Multiplayer Yahtzee

Final Report

Blue Bulldogs

Hailey Boe

Anna Cardinal

Tyler Kamphouse

Jonathan Smoley

Course: CPSC 224 - Software Development

I. Introduction and Project Description

The game we made was Yahtzee. The object of the game is to roll the dice for different scoring combinations, and the player with the highest score when their score card is full wins. On every player's turn, they have the option to roll the dice three times to get the highest score combination possible. After they finish rolling, they must put a score down on their score sheets regardless of if they have any points or not. The player can also choose to stop rolling at any point in their turn. The number of dice, rolls, and sides of the dice are also all variables. At every dice roll, the player can choose to keep some dice and set them aside or reroll their dice. When the player is finished rolling, they must put a score down on the score sheet.

The scoring is separated into two sections, upper and lower. In the upper section, the player can choose to score the most die of a single side, multiplied by the number of dice that have that side. In the lower scoring section, the options are three of a kind, four of a kind, full house, small straight, large straight, Yahtzee, and chance. The player finishes the game when their entire score card sheet is filled out. The goal is to gamble each dice roll and the empty slots on their scorecard to see if they can get the maximum possible points.

In our game, we are going to start on a home screen where there will be a drop-down menu to choose the number of players (max of 8). There will also be a settings option to change the number of dice, dice sides, and dice roll turns, as well as directions for how to play and an option to start the game. After they set the number of players, they will be forced to enter the names for each player on another window. All this information will be collected in player objects. From there they click a button to start playing. As they cycle through turns, a scorecard object will be updated. When all the scorecards are full, registered by booleans, the game will end and continue onto an end window page. The page will display an altered score card that has every player ranked from most to least points. The winner is the player with the most points.

This document serves as a comprehensive overview of both our individual efforts and as a reflection of the project in general. Our project process was not without difficulties, but in general, was somewhat seamless. We started early and met frequently to talk about individual responsibilities. In general, our communication could have been better, but having issues with communication, especially as this is all our first time’s coding in a group, was inevitable. When our communication did start to lack, we made up for it by trying to address it and prevent it from happening in the future. We used GitHub frequently to store project updates, documents, and tasks. We did not use branching to the best of our ability, but it is something we will do in the future.

II. Team Members - Bios and Project Roles

Hailey Boe is a business student interested in computer science. She is majoring in business with a concentration in finance and is minoring in computer science with a concentration in software development. For this project, her responsibilities included breaking the original die class down into the MVC model, and more specifically, Die and DieView. She was also responsible for developing a testing plan with unit, integration, and functional tests, as well as developing a communication plan.

Jonathan Smoley is a Computer Science student interested in DevOps, artificial intelligence, cybersecurity, information technology, and both PC building and gaming. His experience has been limited to working on Algorithm Analysis using C++, algorithm optimization with Assembly, and some Machine Learning with Python. Although, this is his second time working with Java at an educational capacity. For this project, Jonathan conducted the initial implementation and navigation between screens. In addition, he assisted with interconnecting the objects designed by other team members and testing the integration of data between screens. Finally, he spearheaded repository organization and javadoc design.

Anna Cardinal is a Computer Science major at Gonzaga University with an interest in website development. Currently, her background in the computer science field is limited, but she has some experience with Java, Python, and C++. For this project, Anna wrote the code for the ScoreCard class and other affiliated classes like ScoreLine, Leaderboard, PlayerScoreCard, and Player. Additionally, she programmed a lot of the GameWindow class, and the game component classes like GameLabel and GameButton. She also did a lot of the formatting by picking a color palette, a font, and handling a lot of the design of the user interface layout.

Tyler Kamphouse is Majoring in Computer Science. He is interested in App Development and making Arduino Projects at home. He has experience with C++, Java, Swift, and a little python. For the Multiplayer Yahtzee Group Project, Tyler worked on the Hand class and its view elements, along with creating the UML and Sequence diagrams for the presentations and reports.

III. Project Requirements

We had a combination of functional and non-functional requirements for this project. Our general approach was to complete high-priority function requirements first, then high-priority non-functional requirements. After that we moved on to medium-priority requirements, also starting on the functional side and moving over to the non-functional side. Below is a list of all our project requirements including their priority level and whether they were completed.

Functional Requirements:

- Let user pick number of players – high – completed
- Let user navigate to settings screen – medium – incomplete
- Let user pick number of dice – medium – incomplete
- Let user pick number of sides on dice – medium – incomplete
- Let user pick number of rolls per turn – medium – incomplete
- Let user navigate to home screen from settings – medium - incomplete
- Let user navigate to name collection screen – high – complete
- Let user enter all player name – high – complete
- Let user navigate to game screen from name collection screen – high - complete
- Display current player scorecard – high – complete
- Display current hand – high - complete
- Let player select dice to keep – high – complete
- Let player roll hand – high – complete
- Let player select line to record score – high – complete
- Iterate through players in game – high – complete
- Display final scores after game ends – high – complete
- Display winner – high – complete
- Let user navigate to title screen at game end – medium – complete

Non-Functional Requirements:

- Display current player name – high - complete
- Display current roll number – high - complete
- Uniform formatting – high – complete
- Display current turn number – medium – incomplete
- Uniquely color each PlayerScoreCard – medium – complete

For more in-depth project requirements, see Appendix A

IV. Solution Approach

When deciding our GUI approach, we landed on a couple of specific design ideas. Firstly, we wanted to have multiple windows that served different purposes throughout the game. For example, we wanted an introductory screen to display the Yahtzee title and give the user the opportunity to pick the number of players and we wanted this window to be separate from the window that would allow the users to enter unique names for each player. The other design problem we had to figure out was how to be as obvious as possible when it came to guiding user action and letting the user know what to do next. For this we decided that instead of simply disabling swing components, we would only allow components to be visible when we wanted the player to use them. This would prevent the user from attempting to take actions that were currently not allowed as well as give them hints as to what specific action they needed to take next to progress the game.

When it came to our coding approach, our group decided to implement the Model View Controller pattern to the best of our ability. Some of us were more rigid about following this pattern than others and given more time I think our team would work to more closely mirror each other's implementations of the MVC model. We also decided, given the nature of our multi-window approach, to let the GameWindow class handle most of the game mechanics / logic and facilitate communication between the objects throughout the game.

For in-depth UML Diagrams and GameWindow Sequence Diagram, see Appendix B

V. Test Plan

1. Unit Testing

Unit testing was utilized by our group. We planned on having at least two people write and modify existing unit tests from previous projects to fit with this project. These unit tests were used to show our progress on issues in GitHub, and additionally be used to help for Integration tests show our progress on milestones. Every team member at some point was viewing the unit tests to track progress on what needs to be accomplished next and to hold each other accountable for getting functioning work.

Our group expected to have at least one unit test for every class, and classes such as “Hand” have more to be able to test important functions like “roll hand” or “keep selected dice”. These types of unit tests were taken from group member’s previous tests already written and were lightly modified for the assignment. Unit tests for new classes like “Player” and “LeaderBoard” needed unit tests written from scratch. These tested abilities such as “get player score” or “set player name”.

2. Integration Testing

For integration testing, we tried to strategically use integration testing to focus on catching bugs that had not been detected by the unit tests. The areas we were going to focus on using integration testing the most was the junction between our settings frame and our game play frame, and between our main frame and the end frame. There were a lot of setting features, such as the number of players in the settings page, that are vital to the function requirements for the rest of the code. For that junction, we tested to see if the number of selected players in the startWindow changed the number of player’s names to be filled out in the nameWindow. We also checked that the MainWindow cycled through the appropriate number of players received in the startWindow when the user selected the number of players from drop down in the frame before. For the scorecard and main game play through junction, we tested to see that the scorecard had appropriately assembled itself in the endWindow, and displayed the rankings of the players with the proper player scorecard data.

When we worked on code separately, every time we pushed to the repo in GitHub, we ensured that the tests were passing in JUnit. Every time we met in group meetings, we also amply discussed bug issues and potential areas where there were problems. In these meetings, we also talked through current bug problems. Overall, the most important thing we did was add onto new code slowly and test frequently with unit tests. Most every function in each class has a unit test. If the function completes multiple tasks, each behavior is also tested with additional unit tests. In these tests, the data sets were inclusive of all test cases, such as zeros, negative values, and other unordinary input data. After producing these data sets, we then analyzed different structures of the two joint functions and determined the most vulnerable attributes. Afterwards, we were also able to better identify and remedy areas of our code that need to be strengthened.

3. System Testing

3.1. Functional testing:

Functional testing, in general, is highly dependent upon the various functional requirements included in the project plan. As such, the plan is to have a test for those requirements that are more likely to produce errors when faced with a user. For instance, while a title is required for this project, it has a low chance of failing to appear to the user. Therefore, testing resources (i.e., time) would be better spent on a requirement like tracking a user request for the number of players.

Once again, this type of testing determines if the actions performed when running this program are in accordance with the requirements set prior to development. With the

combination of unit testing and integration testing, the only action-wise testing left to be completed pertains specifically to the functions required to make Yahtzee run correctly.

3.2. Performance testing:

We did not plan on having many performance tests.

3.3. User Acceptance Testing:

Every member of the group ran through the program several times throughout the coding process and even when the code is “complete” to check for any bugs that slipped through unit or integration tests. Ideally each member tried to do different things through each playthrough to better test every aspect of the code. Additionally, we planned on having people we know play the game and give feedback, but we did not end up doing this.

VI. Project Implementation Description

GitHub Link: https://github.com/GU-2021-Fall-CPSC224/final-yahtzee-blue-bulldogs

Our implementation of Multiplayer Yahtzee closely mirrors classic Yahtzee, however there were some specific design features we had to include to adapt to a virtual platform. These design features include:

- On each roll, possible scores are displayed in the current player's scorecard, the player can then choose which dice to keep and roll again
- After rolling, the player must record a score on their scorecard, regardless of if they have any points or not
- The player finishes the game when their entire scorecard is filled out
- No more than eight players

In our initial planning, we had considered making this game compatible with “Lizard Spock Yahtzee” rules, however we did not have time.

Our user interface design has also changed somewhat over the course of our project [see Appendix C]. This is partially due to changes in our code functionality, such as deciding against implementing “Lizard Spock Yahtzee,” but some of these changes were just the result of better design ideas coming to us as we worked through our project.

Over the course of our project, our test harness gradually grew as each team member wrote tests for their assigned classes [see Appendix D]. The fact that we split our work up by class made splitting up testing especially easy. The most complexity came from integration testing because at that point we were working with each other's classes and at that point we had to communicate with each other about necessary changes and additions to those classes to make testing easier / fix bugs found through testing.

VII. Future Work

Logically, there are still a few features that could be added or changed in the future. However, we are unsure whether development will continue into later stages.

The most pressing addition to be made would be the addition of user selected configuration. That is, implementing what has been named “Lizard-Spock Yahtzee” by making the hand size, dice type, and roll count variables to be defined. Another gameplay feature to add is a list of total scores for each player. Ideally, it would appear as part of the GameWindow class. It would stay visible through all rounds of the game and change according the the scores recorded on each turn a player takes.

During development, GitHub was used to store the progress that each team member made. The GitHub platform allows certain quality checks to be made every time changes are made. Whenever a change was pushed to the repository on GitHub a workflow filled with these checks was added to a queue where they all eventually failed. To guarantee that development was successful and document that success, more work could be done to set up this tool for future changes that will be made. Sometimes one part of packaging a project is embellishment. In this case, embellishment includes improving code readability. Not much effort was put into organizing a standard format for classes, methods, brackets. Future development could focus on issues such as this. As is quite common, there could be more tests added to this project. Specifically, an integration test for GameWindow that checks if all necessary components are created and used correctly during initialization and even game progression.

VIII. Glossary

Unit Testing: a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Integration Testing: the phase in software testing in which individual software modules are combined and tested as a group.

Functional Testing: a type of testing that seeks to establish whether each application feature works as per the software requirements.

Performance Testing: a software testing process used for testing the speed, response time, stability, reliability, scalability, and resource usage of a software application under a particular workload.

User Acceptance Testing: the last phase of the software testing process, actual software users test the software to make sure it can handle necessary tasks in real-world scenarios

MVC: Stands for Model View Control and is a software pattern. It focuses on separating code into the visible aspects and logical aspects. This helps code become scalable in the future.

UML: Stands for Unified Modeling Language. This is a basic development tool that helps software developers visualize how classes, objects, methods, and variables are interacting with one another on a software design level.

IX. References

“Yahtzee.” Wikipedia. May 6th, 2022. https://en.wikipedia.org/wiki/Yahtzee#:~:text=Yahtzee%20is%20a%20dice%20game,Edwin%20S.%20Lowe%20in%201956

"The History of Yahtzee.” Ultra Board Games. May 6th, 2022. https://www.ultraboardgames.com/yahtzee/history.php

"Integration Testing.” Java T Point. April 9th, 2022. https://www.javatpoint.com/java-swing

X. Appendices

Appendix A

Functional Requirements:

Part	let user pick number of players
Priority	High
Purpose	application lets user select number of players from drop box on intro screen
Inputs / Needs	user selects number of players (1 to 8) through JComboBox
Operators / Actors	TitleWindow, User
Outputs	number of players

Part	let user navigate to setting screen
Priority	Medium
Purpose	application lets user leave starting window to go to a settings screen
Inputs / Needs	user clicks settings button
Operators / Actors	TitleWindow, GameButton, User
Outputs	N/A

Part	let user pick number of dice
Priority	Medium
Purpose	application lets user select number of dice in game from a dropbox on screen
Inputs / Needs	user selects number of dice (5, 6, or 7) from a JComboBox
Operators / Actors	SettingsWindow, User
Outputs	number of dice

Part	let user pick number of sides on dice
Priority	Medium
Purpose	application lets user select number of sides on dice from a dropbox on screen
Inputs / Needs	user selects number of sides on the dice (6, 8, or 12) from a JComboBox
Operators / Actors	SettingsWindow, User
Outputs	number of sides on dice

Part	let user pick number of rolls per turn
Priority	Medium
Purpose	application lets user select number of rolls per turn from a dropbox on screen
Inputs / Needs	user selects number of rolls per turn (2, 3, or 4) from a JComboBox
Operators / Actors	SettingsWindow, User
Outputs	number of rolls per turn

Part	let user navigate to home screen from settings
Priority	Medium
Purpose	application lets user return to homescreen from settings screen
Inputs / Needs	user clicks return to intro screen button
Operators / Actors	SettingsWindow, User
Outputs	Data collected from settings screen (number of dice, number of sides on dice, number of rolls per turn)

Part	let user navigate to name collection screen
Priority	High
Purpose	application lets user leave starting window to go to screen that asks for player names
Inputs / Needs	user clicks play game button
Operators / Actors	TitleWindow, GameButton, User
Outputs	N/A

Part	let user enter all player names
Priority	High
Purpose	application lets user enter player names for every player in game
Inputs / Needs	number of players in game
Operators / Actors	NameWindow, User
Outputs	Player names

Part	let user navigate to main gameplay screen from name collection screen
Priority	High
Purpose	application lets user leave name collection screen to go to main gameplay screen
Inputs / Needs	all players must have been given a name, user clicks play button
Operators / Actors	NameWindow, User
Outputs	Player names

Part	Display Current Player Scorecard
Priority	High
Purpose	application displays current player's scorecards on left side of screen, including possible score with a given hand, and recorded scores from previous turns
Inputs / Needs	current player, current player's ScoreCard
Operators / Actors	GameWindow, Player arraylist, ScoreCard
Outputs	Scoring information

Part	display current hand
Priority	High
Purpose	application displays current dice values of the game hand on the screen
Inputs / Needs	current die values
Operators / Actors	GameWindow, Hand
Outputs	visual representation of die side corresponding to current die values

Part	let player select dice to keep
Priority	High
Purpose	application allows player to select which dice to keep from a goven hand through radiobutton corresponding to the dice
Inputs / Needs	user selects radio buttons corresponding to specific dice
Operators / Actors	GameWindow, Hand, Die
Outputs	N/A

Part	let player roll hand
Priority	High
Purpose	application allows player to roll a given hand, and sets the unselected dice to a new randomly generated value
Inputs / Needs	user clicks roll button
Operators / Actors	GameWindow, Hand, Die
Outputs	new randomly generated die values

Part	let player select line to record score
Priority	High
Purpose	application allows player to choose a scoreline on their scorecard to record their score after they finish rolling their hand for that turn
Inputs / Needs	user selects scoreline through radiobutton, user clicks score button
Operators / Actors	GameWindow, Player, ScoreCard
Outputs	Newly Recorded Score Value

Part	iterate through players in game
Priority	High
Purpose	application iterates through all players in game after each player finishes there turn
Inputs / Needs	player clicks end turn button
Operators / Actors	GameWindow, Player arraylist, current player
Outputs	N/A

Part	display final scores after game ends
Priority	High
Purpose	application displays leaderboard scorecard after every player has filled their scorecard
Inputs / Needs	player scores, player names
Operators / Actors	EndGameWindow, Player arraylist, LeaderBoard
Outputs	all player scores

Part	display winner
Priority	High
Purpose	application displays game winner name under the leaderboear based on who has the highest final score
Inputs / Needs	player scores, winner name
Operators / Actors	EndGameWindow, Player arraylist, LeaderBoard
Outputs	winner name

Part	let user navigate to title screen at game end
Priority	Medium
Purpose	application allows user to return to the homescreen after game ends and displays the leaderboard
Inputs / Needs	user clicks return to title button
Operators / Actors	EndGameWindow
Outputs	N/A

Non Functional Requirements:

Part	display current player name
Priority	high
Purpose	application displays current player's name at the top of the screen to indicate turn
Inputs / Needs	current player name
Operators / Actors	GameWindow, Player arraylist
Outputs	Current player name

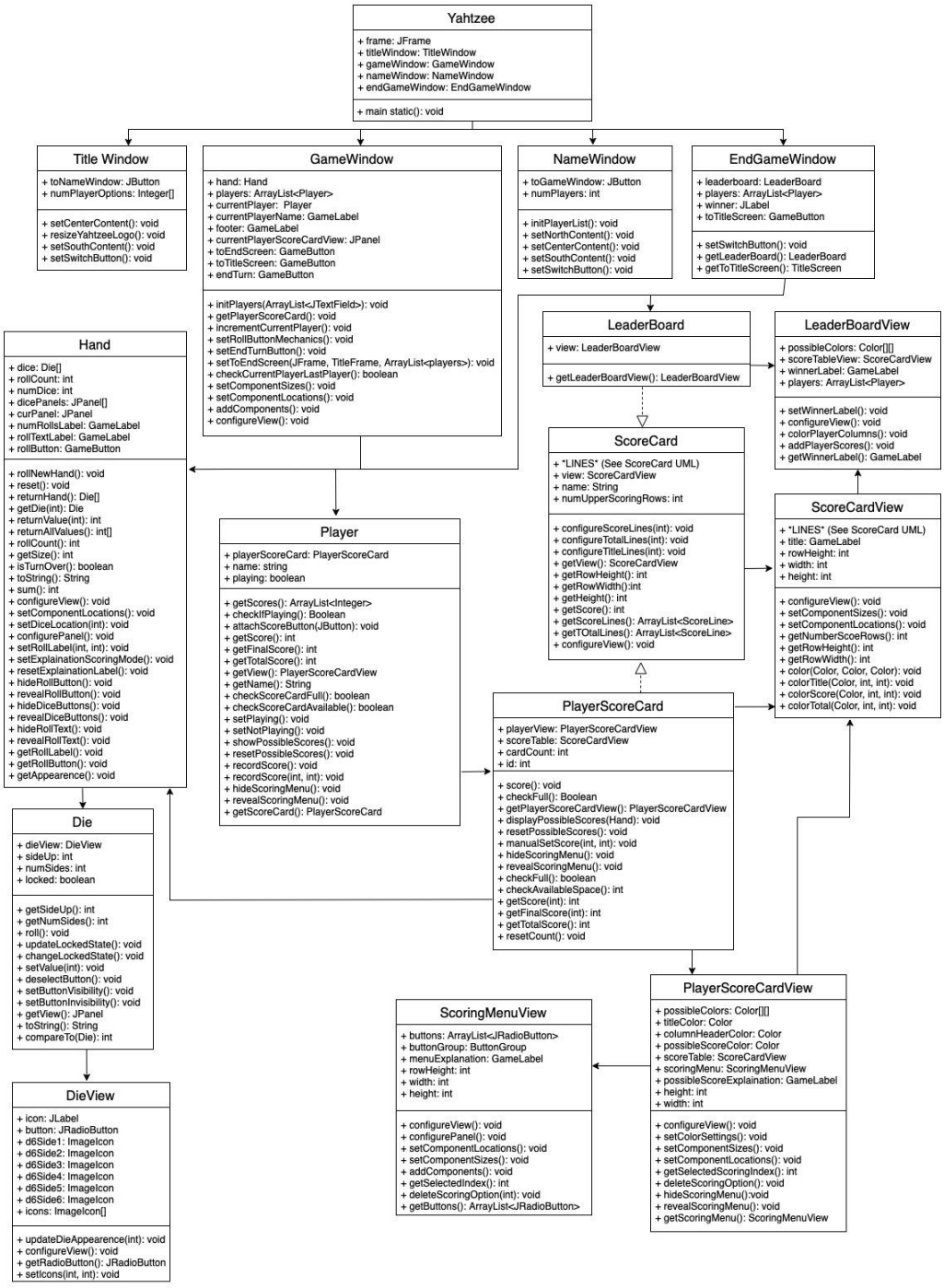
Part	display current roll number
Priority	high
Purpose	application displays what roll the current player is on, as well as how many rolls they have left
Inputs / Needs	number of rolls per turn, current roll number
Operators / Actors	GameWindow, Hand
Outputs	current roll number, number of rolls left

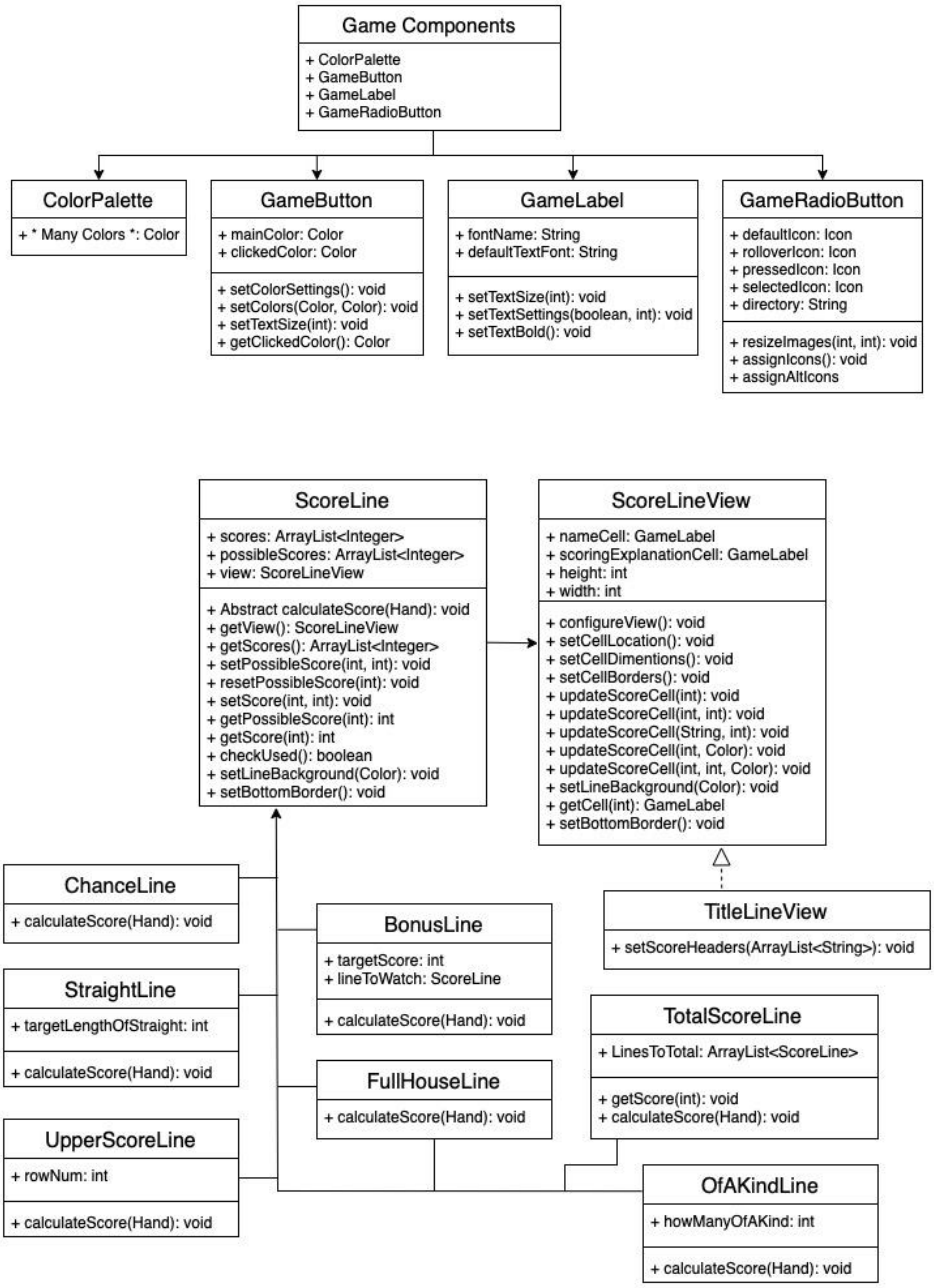
Part	uniform formatting
Priority	high
Purpose	all windows have same color palette / font / other formatting decisions to create unity throughout the game
Inputs / Needs	current player name
Operators / Actors	All windows
Outputs	N/A

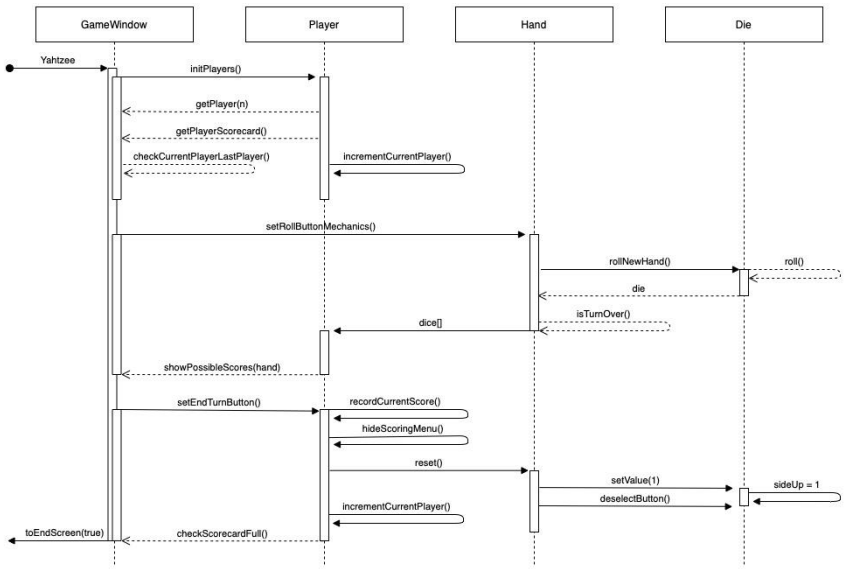
Part	display current turn number
Priority	medium
Purpose	application displays current turn at the top of the screen
Inputs / Needs	current player name
Operators / Actors	GameWindow
Outputs	Turn number

Part	uniquely color each playerscorecard
Priority	medium
Purpose	Makes each player's scorecard distinctive to better differentiate between players
Inputs / Needs	current player name
Operators / Actors	GameWindow, Player arraylist
Outputs	Current player name

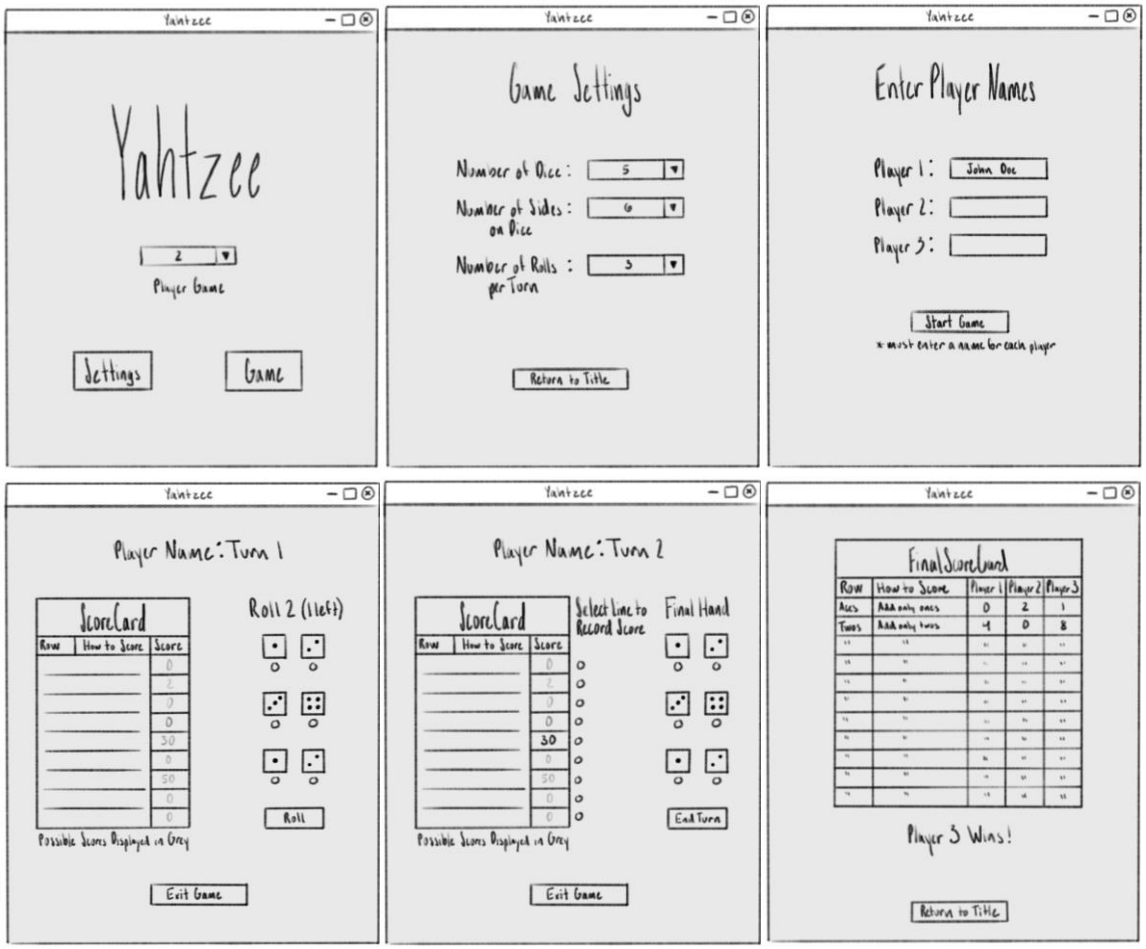
Appendix B

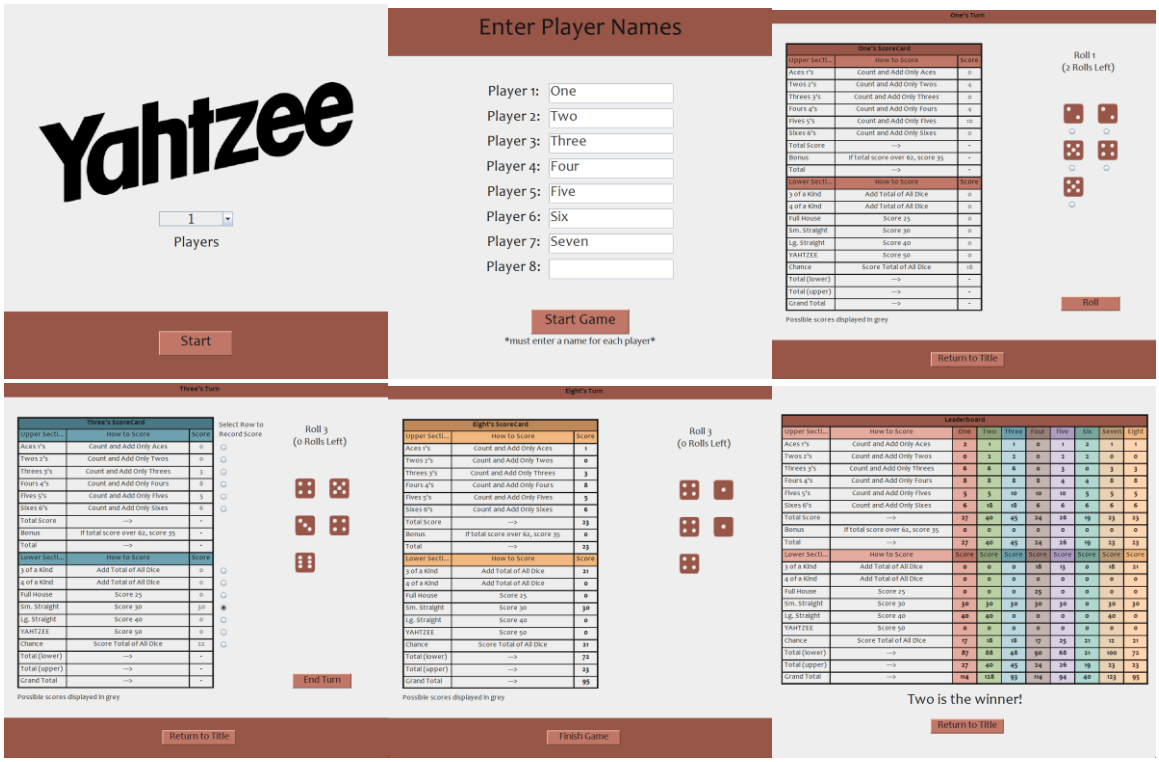

Yahtzee
+ frame: JFrame
+ titleWindow: TitleWindow
+ gameWindow: GameWindow
+ nameWindow: NameWindow
+ endGameWindow: EndGameWindow
+ main static(): void
Title Window
+ toNameWindow: JButton
+ numPlayerOptions: Integer[]
+ setCenterContent(): void
+ resizeYahtzeeLogo(): void
+ setSouthContent(): void
+ setSwitchButton(): void
GameWindow
+ hand: Hand
+ players: ArrayList<Player>
+ currentPlayer: Player
+ currentPlayerName: GameLabel
+ footer: GameLabel
+ currentPlayerScoreCardView: JPanel
+ toEndScreen: GameButton
+ toTitleScreen: GameButton
+ endTurn: GameButton
+ initPlayers(ArrayList<JTextField>): void
+ getPlayerScoreCard(): void
+ incrementCurrentPlayer(): void
+ setRollButtonMechanics(): void
+ setEndTurnButton(): void
+ setToEndScreen(JFrame, TitleFrame, ArrayList<players>): void
+ checkCurrentPlayerLastPlayer(): boolean
+ setComponentSizes(): void
+ setComponentLocations(): void
+ addComponents(): void
+ configureView(): void
NameWindow
+ toGameWindow: JButton
+ numPlayers: int
+ initPlayerList(): void
+ setNorthContent(): void
+ setCenterContent(): void
+ setSouthContent(): void
+ setSwitchButton(): void
EndGameWindow
+ leaderboard: LeaderBoard
+ players: ArrayList<Player>
+ winner: JLabel
+ toTitleScreen: GameButton
+ setSwitchButton(): void
+ getLeaderBoard(): LeaderBoard
+ getToTitleScreen(): TitleScreen
Hand
+ dice: Die[]
+ rollCount: int
+ numDice: int
+ dicePanels: JPanel[]
+ curPanel: JPanel
+ numRollsLabel: GameLabel
+ rollTextLabel: GameLabel
+ rollButton: GameButton
+ rollNewHand(): void
+ reset(): void
+ returnHand(): Die[]
+ getDie(int): Die
+ returnValue(int): int
+ returnAllValues(): int[]
+ rollCount(): int
+ getSize(): int
+ isTurnOver(): boolean
+ toString(): String
+ sum(): int
+ configureView(): void
+ setComponentLocations(): void
+ setDiceLocation(int): void
+ configurePanel(): void
+ setRollLabel(int, int): void
+ setExplainationScoringMode(): void
+ resetExplainationLabel(): void
+ hideRollButton(): void
+ revealRollButton(): void
+ hideDiceButtons(): void
+ revealDiceButtons(): void
+ hideRollText(): void
+ revealRollText(): void
+ getRollLabel(): void
+ getRollButton(): void
+ getAppearence(): void
LeaderBoard
+ view: LeaderBoardView
+ getLeaderBoardView(): LeaderBoardView
LeaderBoardView
+ possibleColors: Color[][]
+ scoreTableView: ScoreCardView
+ winnerLabel: GameLabel
+ players: ArrayList<Player>
+ setWinnerLabel(): void
+ configureView(): void
+ colorPlayerColumns(): void
+ addPlayerScores(): void
+ getWinnerLabel(): GameLabel
ScoreCard
+ *LINES* (See ScoreCard UML)
+ view: ScoreCardView
+ name: String
+ numUpperScoringRows: int
+ configureScoreLines(int): void
+ configureTotalLines(int): void
+ configureTitleLines(int): void
+ getView(): ScoreCardView
+ getRowHeight(): int
+ getRowWidth():int
+ getHeight(): int
+ getScore(): int
+ getScoreLines(): ArrayList<ScoreLine>
+ getTOtalLines(): ArrayList<ScoreLine>
+ configureView(): void
ScoreCardView
+ *LINES* (See ScoreCard UML)
+ title: GameLabel
+ rowHeight: int
+ width: int
+ height: int
+ configureView(): void
+ setComponentSizes(): void
+ setComponentLocations(): void
+ getNumberScoeRows(): int
+ getRowHeight(): int
+ getRowWidth(): int
+ color(Color, Color, Color): void
+ colorTitle(Color, int, int): void
+ colorScore(Color, int, int): void
+ colorTotal(Color, int, int): void
Player
+ playerScoreCard: PlayerScoreCard
+ name: string
+ playing: boolean
+ getScores(): ArrayList<Integer>
+ checkIfPlaying(): Boolean
+ attachScoreButton(JButton): void
+ getScore(): int
+ getFinalScore(): int
+ getTotalScore(): int
+ getView(): PlayerScoreCardView
+ getName(): String
+ checkScoreCardFull(): boolean
+ checkScoreCardAvailable(): boolean
+ setPlaying(): void
+ setNotPlaying(): void
+ showPossibleScores(): void
+ resetPossibleScores(): void
+ recordScore(): void
+ recordScore(int, int): void
+ hideScoringMenu(): void
+ revealScoringMenu(): void
+ getScoreCard(): PlayerScoreCard
PlayerScoreCard
+ playerView: PlayerScoreCardView
+ scoreTable: ScoreCardView
+ cardCount: int
+ id: int
+ score(): void
+ checkFull(): Boolean
+ getPlayerScoreCardView(): PlayerScoreCardView
+ displayPossibleScores(Hand): void
+ resetPossibleScores(): void
+ manualSetScore(int, int): void
+ hideScoringMenu(): void
+ revealScoringMenu(): void
+ checkFull(): boolean
+ checkAvailableSpace(): int
+ getScore(int): int
+ getFinalScore(int): int
+ getTotalScore(): int
+ resetCount(): void
Die
+ dieView: DieView
+ sideUp: int
+ numSides: int
+ locked: boolean
+ getSideUp(): int
+ getNumSides(): int
+ roll(): void
+ updateLockedState(): void
+ changeLockedState(): void
+ setValue(int): void
+ deselectButton(): void
+ setButtonVisibility(): void
+ setButtonInvisibility(): void
+ getView(): JPanel
+ toString(): String
+ compareTo(Die): int
DieView
+ icon: JLabel
+ button: JRadioButton
+ d6Side1: ImageIcon
+ d6Side2: ImageIcon
+ d6Side3: ImageIcon
+ d6Side4: ImageIcon
+ d6Side5: ImageIcon
+ d6Side6: ImageIcon
+ icons: ImageIcon[]
+ updateDieAppearence(int): void
+ configureView(): void
+ getRadioButton(): JRadioButton
+ setIcons(int, int): void
ScoringMenuView
+ buttons: ArrayList<JRadioButton>
+ buttonGroup: ButtonGroup
+ menuExplanation: GameLabel
+ rowHeight: int
+ width: int
+ height: int
+ configureView(): void
+ configurePanel(): void
+ setComponentLocations(): void
+ setComponentSizes(): void
+ addComponents(): void
+ getSelectedIndex(): int
+ deleteScoringOption(int): void
+ getButtons(): ArrayList<JRadioButton>
PlayerScoreCardView
+ possibleColors: Color[][]
+ titleColor: Color
+ columnHeaderColor: Color
+ possibleScoreColor: Color
+ scoreTable: ScoreCardView
+ scoringMenu: ScoringMenuView
+ possibleScoreExplaination: GameLabel
+ height: int
+ width: int
+ configureView(): void
+ setColorSettings(): void
+ setComponentSizes(): void
+ setComponentLocations(): void
+ getSelectedScoringIndex(): int
+ deleteScoringOption(): void
+ hideScoringMenu():void
+ revealScoringMenu(): void
+ getScoringMenu(): ScoringMenuView


Game Components
+ ColorPalette
+ GameButton
+ GameLabel
+ GameRadioButton
ColorPalette
+ * Many Colors *: Color
GameButton
+ mainColor: Color
+ clickedColor: Color
+ setColorSettings(): void
+ setColors(Color, Color): void
+ setTextSize(int): void
+ getClickedColor(): Color
GameLabel
+ fontName: String
+ defaultTextFont: String
+ setTextSize(int): void
+ setTextSettings(boolean, int): void
+ setTextBold(): void
GameRadioButton
+ defaultIcon: Icon
+ rolloverIcon: Icon
+ pressedIcon: Icon
+ selectedIcon: Icon
+ directory: String
+ resizeImages(int, int): void
+ assignIcons(): void
+ assignAltIcons
ScoreLine
+ scores: ArrayList<Integer>
+ possibleScores: ArrayList<Integer>
+ view: ScoreLineView
+ Abstract calculateScore(Hand): void
+ getView(): ScoreLineView
+ getScores(): ArrayList<Integer>
+ setPossibleScore(int, int): void
+ resetPossibleScore(int): void
+ setScore(int, int): void
+ getPossibleScore(int): int
+ getScore(int): int
+ checkUsed(): boolean
+ setLineBackground(Color): void
+ setBottomBorder(): void
ScoreLineView
+ nameCell: GameLabel
+ scoringExplanationCell: GameLabel
+ height: int
+ width: int
+ configureView(): void
+ setCellLocation(): void
+ setCellDimentions(): void
+ setCellBorders(): void
+ updateScoreCell(int): void
+ updateScoreCell(int, int): void
+ updateScoreCell(String, int): void
+ updateScoreCell(int, Color): void
+ updateScoreCell(int, int, Color): void
+ setLineBackground(Color): void
+ getCell(int): GameLabel
+ setBottomBorder(): void
ChanceLine
+ calculateScore(Hand): void
BonusLine
+ targetScore: int
+ lineToWatch: ScoreLine
+ calculateScore(Hand): void
TitleLineView
+ setScoreHeaders(ArrayList<String>): void
StraightLine
+ targetLengthOfStraight: int
+ calculateScore(Hand): void
TotalScoreLine
+ LinesToTotal: ArrayList<ScoreLine>
+ getScore(int): void
+ calculateScore(Hand): void
FullHouseLine
+ calculateScore(Hand): void
UpperScoreLine
+ rowNum: int
+ calculateScore(Hand): void
OfAKindLine
+ howManyOfAKind: int
+ calculateScore(Hand): void

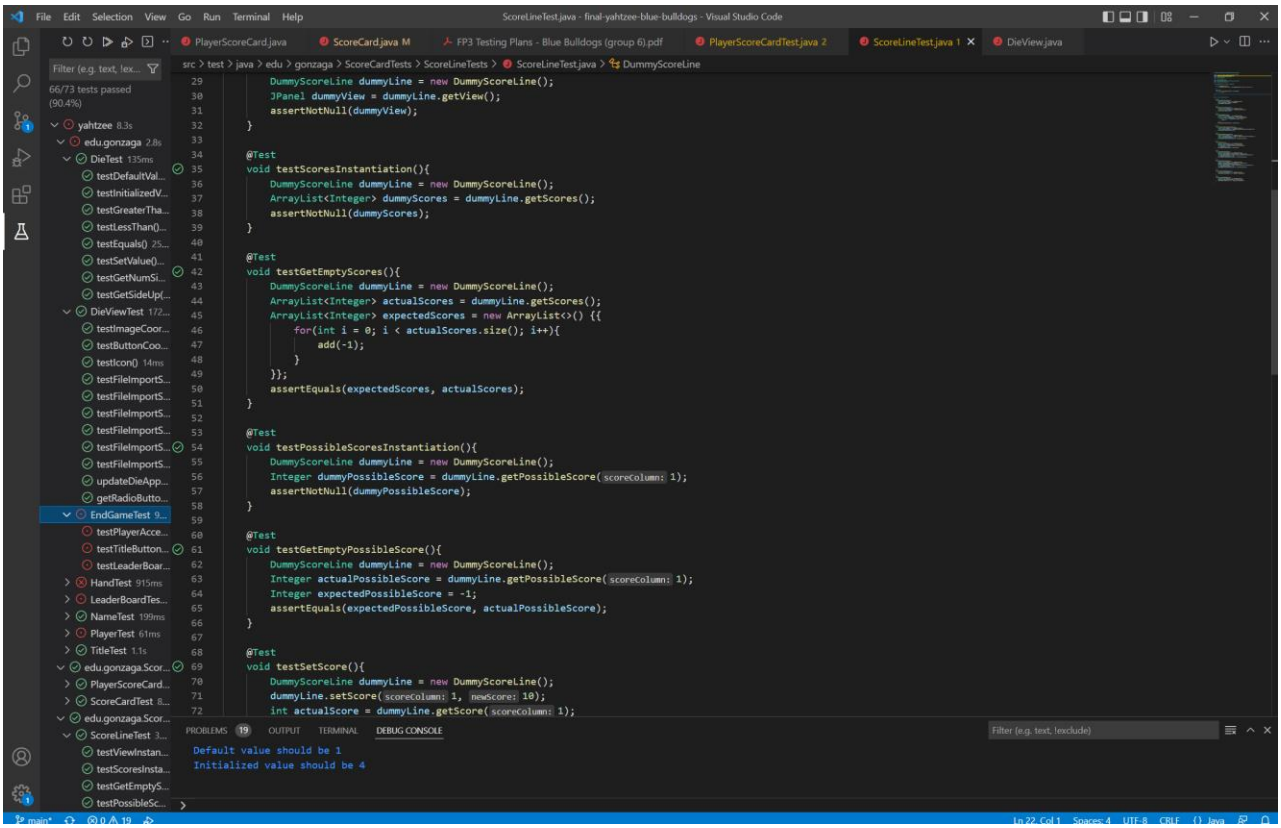

GameWindow
Player
Hand
Die
Yahtzee
initPlayers()
getPlayer(n)
getPlayerScorecard()
checkCurrentPlayerLastPlayer()
incrementCurrentPlayer()
setRollButtonMechanics()
rollNewHand()
roll()
die
isTurnOver()
dice[]
showPossibleScores(hand)
setEndTurnButton()
recordCurrentScore()
hideScoringMenu()
reset()
setValue(1)
sideUp = 1
deselectButton()
incrementCurrentPlayer()
toEndScreen(true)
checkScorecardFull()

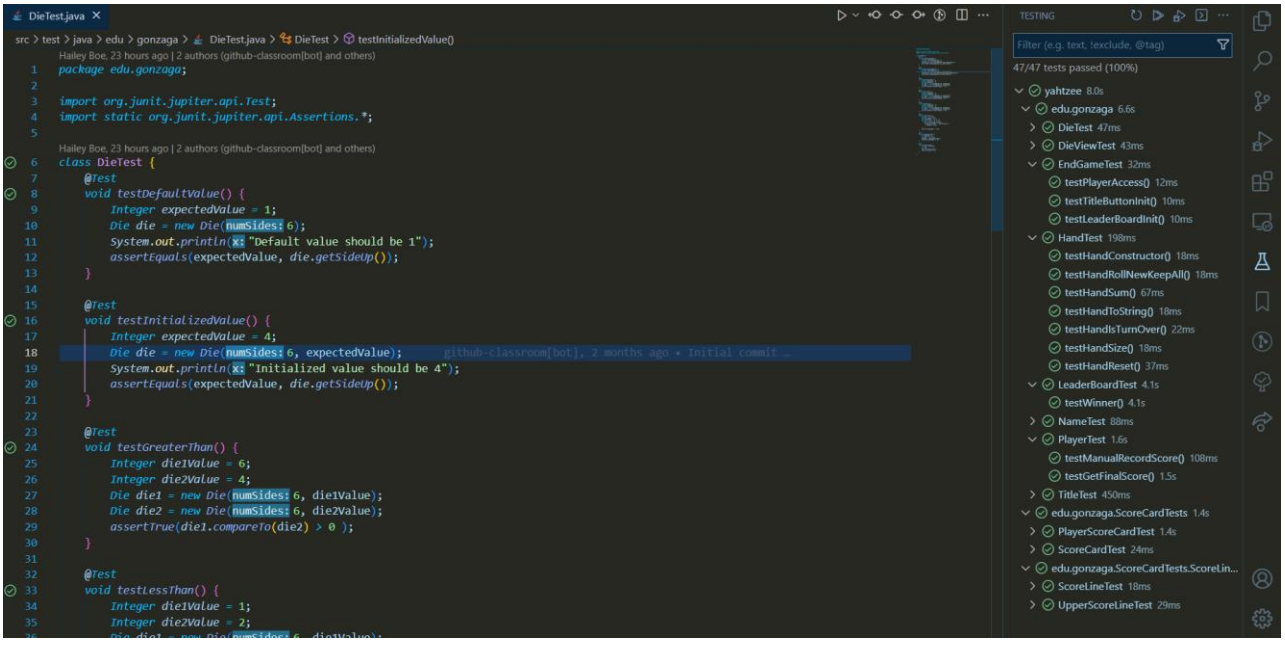
Appendix C


Yahtzee
Yahtzee
2
Player Game
Settings
Game
Yahtzee
Game Settings
Number of Dice :
5
Number of Sides : on Dice
6
Number of Rolls : per Turn
3
Return to Title
Yahtzee
Enter Player Names
Player 1 :
John Doe
Player 2 :
Player 3 :
Start Game
* must enter a name for each player
Yahtzee
Player Name: Turn 1
ScoreCard
Row
How to Score
Score
Roll 2 (1 left)
Roll
Possible Scores Displayed in Grey
Exit Game
Yahtzee
Player Name: Turn 2
ScoreCard
Row
How to Score
Score
Select Line to Record Score
Final Hand
End Turn
Possible Scores Displayed in Grey
Exit Game
Yahtzee
Final Scoreboard
Row
How to Score
Player 1
Player 2
Player 3
Aces
Add only aces
0
2
1
Twos
Add only twos
4
0
8
Player 3 Wins!
Return to Title


Yahtzee
1
Players
Start
Enter Player Names
Player 1: One
Player 2: Two
Player 3: Three
Player 4: Four
Player 5: Five
Player 6: Six
Player 7: Seven
Player 8:
Start Game
must enter a name for each player
One's Turn
Roll 1
(2 Rolls Left)
Roll
Return to Title
Three's Turn
Select Row to Record Score
Roll 3
(0 Rolls Left)
End Turn
Return to Title
Eight's Turn
Roll 3
(0 Rolls Left)
Finish Game
Leaderboard
Two is the winner!
Return to Title

Appendix D


ScoreLineTest.java - final-yahtzee-blue-bulldogs - Visual Studio Code
PlayerScoreCard.java
ScoreCard.java M
FP3 Testing Plans - Blue Bulldogs (group 6).pdf
PlayerScoreCardTest.java 2
ScoreLineTest.java 1
DieView.java
src > test > java > edu > gonzaga > ScoreCardTests > ScoreLineTests > ScoreLineTest.java > DummyScoreLine
66/73 tests passed (90.4%)
yahtzee 8.3s
edu.gonzaga 2.8s
DieTest 135ms
DieViewTest 172...
EndGameTest 9...
HandTest 915ms
LeaderBoardTes...
NameTest 199ms
PlayerTest 61ms
TitleTest 1.1s
edu.gonzaga.Scor...
PlayerScoreCard...
ScoreCardTest 8...
ScoreLineTest 3...
PROBLEMS 19
OUTPUT
TERMINAL
DEBUG CONSOLE
Default value should be 1
Initialized value should be 4


DieTest.java
src > test > java > edu > gonzaga > DieTest.java > DieTest > testInitializedValue()
Hailey Boe, 23 hours ago | 2 authors (github-classroom[bot] and others)
package edu.gonzaga;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class DieTest {
@Test
void testDefaultValue() {
Integer expectedValue = 1;
Die die = new Die(numSides: 6);
System.out.println(x: "Default value should be 1");
assertEquals(expectedValue, die.getSideUp());
}
@Test
void testInitializedValue() {
Integer expectedValue = 4;
Die die = new Die(numSides: 6, expectedValue);
System.out.println(x: "Initialized value should be 4");
assertEquals(expectedValue, die.getSideUp());
}
@Test
void testGreaterThan() {
Integer die1Value = 6;
Integer die2Value = 4;
Die die1 = new Die(numSides: 6, die1Value);
Die die2 = new Die(numSides: 6, die2Value);
assertTrue(die1.compareTo(die2) > 0);
}
@Test
void testLessThan() {
Integer die1Value = 1;
Integer die2Value = 2;
TESTING
47/47 tests passed (100%)
yahtzee 8.0s
edu.gonzaga 6.6s
DieTest 47ms
DieViewTest 43ms
EndGameTest 32ms
testPlayerAccess() 12ms
testTitleButtonInit() 10ms
testLeaderBoardInit() 10ms
HandTest 198ms
testHandConstructor() 18ms
testHandRollNewKeepAll() 18ms
testHandSum() 67ms
testHandToString() 18ms
testHandIsTurnOver() 22ms
testHandSize() 18ms
testHandReset() 37ms
LeaderBoardTest 4.1s
testWinner() 4.1s
NameTest 88ms
PlayerTest 1.6s
testManualRecordScore() 108ms
testGetFinalScore() 1.5s
TitleTest 450ms
edu.gonzaga.ScoreCardTests 1.4s
PlayerScoreCardTest 1.4s
ScoreCardTest 24ms
edu.gonzaga.ScoreCardTests.ScoreLin...
ScoreLineTest 18ms
UpperScoreLineTest 29ms