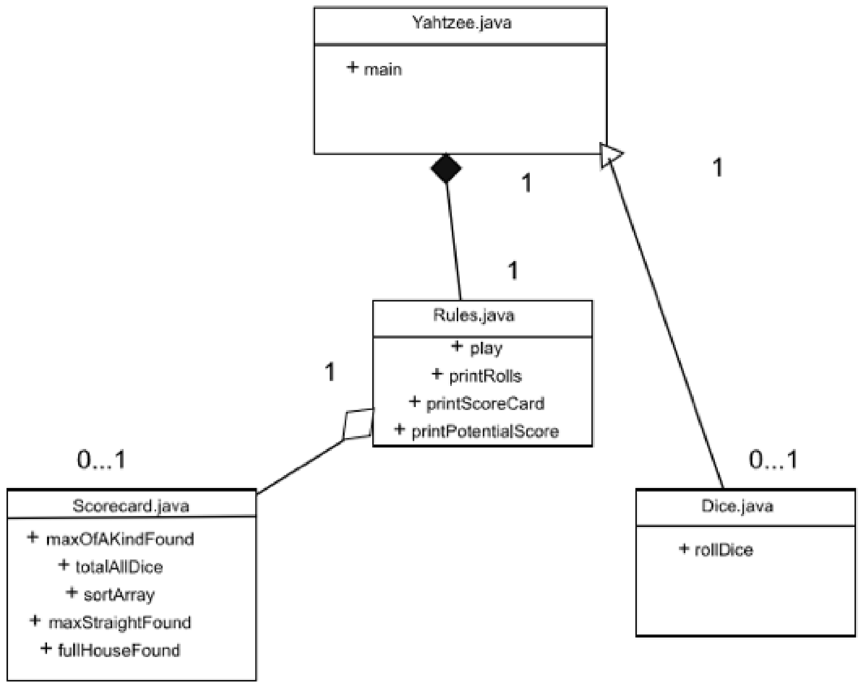

This is your final hand:

2.6 Design Alternative Analysis

- **Design Alternative Analysis 1**
  - **The issue:** The first problem our group faced with the project is how to incorporate each group member's code from Programming Assignment 6 into a foundation for the group project. Each group member had a successful Programming Assignment 6, so we considered everyone's code. We mainly focused our decision on class design structure and studied each group member's UML diagram. We realized Eugene and Brandon had similar class design structures so for the sake of making a decision we thought of their assignments as one. We evaluated three alternatives.
  - **Available alternatives**
    - Alternative 1: Keep all of Benjamin's code. There are many methods per class in his design.
    - Alternative 2: Keep all of Brandon's code. Brandon's code has fewer methods and each method does more work in comparison to Benjamin's code. Eugene's class design structure is similar to Brandon's and will not be evaluated.
    - Alternative 3: Combine the best aspects of Benjamin's and Brandon's class structure.
  - **First alternative**
    - In this design, we have four classes: Calculations, Player, Die, and Yahtzee. Each of the classes contains many methods.
  - **Pros**
    - Since there are many methods in each of the classes the methods do less (in theory, one task). This modularized form of programming generally makes the code easier to read and debug.
    - Code can be adapted for multiple users much easier.
  - **Cons**
    - The design could be considered more complicated because there seems to be more "going on".
    - The comments could be improved.


Yahtzee.java
+ scannerRead: scanner(System.in)
+ playAgain: char
+ player[]: array of type Player
+ numberOfTurns: int //set equal to numberRows
+ hand: ArrayList<Die>
+ numberOfPlayers: int
+ playerNames: String[]
+ playGame(): void
+ rollDie(): Die
+ sortArray(ArrayList<Die>): void
+ keepString(int, String): String
+ displayRules(): void
+ getInput(): void // user's name(s)
+ checkLeaderBoard(): void
Die.java
- jerseyNumber: int
- playerName: enum Player
+ final numberOfSides: int
+ final numberOfDie: int
+ final numberOfRolls: int
+ roll(): void
+ getJerseyNumber(): int
Implements
«interface»
Cloneable
Player.java
+ numberRows: int
- scoringCard: int[]
- canPlaceScoreCard: boolen[]
- scoredRowNumber: int
+ name: String
+ currentScore: int
+ upperScore: int //for determining bonus...
+ ScoreCard(): void
+ scoreCardTracker(ArraList<Die>): void
+ placeUpper(ArraList<Die>): void
+ placeLower(ArraList<Die>): void
+ printUpperChoices(ArraList<Die>): void
+ printLowerChoices(ArraList<Die>): void
+ isFull(): boolean
+ canPrint(): boolean // for printing options
+ totalValue(): int
+ getNumberRows(): int
+ printCompleted(): void
Calculations.java
+ maxOfAKindFound(ArrayList<Die>): int
+ totalAllDice(ArrayList<Die>): int
+ fullTeamFound(ArrayList<Die>): int //similar to maxStraightFound from past
+ allStartersFound(ArrayList<Die>): int //similar to fullHouseFound from past
+ allBenchFound(ArrayList<Die>): int // see above
+ bonusAcheived(Player): bool //use totalValue of Upper scorecard to determine
1
5
5
1
5
1
1...*
1
1
1
1
1

- **Second alternative**
  - In this class design, we have four classes: Dice, Scorecard, Rules, and Yahtzee.
- **Pros**
  - The sorting method is simpler because it does not involve cloneable.
  - When looking at the UML Class Diagram, this design appears to be simpler than the first alternative.
- **Cons**
  - The code is more difficult to read and debug since each method does more than one task.
  - Many of the methods are static.
  - There are very few private variables with getter and setter functions.



- **Third alternative**
  - In this design, we combine Benjamin's and Brandon's Programming Assignment 6 to use as a foundation for the group project.
- **Pros**
  - Combines the best aspects of both assignments.
- **Cons**
  - No group member is an immediate expert of the code.
- **Decision:** Our group elected to utilize the first alternative. The most significant benefit of the first alternative is that the code adapts with the most ease to a multi-user game. We believe the third alternative would be time-intensive to implement considering variable names, method names, etc. would not match up. Benjamin briefed the entire group on his Programming Assignment 6 code, so that all group members feel comfortable with it and can work with it moving forward.
- **Reflection on choice:** We did make the correct decision by choosing the first alternative. We added one additional class to the original design: YahtzeeGUI. The code from first alternative adapted very well to a multiuser game. Benjamin did a great job of briefing the group on his Programming Assignment 6 code. We all felt very comfortable working with it.

- **Design Alternative Analysis 2**
  - **The issue:** The problem that our group is encountering is playing the game with more than one user. Our code has a specific player (user) class, which means that we can easily implement more than one user playing the game at a time. When playing a game of Yahtzee in real life, we usually play the game one round at a time, however when we were looking at our code, it works most efficiently when we play the game one user at a time, and then clearing the table, and playing again with the next user.
  - **Available alternatives**
    - Alternative 1: Users play at the same time
    - Alternative 2: Users play at different times
  - **First alternative**
    - In this design, we have the users play at the same time. Figuratively, one user does their turn and then "hands over" the dice to the next user so that they can do their turn.
  - **Pros**
    - This option allows the game to finish much quicker, as each user will be allowed to go one at a time.
    - Users do not spend as much time waiting, since each user's turns will be a lot closer together rather than one whole game at a time.
    - The game will feel less like a single user game and will cause users to be more competitive.
  - **Cons**
    - This will take much longer to implement.
    - This will cause more memory to be used and will make our code less efficient.
    - We want the users to feel more competitive with the leaderboard than with the other users.
  - **Second alternative**
    - In this design, we have users play one at a time, so that each user plays their own game, and then "passes" the dice on to the next user.
  - **Pros**
    - This would be much simpler to implement.
    - We could get rid of the memory at the end of the round, and then reuse the storage for the next play through of the game.
    - The users will feel like they are playing against the leaderboard rather than each other.
  - **Cons**
    - There is a longer wait time between turns since each user must wait until their turn to play the full game.
    - The is not the traditional way a multiuser (multiplayer) Yahtzee game is played.
  - **Decision:** We chose to go with alternative one, because although it would be more difficult to implement, we want to have an emphasis on realism. Alternative one will look more similar to a real game of Yahtzee than the other alternative. We also do not want the non-playing users to get bored as they wait for their turn to play the game.
  - **Reflection on choice:** We are very happy we decided on the first alternative. Having the users all play the game at the same time and "passing the dice" back and forth resembles a traditional Yahtzee game. This alternative keeps all of the users engaged with the game.

- **Design Alternative Analysis 3**
  - **The issue:** The problem our group is facing is how to implement the GU Men's Basketball Player Numbers. Specifically, we are focusing on how to randomly generate a player and how to sort the players. We initially were planning to use their jersey numbers. However, the problem is that there is a wide range of jersey numbers (anywhere from 1 to 33) and the jersey numbers are not consecutive (i.e. there are players with jersey numbers 1, 2, 3 and 5, but no number 4). This presents a problem because we cannot simply use the random function in Java. We explored two alternatives for this problem.
  - **Available alternatives**
    - Alternative 1: Utilize a random function with enum values
    - Alternative 2: Assign consecutive integer values between 1 and 15 to each player and utilize the random function we are familiar with
  - **First alternative**
    - In this design, we utilize a random function in Java for enum values. Each of the user's names is an enum value, so it would be possible with the function to select one.
  - **Pros**
    - This option is rather straightforward and we found examples of how to implement it on the internet. One example we found was randomly selecting a card suit (i.e. Spades, Hearts, Diamonds, Clubs). Our case would be similar.
  - **Cons**
    - We have never used the random function with enum values, so it is not something we are very familiar with.

  An example found on the internet of implementing a random function with enum values is below. We would replace the card suits with the names of the players on the roster and modify the range to be 0 to 14.

```
public enum CardSuit
{
        Spades,
        Hearts,
        Diamonds,
        Clubs
}

void Start ()
{
        cardSuit = (CardSuit)Random.Range(0, 3);
}
```

  - **Second alternative**
    - In this design, each player on the roster is assigned an integer value between 1 and 15. We utilize the "normal" and familiar random function to generate a random integer value between 1 and 15. Then, the number generated is matched back to the corresponding player on the roster and further operations are conducted with that player.
  - **Pros**
    - We completely understand what is happening here.
  - **Cons**
    - It will require many lines of code in comparison to the first alternative.

```
/**
* "Rolls" the die by finding a random double between 0.0 and 1.0.
* That result is multiplied by the number of sides + 1 and is
* casted to an integer. A series of conditions are checked, such that
* the result of the roll is matched with its corresponding player
* from the roster. In addition, the player is assigned his
* corresponding position and status.
* @param N/A
* @returns N/A
* @throws N/A
*/

public void roll(){
        jerseyNumber = (int)(Math.random() * numberOfSides + 1);

        if(jerseyNumber == 3 || jerseyNumber == 4 ||
          jerseyNumber == 5 || jerseyNumber == 7 ||
          jerseyNumber == 13 || jerseyNumber == 15)
                position = "FORWARD";
        else
                position = "GUARD";

        if(jerseyNumber == 9 || jerseyNumber == 10 ||
          jerseyNumber == 11 || jerseyNumber == 13 ||
          jerseyNumber == 15)
                status = "STARTER";
        else
                status = "BENCH";

        if(jerseyNumber == 1)
                name = "Joel Ayayi";
        else if(jerseyNumber == 2)
                name = "Jack Beach";
        else if(jerseyNumber == 3)
                name = "Brandon Clarke";
        else if(jerseyNumber == 4)
                name = "Rui Hachimura";
        else if(jerseyNumber == 5)
                name = "Jeremy Jones";
        else if(jerseyNumber == 6)
                name = "Corey Kispert";
        else if(jerseyNumber == 7)
                name = "Jacob Larson";
        else if(jerseyNumber == 8)
                name = "Alex Martin";
        else if(jerseyNumber == 9)
                name = "Silas Melson";
        else if(jerseyNumber == 10)
                name = "Zach Norvell Jr.";
        else if(jerseyNumber == 11)
                name = "Josh Perkins";
        else if(jerseyNumber == 12)
                name = "Brian Pete";
        else if(jerseyNumber == 13)
                name = "Killian Tillie";
        else if(jerseyNumber == 14)
                name = "Jesse Wade";
        else
                name = "Jonathan Williams";
}
```

– **Decision:** We elected to go with the second alternative because we understood it the best. We know it will work and how it will work, so we were simply just most comfortable with it. The user will not know the difference from what they see and interact with.

– **Reflection on choice:** We slightly modified what the second alternative means. However, we are happy with the choice we made to choose the second alternative. We do assign an integer value between 1 and 15 to each player and we did utilize the if/else if/else conditions as written above. Instead of matching that value back to the player's corresponding jersey number we instead just use the integer value 1 - 15. We call that assigned value their scaling factor. When we utilized player's jersey numbers in the game, the scoring became "clunky" because there is a considerable spread between players' jersey numbers (0 - 33). This made certain players way more "valuable" than we ever intended. We are comfortable with the code behind this alternative.

Part 3

Testing

3.1 Test Plan

- **Starting a game:** At the start of the game, the user has the option to view the leaderboard (top 3 highest scores in game history). The user also has the option to view the game instructions, as well as each basketball player's name, their number, their position, and if they are a starter or bench, as this is key to scoring for the rest of the game. When starting the game, the user will be prompted to select the number of users for the game and then to enter each user's name. As the users play the game, the game will state who's turn it is.

Requirement ID	**Requirement**	**Verified By**
1.0	Starting a game	Test Case 1
1.1	Leaderboard displayed (by click)	Test Case 1 Step D
1.2	Instructions displayed (by click)	Test Case 1 Step E
1.3	Enter number of users	Test Case 1 Step F
1.4	Enter user's name(s)	Test Case 1 Step G
1.5	Display who's turn it is	Test Case 1 Step H

- **Rolling a game:** To roll a new hand the first step is to roll all five dice, randomly generating five Gonzaga basketball players. The game will state which roll number the current roll is. The game will prompt the user to select which players to keep in their hand, and which to reroll. The dice that are not kept will be rerolled. At this point, the user's turn will proceed to scoring if one of two conditions is met: either all of the dice have been kept or all available rolls have been used.

Requirement ID	**Requirement**	**Verified By**
2.0	Rolling a game	Test Case 2
2.1	Roll all the dice by clicking the ball to roll	Test Case 2 Step A
2.2	Display the pictures of the five Gonzaga basketball players rolled	Test Case 2 Step B
2.3	Display the current roll number	Test Case 2 Step C
2.4	User clicks the players they want to keep	Test Case 2 Step D
2.5	Appropriate rerolls are made	Test Case 2 Step E
2.5	Proceed to scoring if either all the dice have been kept or all rolls have been used, else continue rolling	Test Case 2 Step F

- **Scoring a hand:** After each turn, the unused scoreboard lines and the value the user is eligible score on those lines is displayed. The user's current score (before scoring the current hand) is displayed. The user is prompted to input a line number they wish to score. The game only allows the user to choose from unused lines.

Requirement ID	Requirement	Verified By
3.0	Scoring a hand	Test Case 3
3.1	Display available lines to score (all lines unused) as well as the points they are eligible to score on each	Test Case 3 Step A
3.2	Display current score (before scoring the current hand)	Test Case 3 Step B
3.3	User selects scoreline to score (bad input not accepted)	Test Case 3 Step C
3.4	User has/does not have more rolls remaining	Test Case 3 Step D
3.5	Case to switch to the next user or case to transition to finishing a game	Test Case 3 Step E

- **Finishing a game** The game is over once all users have scored all lines. At that point the grand total for each user is displayed (including the bonus, if applicable). If any user's score qualifies them for the leaderboard, the leaderboard will be updated. The users will be thanked for playing and informed to view the leaderboard or to play again they must close the current window and reload the game.

Requirement ID	Requirement	Verified By
4.0	Finishing a game	Test Case 4
4.1	Determine all users have scored all lines	Test Case 4 Step A
4.2	Display each user's grand total (including the bonus, if applicable)	Test Case 4 Step B
4.3	Determine if any user's score qualifies for the leaderboard	Test Case 4 Step C
4.4	Update the leaderboard if necessary	Test Case 4 Step D
4.5	Thank the user(s) for playing and inform them to view the leaderboard or to play again they must close the current window and reload the game	Test Case 4 Step E

Test Cases

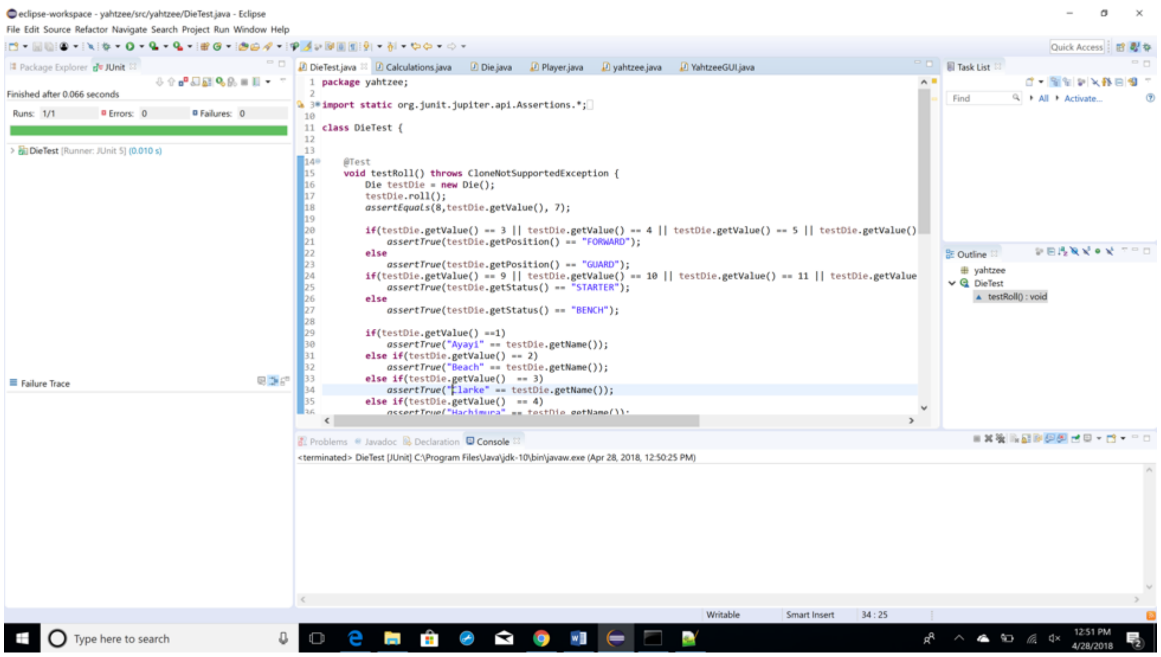
1. Test Case 1: New Game
   - A. Launch game
   - B. Display main menu
   - C. Provide options for Play, Leaderboard, Instructions
   - D. Display the leaderboard when the user clicks "Leaderboard"
     - i. Return to the main menu by clicking "Main Menu"
   - E. Display the instructions when the user clicks "Instructions"
     - i. Return to the main menu by clicking "Main Menu"
   - F. User clicks desired number of users (1-4)
   - G. Enter user's name(s)
   - H. Display who's turn it is

2. Test Case 2: Rolling a hand
   - A. Roll all the dice by click the ball to roll
   - B. Display the pictures of the five Gonzaga basketball players rolled
   - C. Display the current roll number
   - D. User clicks the players they want to keep
     - i. User makes a selection of keeps and rerolls
     - ii. User decides to keep all of the dice
   - E. Appropriate rerolls are made
   - F. Proceed to scoring if either all the dice have been kept or all rolls have been used, else continue rolling
3. Test Case 3: Scoring a hand
   - A. Display available lines to score (all lines unused) as well as the points they are eligible to score on each
   - B. Display current score (before scoring the current hand)
   - C. User selects scoreline to score
     - i. Bad input not accepted
   - D. Determine number of rolls remaining
     - i. If user has more rolls remaining, return to Test Case Step A
     - ii. If user does not have rolls remaining, continue...
   - E. Determine how many users still need to play the game
     - i. If more users still need to play, return to Test Case Step A
     - ii. If no users still need to play, continue...
4. Test Case 4: Finishing a game
   - A. Determine all users have scored all lines
   - B. Display each user's grand total (including the bonus, if applicable)
   - C. Determine if any user's score qualifies for the leaderboard
   - D. Update the leaderboard if necessary
   - E. Thank the user(s) for playing and inform them to view the leaderboard or to play again they must close the current window and reload the game

3.2 JUnit Testing

We utilized JUnit, a unit testing framework for the Java programming language, to test our code. The code and screenshots for our JUnit testing have been turned in separately. Below are the screenshots of successful JUnit testing, which includes a small sampling of the code utilized for the test.

- **JUnit Testing for Die.java**



Finished after 0.066 seconds

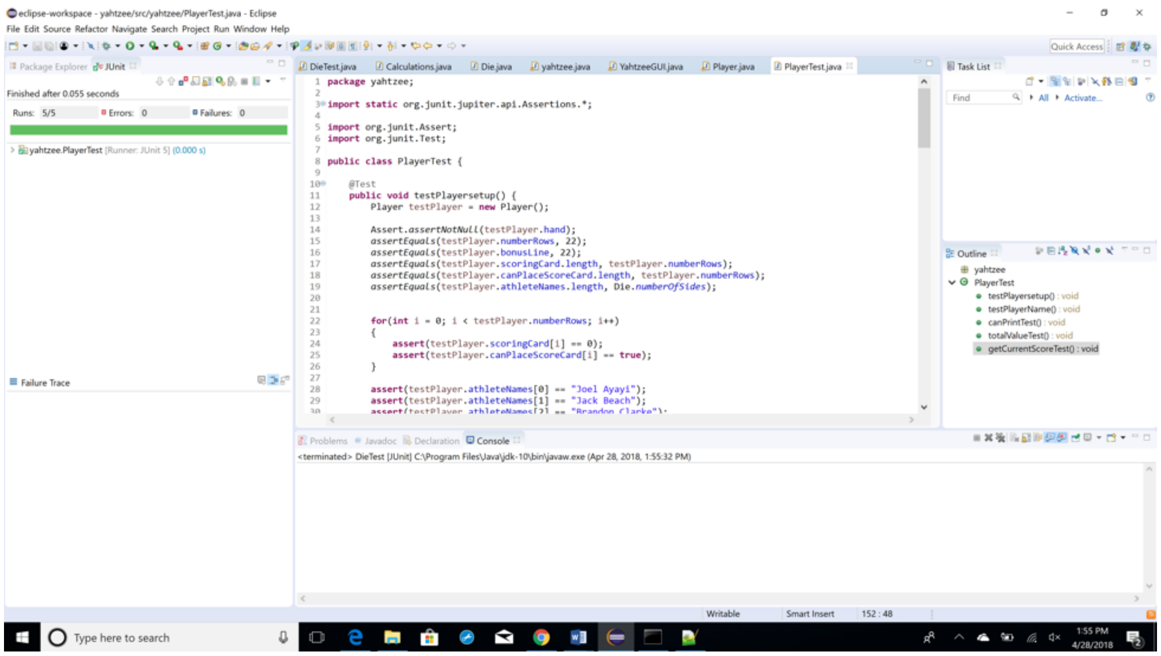
Runs: 1/1 Errors: 0 Failures: 0

DieTest [Runner: JUnit 5] (0.010 s)

```java
package yahtzee;

import static org.junit.jupiter.api.Assertions.*;

class DieTest {


    @Test
    void testRoll() throws CloneNotSupportedException {
        Die testDie = new Die();
        testDie.roll();
        assertEquals(8,testDie.getValue(), 7);

        if(testDie.getValue() == 3 || testDie.getValue() == 4 || testDie.getValue() == 5 || testDie.getValue()
            assertTrue(testDie.getPosition() == "FORWARD");
        else
            assertTrue(testDie.getPosition() == "GUARD");
        if(testDie.getValue() == 9 || testDie.getValue() == 10 || testDie.getValue() == 11 || testDie.getValue
            assertTrue(testDie.getStatus() == "STARTER");
        else
            assertTrue(testDie.getStatus() == "BENCH");

        if(testDie.getValue() ==1)
            assertTrue("Ayayi" == testDie.getName());
        else if(testDie.getValue() == 2)
            assertTrue("Beach" == testDie.getName());
        else if(testDie.getValue()  == 3)
            assertTrue("Clarke" == testDie.getName());
        else if(testDie.getValue()  == 4)
            assertTrue("Hachimura" == testDie.getName());
```

```
<terminated> DieTest [JUnit] C:\Program Files\Java\jdk-10\bin\javaw.exe (Apr 28, 2018, 12:50:25 PM)
```

- **JUnit Testing for Player.java**



Finished after 0.055 seconds

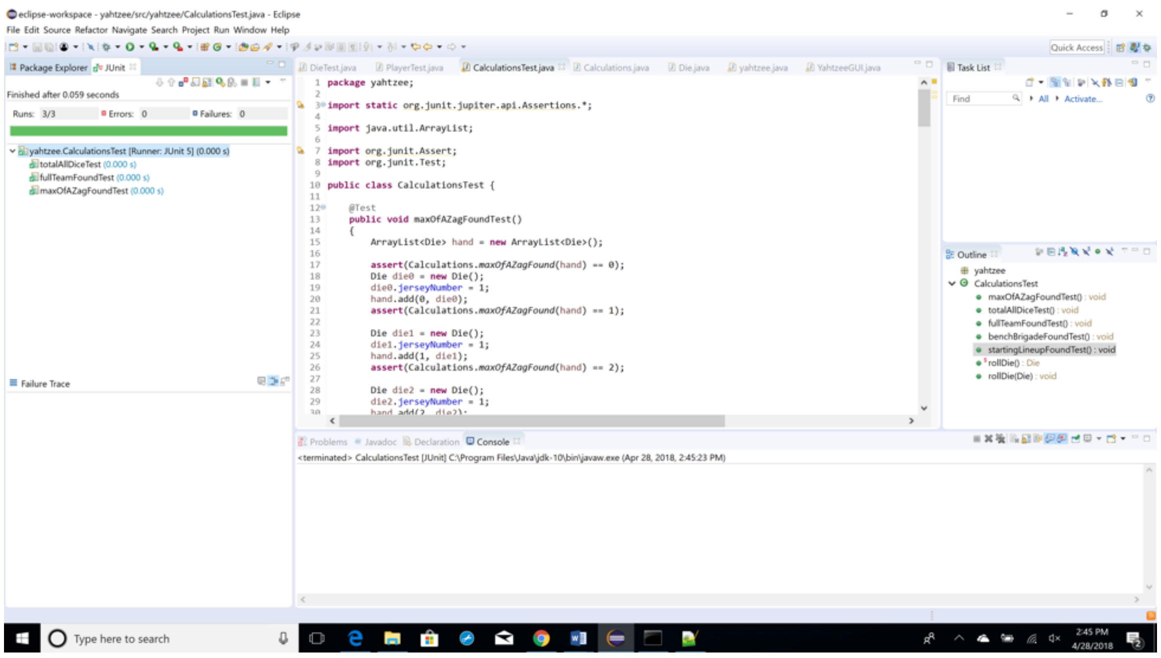
Runs: 5/5 Errors: 0 Failures: 0

yahtzee.PlayerTest [Runner: JUnit 5] (0.000 s)

```java
package yahtzee;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.Assert;
import org.junit.Test;

public class PlayerTest {

    @Test
    public void testPlayersetup() {
        Player testPlayer = new Player();

        Assert.assertNotNull(testPlayer.hand);
        assertEquals(testPlayer.numberRows, 22);
        assertEquals(testPlayer.bonusLine, 22);
        assertEquals(testPlayer.scoringCard.length, testPlayer.numberRows);
        assertEquals(testPlayer.canPlaceScoreCard.length, testPlayer.numberRows);
        assertEquals(testPlayer.athleteNames.length, Die.numberOfSides);


        for(int i = 0; i < testPlayer.numberRows; i++)
        {
            assert(testPlayer.scoringCard[i] == 0);
            assert(testPlayer.canPlaceScoreCard[i] == true);
        }

        assert(testPlayer.athleteNames[0] == "Joel Ayayi");
        assert(testPlayer.athleteNames[1] == "Jack Beach");
        assert(testPlayer.athleteNames[2] == "Brandon Clarke");
```

```
<terminated> DieTest [JUnit] C:\Program Files\Java\jdk-10\bin\javaw.exe (Apr 28, 2018, 1:55:32 PM)
```

- **JUnit Testing for Calculations.java**



eclipse-workspace - yahtzee/src/yahtzee/CalculationsTest.java - Eclipse

Finished after 0.059 seconds

Runs: 3/3 Errors: 0 Failures: 0

```
yahtzee.CalculationsTest [Runner: JUnit 5] (0.000 s)
    totalAllDiceTest (0.000 s)
    fullTeamFoundTest (0.000 s)
    maxOfAZagFoundTest (0.000 s)
```

```java
package yahtzee;

import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;

import org.junit.Assert;
import org.junit.Test;

public class CalculationsTest {

    @Test
    public void maxOfAZagFoundTest()
    {
        ArrayList<Die> hand = new ArrayList<Die>();

        assert(Calculations.maxOfAZagFound(hand) == 0);
        Die die0 = new Die();
        die0.jerseyNumber = 1;
        hand.add(0, die0);
        assert(Calculations.maxOfAZagFound(hand) == 1);

        Die die1 = new Die();
        die1.jerseyNumber = 1;
        hand.add(1, die1);
        assert(Calculations.maxOfAZagFound(hand) == 2);

        Die die2 = new Die();
        die2.jerseyNumber = 1;
```

```
<terminated> CalculationsTest [JUnit] C:\Program Files\Java\jdk-10\bin\javaw.exe (Apr 28, 2018, 2:45:23 PM)
```

Part 4

Conclusion

4.1 Reflection on working as a team

This was the first long-term project each of us has done. Additionally, this was the first time any of us have worked with a group (we have worked individually or with a partner previously). As a side note, each of us enjoyed this project and liked the long-term approach that justified a serious investment in the project. The three of us enjoyed working together and became closer in the process. We were friends going into the project. Working together did create some frustrations at times (i.e. finding times to meet). However, we did not have any disagreements over how to design or implement the game. All of us were very understanding of each other's opinions and we agreed on a compromise when applicable. In terms of managing the workload, we had a good balance of meeting and working together and dividing work and completing it on our own. Often times there was a pattern we followed for any given task. However, learning how to do the task the first time could be challenging. This is when meeting and working together was beneficial. Then, a group member would feel comfortable finishing the work on their own. Each of us became "experts" in certain areas of the project and we utilized this to our benefit. In terms of each group member's contribution we had a good balance of specializing in an area and still helping each other (thus knowing the other components). Ben's main contributions were working on the non-GUI code (he supplied his code from Programming Assignment 6), working with the other two members on the GUI code to implement his code with the GUI, and JUnit testing and testing in general. Eugene's main contributions were preparing most of the deliverables throughout the project (i.e. user experience mockup, test plan, final report) and working with a few specific areas in the GUI: researching JSwing elements, panel layout, main menu panel, leaderboard panel, game instruction panels, and graphics. Brandon's main contributions were working on the UML components (class and sequence diagrams), assisting with deliverables, helping transition Ben's code from Programming Assignment 6 to meet our requirements for the game, and much of the GUI components relating to the actual playing of the game. We regularly met to review each other's work. We also met at the end of the project to complete a final review.

4.2 Existing/remaining issues with the game

Fortunately, but not by chance, no issues exist or remain with the game based on our functional requirements. There are improvements that could be made that will be discussed next, however, they are not current issues.

4.3 Future work/extensions with the game

If we were to continue working on this project beyond the current timeline, there are two main areas we identify for future work/extensions with the game:

- **Improved GUI Appearance:** For example, it would look nicer if the panel background was a thematic image, as opposed to a plain color.

- **Enhanced User-Input Validation:** For example, it would have been more user-friendly if an error message appeared if the user entered bad input (i.e. a user name that is NOT one word). Our game does not take bad input. The user is allowed to retry and enter valid input. For instance, on the scoring card if the user enters an invalid line to score and clicks submit, the next button does not appear. Once the user does enter valid input, the next button does appear and the user can continue with the game as expected.