Programmer's Manual

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

Programmer's Manual – Table of Contents

Requirements Documentation

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

Requirements Documentation

1. Introduction

1.1. Scope of the Product

Proposed Project Summary

For the semester project, team 1 proposes to develop a version of the popular dice game Farkle. Our proposed version will be a standalone application utilizing a graphic user interface. The first iteration of development will yield a one player version of the game, and two player support will be added in a second iteration with the second player being a live person using the same computer as the first player. If time permits, a third iteration will yield a two player version that can be played against a computer opponent with probability determining the play of the computer opponent.

Traditional Rules of Farkle

Farkle is a game typically played with six dice and more than one player. The traditional rules dictate that each player takes turns rolling the dice in succession, with each turn producing a score. The score produced from the players current turn is added to that players previous score accumulation. The goal is to be the first player to reach 10,000 points. The scoring is generated as follows:

1. At the beginning of the turn, the player rolls all 6 dice.
2. Scoring for each roll is as follows: Each 1 = 100 points, each 5 = 50 points, (3) 1s = 1000 points, (3) 2s = 200 points, (3) 3s = 300 points, (3) 4s = 400 points, (3) 5s = 500 points, (3) 6s = 600 points, a straight = 1500 points, and more than three of a kind doubles the value for each additional match (e.g. (5) 3s = 300 x 2 x 2 = 1200 points).
3. Dice resulting in a score are chosen and removed by the player (the player must pick up at least one scoring die, but does not need to pick up all scoring die), and the player decides if they want to roll with the remaining dice or pass to the next player.
4. At least one die must be set aside after each roll.
5. A “farkle” occurs when a roll results in no points. All points accrued during that turn are forfeit, and play is passed to the next player.
6. If the player has scored all six dice, he or she can roll again with all six dice.
7. Once a player has reached the winning point total, each successive player has one last chance to score enough points to surpass the leader.

Proposed Changes to the Traditional Rules for this Version of the Game

Traditionally, Farkle is a multiplayer game with the first player to a given point total being named the winner of the game. The primary play for our proposed version will center on a single player. Two player options will be available in the final release, but we anticipate most players will opt for single player gameplay making it the primary focus of this application. To differentiate single player play from multiplayer play, the single player version will limit the number of turns given to the

player to ten with the primary goal being to maximize the total points for each game. In a sense, the player will be playing against himself or herself trying to beat a previous high score.

Programming Language

The application will be written in Java using JDK version 7.

Platform and Software Requirements

The application will be developed and tested on the Microsoft Windows 7 platform with the Java Runtime Environment 1.7+ installed.

1.2. Definitions, Acronyms, and Abbreviations

- JRE – Java Runtime Environment version 7 or newer
- JDK – Java Development Kit version 7

1.3. References

- The Farkle game rules were adapted from the Farkle Wikipedia page located at: http://en.wikipedia.org/wiki/Farkle
- Single player game play was adapted from the single player game version of the Farkle application found on Facebook at the following location: https://www.facebook.com/farkleonline
- The scoring summary image used in this application was adapted from the scoring summary image used in the Farkle application found on Facebook at the following location: https://www.facebook.com/farkleonline
- The application launcher was created using Visual Studio along with the InstallShield plugin from Flexera Software.
- Version control was implemented using Google Code.
- The application was packaged into a .exe file using Maven.

2. General Description

2.1. Product Perspective

The software is being built to allow the end user to play the popular dice game, Farkle, even if that user does not have a set of dice, or another player to play against. This software will simplify playing this game, and speed up overall gameplay. Without the need for actual dice, and with the addition of automatic scoring, this version of the game can be played multiple times in the time it takes to play one game in the traditional manner.

2.2. Product Functions

This game will allow the end user to play Farkle in multiple modes. The user can elect to play a single player version of the game that departs from the traditional rules by limiting the number of turns that player can take to a total of ten. The goal of this single player mode is to accumulate as many points as possible within the limited number of turns, trying to beat previous high scores. The user can also elect to play a two player version of the game that follows the traditional rules of Farkle, by

proclaiming the winner to be the first to 10,000 points. This two player mode can be played against another user, or against the computer.

2.3. User Characteristics

The anticipated end user is a basic user of the Windows operating system. The user is installing and using this application to take a break from his or her normal activities. This application will provide a few minutes of mindless enjoyment, or a means of procrastination, for the typical university student or professional employee. The application must be easy to install with limited requirements to properly serve this anticipated user.

2.4. General Constraints

As a standalone application, this software does not have many constraints. It is not designed for, nor does it work over, a network connection. Multiplayer mode is limited to two players, and if the user elects to play against another live user, both players must take turns at the same computer.

2.5. Assumptions and Dependencies

This software requires that the end user has the Java Runtime Environment (JRE) version 7 or newer installed. Though it may run on other operating systems with JRE installed, it was developed and optimized for Windows 7. A minimum display resolution of 1200 x 800 pixels is required.

3. Specific Requirements

1.0.0 Graphic User Interface

1.1.0 Select Game Mode Option Box – Upon opening the application, the user is greeted with an option box that includes all configuration options for gameplay. These options include “1 Player Mode”, “2 Player Mode”, “Human Opponent” (if two player mode is selected), “Computer Opponent” (if two player mode is selected), and text fields to enter the associated player names. Also included is a “Start” button and a “Close” button (both of which are always enabled). This option dialog box should pop up over the main GUI set to a solid green color (section 1.2.0).

- 1.1.1 If the user selects the “Close” button at any time, the application closes.
- 1.1.2 The “1 Player Mode” is highlighted by default when the application is first opened, and a blank text field for player one’s name is displayed.
  - 1.1.2.a If the user selects the “Start” button with “1 Player Mode” highlighted and the “Player One Name” field empty, the one player GUI (section 1.3.0) opens with the name “Jacob” assigned to player one.
  - 1.1.2.b If the user selects the “Start” button with “1 Player Mode” highlighted and a name supplied in the “Player One Name” text field, the one player GUI (section 1.3.0) opens with the provided name assigned to player one.

1.1.3 If the user highlights the “2 Player Mode” option, the “1 Player Mode” option is deselected, and two more options appear (“Human Opponent” and “Computer Opponent”). The “Human Opponent” option is highlighted by default.

1.1.4 When the “Human Opponent” option is highlighted, two text fields are displayed, labeled “Player One Name”, and “Player Two Name”.

1.1.4.a If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field is empty, the “Player Two Name” field is empty, and the user selects the “Start” button, the two player mode GUI (section 1.4.0) is opened, and the names “Jacob” and “Brant” are assigned to player 1 and player 2, respectively.

1.1.4.b If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field is empty, the “Player Two Name” field contains a name, and the user selects the “Start” button, the two player mode GUI (section 1.4.0) is opened, the name “Jacob” is assigned to player 1, and the supplied name is assigned to player 2.

1.1.4.c If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field contains a name, the “Player Two Name” field is empty, and the user selects the “Start” button, the two player mode GUI (section 1.4.0) is opened, the name “Brant” is assigned to player 2, and the supplied name is assigned to player 1.

1.1.4.d If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field contains a name, the “Player Two Name” field contains a name, and the user selects the “Start” button, the two player mode GUI (section 1.4.0) is opened, and the supplied names are assigned to player 1 and player 2 accordingly.

1.1.5 When the “Computer Opponent” option is highlighted, the text fields for “Player One Name” and “Player Two Name” are displayed, but the “Player Two Name” field is disabled, and “Computer” is supplied for the “Player Two Name”.

1.1.5.a If “Two Player Mode” is highlighted, “Computer Opponent” is highlighted, the “Player One Name” field is empty, and the user selects the “Start” button, the two player mode GUI (section 1.4.0) is opened, the name “Jacob” is assigned to player 1, and “Computer” is assigned to player 2.

1.1.5.b If “Two Player Mode” is highlighted, “Computer Opponent” is highlighted, the “Player One Name” field contains a name, and the user selects the “Start” button, the two player mode GUI (section

1.4.0) is opened, the supplied name is assigned to player 1, and "Computer" is assigned to player 2.

1.2.0 Items common to the user interface for both modes

1.2.1 The center of the screen shall display the six dice used during gameplay. These dice shall display the name of the game, "Farkle", until the user selects the roll button for the first time.

1.2.2 The total turn score shall be displayed above the dice in the center of the screen, and the score for the selected dice of the current roll shall be displayed directly below the turn score in the center of the screen. This score shall be updated as each die is selected.

1.2.3 Rules for the scoring combinations shall be displayed on the right side of the screen.

1.2.4 A "Roll Dice" button shall be displayed below the dice in the middle of the screen.

1.2.5 A "Bank Score" button shall be displayed below the dice in the middle of the screen, and shall initially be disabled.

1.2.6 After each roll, dice that have previously been selected, scored, and locked shall be shaded to indicate they will not be available on the next roll, and the turn accumulated score shall be updated.

1.2.7 If any roll results in 0 points, the word "Farkle" is prominently displayed, and 0 points is displayed in the accumulated turn score above the dice. The dice retain their current values that resulted in the Farkle. After the Farkle message is displayed, the dice still retain the values that resulted in the Farkle, but all dice are unlocked and play passes to the next player (in two player mode), or to the next turn (in one player mode).

1.2.8 A "Select All" button shall be displayed below the dice in the middle of the screen, and shall be initially disabled.

1.2.9 During a current roll, current dice selected by the user shall be indicated with a yellow border around each selected die, and the score for the currently selected dice shall be updated above the dice.

1.2.10 A menu shall be displayed at the top of the main GUI with one main option, "File", and five sub options: "Hint", "New Game", "Restart Game", "Reset High Score", and "Quit".

1.2.10.a If the user selects the "New Game" option, the select game mode option box (section 1.1.0) is displayed.

1.2.10.b If the user selects the "Restart Game" option, the current game with all current configurations (player mode, player names, and player types) is restarted.

1.2.10.c If the user selects the "Quit" option, the application is closed.

1.2.10.d If the user selects the "Hint" option, the dice combination for the highest possible score for the current roll is displayed.

1.2.10.e The "Hint" option shall only be available after a player has rolled, and before that player has selected any dice.

1.2.10.f If the user selects the "Reset High Score" option, the high score is reset to 0.

1.2.10.g The "New Game" option and the "Reset Game" option shall be disabled while the computer is taking its turn.

1.3.0 <u>One player mode graphic user interface</u>

1.3.1 The title of the window shall display: "Farkle – Single Player Mode".

1.3.2 The overall point total for the current game shall be displayed on the upper left hand corner of the screen, just below the player's name.

1.3.3 The left side of the screen shall have an area to display the point total for each of the ten turns taken in single player mode.

1.3.4 The current turn shall be indicated by highlighting that turn on the left side of the screen. This turn shall be highlighted as soon as the previous turn ends (which occurs after the player selects the "Bank Score" button, or after the Farkle message animation completes), and before the player selects the "Roll Dice" button for the current turn.

1.3.5 The current highest achieved score shall be displayed on the lower left hand corner of the screen. This score shall initially be set to 0 points.

1.3.6 The top of the left hand side of the screen shall display "Player: ", along with the provided name of the player.

1.4.0 <u>Two player mode graphic user interface.</u>

1.4.1 The title of the window shall display, "Farkle – Two Player Mode".

1.4.2 The left side of the screen shall have an area to display the overall accumulated point total for each player. This takes the place of the area displaying the point total for each turn in the one player mode graphic user interface.

1.4.2.a Each player shall be indicated in the following manner: "Player: " along with the provided player's name, or "Computer" if a "Computer Opponent" has been selected, followed by the running point total for the current game for that player.

1.4.3 The player whose turn it is shall be indicated by highlighting that player's current turn on the left side of the screen. The turn shall be highlighted after the previous player's roll ends and before the player selects the "Roll Dice" button for the first time for the current turn.

1.4.4 The first player to meet the minimum total point threshold required to win the game (equal to 10,000 points) shall be highlighted in a different color to indicate each subsequent player has one more turn to try and beat that player's score.

1.4.5 The turn totals for each player shall be displayed in a scroll pane below that player's name and game score. This scroll pane shall initially display 5 turns, adding additional turns after they are taken. The scrolling ability shall be enabled at the beginning of the 11^{th} turn.

1.4.6 After a player has surpassed the 10,000 point threshold, a message dialog box is displayed indicating the other player has one last turn to try and beat that player's total score.

1.5.0 Conclusion of game option box

1.5.1 At the conclusion of the game, an option box shall be displayed with the player's overall score for the completed game (in one player mode), or the winner of the current game (in two player mode). This option box shall include three options: "Play Again?", "Main Menu", and "Quit".

1.5.1.a If the user selects the "Play Again?" button, the game will be restarted with all of the same configuration options of the previous game (player mode, player's names, and player types).

1.5.1.b If the user selects the "Main Menu" button, the select game mode option box will be displayed (see section 1.1.0).

1.5.1.c If the user selects the "Quit" button, the application will immediately close.

1.5.1.d If the user selects the close button in the upper portion of the conclusion of game option box, the application will immediately close.

2.0.0 Game Modes

2.1.0 When one player mode is selected, the one player mode graphic user interface (section 1.3.0) is displayed with the name "Farkle" displayed on the dice, the "Bank Score" button disabled, the "Select All" button disabled, and turn number one highlighted. The user will have ten turns to try and get as many points as possible.

2.1.1 Each turn is taken according to the requirements of section 4.0.0.

2.1.2 The game ends at the conclusion of the tenth turn, and the player's score is compared to the current high score.

2.1.3 If the player's score is greater than the current high score, a congratulatory message is displayed, and the player's score replaces the previous high score.

2.1.4 The conclusion of game option box (section 1.5.0) is displayed at the completion of the tenth turn.

2.2.0 When two player mode against a live person is selected, the two player mode graphic user interface (section 1.4.0) is displayed with the name "Farkle" displayed on the dice, the "Bank Score" button disabled, the "Select All" button disabled, and player one highlighted indicating it is his or her turn.

2.2.1 Each turn is taken according to the requirements of section 4.0.0. The current player for each turn is highlighted during that player's turn.

2.2.2 The first player to surpass 10,000 total points at the end of a given turn is highlighted in a different color.

2.2.3 The other player has one more turn to try and surpass the point total of the first player to surpass 10,000 points.

2.2.4 The conclusion of game option box (section 1.5.0) is displayed after a player wins.

2.3.0 When two player mode against the computer is selected, the two player mode graphic user interface (section 1.4.0) is displayed with the name "Farkle" displayed

on the dice, the "Bank Score" button disabled, the "Select All" button disabled, and player one highlighted indicating it is his turn.

2.3.1 Each turn is taken according to the requirements of section 4.0.0. The current player for each turn is highlighted during that players turn.

2.3.2 Decisions made during the computer player's turn are chosen in accordance with requirements section 5.0.0.

2.3.3 The first player to surpass 10,000 total points at the end of a given turn is highlighted in a different color.

2.3.4 The other player has one more turn to try and surpass the point total of the first player to surpass 10,000 points.

2.3.5 The conclusion of game option box (section 1.5.0) is displayed after a player wins.

3.0.0 Dice

3.1.0 Farkle is played with six standard 6 sided dice with each side numbered from 1 through 6 (inclusive).

3.2.0 Each die that is rolled shall be assigned a random value from 1 to 6 (inclusive) at the conclusion of the roll.

4.0.0 Player's Turn

4.1.0 At the beginning of the turn, the turn score is set to 0. The player selects the "Roll Dice" button, and all 6 dice are rolled in accordance with requirement 3.2.0. The "Select All" button is enabled after the initial roll takes place.

4.2.0 The resulting roll is analyzed according to requirement 6.0.0 to determine if the player farkled. A farkle occurs if the roll results in 0 points.

4.3.0 If the player did not farkle, he or she must select at least one scoring die. The score for the selected dice is calculated according to requirement 5.0.0, and is updated after each die selection. The score for the selected dice is displayed in accordance with section 1.2.9. If any of the selected dice does not contribute to the score, a selected dice score of 0 is displayed and the "Roll Dice" and "Bank Score" buttons are disabled.

4.4.0 When all of the selected dice contribute to the point total for the roll, the "Roll Dice" button is enabled, and the roll point total is added to the running point total for the current turn.

4.5.0 If the current turn score is greater than or equal to 300, the bank button is enabled.

4.6.0 If the player elects to roll again the selected dice are locked, the remaining dice are rolled, and the process returns to requirement 4.2.0.

4.7.0 If all six dice have been selected, and they all contribute to the turns point total, the player is issued a bonus roll indicated with a message in the current turn box. All selected and locked dice are deselected and unlocked, and the process returns to requirement 4.1.0.

4.8.0 If the player selects the bank button, the current turn point total is added to the player's game point total, and the turn is over.

4.9.0 If the player farkles on any roll during the current turn, that player loses all points accumulated during the current turn, the farkle message is displayed per requirement 1.2.7, and the turn is over.

4.10.0 When the turn is over all dice are unlocked while continuing to display their last value , the “Roll Dice” button is enabled, the “Bank Score” button is disabled, the “Select All” button is disabled, the current turn point total is set to 0, the current roll point total is set to 0, and play passes to the next player (two player mode) or the next turn (single player mode) by highlighting the appropriate player or turn on the left hand side of the screen.

5.0.0 Computer player

5.1.0 The computer player takes its turn in accordance with requirement 4.0.0, and the dice selection, as well as the decision to continue rolling the dice, are made in accordance with the following requirements.

5.2.0 After each roll, the computer player always selects the maximum scoring combination of dice.

5.3.0 If the current turn point total is less than the goal calculated in section 5.5.0, the computer always rolls again.

5.4.0 If the previous roll resulted in a bonus roll, the computer always rolls again.

5.5.0 The computer’s goal is calculated after each roll. This goal is pseudo-randomly selected as 300 fifty percent of the time, 600 thirty percent of the time, and 1000 twenty percent of the time.

6.0.0 Scoring

6.1.0 Each 1 rolled is worth 100 points.

6.2.0 Each 5 rolled is worth 50 points.

6.3.0 Three 1’s are worth 1000 points.

6.4.0 Three of a kind of any value other than 1 is worth 100 times the value of the die (e.g. three 4’s is worth 400 points).

6.5.0 Four, five, or six of a kind is scored by doubling the three of a kind value for every additional matching die (e.g. five 3’s would be scored as 300 X 2 X 2 = 1200).

6.6.0 Three distinct doubles (e.g. 1-1-2-2-3-3) is worth 750 points. This scoring rule does not include the condition of rolling four of a kind along with a pair (e.g. 2-2-2-2-3-3 is worth does not satisfy the three distinct doubles scoring rule).

6.7.0 A straight (e.g. 1-2-3-4-5-6), which can only be achieved when all 6 dice are rolled, is worth 1500 points.

Design Documentation

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

Team 1: Curtis Brown, Jacob Davidson, Brant Mullinix
Course: CSC 478 - B
Farkle Project: UML Diagram
Date: 11/29/2014
Version: 3.0.0

Game

-bonusTurn : boolean
-controller : FarkleController
-currentPlayer : int
-gameMode : GameMode
-players : Player[]
-prefs : Preferences

+Game(GameMode, FarkleController)
+bank() : int
+calculateHighestScore(List<Integer>) : Object[]
+calculateScore(List<Integer>, boolean) : int
+farkle() : void
+getCurrentPlayer() : Player
+getGameMode() : GameMode
+getGameScoreForCurrentPlayer() : int
+getGameScoreForPlayer(int) : int
+getHighScore() : int
+getNextPlayer() : int
+getPlayerName(int) : String
+getPlayerNumberForCurrentPlayer() : int
+getPlayers() : Player[]
+getPlayerTypeForCurrentPlayer() : PlayerType
+getRollScores() : int
+getTurnNumberForCurrentPlayer() : int
+getWinningPlayerInfo() : String[]
+isBonusTurn() : boolean
+processHold(int) : void
+processRoll() : void
+resetGame() : void
+resetHighScore() : void
+setBonusTurn(boolean) : void
+setCurrentPlayer(int) : void
+setHighScore(int) : void
+setPlayerName(int, String) : void
+setPlayerType(int, PlayerType) : void

1
1..2

Player

-gameScore : int
-playerName : String
-playerNumber : int
-rollNumber : int
-rollScore : HashMap<Integer, Integer>
-turnNumber : int
-turnScores : ArrayList<Integer>
-type : PlayerType

+Player(int)
+endTurn(boolean) : void
+getGameScore() : int
+getPlayerName() : String
+getPlayerNumber() : int
+getRollNumber() : int
+getRollScore() : HashMap<Integer, Integer>
+getRollScores() : int
+getTurnNumber() : int
+getTurnScores() : ArrayList<Integer>
+getType() : PlayerType
+resetRollScores() : void
+resetTurnScores() : void
+scoreRoll(int) : void
+setGameScore(int) : void
+setPlayerName(String) : void
+setRollNumber(int) : void
+setRollScore(HashMap<Integer, Integer>) : void
+setTurnNumber(int) : void
+setTurnScores(ArrayList<Integer>) : void
+setType(PlayerType) : void

FarkleController

~farkleGame : Game
~farkleOptions : FarkleOptionsDialog
~farkleUI : FarkleUI
~isLastTurn : boolean
~isTest : boolean
~rand: Random
~POINT_THRESHOLD : int

+main(String[]) : void
+FarkleController(boolean)
+actionPerformed(ActionEvent) : void
+asynchronousRoll() : void
+bank() : void
+bankHandler() : void
+checkHighScore(int) : boolean
+compDecision() : void
+endGame(boolean, boolean, boolean) : void
+farkle() : void
+gameEnded(boolean, boolean) : boolean
+getHighestPossibleScore() : int
+getHighestScoringDieValues() : List<Integer>
+isHintAvailable() : boolean
+isResetHighScoreAvailable() : boolean
+isResetOrNewGameAvailable() : boolean
+mouseClicked(MouseEvent) : void
+mouseEntered(MouseEvent) : void
+mouseExited(MouseEvent) : void
+mousePressed(MouseEvent) : void
+mouseReleased(MouseEvent) : void
+newGame(FarkleOptionsDialog) : void
+replayGame() : void
+resetHighScore() : void
+rollHandler() : void
+setTurnScore(int, int, int) : void
+setUI(FarkleUI) : void
+setUIHighScore(int) : void
+tryToEndGame() : void

1
1

FarkleOptionsDialog

-closeButton : JButton
-computerPlayerLabel : JLabel
-gameMode : GameMode
-gameModeOptionTitle : JLabel
-greenBackground : Color
-humanPlayerLabel : JLabel
-multiPlayerLabel : JLabel
-player1Name : String
-player2Name : String
-playerModeSelectionPanel : JPanel
-playerModeSelectLabel : JLabel
-playerNamesLabel : JLabel
-playerOneName : JTextField
-playerOneNameLabel : JLabel
-playerOnePanel : JPanel
-playerTwoName : JTextField
-playerTwoNameLabel : JLabel
-playerTwoNamePanel : JPanel
-playerType : PlayerType
-playerTypeSelectionPanel : JPanel
-playerTypeSelectLabel : JLabel
-singlePlayerLabel : JLabel
-startButton : JButton
-serialVersionUID : long

+FarkleOptionsDialog()
+getGameMode() : GameMode
+getJLabel(int) : JLabel
+getJPanel(int) : JPanel
+getJTextField(int) : JTextField
+getPlayer1Name() : String
+getPlayer2Name() : String
+getPlayerType() : PlayerType
+mouseClicked(MouseEvent) : void
+mouseEntered(MouseEvent) : void
+mouseExited(MouseEvent) : void
+mousePressed(MouseEvent) : void
+mouseReleased(MouseEvent) : void
+setPlayer1Name(String) : void
+setPlayer2Name(String) : void
+setPlayerType(PlayerType) : void
+showWindow() : void

Program Entry

FarkleUI

-audioStream : AudioInputStream
-bankBtn : JButton
-bankSounds : ArrayList<URL>
-bonusSound : URL
-controller : FarkleController
-dice : Die[]
-farkleMessage : JDialog
-greenBackground : Color
-highScore : JLabel
-highScoreTitle : JLabel
-isFirstRun : boolean
-isTest : boolean
-player1GameScore : JLabel
-player1GameScoreLabel : JLabel
-player1Name : JLabel
-player1NameLabel : JLabel
-player1ScoreLabels : ArrayList<JLabel>
-player1ScorePanel : JPanel
-player1Scores : ArrayList<JLabel>
-player1ScrollBar : JScrollBar
-player2GameScore : JLabel
-player2GameScoreLabel : JLabel
-player2Name : JLabel
-player2NameLabel : JLabel
-player2ScoreLabels : ArrayList<JLabel>
-player2ScorePanel : JPanel
-player2Scores : ArrayList<JLabel>
-player2ScrollBar : JScrollBar
-rollBtn : JButton
-rollScore : JLabel
-rollSounds : ArrayList<URL>
-runningScore : JLabel
-selectAllBtn : JButton
-serialVersionUID : long

+FarkleUI(FarkleController)
+addHighScore(Jpanel) : void
+addTurnScore(int, int) : void
+buildDicePanel() : void
+buildPlayerPanel(GameMode) : void
+createButtonPanel() : Jpanel[]
+createDiceGridPanel() : Jpanel
+createDiceHeaderPanel() : JPanel
+createFarkleMenuBar() : JMenuBar
-createPlayerNamePanel(int) : JPanel
-createPlayerScorePanel(int, int) : JScrollPane
+createScoreGuidePanel() : void
+diceFirstRun() : void
+disableDice() : void
+displayMessage(String, String) : void
+displayYesNoMessage(String, String) : boolean
+dispose() : void
+getBankBtn() : JButton
+getDice(DieState…) : ArrayList<Die>
+getDieValues(DieState…) : List<Integer>
+getFarkleMessage() : JDialog
+getGameScore(int) : int
+getHighScore() : int
+getPlayerScoreLabels(int) : ArrayList<JLabel>
+getPlayerScores(int) : ArrayList<JLabel>
+getRollBtn() : JButton
+getRunningScore() : int
+getSelectAllBtn() : JButton
-getSounds() : void
+getTurnScore(int, int)
+highlightTurn(JLabel, JLabel, boolean) : void
+highlightTurnScore(int, int, boolean) : void
+initUI() : void
+lockScoredDice() : void
+playBankSound() : void
+playBonusSound() : void
+playRollSound() : void
+resetDice() : void
+resetDicePanel(): void
+resetScores(int) : void
+rollDice() : void
+selectAllDice() : void
+selectDice(List<Integer>) : void
+setGameScore(int, int) : void
+setHighScore(int) : void
+setPlayerName(int, String) : void
+setRollScore(int) : void
+setRunningScore(int) : void
+setTurnScore(int, int, int) : void
+unHighlightAllTurnScores(int) : void
+unHighlightTurn(JLabel, JLabel) : void

1
1
1
1
1
6

<<Enumeration>> PlayerType

COMPUTER
USER

<<Enumeration>> GameMode

MULTIPLAYER
SINGLEPLAYER

<<Uses>>

DieClickEvent

-serialVersionUID : long

+DieClickEvent(Component, int, long, int, int, int, int, boolean)

FarkleMessage

~farkleCenterMsg : JLabel
~farkleLeftMsg : JLabel
~farkleRightMsg : JLabel
~farkleSound : URL
-serialVersionUID : long

+FarkleMessage()
+playFarkleSound() : void
+setVisible(boolean) : void

<<Enumeration>> DieState

DISABLED
HELD
SCORED
UNHELD

Die

-icon : Icon
-random : Random
-state : DieState
-value : char
-values : char[]
-ARC : int
-BORDER_GAP : int
-HELD_BORDER : BORDER
-OUT_FRAME : int
-OVAL_RADIUS : int
-serialVersionUI : long
-SML_GAP : int
STROKE_WIDTH float:
-UNHELD_BORDER : BORDER

+Die(FarkleController)
-fillOval(Graphics2D, int, int) : void
+getIcon() : Icon
+getState() : DieState
+getValue() : int
+isHeld() : boolean
+isScored() : boolean
+isUnheld() : boolean
+paintComponent(Graphics) : void
+roll() : void
+setIcon(Icon) : void
+setState(DieState) : void
+setValue(int) : void

Pseudocode

The following psuedocode outlines the two most important algorithms of the Farkle applications. These algorithms are included in the calculateScore(List<Integer> roll, Boolean checkHeldDice), and calculateHighestScore(List<Integer> dice) methods of the Game class. The calculateScore method calculates the Farkle score for a list of integers that represents a dice roll, and is used to calculate the score of a player's selected dice and to determine if a player Farkled after a given roll. The calculatedHighestScore method determines the combination of a list of integers (representing a dice roll) that results in the highest score and calculates that score. The calculateHighestScore method is used to determine the dice a computer player selects after a given roll, and to provide the list of dice that results in the highest score when the user selects the "Hint" option from the file menu.

1. Calculate Score Method in the Game Class

(Fulfills all requirements from section 6.0.0 Scoring of the Requirements Document)

CalculateHighScore (Takes a list of integers, roll, and a Boolean flag, checkHeldDice, as parameters) returns an integer:

Initialize an integer variable, *calculatedScore*, that will store the calculated score;
If *roll* is not null:
Set *calculatedScore* to 0;
If *roll* is not empty:
Initialize a Boolean variable, *incorrectValuePresent*, and set it to false. This flag is set to true if any incorrect values are present;
For each value in *roll*:
If the value is not equal to an integer from 1 to 6 (inclusive), set *incorrectValuePresent* to true;
If *incorrectValuePresent* is false:
Initialize a Boolean variable, *oneOrTwoStrictDie*, and set it to false. This flag is set to true if there are only one or two dice in the list;
Initialize an integer variable, *numberOfDieHeld*, and set it to 0. This variable counts the number of held dice;
Initialize a Boolean, *isStraight*, and set it to true. This flag checks for a straight;
Initialize a Boolean, *isThreePair*, and set it to true. This flag checks for three pairs;
Initialize an int, *countOfPairs*, and set it to 0. This variable counts the number of pairs;
Initialize an array of integers, *countedDie*, and set it to a size of 6. This array will count the number of each value of dice present in the roll;
For each value in *roll*:
Increment the integer at that values index in *countedDie*, effectively counting the number of each value of die in present in the roll;
For each value in the *countedDie*:
Initialize an integer variable, *currentCount*, and set it equal to the value in *countedDie*;

If the current value is located at index 1, 2, 3, or 5 (representing 2, 3, 4, or 6 valued die), and *currentCount* is 1 or 2, set the *oneOrTwoStrictDie* flag to true;
If *currentCount* does not equal 1, we know that the roll cannot be a straight. Set the *isStraight* flag to false;
If *currentCount* does not equal 0 or 1, we know that the roll cannot contain three pairs. Set the *isThreePair* flag to false;
If *currentCount* equals 2, increment *countOfPairs*;
Initialize a double variable, *temp,* that will be used for power calculations;
Switch on the current die value, represented by the current index of the *countedDie* array;

Case 0 (which represents a die value of 1):
If *currentCount* is greater than 0 and less than 3:
Add *currentCount* * 100 to *calculatedScore*;
Else if *currentCount* is greater than 3:
Set *temp* to 2^(*currentCount* – 3);
Add 1000 * *temp* to *calculatedScore*;
Break;

Case 4 (which represents a die value of 5):
If *currentCount* is greater than 0 and less than 3:
Add *currentCount* * 50 to *calculatedScore*;
Else if *currentCount* is greater than 3:
Set *temp* to 2^(*currentCount* – 3);
Add 500 * *temp* to *calculatedScore*;
Break;

Default (which represents a die value of 2, 3, 4, or 6):
If *currentCount* is greater than 3:
Set *temp* to 2^(*currentCount* – 3);
Add the Current die value * *temp* to *calculatedScore*;
Else If *isStraight* equals true, the current die value is 6, and the size of *roll* is 6:
Add 1500 to *calculatedScore*;
Else If *isThreePair* equals true, the current die value is 6, and the size of *roll* is 6:
Add 750 to *calculatedScore*;
Break;

If *checkHeldDice* is true, we need to return 0 if any of the dice do not contribute to the score:
If *countOfPairs* does not equal 3:
Set *isThreePair* to false indicating the roll does not contain 3 pairs;
If *oneOrTwoStrictDie* is true, *isStraight* is false, and *isThreePair* is false:

Set *calculatedScore* to 0, because we know that 2, 3, 4, or 6 occurs once or twice, and the roll does not contain three pair or a straight.

If *numberOfHeldDie* is equal to 0:

Set *calculatedScore* to 0

Else the list of integers representing the roll is null:

Set *calculatedScore* to 0;

Return *calculatedScore*;

2. Calculate Highest Score Method in the Game Class

(Fulfills the requirements of sections 1.2.10.d and 5.2.0 of the Requirements Document)

CalculateHighestScore (Takes a list of integers, dice as a parameters) returns an Object array:

Initialize integer variable, *highestScorePossible*, and set it to 0. This variable will store the calculated score for the combination of integers that result in the highest score;

Initialize an Object array, *highestScore*, and set it to a size of 2. This is the array that will be returned;

Initialize a List of Integers, *highestScoringCombination*. This variable will hold the list of dice that result in the highest score;

If *dice* is not null, and the size of dice is greater than 0:

Initialize a variable, *magicNumber*, set it equal to the number of dice ^ 2 minus 1. This is the total number of combinations of dice that can be selected for the given roll.

For every value from 1 to *magicNumber*:

Initialize a list of integers, *tempList*, that will store the list of dice for the current permutation;

Initialize a character array, *binaryNumber*, set it to the same size as the *dice* list, and fill it with the character "0". This array will hold the character representation of the binary number for the current value;

Convert the current value to a character array representing the binary number, and set *binaryNumber* to this character array;

For each character in the *binaryNumber* array:

If the character is a "1", add the die value in the associated position of the *dice* list to the *tempList* list;

Initialize an integer variable, *tempScore*, and set it equal to the calculated score for the current *tempList*. This score is calculated by passing *tempList* and false to the calculateScore method;

If *tempScore* is greater than *highestPossibleScore*:

Set *highestScorePossible* equal to *tempScore*;

Clear the *highestScoringCombination* list;

Set *highestScoringCombination* equal to *tempList;*

Set the first element of the *highestScore* array equal to *highestScorePossible;*

Set the second element of the *highestScore* array equal to *highestScoringCombination*;

Return the *highestScore* array;

Decision Tables

1. Game Mode Options Box Decision Table

(Fulfills requirements 1.1.0, 1.1.2, 1.1.3, 1.1.4, and 1.1.5 of the Requirements Document.)

Conditions	Rules		
Single Player Mode	✓	✗	✗
Two Player Mode	✗	✓	✓
Human Opponent	✗	✓	✗
Computer Opponent	✗	✗	✓
Actions			
Display Player 1 Name Text Field	✓	✓	✓
Display Player 2 Name Text Field	✗	✓	✓
Enable Player 2 Name Text Field	✗	✓	✗
Display Opponent Type Selection	✗	✓	✗
Supply the Name "Computer" to the Player 2 Name Text Field	✗	✗	✓

2. Game Initialization Decision Table

(Fulfills requirements 1.1.2.a, 1.1.2.b, 1.1.4.a, 1.1.4.b, 1.1.4.c, 1.1.4.d, 1.1.5.a, and 1.1.5.b of the Requirements Document.)

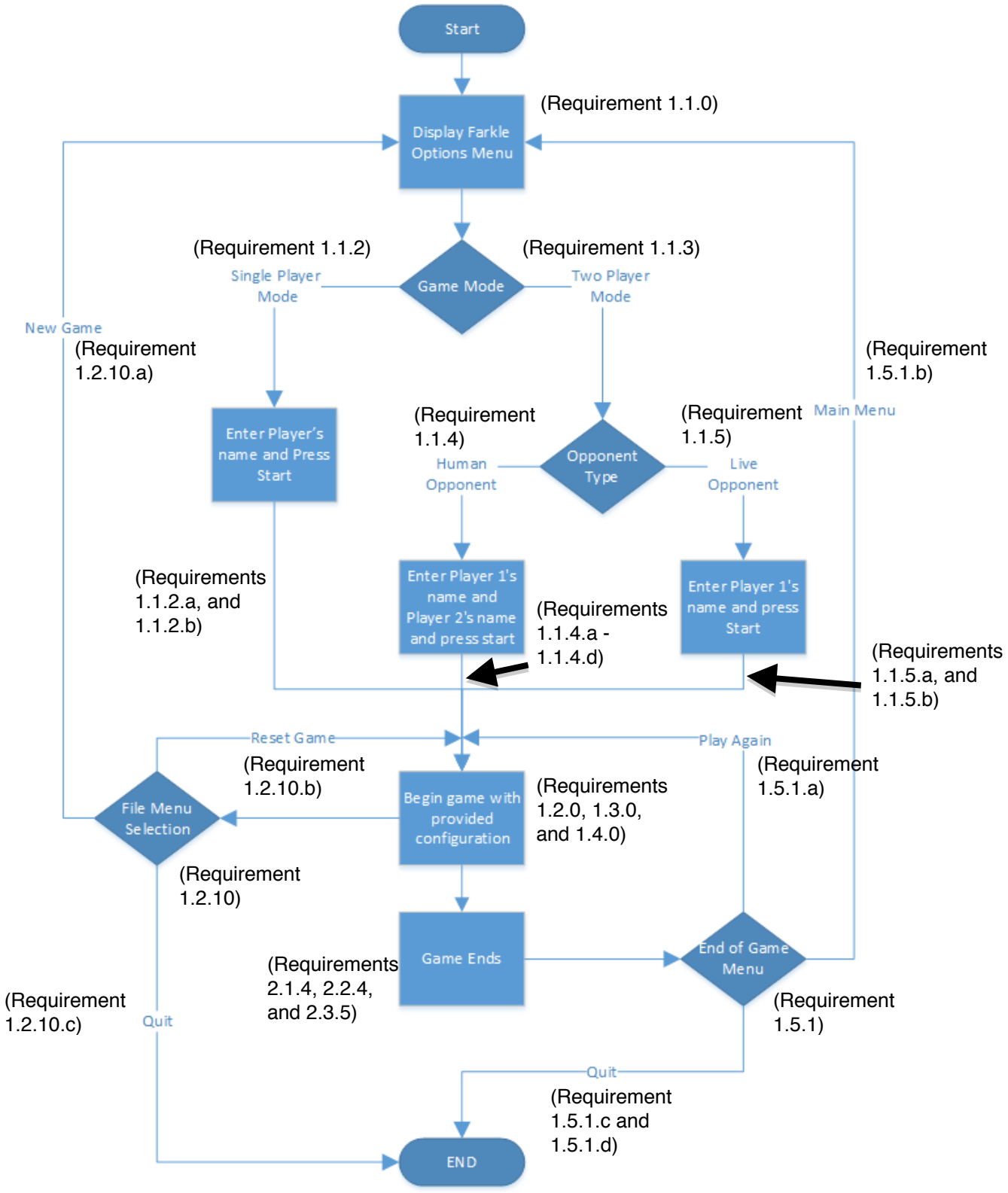
Conditions	Rules							
Single Player Mode	✓	✓	✗	✗	✗	✗	✗	✗
Two Player Mode	✗	✗	✓	✓	✓	✓	✓	✓
Human Opponent	✗	✗	✓	✓	✓	✓	✗	✗
Computer Opponent	✗	✗	✗	✗	✗	✗	✓	✓
Player 1 name supplied	✗	✓	✗	✓	✗	✓	✗	✓
Player 2 name supplied	✗	✗	✗	✗	✓	✓	✗	✗
Actions								
Open the Single Player GUI	✓	✓	✗	✗	✗	✗	✗	✗
Open the Two Player GUI	✗	✗	✓	✓	✓	✓	✓	✓
Assign the Supplied Name to Player 1	✗	✓	✗	✓	✗	✓	✗	✓
Assign the Supplied Name to Player 2	✗	✗	✗	✗	✓	✓	✗	✗
Assignt the Name "Jacob" to Player 1	✓	✗	✓	✗	✓	✗	✓	✗
Assign the Name "Brant" to Player 2	✗	✗	✓	✓	✗	✗	✗	✗
Assign the Name "Computer" to Player 2	✗	✗	✗	✗	✗	✗	✓	✓

3. Dice Roll Decision Table

(Fulfills requirements 4.1.0, 4.2.0, 4.3.0, 4.4.0, 4.5.0, and 4.7.0 of the Requirements Document.)

Conditions	Rules									
Dice Have Been Rolled	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dice Have Been Selected All of Which Are Scoreable	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
Dice Have Been Selected Some of Which Are Not Scoreable	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
No Dice Have Been Selected	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
No Dice for the Current Roll Are Scoreable	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
The Turn Total is Greater than 300 Points	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓
All Dice Have Been Selected and All Are Scoreable	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Actions										
Roll Button is Enabled	✓	✗	✗	✗	✓	✓	✗	✗	✓	✓
Select All Button is Enabled	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗
Bank Button is Enabled	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Farkle Message is Displayed	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Bonus Roll is Issued	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Roll Score is 0	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
Roll Score is Added to the Running Turn Score	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
Turn Score is Set to 0	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗

Farkle Application Control Diagram



Source Code

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

```java
package com.lotsofun.farkle;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.Stroke;
import java.awt.image.BufferedImage;
import java.util.Random;
import javax.swing.BorderFactory;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.border.Border;

/**
 * Displays a die and its state in the graphic user interface with the a value
 * from 1 to 6 (inclusive). The die value is determined via the roll method.
 * Tracks the state of the die, boxing a held die in a yellow border, and
 * shading a scored die in gray.
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public class Die extends JLabel {

    /** Serialize this to comply with component conventions */
    private static final long serialVersionUID = 1L;

    /** The standard border gap used */
    private static final int BORDER_GAP = 3;

    /** The border used for an unheld die */
    private static final Border UNHELD_BORDER = BorderFactory
            .createEmptyBorder(BORDER_GAP, BORDER_GAP, BORDER_GAP, BORDER_GAP);

    /** The border used for a held die */
    private static final Border HELD_BORDER = BorderFactory
            .createCompoundBorder(BorderFactory.createEmptyBorder(BORDER_GAP,
                    12, BORDER_GAP, 12), BorderFactory.createLineBorder(
                    Color.YELLOW, BORDER_GAP));

    /** Standard die image width and height */
    private static final int OUT_FRAME = 55;

    /** Die corner radius */
    private static final int ARC = 8;

    /** Standard line thickness */
    private static final float STROKE_WIDTH = 2f;

    /** Small gap used around die graphic */
    private static final int SML_GAP = 1;

    /** Radius of the die value dots */
    private static final int OVAL_RADIUS = 12;
```

```
58
59      /** Random number used to generate die values for a given roll */
60      private Random random = new Random();
61
62      /** The created die image */
63      private Icon icon;
64
65      /** The value of the die */
66      private char value;
67
68      /** The state of the die */
69      private DieState state;
70
71      /** The possible numeric die values */
72      private final char[] values = { '1', '2', '3', '4', '5', '6' };
73
74      /**
75       * Constructor takes a reference to the controller to add as a listener, and
76       * sets the initial die state to unheld.
77       *
78       * @param controller
79       *            MouseListener this will reference
80       */
81      public Die(FarkleController controller) {
82
83          // Call the default JLabel constructor
84          super();
85
86          // Set the die state to unheld
87          this.state = DieState.UNHELD;
88
89          // Add a mouse listener to this die from the controller
90          this.addMouseListener(controller);
91      }
92
93      /**
94       * Sets the die value to a random number and repaints it
95       */
96      public void roll() {
97
98          /*********************************************
99           * 3.2.0: Each die that is rolled shall be assigned a random value from
100          * 1 to 6 (inclusive) at the conclusion of the roll.
101          *********************************************/
102         /*********************************************
103          * 4.6.0: If the player elects to roll again, the remaining dice are
104          * rolled and the process returns to requirement 4.2.0.
105          *********************************************/
106         // Roll die only if die is unheld
107         if (this.state == DieState.UNHELD) {
108
109             // Set the die value to a random char from 1 to 6
110             value = (char) values[random.nextInt(6)];
111
112             // Repaint the die
113             this.repaint();
114         }
```

```
115     }
116
117     /**
118      * Draw the Die object, set the face value to the current value of the die,
119      * and decorate it according to the die state
120      */
121     @Override
122     public void paintComponent(Graphics g) {
123
124         // Cast the Graphics object as a Graphics2D
125         Graphics2D g2 = (Graphics2D) g;
126         super.paintComponent(g2);
127
128         // Set the image size
129         BufferedImage img = new BufferedImage(OUT_FRAME, OUT_FRAME,
130                 BufferedImage.TYPE_INT_ARGB);
131         g2 = img.createGraphics();
132
133         // Set the stroke width
134         Stroke stroke = new BasicStroke(STROKE_WIDTH);
135
136         // The die is white if it is not in a scored state
137         if (this.state != DieState.SCORED) {
138             g2.setColor(Color.white);
139
140             // Else, the die is in a scored state, set its color to gray
141         } else {
142             g2.setColor(Color.gray);
143         }
144
145         // Generate the die
146         g2.fillRoundRect(0, 0, OUT_FRAME, OUT_FRAME, ARC, ARC);
147         g2.setColor(Color.black);
148         g2.setStroke(stroke);
149         g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
150                 RenderingHints.VALUE_ANTIALIAS_ON);
151         g2.drawRoundRect(SML_GAP, SML_GAP, OUT_FRAME - SML_GAP * 2, OUT_FRAME
152                 - SML_GAP * 2, ARC, ARC);
153         g2.setColor(Color.black);
154
155         // Add the die dots based on the current die value
156         switch (value) {
157         case '1':
158             fillOval(g2, 1, 1);
159             break;
160         case '2':
161             fillOval(g2, 0, 0);
162             fillOval(g2, 2, 2);
163             break;
164         case '3':
165             fillOval(g2, 0, 0);
166             fillOval(g2, 1, 1);
167             fillOval(g2, 2, 2);
168             break;
169         case '4':
170             fillOval(g2, 0, 0);
171             fillOval(g2, 0, 2);
```

```
172             fillOval(g2, 2, 0);
173             fillOval(g2, 2, 2);
174             break;
175         case '5':
176             fillOval(g2, 0, 0);
177             fillOval(g2, 0, 2);
178             fillOval(g2, 1, 1);
179             fillOval(g2, 2, 0);
180             fillOval(g2, 2, 2);
181             break;
182         case '6':
183             fillOval(g2, 0, 0);
184             fillOval(g2, 0, 1);
185             fillOval(g2, 0, 2);
186             fillOval(g2, 2, 0);
187             fillOval(g2, 2, 1);
188             fillOval(g2, 2, 2);
189             break;
190 
191         /*
192          * The following die values are used to display "Farkle" On the dice
193          * when starting a new game.
194          */
195         case 'f':
196             fillOval(g2, 0, 0);
197             fillOval(g2, 1, 0);
198             fillOval(g2, 2, 0);
199             fillOval(g2, 0, 1);
200             fillOval(g2, 0, 2);
201             fillOval(g2, 1, 1);
202             break;
203         case 'a':
204             fillOval(g2, 2, 0);
205             fillOval(g2, 2, 2);
206             fillOval(g2, 0, 1);
207             fillOval(g2, 1, 1);
208             break;
209         case 'r':
210             fillOval(g2, 0, 0);
211             fillOval(g2, 1, 0);
212             fillOval(g2, 2, 0);
213             fillOval(g2, 0, 1);
214             fillOval(g2, 0, 2);
215             break;
216         case 'k':
217             fillOval(g2, 0, 0);
218             fillOval(g2, 1, 0);
219             fillOval(g2, 2, 0);
220             fillOval(g2, 0, 2);
221             fillOval(g2, 2, 2);
222             fillOval(g2, 1, 1);
223             break;
224         case 'l':
225             fillOval(g2, 0, 0);
226             fillOval(g2, 1, 0);
227             fillOval(g2, 2, 0);
228             fillOval(g2, 2, 2);
```

```
229             fillOval(g2, 2, 1);
230             break;
231         case 'e':
232             fillOval(g2, 0, 0);
233             fillOval(g2, 1, 0);
234             fillOval(g2, 2, 0);
235             fillOval(g2, 0, 1);
236             fillOval(g2, 0, 2);
237             fillOval(g2, 1, 1);
238             fillOval(g2, 2, 0);
239             fillOval(g2, 2, 2);
240             fillOval(g2, 2, 1);
241             break;
242         default:
243             break;
244         }
245
246         // Done creating the die graphic, dispose of the Graphics2D object
247         g2.dispose();
248
249         // Set the Image Icon for this JLabel to the created image
250         this.setIcon(new ImageIcon(img));
251
252         // Repaint the JLabel
253         this.repaint();
254     }
255
256     /**
257      * Adds a black dot to the die graphic at the specified location
258      *
259      * @param g2
260      *             The graphics2D object
261      * @param row
262      *             Integer row number for the dot
263      * @param col
264      *             Integer column number for the dot
265      */
266     private void fillOval(Graphics2D g2, int row, int col) {
267
268         // Calculate the width of the die (minus the stroke width)
269         double rectWidth = OUT_FRAME - 4 * STROKE_WIDTH;
270
271         // Convert the specified column to an absolute x location for the
272         // graphic
273         int x = (int) (2 * STROKE_WIDTH - OVAL_RADIUS / 2 + (col + 0.5)
274                 * rectWidth / 3);
275
276         // Convert the specified row to an absolute y location for the graphic
277         int y = (int) (2 * STROKE_WIDTH - OVAL_RADIUS / 2 + (row + 0.5)
278                 * rectWidth / 3);
279
280         // Draw the black dot at the specified location.
281         g2.fillOval(x, y, OVAL_RADIUS, OVAL_RADIUS);
282     }
283
284     /**
285      * Get the Icon for this JLabel
```

```
286      */
287     @Override
288     public Icon getIcon() {
289         return icon;
290     }
291 
292     /**
293      * Set the Icon for this JLabel
294      */
295     @Override
296     public void setIcon(Icon icon) {
297         this.icon = icon;
298     }
299 
300     /**
301      * If the value isn't a number then 0 is returned
302      * else the value is returned as an integer
303      *
304      * @return The int die value
305      */
306     public int getValue() {
307 
308         // Return the numeric value of the char representing the die value
309         int retVal = 0;
310         retVal = Character.digit(value, 10);
311         return (retVal == -1) ? 0 : retVal;
312     }
313 
314     /**
315      * Set the die to a specified value
316      *
317      * @param value
318      *            int value
319      */
320     public void setValue(int value) {
321 
322         // Cast the value to a char and set it
323         this.value = (char) value;
324 
325         // Repaint the die
326         this.repaint();
327     }
328 
329     /**
330      * Determine if the die is in a held state
331      *
332      * @return True if the die is held
333      */
334     public boolean isHeld() {
335         return (this.state == DieState.HELD);
336     }
337 
338     /**
339      * Determine if the die is in a scored state
340      *
341      * @return True if the die is scored
342      */
```

```
343     public boolean isScored() {
344         return (this.state == DieState.SCORED);
345     }
346 
347     /**
348      * Determine if the die is in an unheld state
349      *
350      * @return True if the die is unheld
351      */
352     public boolean isUnheld() {
353         return (this.state == DieState.UNHELD);
354     }
355 
356     /**
357      * Set the state of this die
358      *
359      * @param dieState
360      *            Desired state
361      */
362     public void setState(DieState dieState) {
363 
364         // Set the die state
365         this.state = dieState;
366 
367         // If the new die state is held, set the border to held
368         if (dieState == DieState.HELD) {
369             this.setBorder(HELD_BORDER);
370 
371             // Else, set the border to unheld
372         } else {
373             this.setBorder(UNHELD_BORDER);
374         }
375     }
376 
377     /**
378      * Get the state of this die
379      *
380      * @return The DieState of this die
381      */
382     public DieState getState() {
383         return this.state;
384     }
385 }
386 
```

```java
package com.lotsofun.farkle;

import java.awt.Component;
import java.awt.event.MouseEvent;

/**
 * Custom click event for the automated player
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public class DieClickEvent extends MouseEvent {

    /** Serialize this to comply with component conventions */
    private static final long serialVersionUID = 1L;

    /**
     * Constructor: builds a simulated mouse click
     *
     * @param source
     *            The clicked component
     * @param id
     *            Integer indicating the type of event
     * @param when
     *            Long that gives the time the event occurred
     * @param modifiers
     *            Keys down during the event
     * @param x
     *            Horizontal x mouse position
     * @param y
     *            Vertical y mouse position
     * @param clickCount
     *            The number of mouse clicks associated with the event
     * @param popupTrigger
     *            True if this event is a trigger for a popup menu
     */
    public DieClickEvent(Component source, int id, long when, int modifiers,
            int x, int y, int clickCount, boolean popupTrigger) {
        super(source, id, when, modifiers, x, y, clickCount, popupTrigger);
    }
}

```

DieState.java

```java
package com.lotsofun.farkle;

/**
 *
 * Enum for DieStates
 * UNHELD - Available to be rolled
 * HELD - Not
 * available to be rolled but eligible for scoring selection
 * SCORED - Not
 * available to be rolled or selected for scoring
 * DISABED - Prevents die
 * selection
 *
 * @author Curtis Brown
 * @version 3.0.0
 *
 */
public enum DieState {
    UNHELD, HELD, SCORED, DISABLED
}

```

```java
package com.lotsofun.farkle;

import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.List;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.JOptionPane;

/**
 * Controls the communication between the FarkleUI (the view) and Game (the
 * model) classes. Also serves as the entry point for the application.
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public class FarkleController implements ActionListener, MouseListener {

    /** The view of the application */
    FarkleUI farkleUI;

    /** The model of the application */
    Game farkleGame;

    /** The point threshold for two player mode */
    final int POINT_THRESHOLD = 10000;

    /** Flag to check if this the last turn */
    boolean isLastTurn = false;

    /** The farkle options dialog box */
    FarkleOptionsDialog farkleOptions = null;

    /** Used for testing the controller */
    boolean isTest = false;

    /** Random number generator for computer decisions */
    Random rand = new Random();

    /**
     * The entry point for the application
     *
     * @param args
     */
    public static void main(String[] args) {

        // Instantiate the Farkle Controller
        final FarkleController controller = new FarkleController(false);

        // Pass the controller to a new FarkleUI
        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
```

```
58                  new FarkleUI(controller);
59              }
60          });
61      }
62
63      /**
64       * Constructor: Instantiate the Farkle Controller
65       *
66       * @param isTest
67       *             Boolean true if testing
68       */
69      public FarkleController(boolean isTest) {
70          this.isTest = isTest;
71      }
72
73      /**
74       * Set a reference to the UI object
75       *
76       * @param farkle
77       *             the FarkleUI object to set
78       */
79      public void setUI(FarkleUI farkle) {
80          this.farkleUI = farkle;
81      }
82
83      /**
84       * Handle farkle by sending calls to both the Model and View
85       */
86      public void farkle() {
87
88          /*************************************************
89           * 1.2.7: If any roll results in 0 points, the word "Farkle" is
90           * prominently displayed, and 0 points is displayed in the accumulated
91           * turn score above the dice. The dice retain their current values that
92           * resulted in the Farkle. After the Farkle message is displayed, the
93           * dice still retain the values that resulted in the Farkle, but all
94           * dice are unlocked and play passes to the next player (in two player
95           * mode), or to the next turn (in one player mode).
96           *************************************************/
97          /*************************************************
98           * 4.9.0: If the player farkles on any roll during the current turn,
99           * that player loses all points accumulated during the current turn, the
100          * farkle message is displayed per requirement 1.2.7, and the turn is
101          * over.
102          *************************************************/
103         /*************************************************
104          * 4.10.0:When the turn is over all dice are unlocked while continuing
105          * to display their last value , the "Roll Dice" button is enabled,the
106          * "Bank Score" button is disabled, the "Select All" button is disabled,
107          * the current turn point total is set to 0, the current roll point
108          * total is set to 0, and play passes to the next player (two player
109          * mode) or the next turn (single player mode) by highlighting the
110          * appropriate player or turn on the left hand side of the screen.
111          *************************************************/
112         /*
113          * farkleUI: Disable the dice, set the running score and roll score to
114          * 0, disable the "Select All" button, disable the "Roll Dice" Button,
```

```
115         * show the FarkleMessage, and unhighlight all turn scores for the
116         * current player.
117         */
118        farkleUI.disableDice();
119        farkleUI.setRunningScore(0);
120        farkleUI.setRollScore(0);
121        farkleUI.getSelectAllBtn().setEnabled(false);
122        farkleUI.getRollBtn().setEnabled(false);
123        if (isTest == false) {
124            farkleUI.getFarkleMessage().setLocationRelativeTo(farkleUI);
125            farkleUI.getFarkleMessage().setVisible(true);
126        }
127        farkleUI.unHighlightAllTurnScores(farkleGame
128                .getPlayerNumberForCurrentPlayer());
129
130        /*
131         * Set the player's turn score and call
132         * the Game's farkle method
133         */
134        setTurnScore(farkleGame.getPlayerNumberForCurrentPlayer(),
135                farkleGame.getTurnNumberForCurrentPlayer(), 0);
136        farkleGame.farkle();
137
138        // Determine if the game has ended
139        tryToEndGame();
140
141        /*****************************************************
142         * 1.3.4 - Turn Highlighting
143         *****************************************************/
144        // Highlight the current turn for the new current player after the
145        // farkle
146        farkleUI.highlightTurnScore(
147                farkleGame.getPlayerNumberForCurrentPlayer(),
148                farkleGame.getTurnNumberForCurrentPlayer(), false);
149
150
151        // If the new current player is a computer player, set the running score
152        // to 0, and have the computer take its turn
153        if (farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.COMPUTER) {
154            farkleUI.setRunningScore(0);
155            asynchronousRoll();
156
157            // Else, enable the roll button for the new current player
158        } else {
159            farkleUI.getRollBtn().setEnabled(true);
160        }
161    }
162
163    /**
164     * Handle bank by sending calls to both the Model and the View
165     */
166    public void bank() {
167
168        // Retrieve the current player number from the farkleGame
169        int player = farkleGame.getPlayerNumberForCurrentPlayer();
170
171        /*****************************************************
```

```
172          * 4.8.0: If the player selects the bank button, the current turn point
173          * total is added to the player's game point total, and the turn is
174          * over.
175          ************************************************/
176         /************************************************
177          * 4.10.0:When the turn is over all dice are unlocked while continuing
178          * to display their last value , the "Roll Dice" button is enabled,the
179          * "Bank Score" button is disabled, the "Select All" button is disabled,
180          * the current turn point total is set to 0, the current roll point
181          * total is set to 0, and play passes to the next player (two player
182          * mode) or the next turn (single player mode) by highlighting the
183          * appropriate player or turn on the left hand side of the screen.
184          ************************************************/
185         /*
186          * farkleUI: Disable the dice, set the running score and roll score to
187          * 0, disable the "Select All" button, disable the "Bank" Button, set
188          * the updated game score, unhighlight all turn scores for the current
189          * player, and check the game score against the high score.
190          */
191         farkleUI.setRunningScore(0);
192         farkleUI.setRollScore(0);
193         farkleUI.disableDice();
194         farkleUI.unHighlightAllTurnScores(player);
195         farkleUI.setGameScore(player, farkleGame.bank());
196         farkleUI.getBankBtn().setEnabled(false);
197         farkleUI.getSelectAllBtn().setEnabled(false);
198
199         // If this is a single player game, check the high score
200         if (farkleGame.getGameMode() == GameMode.SINGLEPLAYER)
201             checkHighScore(player);
202
203         // Determine if the game has ended
204         tryToEndGame();
205
206         /************************************************
207          * 1.3.4 - Turn Highlighting
208          ************************************************/
209
210         // Highlight the current turn for the new current player after the bank
211         farkleUI.highlightTurnScore(
212                 farkleGame.getPlayerNumberForCurrentPlayer(),
213                 farkleGame.getTurnNumberForCurrentPlayer(), false);
214
215         // If this isn't the last turn, set the turn score for the new current
216         // player to 0
217         if (!isLastTurn) {
218             setTurnScore(farkleGame.getPlayerNumberForCurrentPlayer(),
219                     farkleGame.getTurnNumberForCurrentPlayer(), 0);
220         }
221
222         /*
223          * If the new current player is a computer player, set the running score
224          * to 0, disable the "Roll Dice" button, and have the computer take its
225          * turn
226          */
227         if (farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.COMPUTER) {
228             farkleUI.setRunningScore(0);
```

```
229             farkleUI.getRollBtn().setEnabled(false);
230             asynchronousRoll();
231         }
232
233         // Enable the "Roll Dice" button for the new current player
234         farkleUI.getRollBtn().setEnabled(true);
235     }
236
237     /**
238      * Checks given player's game score to see if it is greater than or equal to
239      * the saved high score. If it is, it saves the new high score and updates
240      * the FarkleUI with the new high score.
241      *
242      * @param playerNumber
243      *            int representing the player to check
244      * @return true if player's score is the new high score
245      */
246     public boolean checkHighScore(int playerNumber) {
247
248         // Check the player's game score against the saved high score
249         if (farkleGame.getGameScoreForPlayer(playerNumber) >= farkleGame
250                 .getHighScore()) {
251
252             // Save the new high score
253             farkleGame.setHighScore(farkleGame
254                     .getGameScoreForPlayer(playerNumber));
255
256             // Set the farkle UI to display the new high score
257             farkleUI.setHighScore(farkleGame.getHighScore());
258
259             return true;
260         }
261
262         // This score is not greater than or equal to the saved high score
263         return false;
264     }
265
266     /**
267      * Used to set the high score in the UI on init
268      *
269      * @param highScore
270      *            integer representing the high score
271      */
272     public void setUIHighScore(int highScore) {
273
274         // Set the high score displayed in the Farkle UI
275         if (null != farkleUI) {
276             farkleUI.setHighScore(highScore);
277         }
278     }
279
280     /**
281      * Resets the high score stored in the preferences to the default value (0)
282      */
283     public void resetHighScore() {
284
285         // Reset the high score in the model
```

```
286         farkleGame.resetHighScore();
287 
288         // Set the high score in the FarkleUI to the reset score
289         farkleUI.setHighScore(farkleGame.getHighScore());
290     }
291 
292     /**
293      * Determine if this is the last turn in single player mode or two player
294      * mode, or if the game has ended for single player mode or two player mode
295      */
296     public void tryToEndGame() {
297 
298         // Check if the game has ended in single player mode
299         if (farkleGame.getGameMode() == GameMode.SINGLEPLAYER) {
300 
301             // If the current turn is 10, this is the last turn
302             if (farkleGame.getTurnNumberForCurrentPlayer() == 10) {
303                 isLastTurn = true;
304             }
305 
306             /*************************************************
307              * 2.1.2 - The game ends at the conclusion of the tenth turn, and
308              * the player's score is compared to the current high score.
309              *************************************************/
310             // If the current turn is greater than 10, end the game
311             if (farkleGame.getTurnNumberForCurrentPlayer() > 10) {
312 
313                 /*************************************************
314                  * 2.1.3 - If the player's score is greater than the current
315                  * high score, a congratulatory message is displayed, and the
316                  * player's score replaces the previous high score.
317                  *************************************************/
318                 // If the player acheived a high score, display a message
319                 if (checkHighScore(1)) {
320                     farkleUI.displayMessage(
321                             "Congrats! You achieved a new high score.",
322                             "High Score");
323                 }
324 
325                 /*************************************************
326                  * 2.1.4 - The conclusion of game option box is displayed at the
327                  * completion of the tenth turn.
328                  *************************************************/
329                 // End the game.
330                 endGame(false, false, false);
331             }
332 
333             // Check if the game has ended in two player mode
334         } else if (farkleGame.getGameMode() == GameMode.MULTIPLAYER) {
335 
336             // Determine if either player has surpassed the threshold
337             int player1Score = farkleGame.getGameScoreForPlayer(1);
338             int player2Score = farkleGame.getGameScoreForPlayer(2);
339             if (player1Score >= POINT_THRESHOLD
340                     || player2Score >= POINT_THRESHOLD) {
341 
342                 /*************************************************
```

```
343                  * 1.4.4 - The first player to meet the minimum total point
344                  * threshold required to win the game (equal to 10,000 points)
345                  * shall be highlighted in a different color to indicate each
346                  * subsequent player has one more turn to try and beat that
347                  * playerâ€™s score.
348                  ***************************************************/
349                 /***************************************************
350                  * 2.2.2 - The first player to surpass 10,000 total points at
351                  * the end of a given turn is highlighted in a different color.
352                  ***************************************************/
353                 /***************************************************
354                  * 2.2.3 - The other player has one more turn to try and surpass
355                  * the point total of the first player to surpass 10,000 points.
356                  ***************************************************/
357                 /***************************************************
358                  * 2.3.3 - The first player to surpass 10,000 total points at
359                  * the end of a given turn is highlighted in a different color.
360                  ***************************************************/
361                 /***************************************************
362                  * 2.3.4 - The other player has one more turn to try and surpass
363                  * the point total of the first player to surpass 10,000 points.
364                  ***************************************************/
365                 /***************************************************
366                  * 1.4.6 - After a player has surpassed the 10,000 point
367                  * threshold, a message dialog box is displayed indicating the
368                  * other player has one last turn to try and beat that
369                  * playerâ€™s total score.
370                  ***************************************************/
371                 /*
372                  * If it is not the last turn, display a message that the other
373                  * player has one more turn. an set the isLastTurn flag
374                  */
375                 if (!isLastTurn) {
376                     int player = (player1Score >= POINT_THRESHOLD) ? 1 : 2;
377                     if (isTest == false) {
378                         farkleUI.displayMessage(
379                                 farkleGame.getPlayerName(player)
380                                         + " has scored "
381                                         + ""
382                                         + farkleGame
383                                                 .getGameScoreForPlayer(player)
384                                         + " points\n"
385                                         + "This is your last turn to try to beat them.\n"
386                                         + "Good Luck!", "Last Turn");
387                     }
388                     isLastTurn = true;
389 
390                     /***************************************************
391                      * 2.2.4 - The conclusion of game option box is displayed
392                      * after a player wins
393                      ***************************************************/
394                     /***************************************************
395                      * 2.3.5 - The conclusion of game option box is displayed
396                      * after a player wins
397                      ***************************************************/
398                     // Else, reset the last turn flag and end the game
399                 } else {
```

```
400                     isLastTurn = false;
401                     endGame(false, false, false);
402                 }
403             }
404         }
405     }
406
407     /**
408      * Unhighlights all player's turns, and resets the current game or starts a
409      * new game based on the passed parameters. If resetOnly and mainMenu are
410      * both false, displays the end of game dialog and gets the user selection
411      * to replay the game or go to the main menu. Otherwise, the game is
412      * replayed.
413      *
414      * @param resetOnly
415      *            Boolean true if the game is to be reset
416      * @param mainMenu
417      *            Boolean true if the main menu is to be displayed
418      * @param testReplayGame
419      *            Boolean true if this is a test of this method
420      */
421     public void endGame(boolean resetOnly, boolean mainMenu,
422             boolean testReplayGame) {
423
424         // Flag to determine if the current game is to be replayed
425         boolean replayGame = (isTest) ? testReplayGame : gameEnded(resetOnly,
426                 mainMenu);
427
428         // Unhighlight all player turns for player 1
429         farkleUI.unHighlightAllTurnScores(1);
430
431         // If two player mode, unhighlight player turns for player 2 and reset
432         // the turn score labels
433         if (farkleGame.getGameMode() == GameMode.MULTIPLAYER) {
434
435             farkleUI.unHighlightAllTurnScores(2);
436
437             // Rebuild the GUI to clear scpre labels for both players
438             farkleUI.getContentPane().removeAll();
439
440             // Clear the score labels and scores for both players
441             farkleUI.getPlayerScoreLabels(1).clear();
442             farkleUI.getPlayerScoreLabels(2).clear();
443             farkleUI.getPlayerScores(1).clear();
444             farkleUI.getPlayerScores(2).clear();
445
446             // Build the dice panel
447             farkleUI.buildDicePanel();
448
449             // Build the scoring guide panel
450             farkleUI.createScoreGuidePanel();
451
452             // Build the player panel for the selected game mode
453             farkleUI.buildPlayerPanel(farkleOptions.getGameMode());
454
455         }
456
```

```
457         // If replayGame is true, reset the current game
458         if (replayGame) {
459             replayGame();
460             farkleUI.diceFirstRun();
461 
462             // Else show the main menu
463         } else {
464             replayGame();
465             farkleUI.initUI();
466         }
467     }
468 
469     /**
470      * Displays the farkle options dialog, and creates a new game based on the
471      * chosen settings.
472      *
473      * @param options
474      *            FarkleOptionsDialog used for testing, set to null if not testing
475      */
476     public void newGame(FarkleOptionsDialog options) {
477 
478         // Set the window title
479         farkleUI.setTitle("Farkle");
480 
481         // Repaint the farkleUI to remove all panels before displaying options
482         farkleUI.repaint();
483 
484         // If options is null, display the farkle options dialog
485         if (null == options) {
486             // Get the game options
487             farkleOptions = new FarkleOptionsDialog(farkleUI);
488             farkleOptions.showWindow();
489 
490             // Else this is used for testing, set farkleOptions to the passed
491             // options
492         } else {
493             farkleOptions = options;
494         }
495 
496         /*****************************************************
497          * 1.2.0 - Items common to the user interface for both modes
498          *****************************************************/
499         // Build the dice panel
500         farkleUI.buildDicePanel();
501 
502         // Build the scoring guide panel
503         farkleUI.createScoreGuidePanel();
504 
505         /*****************************************************
506          * 2.0.0 - Game Modes
507          *****************************************************/
508         // Build the player panel for the selected game mode
509         farkleUI.buildPlayerPanel(farkleOptions.getGameMode());
510 
511         // Create the game object no matter what the mode is
512         if (null != farkleOptions.getGameMode()) {
513             farkleGame = new Game(farkleOptions.getGameMode(), this);
```

```
514         }
515
516         /**************************************************
517          * 1.1.2a - If the user selects the "Start" button with "1 Player Mode"
518          * highlighted and the "Player One Name" field empty, the one player GUI
519          * opens with the name "Jacob" assigned to player one.
520          **************************************************/
521         // Set the first player's name to "Jacob" if it was not provided
522         if (farkleOptions.getPlayer1Name().length() == 0) {
523             farkleUI.setPlayerName(1, "Jacob");
524             farkleGame.setPlayerName(1, "Jacob");
525
526             // Else, set the first player's name to the provided name
527         } else {
528
529             /**************************************************
530              * 1.1.2b - If the user selects the "Start" button with
531              * "1 Player Mode" highlighted and a name supplied in the
532              * "Player One Name" text field, the one player GUI opens with the
533              * provided name assigned to player one.
534              **************************************************/
535             farkleUI.setPlayerName(1, farkleOptions.getPlayer1Name());
536             farkleGame.setPlayerName(1, farkleOptions.getPlayer1Name());
537         }
538
539         // If it is a multiplayer game
540         if (farkleOptions.getGameMode() == GameMode.MULTIPLAYER) {
541
542             /**************************************************
543              * 1.4.1 - The title of the window shall display,
544              * "Farkle - Two Player Mode".
545              **************************************************/
546             farkleUI.setTitle("Farkle - Two Player Mode");
547
548             // If the opponent is a human player
549             if (farkleOptions.getPlayerType() == PlayerType.USER) {
550
551                 /**************************************************
552                  * 1.1.4.a - If "Two Player Mode" is highlighted,
553                  * "Human Opponent" is highlighted, the "Player One
554                  * Name" field is empty, the "Player Two Name" field is empty,
555                  * and the user selects the "Start" button, the two player mode
556                  * GUI is opened, and the names "Jacob" and "Brant" are assigned
557                  * to player 1 and player 2, respectively.
558                  **************************************************/
559                 /**************************************************
560                  * 1.1.4.c - If "Two Player Mode" is highlighted,
561                  * "Human Opponent" is highlighted, the "Player One
562                  * Name" field contains a name, the "Player Two Name" field is
563                  * empty, and the user selects the "Start" button, the two
564                  * player mode GUI is opened, the name "Brant" is assigned to
565                  * player 2, and the supplied name is assigned to player 1.
566                  **************************************************/
567                 // Set the player's name to "Brant" if it was not provided
568                 if (farkleOptions.getPlayer2Name().length() == 0) {
569                     farkleUI.setPlayerName(2, "Brant");
570                     farkleGame.setPlayerName(2, "Brant");
```

```
571
572                 /**************************************************
573                  * 1.1.4.b - If "Two Player Mode" is highlighted,
574                  * "Human Opponent" is highlighted, the "Player One
575                  * Name" field is empty, the "Player Two Name" field
576                  * contains a name, and the user selects the "Start" button,
577                  * the two player mode GUI is opened, the name "Jacob" is
578                  * assigned to player 1, and the supplied name is assigned
579                  * to player 2.
580                  **************************************************/
581                 /**************************************************
582                  * 1.1.4.d - If "Two Player Mode" is highlighted,
583                  * "Human Opponent" is highlighted, the "Player One
584                  * Name" field contains a name, the "Player Two Name" field
585                  * contains a name, and the user selects the "Start" button,
586                  * the two player mode GUI is opened, and the supplied names
587                  * are assigned to player 1 and player 2 accordingly.
588                  **************************************************/
589                 // Else set the player's name to the provided name
590             } else {
591                 farkleUI.setPlayerName(2, farkleOptions.getPlayer2Name());
592                 farkleGame.setPlayerName(2, farkleOptions.getPlayer2Name());
593             }
594         }
595
596         /**************************************************
597          * 1.1.5.a - If "Two Player Mode" is highlighted,
598          * "Computer Opponent" is highlighted, the "Player One Name" field
599          * is empty, and the user selects the "Start" button, the two player
600          * mode GUI is opened, the name "Jacob" is assigned to player 1, and
601          * "Computer" is assigned to player 2.
602          **************************************************/
603         /**************************************************
604          * 1.1.5.b - If "Two Player Mode" is highlighted,
605          * "Computer Opponent" is highlighted, the "Player One Name" field
606          * contains a name, and the user selects the "Start" button, the two
607          * player mode GUI is opened, the supplied name is assigned to
608          * player 1, and "Computer" is assigned to player 2.
609          **************************************************/
610         /*
611          * Else, the opponent is a computer player. Set the name and player
612          * type to Computer
613          */
614         else {
615             farkleUI.setPlayerName(2, "Computer");
616             farkleGame.setPlayerName(2, "Computer");
617             farkleGame.setPlayerType(2, PlayerType.COMPUTER);
618         }
619
620         // Else, it is a single player game.
621     } else {
622         /**************************************************
623          * 1.3.1 - The title of the window shall display:
624          * "Farkle - Single Player Mode".
625          **************************************************/
626         farkleUI.setTitle("Farkle - Single Player Mode");
627     }
```

```
628
629         /*****************************************************
630          * 2.1.0 - When one player mode is selected, the one player mode graphic
631          * user interface is displayed with the name "Farkle" displayed on the
632          * dice, the "Bank Score" button disabled, the "Select All" button
633          * disabled, and turn number one highlighted. The user will have ten
634          * turns to try and get as many points as possible.
635          *****************************************************/
636         /*****************************************************
637          * 2.2.0 - When two player mode against a live person is selected, the
638          * two player mode graphic user interface is displayed with the name
639          * "Farkle" displayed on the dice, the "Bank  Score" button disabled,
640          * the "Select All" button disabled, and player one highlighted
641          * indicating it is his or her turn.
642          *****************************************************/
643         /*****************************************************
644          * 2.3.0 - When two player mode against the computer is selected, the
645          * two player mode graphic user interface is displayed with the name
646          * "Farkle" displayed on the dice, the "Bank  Score" button disabled,
647          * the "Select All" button disabled, and player one highlighted
648          * indicating it is his turn.
649          *****************************************************/
650         // Reset the dice to display "Farkle"
651         farkleUI.diceFirstRun();
652
653         /*****************************************************
654          * 1.3.4 - Turn Highlighting
655          *****************************************************/
656         farkleUI.highlightTurnScore(
657                 farkleGame.getPlayerNumberForCurrentPlayer(),
658                 farkleGame.getTurnNumberForCurrentPlayer(), false);
659
660         // Pack the frame
661         farkleUI.pack();
662
663         // Discard the unneeded options dialog
664         farkleOptions.dispose();
665     }
666
667     /**
668      * Reset the game with the current configuration
669      */
670     public void replayGame() {
671
672         // Reset the model
673         farkleGame.resetGame();
674
675         // Set the UI turn score for player 1 to 0
676         setTurnScore(1, 1, 0);
677
678         // Set the UI roll score to 0
679         farkleUI.setRollScore(0);
680
681         // Reset the UI Dice Panel
682         farkleUI.resetDicePanel();
683
684         // Highlight the current turn for the current player
```

```
685         farkleUI.highlightTurnScore(
686                 farkleGame.getPlayerNumberForCurrentPlayer(),
687                 farkleGame.getTurnNumberForCurrentPlayer(), false);
688     }
689 
690     /**
691      * UI accessor method to simplify the process of updating a specific score
692      * label
693      *
694      * @param player
695      *            player who's turn score is to be set
696      * @param turn
697      *            the turn for which to set the score
698      * @param score
699      *            the score to set the turn to
700      */
701     public void setTurnScore(int player, int turn, int score) {
702         if (null != farkleUI) {
703             farkleUI.setTurnScore(player, turn, score);
704         }
705     }
706 
707     /**
708      * Call the appropriate method based on the button selected
709      */
710     @Override
711     public void actionPerformed(ActionEvent arg0) {
712 
713         /*****************************************************
714          * 2.1.1: Each turn is taken according to the requirements of section
715          * 4.0.0.
716          *****************************************************/
717         /*****************************************************
718          * 2.2.1 - Each turn is taken according to the requirements of section
719          * 4.0.0. The current player for each turn is highlighted during that
720          * player's turn.
721          *****************************************************/
722         /*****************************************************
723          * 2.3.1 - Each turn is taken according to the requirements of section
724          * 4.0.0. The current player for each turn is highlighted during that
725          * players turn.
726          *****************************************************/
727         /*****************************************************
728          * 4.1.0 - At the beginning of the turn, the turn score is set to 0. The
729          * player selects the "Roll Dice" button, and all 6 dice are rolled in
730          * accordance with requirement 3.2.0. The "Select All" button is enabled
731          * after the initial roll takes place.
732          *****************************************************/
733         // If Roll button clicked call the rollHandler()
734         if (arg0.getSource() == farkleUI.getRollBtn()) {
735             rollHandler();
736         }
737         // If Bank button clicked call the bankHandler()
738         else if (arg0.getSource() == farkleUI.getBankBtn()) {
739             bankHandler();
740         }
741         // If Select All button clicked, call the fakleUI selectAllDice()
```

```
742         else if (arg0.getSource() == farkleUI.getSelectAllBtn()) {
743             farkleUI.selectAllDice();
744         }
745     }
746 
747     /**
748      * Not used
749      */
750     @Override
751     public void mouseClicked(MouseEvent arg0) {
752         // Auto-generated method stub
753 
754     }
755 
756     /**
757      * Not used
758      */
759     @Override
760     public void mouseEntered(MouseEvent arg0) {
761         // Auto-generated method stub
762 
763     }
764 
765     /**
766      * Not used
767      */
768     @Override
769     public void mouseExited(MouseEvent arg0) {
770         // Auto-generated method stub
771 
772     }
773 
774     /**
775      * Determine the action to take when the player selects a die
776      */
777     @Override
778     public void mousePressed(MouseEvent arg0) {
779 
780         // Flag to determine if the user can select a die
781         boolean isClickIgnored;
782 
783         /*
784          * If the current player is PlayerType.COMPUTER, ignore all user clicks
785          * so the player cannot select dice during the computer's turn
786          */
787         if (farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.COMPUTER
788                 && (arg0 instanceof DieClickEvent) == false) {
789             isClickIgnored = true;
790 
791             // Else, this is a human player, do not ignore the click
792         } else {
793             isClickIgnored = false;
794         }
795 
796         // If the click is not ignored
797         if (isClickIgnored == false) {
798 
```

```
799             // Determine which die was clicked
800             Die d = (Die) arg0.getSource();
801 
802             /**************************************************
803              * 1.2.9 - During a current roll, current dice selected by the user
804              * shall be indicated with a yellow border around each selected die,
805              * and the score for the currently selected dice shall be updated
806              * above the dice.
807              **************************************************/
808             /*
809              * If the value of the die > 0, and it is not disabled, toggle its
810              * state
811              */
812             if (d.getValue() > 0 && d.getState() != DieState.DISABLED) {
813                 if (d.getState() == DieState.HELD) {
814                     d.setState(DieState.UNHELD);
815                 } else if (d.getState() == DieState.UNHELD) {
816                     d.setState(DieState.HELD);
817                 }
818 
819                 // If the die's state is not SCORED
820                 if (d.isScored() == false) {
821 
822                     // Get the value of the HELD dice
823                     int rollScore = farkleGame.calculateScore(
824                             farkleUI.getDieValues(DieState.HELD), true);
825 
826                     // Tell the model about it
827                     farkleGame.processHold(rollScore);
828                     farkleUI.setRollScore(rollScore);
829 
830                     // Get the running score from the model
831                     int runningScore = farkleGame.getRollScores();
832 
833                     // Update the UI based on the model's response
834                     if (runningScore > 0
835                             && farkleGame.getPlayerTypeForCurrentPlayer() !=
    PlayerType.COMPUTER) {
836                         farkleUI.getRollBtn().setEnabled(true);
837                     } else {
838                         farkleUI.getRollBtn().setEnabled(false);
839                     }
840 
841                     /**************************************************
842                      * 4.5.0: If the current turn point total is greater than or
843                      * equal to 300, the bank button is enabled.
844                      **************************************************/
845                     // Enable the bank button if the score is >= 300
846                     if (runningScore >= 300
847                             && farkleGame.getPlayerTypeForCurrentPlayer() !=
    PlayerType.COMPUTER) {
848                         farkleUI.getBankBtn().setEnabled(true);
849                     } else {
850                         farkleUI.getBankBtn().setEnabled(false);
851                     }
852 
853                     /**************************************************
```

```
854                      * 4.4.0: When all of the selected dice contribute to the
855                      * point total for the roll, the roll button is enabled and
856                      * the roll point total is added to the running point total
857                      * for the current turn.
858                      ************************************************/
859                     // Don't allow a user to roll with no scoring dice held
860                     if (rollScore > 0
861                             && farkleGame.getPlayerTypeForCurrentPlayer() !=
    PlayerType.COMPUTER) {
862                         farkleUI.getRollBtn().setEnabled(true);
863                     } else {
864                         farkleUI.getRollBtn().setEnabled(false);
865                         farkleUI.getBankBtn().setEnabled(false);
866                     }
867 
868                     /************************************************
869                      * 4.7.0 - If all six dice have been selected, and they all
870                      * contribute to the turns point total, the player is issued
871                      * a bonus roll indicated with a message in the current turn
872                      * box. All selected and locked dice are deselected and
873                      * unlocked, and the process returns to requirement 4.1.0.
874                      ************************************************/
875                     if ((farkleUI.getDice(DieState.HELD).size()
876                             + farkleUI.getDice(DieState.SCORED).size() == 6)
877                             && (rollScore > 0)) {
878                         farkleUI.disableDice();
879                         farkleUI.playBonusSound();
880 
881                         /************************************************
882                          * 1.3.4 - Turn Highlighting
883                          ************************************************/
884                         farkleUI.highlightTurnScore(
885                                 farkleGame.getPlayerNumberForCurrentPlayer(),
886                                 farkleGame.getTurnNumberForCurrentPlayer(),
887                                 true);
888                         farkleGame.setBonusTurn(true);
889                     }
890 
891                     // Update the running score label
892                     farkleUI.setRunningScore(runningScore);
893                 }
894             }
895         }
896     }
897 
898     /**
899      * Not used
900      */
901     @Override
902     public void mouseReleased(MouseEvent arg0) {
903         // Auto-generated method stub
904 
905     }
906 
907     /**
908      * If mainMenu is true, returns false indicating a new game is to be started
909      * and the main menu is to be displayed. If resetOnly is true and main menu
```

```
910      * is false, returns true indicating the game is to be reset with the
911      * current game configuration. Displays the end of game option box if both
912      * resetOnly and MainMenu are false, and returns the user's selection (false
913      * for main menu or true for reset game). If the user selects "Quit" from
914      * the end of game option box, the application closes.
915      *
916      * @param resetOnly
917      *            true if the game is to be reset
918      * @param mainMenu
919      *            true if a new game is to be started
920      * @return false indicating a new game should be started, or true indicating
921      *         the current game is to be reset.
922      */
923     public boolean gameEnded(boolean resetOnly, boolean mainMenu) {
924         boolean retVal = true;
925 
926         /*
927          * If resetOnly and mainMenu are both set to false, display the end of
928          * game dialog box, and return the user's selection.
929          */
930         if ((!resetOnly) && (!mainMenu)) {
931 
932             /*************************************************
933              * 1.5.0 - Conclusion of game option box
934              *************************************************/
935             /*************************************************
936              * 1.5.1 - At the conclusion of the game, an option box shall be
937              * displayed with the player's overall score for the completed game
938              * (in one player mode), or the winner of the current game (in two
939              * player mode). This option box shall include three options:
940              * "Play Again?", "Main Menu", and "Quit".
941              *************************************************/
942             String message = "";
943 
944             // Get the winner's information
945             String[] winnerStats = farkleGame.getWinningPlayerInfo();
946 
947             // If the game mode is single player, construct the appropriate
948             // message
949             if (farkleGame.getGameMode() == GameMode.SINGLEPLAYER) {
950                 message = "Total Score: " + farkleGame.getGameScoreForPlayer(1);
951 
952                 /*
953                  * Else if the game mode is two player and there is a distinct
954                  * winner construct the appropriate message.
955                  */
956             } else if (winnerStats.length > 0 && winnerStats.length < 3) {
957                 message = winnerStats[0] + " wins with a total " + "score of "
958                         + winnerStats[1] + "!";
959 
960                 /*
961                  * Else if the game mode is two player, and there was a tie,
962                  * construct the appropriate message.
963                  */
964             } else if (winnerStats.length == 3) {
965                 message = winnerStats[0] + " and " + winnerStats[1] + " "
966                         + "tied with a total score of " + winnerStats[2];
```

```
967
968             // Else, something went wrong. Construct the appropriate message
969         } else {
970             message = "A winner couldn't be determined.\nSomething is broken.";
971         }
972
973         // Display the JOptionPane, and get the user's response
974         Object[] options = { "Play Again", "Main Menu", "Exit" };
975         int userResponse = JOptionPane.showOptionDialog(farkleUI, message
976                 + "\nWhat would you like to do?", "Game Over",
977                 JOptionPane.YES_NO_CANCEL_OPTION,
978                 JOptionPane.QUESTION_MESSAGE, null, options, options[2]);
979
980         /*****************************************************
981          * 1.5.1.a - If the user selects the "Play Again?" button, the game
982          * will be restarted with all of the same configuration options of
983          * the previous game (player mode, playerâ€™s names, and player
984          * types).
985          *****************************************************/
986         // If the user selects "Play Again?" return true
987         if (userResponse == 0) {
988             retVal = true;
989         }
990
991         /*****************************************************
992          * 1.5.1.b - If the user selects the "Main Menu" button, the select
993          * game mode option box will be displayed.
994          *****************************************************/
995
996         // Else if the user selects "Main Menu", return false
997         else if (userResponse == 1) {
998             retVal = false;
999         }
1000
1001        /*****************************************************
1002         * 1.5.1.c - If the user selects the "Quit" button, the application
1003         * will immediately close.
1004         *****************************************************/
1005        /*****************************************************
1006         * 1.5.1.d - If the user selects the close button in the upper
1007         * portion of the conclusion of game option box, the application
1008         * will immediately close.
1009         *****************************************************/
1010        /*
1011         * Else, the user selected "Quit" or the close window button, close
1012         * the application
1013         */
1014        else {
1015            farkleUI.dispose();
1016        }
1017    }
1018
1019    // If main menu is true, return false indicating a new game is to be
1020    // started
1021    if (mainMenu) {
1022        retVal = false;
1023    }
```

```
1024
1025        // Disable the bank button
1026        farkleUI.getBankBtn().setEnabled(false);
1027
1028        // Reset the scores, running score, and game scores for all players
1029        farkleUI.setRunningScore(0);
1030        farkleUI.setGameScore(1, 0);
1031        farkleUI.resetScores(1);
1032        if (farkleGame.getGameMode() == GameMode.MULTIPLAYER) {
1033            farkleUI.resetScores(2);
1034            farkleUI.setGameScore(2, 0);
1035        }
1036
1037        // Return the selected option
1038        return retVal;
1039    }
1040
1041    /**
1042     * Checks if the player selected bank after receiving a bonus roll, and
1043     * displays a warning message if that is the case. If not, calls the bank
1044     * method and plays the bank sound. If the next player is a computer player,
1045     * disables the roll button so the user cannot select it while the computer
1046     * is taking its turn.
1047     */
1048    public void bankHandler() {
1049
1050        /*
1051         * If the player banked after receiving a bonus turn, ask the user if
1052         * they're sure they don't want to roll again
1053         */
1054        if (farkleGame.isBonusTurn()
1055                && farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.USER) {
1056            boolean useBonusRoll = farkleUI
1057                    .displayYesNoMessage(
1058                            "You used all your dice so you earned a bonus turn.\n"
1059                                    + "If you bank your score now, you'll lose your bonus
    turn.\n"
1060                                    + "Are you sure you want to bank now?",
1061                            "Warning!");
1062            if (useBonusRoll) {
1063                bank();
1064                farkleUI.playBankSound();
1065                farkleGame.setBonusTurn(false);
1066            }
1067
1068            // Else call the bank() method and play the bank sound
1069        } else {
1070            bank();
1071            farkleUI.playBankSound();
1072        }
1073
1074        // If the new current player is a compuer player, disable the roll
1075        // button
1076        if (farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.COMPUTER) {
1077            farkleUI.getRollBtn().setEnabled(false);
1078        }
1079    }
```

```
1080
1081    /**
1082     * Resets the farkleGame bonus turn flag, resets disabled dice, animates a
1083     * roll action, and checks for a Farkle. The action buttons are set
1084     * accordingly based on the roll outcome and player type.
1085     */
1086    public void rollHandler() {
1087
1088        /*****************************************************
1089         * 4.0.0 - Player's turn
1090         *****************************************************/
1091
1092        // If the previos roll was a bonus turn, set BonusTurn to false
1093        if (farkleGame.isBonusTurn()) {
1094            farkleGame.setBonusTurn(false);
1095        }
1096
1097        // If the dice are disabled (which occurs during a bank or farkle),
1098        // reset them
1099        if (farkleUI.getDieValues(DieState.DISABLED).size() > 0) {
1100            farkleUI.resetDice();
1101        }
1102
1103        /*****************************************************
1104         * 1.2.6 - After each roll, dice that have previously been selected,
1105         * scored, and locked shall be shaded to indicate they will not be
1106         * available on the next roll, and the turn accumulated score shall be
1107         * updated.
1108         *****************************************************/
1109        /*****************************************************
1110         * 4.6.0 - If the player elects to roll again the selected dice are
1111         * locked, the remaining dice are rolled, and the process returns to
1112         * requirement 4.2.0.
1113         *****************************************************/
1114        // Lock all scored dice
1115        farkleUI.lockScoredDice();
1116
1117        // Create a timer to animate the roll
1118        Timer rollAnimationTimer = new Timer();
1119        rollAnimationTimer.scheduleAtFixedRate(new TimerTask() {
1120
1121            // The variable used to count the number of times the dice have
1122            // rolled
1123            int count = 0;
1124
1125            @Override
1126            public void run() {
1127
1128                // Roll the dice ten times
1129                farkleUI.rollDice();
1130                count++;
1131
1132                // After the tenth roll, process it
1133                if (count >= 10) {
1134
1135                    // Tell the model this is a completed roll
1136                    farkleGame.processRoll();
```

```
1137
1138                    /**************************************************
1139                     * 4.2.0: The resulting roll is analyzed according to
1140                     * requirement 6.0.0 to determine if the player farkled. A
1141                     * farkle occurs if the roll results in 0 points.
1142                     **************************************************/
1143                    // Score any UNHELD dice
1144                    int rollScore = farkleGame.calculateScore(
1145                            farkleUI.getDieValues(DieState.UNHELD), false);
1146
1147                    // If it's farkle
1148                    if (rollScore == 0) {
1149
1150                        // Call the farkle method to update the model and view
1151                        farkle();
1152
1153                        // Disable the bank button
1154                        farkleUI.getBankBtn().setEnabled(false);
1155
1156                        // If the next player is not a computer player, enable
1157                        // the roll button
1158                        if (farkleGame.getPlayerTypeForCurrentPlayer() !=
    PlayerType.COMPUTER) {
1159                            farkleUI.getRollBtn().setEnabled(true);
1160                        }
1161
1162                        // Else, this roll did not result in a farkle
1163                    } else {
1164
1165                        // If the player is a computer player, determine the
1166                        // computer's decision
1167                        if (farkleGame.getPlayerTypeForCurrentPlayer() ==
    PlayerType.COMPUTER) {
1168                            compDecision();
1169                        }
1170                    }
1171
1172                    // Cancel this timer task
1173                    this.cancel();
1174                }
1175            }
1176
1177        }, 0, 50);
1178
1179        // Play roll sound
1180        farkleUI.playRollSound();
1181
1182        /**************************************************
1183         * 4.3.0 - If the player did not farkle, he or she must select at least
1184         * one scoring die. The score for the selected dice is calculated
1185         * according to requirement 5.0.0, and is updated after each die
1186         * selection. The score for the selected dice is displayed in accordance
1187         * with section 1.2.9. If any of the selected dice does not contribute
1188         * to the score, a selected dice score of 0 is displayed and the
1189         * "Roll Dice" and "Bank Score" buttons are disabled.
1190         **************************************************/
1191        // Disable the Roll and Bank buttons
```

```
1192        farkleUI.getRollBtn().setEnabled(false);
1193        farkleUI.getBankBtn().setEnabled(false);
1194
1195        // Enable Select All Button if this is not the computer player
1196        if (farkleGame.getPlayerTypeForCurrentPlayer() != PlayerType.COMPUTER) {
1197            farkleUI.getSelectAllBtn().setEnabled(true);
1198        }
1199
1200        /*************************************************
1201         * 1.3.4 - Turn Highlighting
1202         *************************************************/
1203        farkleUI.highlightTurnScore(
1204                farkleGame.getPlayerNumberForCurrentPlayer(),
1205                farkleGame.getTurnNumberForCurrentPlayer(), false);
1206
1207        // Set the roll score to 0
1208        farkleUI.setRollScore(0);
1209
1210    }
1211
1212    /**
1213     * Causes the thread to sleep for a split second giving the illusion of
1214     * playing against a live player. Calculates and selects the highest scoring
1215     * combination of dice for a given roll. Calculates a pseudo random target
1216     * for the computer to achieve during a given roll, and banks the turn if
1217     * that target is achieved on a roll that was not awarded a bonus turn.
1218     * Otherwise, the computer rolls again and calculates a new target for the
1219     * turn.
1220     */
1221    public void compDecision() {
1222
1223        // Delay the thread to give the illusion of playing against a live
1224        // player
1225        try {
1226            Thread.sleep(750);
1227        } catch (InterruptedException e) {
1228            e.printStackTrace();
1229        }
1230
1231        /*************************************************
1232         * 2.3.2 - Decisions made during the computer player's turn are chosen
1233         * in accordance with requirements section 5.0.0.
1234         *************************************************/
1235        /*************************************************
1236         * 5.0.0 - Computer Player
1237         *************************************************/
1238        /*************************************************
1239         * 5.1.0 - The computer player takes its turn in accordance with
1240         * requirement 4.0.0, and the dice selection, as well as the decision to
1241         * continue rolling the dice, are made in accordance with the following
1242         * requirements.
1243         *************************************************/
1244        // The goal the computer is trying to achieve
1245        int goal = 0;
1246
1247        // Determine the highest scoring dice that can be selected
1248        List<Integer> highestScoringDieValues = getHighestScoringDieValues();
```

```
1249
1250        /*****************************************************
1251         * 5.2.0 - After each roll, the computer player always selects the
1252         * maximum scoring combination of dice.
1253         *****************************************************/
1254        // Select the highest scoring combination of dice
1255        try {
1256            farkleUI.selectDice(highestScoringDieValues);
1257        } catch (InterruptedException e) {
1258            e.printStackTrace();
1259        }
1260
1261        /*****************************************************
1262         * 5.5.0 - The computer's goal is calculated after each roll. This
1263         * goal is pseudo-randomly selected as 300 fifty percent of the time,
1264         * 600 thirty percent of the time, and 1000 twenty percent of the time.
1265         *****************************************************/
1266        /*
1267         * Randomly choose a goal score for this roll.
1268         */
1269        int n = rand.nextInt(10) + 1;
1270        if (n <= 5) {
1271            goal = 300;
1272        } else if (n <= 8) {
1273            goal = 600;
1274        } else {
1275            goal = 1000;
1276        }
1277
1278        /*****************************************************
1279         * 5.3.0 - If the current turn point total is less than the goal
1280         * calculated in section 5.5.0, the computer always rolls again.
1281         *****************************************************/
1282        /*****************************************************
1283         * 5.4.0 If the previous roll resulted in a bonus roll, the computer
1284         * always rolls again.
1285         *****************************************************/
1286        /*
1287         * If the current turn score is less than the goal, or the computer
1288         * achieved a bonus turn, the computer should always roll again.
1289         */
1290        if (farkleUI.getRunningScore() < goal || farkleGame.isBonusTurn()) {
1291            rollHandler();
1292
1293            // Else, the computer banks the current turn.
1294        } else {
1295            bankHandler();
1296        }
1297    }
1298
1299    /**
1300     * Returns true if no dice are currently held and the dice have been rolled.
1301     * Used to enable/disable the hint option in the file menu.
1302     *
1303     * @return boolean true if a hint is available
1304     */
1305    public boolean isHintAvailable() {
```

```
1306
1307        // If no dice are held determine if the dice have been rolled
1308        if (farkleUI.getDice(DieState.HELD).size() == 0) {
1309
1310            // If the dice have been rolled, return true
1311            for (Die d : farkleUI.getDice(DieState.UNHELD)) {
1312                if (d.getValue() == 0) {
1313
1314                    // The dice have not been rolled, return false
1315                    return false;
1316                }
1317            }
1318            return true;
1319        }
1320
1321        // At least one die is held, return false
1322        return false;
1323    }
1324
1325    /**
1326     * Since Swing and, consequently, our design isn't thread-safe, don't allow
1327     * the game to be reset or a new game to be created while the automated
1328     * player is taking its turn.
1329     *
1330     * @return boolean true if the computer is not taking its turn
1331     */
1332    public boolean isResetOrNewGameAvailable() {
1333
1334        // If the current player is the computer, return false
1335        if (farkleGame.getGameMode() == GameMode.MULTIPLAYER
1336                && farkleGame.getPlayerTypeForCurrentPlayer() == PlayerType.COMPUTER) {
1337            return false;
1338
1339            // Else, return true
1340        } else {
1341            return true;
1342        }
1343    }
1344
1345    /**
1346     * Returns true if the game mode is GameMode.SINGLEPLAYER.
1347     * Used to enable/disable the reset high score option in the file menu.
1348     *
1349     * @return boolean true if a reset high score is available
1350     */
1351    public boolean isResetHighScoreAvailable() {
1352        // If the game mode is GameMode.MULTIPLAYER return false
1353        if (farkleGame.getGameMode() == GameMode.MULTIPLAYER) {
1354            return false;
1355
1356            // Else, return true
1357        } else {
1358            return true;
1359        }
1360    }
1361
1362    /**
```

```
1363     * Get the highest possible score of the rolled dice
1364     *
1365     * @return int score
1366     */
1367    public int getHighestPossibleScore() {
1368
1369        // Determine the highest possible scoring set of dice
1370        Object[] results = farkleGame.calculateHighestScore(farkleUI
1371                .getDieValues(DieState.UNHELD));
1372        try {
1373
1374            // Return the score for that set of dice
1375            return (Integer) results[0];
1376        } catch (ClassCastException e) {
1377            throw e;
1378        }
1379    }
1380
1381    /**
1382     * Get a list of the dice to select to achieve the highest possible score
1383     * for a given roll of the dice
1384     *
1385     * @return List of Die objects
1386     */
1387    @SuppressWarnings("unchecked")
1388    public List<Integer> getHighestScoringDieValues() {
1389
1390        // Determine the highest possible scoring set of dice
1391        Object[] results = farkleGame.calculateHighestScore(farkleUI
1392                .getDieValues(DieState.UNHELD));
1393        try {
1394            // Return the list of dice to select for highest possible score
1395            return (List<Integer>) results[1];
1396        } catch (ClassCastException e) {
1397            throw e;
1398        }
1399    }
1400
1401    /**
1402     * Roll the dice in their own thread so the call can sleep without blocking
1403     * Swing's Event Dispatch Thread. Used for animating the computer's turn.
1404     */
1405    public void asynchronousRoll() {
1406        Thread t = new Thread(new Runnable() {
1407            @Override
1408            public void run() {
1409                try {
1410                    Thread.sleep(1000);
1411                    rollHandler();
1412                } catch (InterruptedException e) {
1413                    e.printStackTrace();
1414                }
1415            }
1416        });
1417
1418        t.start();
1419    }
```

```
1420 }
1421
```

```java
package com.lotsofun.farkle;

import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Image;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.math.BigDecimal;
import java.net.URL;
import javax.imageio.ImageIO;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;
import javax.swing.ImageIcon;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.Timer;

/**
 * Displays a Farkle message on the screen, and plays the Farkle sound
 *
 * @author Curtis Brown
 * @version 3.0.0
 *
 */
public class FarkleMessage extends JDialog {

    /** Serialize this to comply with component conventions */
    private static final long serialVersionUID = 1L;

    /** The center Farkle Message JLabel */
    JLabel farkleCenterMsg = null;

    /** The left Farkle Message JLabel */
    JLabel farkleLeftMsg = null;

    /** The right Farkle Message JLabel */
    JLabel farkleRightMsg = null;

    /** The location of the farkle sound */
    URL farkleSound = null;

    /**
     * Constructor: Build the Farkle Message JDialog
     */
    public FarkleMessage() {

        // Set the modality to display this in front of all windows
        this.setModalityType(ModalityType.APPLICATION_MODAL);

        // Instantiate the image objects for the Farkle Message
        Image farkleCenterImg = null;
        Image farkleLeftImg = null;
```

```
58          Image farkleRightImg = null;
59
60          // Set the preferred size, decoration, and background color
61          this.setPreferredSize(new Dimension(1024, 768));
62          this.setUndecorated(true);
63          this.setBackground(new Color(0, 0, 0, 0));
64
65          // Set the Farkle Images and sound
66          try {
67              farkleCenterImg = ImageIO.read(getClass().getResource(
68                      "/images/FarkleCenter.png"));
69              farkleLeftImg = ImageIO.read(getClass().getResource(
70                      "/images/FarkleLeft.png"));
71              farkleRightImg = ImageIO.read(getClass().getResource(
72                      "/images/FarkleRight.png"));
73
74              farkleSound = getClass().getResource("/sounds/farkle.wav");
75
76          } catch (IOException e) {
77              e.printStackTrace();
78          }
79
80          // Set the JLabels with the Farkle images
81          farkleCenterMsg = new JLabel(new ImageIcon(
82                  farkleCenterImg
83                          .getScaledInstance(1000, 200, Image.SCALE_SMOOTH)));
84          farkleRightMsg = new JLabel(
85                  new ImageIcon(farkleRightImg.getScaledInstance(1000, 755,
86                          Image.SCALE_SMOOTH)));
87          farkleLeftMsg = new JLabel(new ImageIcon(
88                  farkleLeftImg.getScaledInstance(1000, 755, Image.SCALE_SMOOTH)));
89
90          // Add the JLabels to this JDialog
91          this.add(farkleCenterMsg);
92          this.add(farkleRightMsg);
93          this.add(farkleLeftMsg);
94
95          // Size the window to fit the preferred size and layout of its
96          // subcomponents
97          this.pack();
98
99      }
100
101     /**
102      * Animates the Farkle Message, and plays the farkle sound.
103      *
104      * @param visible
105      *            Boolean - display the Farkle Message
106      */
107     @Override
108     public void setVisible(boolean visible) {
109
110         // If visible is true, display the Farkle message
111         if (visible) {
112
113             // Timer used to animate the message
114             final Timer fadeTimer = new Timer(50, null);
```

```
115
116             // ActionListener for the fade timer
117             ActionListener listener = new ActionListener() {
118
119                 // Set the opacity to 1
120                 BigDecimal opacity = BigDecimal.valueOf(1);
121
122                 // Number of steps in the animation of the Farkle message
123                 int step = 4;
124
125                 /**
126                  * Animate the message in the actionPerformed method
127                  */
128                 @Override
129                 public void actionPerformed(ActionEvent e) {
130
131                     // Start playing the Farkle sound when the JDialog is first
132                     // diaplayed
133                     if (step == 4) {
134                         playFarkleSound();
135                         step--;
136                     }
137
138                     /*
139                      * If step = 0, the animation is complete, reset to farkle
140                      * left message, set step back to 4, and hide this JDialog
141                      */
142                     if (step == 0) {
143                         fadeTimer.stop();
144                         FarkleMessage.this.setVisible(false);
145                         FarkleMessage.this.getContentPane().add(farkleLeftMsg);
146                         step = 4;
147                     }
148
149                     // If the opacity != 0, keep fading
150                     if (opacity.compareTo(BigDecimal.valueOf(0)) > 0) {
151                         fade();
152
153                         // If the opacity <= 0, change the
154                         // JLabel and decrement step
155                     } else if (opacity.compareTo(BigDecimal.valueOf(0)) <= 0) {
156                         switch (step) {
157                         case 3:
158                             remove(farkleLeftMsg);
159                             break;
160                         case 2:
161                             remove(farkleRightMsg);
162                             break;
163                         case 1:
164                             remove(farkleCenterMsg);
165                             break;
166                         }
167                         step--;
168                     }
169                 }
170
171                 /**
```

```
172                 * Decrement the opacity, and set the JDialog to the new opacity
173                 */
174                public void fade() {
175                    if (step == 1) {
176                        opacity = opacity.subtract(BigDecimal.valueOf(0.05));
177                    } else {
178                        opacity = opacity.subtract(BigDecimal.valueOf(0.1));
179                    }
180                    FarkleMessage.this.setOpacity(opacity.floatValue());
181                }
182
183                /**
184                 * Remove the provided JLabel, add a different JLabel, and set
185                 * opacity to 1.
186                 *
187                 * @param c
188                 *            the JLabel to remove
189                 */
190                public void remove(Component c) {
191
192                    // Remove the provided JLabel
193                    FarkleMessage.this.remove(c);
194
195                    // If the message has not finished playing, set the opacity
196                    // to 1
197                    if (step > 0) {
198                        opacity = BigDecimal.valueOf(1.00);
199                    }
200
201                    // Add the next JLabel in the sequence based in the removed
202                    // JLabel
203                    if (c.equals(farkleLeftMsg)) {
204                        FarkleMessage.this.getContentPane().add(farkleRightMsg);
205                    } else if (c.equals(farkleRightMsg)) {
206                        FarkleMessage.this.getContentPane()
207                                .add(farkleCenterMsg);
208                    }
209
210                    // Size the window to fit the preferred size and layout of
211                    // its subcomponents
212                    FarkleMessage.this.pack();
213
214                    // Play the farkle sound again
215                    if (step > 1) {
216                        playFarkleSound();
217                    }
218                }
219            };
220
221            // Add the action listener to the fade timer and start it
222            fadeTimer.addActionListener(listener);
223            fadeTimer.start();
224
225            // Show this JDialog
226            super.setVisible(true);
227        } else {
228
```

```
229             // Hide this JDialog
230             super.setVisible(false);
231         }
232     }
233 
234     /**
235      * Play the Farkle sound
236      */
237     public void playFarkleSound() {
238 
239         // Use AudioStream and Clip to play the Farkle Sound
240         try {
241             AudioInputStream audioStream;
242             audioStream = AudioSystem.getAudioInputStream(farkleSound);
243             Clip clip = AudioSystem.getClip();
244             clip.open(audioStream);
245             clip.start();
246         } catch (UnsupportedAudioFileException x) {
247             x.printStackTrace();
248         } catch (LineUnavailableException y) {
249             y.printStackTrace();
250         } catch (IOException z) {
251             z.printStackTrace();
252         }
253     }
254 }
255 
```

```java
package com.lotsofun.farkle;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

/**
 * Displays a dialog box used to select game mode options, including single
 * player or two player, human opponent or computer opponent, and setting
 * appropriate names for each player. <br />
 * <br />
 * Fulfills requirement 1.0.0:<br />
 * <br />
 * "Select Game Mode Option Box - Upon opening the application, the user is
 * greeted with an option box that includes all configuration options for
 * gameplay. These options include "1 Player Mode", "2 Player Mode", "Human
 * Opponent" (if two player mode is selected), "Computer Opponent" (if two
 * player mode is selected), and text fields to enter the associated player
 * names. Also included is a "Start" button and a "Close" button (both of which
 * are always enabled). This option dialog box should pop up over the main GUI
 * set to a solid green color."
 *
 * @author Jacob Davidson
 * @version 3.0.0
 */
public class FarkleOptionsDialog extends JDialog implements MouseListener {

    /** Serialize this to comply with component conventions */
    private static final long serialVersionUID = 1L;

    /** The primary background color */
    private Color greenBackground = new Color(35, 119, 34);

    /** The label for the player mode selection */
    private JLabel playerModeSelectLabel = new JLabel(" Select Player Mode:");

    /** The single player mode selection label */
    private JLabel singlePlayerLabel = new JLabel("One Player Mode",
            SwingConstants.CENTER);

    /** The two player mode selection label */
    private JLabel multiplayerLabel = new JLabel("Two Player Mode",
            SwingConstants.CENTER);
```

```
58
59      /** The label for the opponent player type selection */
60      private JLabel playerTypeSelectLabel = new JLabel(" Select Opponent Type:");
61
62      /** The human opponent player type selection label */
63      private JLabel humanPlayerLabel = new JLabel("Human Opponent",
64              SwingConstants.CENTER);
65
66      /** The computer opponent player type selection label */
67      private JLabel computerPlayerLabel = new JLabel("Computer Opponent",
68              SwingConstants.CENTER);
69
70      /** The label for the title of the game mode options menu */
71      private JLabel gameModeOptionTitle = new JLabel("Game Mode Options",
72              SwingConstants.CENTER);
73
74      /** The title of the enter player names portion of the JDialog */
75      private JLabel playerNamesLabel = new JLabel("Enter Player Names:");
76
77      /** The label for entering player onne's name */
78      private JLabel playerOneNameLabel = new JLabel("Player One:");
79
80      /** The label for entering player two's name */
81      private JLabel playerTwoNameLabel = new JLabel("Player Two:");
82
83      /** The panel for entering player one's name */
84      private JPanel playerOneNamePanel = new JPanel();
85
86      /** The panel for entering player two's name */
87      private JPanel playerTwoNamePanel = new JPanel();
88
89      /** The panel for player mode selection */
90      private JPanel playerModeSelectionPanel = new JPanel();
91
92      /** The panel for player type selection */
93      private JPanel playerTypeSelectionPanel = new JPanel();
94
95      /** The text field for entering player one's name */
96      private JTextField playerOneName = new JTextField(5);
97
98      /** The text field for entering player two's name */
99      private JTextField playerTwoName = new JTextField(5);
100
101     /** The "Start" button */
102     private JButton startButton = new JButton("Start");
103
104     /** The "Close" button */
105     private JButton closeButton = new JButton("Close");
106
107     /** The provided name for player 1 */
108     private String player1Name = "";
109
110     /** The provided name for player 2 */
111     private String player2Name = "";
112
113     /** The selected game mode */
114     private GameMode gameMode = GameMode.SINGLEPLAYER;
```

```
115
116     /** The selected opponent player type */
117     private PlayerType playerType = PlayerType.USER;
118
119     /**
120      * The Farkle Options Dialog box constructor
121      *
122      * @param frame
123      *            The parent JFrame for this dialog
124      */
125     public FarkleOptionsDialog(JFrame frame) {
126         // Pass the JFrame to the super constructor
127         super(frame);
128
129         // Set the modality to block all top level windows
130         this.setModalityType(ModalityType.APPLICATION_MODAL);
131
132         // Set the layout of the JDialog box
133         BorderLayout layout = new BorderLayout();
134         this.setLayout(layout);
135
136         // Set the size and decoration properties of the JDialog box
137         this.setResizable(false);
138         this.setUndecorated(true);
139         this.setPreferredSize(new Dimension(750, 250));
140         this.setBackground(greenBackground);
141
142         // window is the main JPanel used to create a border
143         JPanel window = new JPanel(new BorderLayout());
144         window.setBorder(BorderFactory.createCompoundBorder(
145                 BorderFactory.createLineBorder(Color.YELLOW, 3),
146                 BorderFactory.createLineBorder(greenBackground, 5)));
147
148         // The gameModePanel is the center of the window panel, and will house
149         // the
150         // game mode selection options, opponent type selection options, and
151         // player
152         // name text fields
153         JPanel gameModePanel = new JPanel();
154         gameModePanel.setLayout(new GridLayout(1, 2, 10, 0));
155         gameModePanel
156                 .setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
157         gameModePanel.setBackground(greenBackground);
158
159         /*************************************************
160          * 1.1.2 - The "1 Player Mode" is highlighted by default when the
161          * application is first opened, and a blank text field for player one's
162          * name is displayed.
163          *************************************************/
164         // Create the single player option and highlight it as the default
165         singlePlayerLabel.setName("1");
166         singlePlayerLabel.setOpaque(true);
167         singlePlayerLabel.setBackground(Color.YELLOW);
168         singlePlayerLabel.addMouseListener(this);
169
170         // Create the two player option
171         multiplayerLabel.setName("2");
```

```
172         multiplayerLabel.setOpaque(true);
173         multiplayerLabel.addMouseListener(this);
174         multiplayerLabel.setBackground(greenBackground);
175
176         // The playerTypePanel is the panel beneath the player mode options, and
177         // is only displayed
178         // if 2 player mode is selected.
179         JPanel playerTypePanel = new JPanel();
180         playerTypePanel.setLayout(new GridLayout(2, 1, 0, 5));
181         playerTypePanel.setBackground(greenBackground);
182
183         // The playerModeSelectionPanel contains the player modes
184         playerModeSelectionPanel.setLayout(new GridLayout(3, 1, 0, 5));
185         playerModeSelectionPanel.setBackground(greenBackground);
186
187         // The playerTypeSelectionPanel contains the player types
188         playerTypeSelectionPanel.setLayout(new GridLayout(3, 1, 0, 5));
189         playerTypeSelectionPanel.setBackground(greenBackground);
190
191         // Create the human player type option
192         humanPlayerLabel.setName("3");
193         humanPlayerLabel.setOpaque(true);
194         humanPlayerLabel.addMouseListener(this);
195         humanPlayerLabel.setBackground(greenBackground);
196
197         // Create the computer player type option
198         computerPlayerLabel.setName("4");
199         computerPlayerLabel.setOpaque(true);
200         computerPlayerLabel.addMouseListener(this);
201         computerPlayerLabel.setBackground(greenBackground);
202
203         // Set the font and colors for the playerModeSelectLabel and add to
204         // the playerModeSelectionPanel
205         playerModeSelectLabel.setFont(new Font("Arial Black", Font.PLAIN, 14));
206         playerModeSelectLabel.setForeground(Color.WHITE);
207         playerModeSelectionPanel.add(playerModeSelectLabel);
208
209         // Set the font for the singlePlayerLabel and add to the
210         // playerModeSelectionPanel
211         singlePlayerLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
212         playerModeSelectionPanel.add(singlePlayerLabel);
213
214         // Set the font and color for the multiPlayerLabel and add to the
215         // playerModeSelectionPanel
216         multiplayerLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
217         multiplayerLabel.setForeground(Color.WHITE);
218         playerModeSelectionPanel.add(multiplayerLabel);
219
220         // Set the font and color for the playerTypeSelectLabel and add to the
221         // playerTypeSelectionPanel
222         playerTypeSelectLabel.setFont(new Font("Arial Black", Font.PLAIN, 14));
223         playerTypeSelectLabel.setForeground(Color.WHITE);
224         playerTypeSelectionPanel.add(playerTypeSelectLabel);
225
226         // Set the font for the humanPlayerLabel and add to the
227         // playerTypeSelectionPanel
228         humanPlayerLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
```

```
229         playerTypeSelectionPanel.add(humanPlayerLabel);
230
231         // Set the font and color for the computerPlayerLabel and add to the
232         // playerTypeSelectionPanel
233         computerPlayerLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
234         computerPlayerLabel.setForeground(Color.WHITE);
235         playerTypeSelectionPanel.add(computerPlayerLabel);
236
237         // Add the playerModeSelectionPanel and the playerTypeSelectionPanel to
238         // the playerTypePanel. The playerTypeSelectionPanel will initially be
239         // hidden
240         playerTypePanel.add(playerModeSelectionPanel);
241         playerTypePanel.add(playerTypeSelectionPanel);
242         playerTypeSelectionPanel.setVisible(false);
243
244         // Create the playerNamesPanel that will house the player name text
245         // boxes on the right side of the JDialog box
246         JPanel playerNamesPanel = new JPanel();
247         playerNamesPanel.setLayout(new GridLayout(6, 1, 0, 5));
248         playerNamesPanel.setBackground(greenBackground);
249
250         // Set the font and color for the playerNames label and add it to the
251         // playerNamesPanel
252         playerNamesLabel.setFont(new Font("Arial Black", Font.PLAIN, 14));
253         playerNamesLabel.setForeground(Color.WHITE);
254         playerNamesPanel.add(playerNamesLabel);
255
256         // Create the playerOneNamePanel that will house the player one name
257         // text box
258         playerOneNamePanel.setLayout(new BorderLayout(10, 0));
259         playerOneNamePanel.setBackground(greenBackground);
260
261         // Set the font and color for the playerOneNameLabel, and add it to the
262         // playerOneNamePanel
263         playerOneNameLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
264         playerOneNameLabel.setForeground(Color.WHITE);
265         playerOneNamePanel.add(playerOneNameLabel, BorderLayout.WEST);
266
267         // Add the playerOneName text box to the playerOneNamePanel
268         playerOneNamePanel.add(playerOneName, BorderLayout.CENTER);
269
270         // Add the playerOneName panel to the playerNamesPanel
271         playerNamesPanel.add(playerOneNamePanel);
272
273         // Create the playerTwoNamePanel that will house the player two name
274         // text box
275         playerTwoNamePanel.setLayout(new BorderLayout(8, 0));
276         playerTwoNamePanel.setBackground(greenBackground);
277
278         // Set the font and color for the playerTwoNameLabel, and add it to the
279         // playerTwoNamePanel
280         playerTwoNameLabel.setFont(new Font("Arial Black", Font.PLAIN, 12));
281         playerTwoNameLabel.setForeground(Color.WHITE);
282         playerTwoNamePanel.add(playerTwoNameLabel, BorderLayout.WEST);
283
284         // Add the playerTwoName text box to the playerTwoNamePanel
285         playerTwoNamePanel.add(playerTwoName, BorderLayout.CENTER);
```

```
286
287         // Add the playerTwoNamePanel to the playerNamesPanel
288         playerNamesPanel.add(playerTwoNamePanel);
289
290         // The playerTwoNamePanel will initially be hidden
291         playerTwoNamePanel.setVisible(false);
292
293         // Add the playerTypePanel and playerNamesPanel to the gameModePanel
294         gameModePanel.add(playerTypePanel);
295         gameModePanel.add(playerNamesPanel);
296
297         // Create the buttonPanel that will house the "start" and "close"
298         // buttons.
299         JPanel buttonPanel = new JPanel();
300         buttonPanel.setBackground(Color.WHITE);
301
302         // Create the "Start" button, and add the actionListener
303         startButton.addActionListener(new ActionListener() {
304             @Override
305             public void actionPerformed(ActionEvent e) {
306
307                 // Set the provided player names
308                 setPlayer1Name(playerOneName.getText());
309                 setPlayer2Name(playerTwoName.getText());
310
311                 // Hide this panel
312                 FarkleOptionsDialog.this.setVisible(false);
313             }
314         });
315
316         // Add the "start" button to the buttonPanel
317         buttonPanel.add(startButton);
318
319         /*************************************************
320          * 1.1.1 If the user selects the "Close" button at any time, the
321          * application closes.
322          *************************************************/
323         closeButton.addActionListener(new ActionListener() {
324             @Override
325             public void actionPerformed(ActionEvent e) {
326
327                 // Dispose this window
328                 FarkleOptionsDialog.this.dispose();
329
330                 // Close the application
331                 System.exit(0);
332             }
333         });
334
335         // Add the "close" button to the button panel
336         buttonPanel.add(closeButton);
337
338         // Create the gameModeTitlePanel that will display the title of this
339         // JDialog box,
340         // and set the background color
341         JPanel gameModeTitlePanel = new JPanel();
342         gameModeTitlePanel.setBackground(Color.WHITE);
```

```
343
344         // Set the font of the gameModeOptionTitle and add it to the
345         // gameModeTitlePanel
346         gameModeOptionTitle.setFont(new Font("Arial Black", Font.PLAIN, 14));
347         gameModeTitlePanel.add(gameModeOptionTitle);
348
349         // Add the gameModeTitlePanel to the top of the window panel
350         window.add(gameModeTitlePanel, BorderLayout.NORTH);
351
352         // Add the gameModePanel to the center of the window panel
353         window.add(gameModePanel, BorderLayout.CENTER);
354
355         // Add the buttonPanel to the bottom of the window panel
356         window.add(buttonPanel, BorderLayout.SOUTH);
357
358         // Add the window panel to this JDialog box
359         this.add(window, BorderLayout.CENTER);
360
361         // Size the window to fit the preferred size and layouts of its
362         // subcomponents
363         this.pack();
364
365         // Set the location of the window relative to the parent frame
366         this.setLocationRelativeTo(frame);
367     }
368
369     /**
370      * Make the Farkle Options Dialog box visible
371      */
372     public void showWindow() {
373         this.setVisible(true);
374     }
375
376     /**
377      * Get the selected opponent player type
378      *
379      * @return PlayerType - Opponent Player Type selection (PlayerType.USER or
380      *         PlayerType.COMPUTER)
381      */
382     public PlayerType getPlayerType() {
383         return playerType;
384     }
385
386     /**
387      * Set the opponent player type
388      *
389      * @param playerType
390      *            PlayerType.USER or PlayerType.COMPUTER
391      */
392     public void setPlayerType(PlayerType playerType) {
393         this.playerType = playerType;
394     }
395
396     /**
397      * Get the provided player 1 name String
398      *
399      * @return The provided player 1 name
```

```
400      */
401     public String getPlayer1Name() {
402         return player1Name;
403     }
404 
405     /**
406      * Set the player 1 name String
407      *
408      * @param player1Name
409      *            Set the player 1 name String
410      */
411     public void setPlayer1Name(String player1Name) {
412         this.player1Name = player1Name;
413     }
414 
415     /**
416      * Get the provided player 2 name String
417      *
418      * @return The provided player 2 name
419      */
420     public String getPlayer2Name() {
421         return player2Name;
422     }
423 
424     /**
425      * Set the player 2 name String
426      *
427      * @param player2Name
428      *            Set the player 1 name String
429      */
430     public void setPlayer2Name(String player2Name) {
431         this.player2Name = player2Name;
432     }
433 
434     /**
435      * Get the selected Game Mode
436      *
437      * @return The selected GameMode (GameMode.SINGLEPLAYER or
438      *         GameMode.MULTIPLAYER)
439      */
440     public GameMode getGameMode() {
441         return gameMode;
442     }
443 
444     /**
445      * Set the Game Mode
446      *
447      * @param gameMode
448      *            GameMode.SINGLEPLAYER or GameMode.MULTIPLAYER
449      */
450     public void setGameMode(GameMode gameMode) {
451         this.gameMode = gameMode;
452     }
453 
454     /**
455      * Change the settings based on the user's selection
456      */
```

```
457     @Override
458     public void mouseClicked(MouseEvent e) {
459 
460         // The single player game mode has been selected
461         if (e.getComponent().getName().equals("1")) {
462 
463             // Highlight the singlePLayerLabel
464             singlePlayerLabel.setBackground(Color.YELLOW);
465             singlePlayerLabel.setForeground(Color.BLACK);
466 
467             // Deselect the multiPlayerLabel
468             multiplayerLabel.setBackground(greenBackground);
469             multiplayerLabel.setForeground(Color.WHITE);
470 
471             // Hide the playerTypeSelectionPanel
472             playerTypeSelectionPanel.setVisible(false);
473 
474             // Deselect all player types
475             humanPlayerLabel.setBackground(greenBackground);
476             computerPlayerLabel.setBackground(greenBackground);
477 
478             // Hide the playerTwoNamePanel
479             playerTwoNamePanel.setVisible(false);
480             playerTwoName.setText("");
481 
482             // Set the game mode to GameMode.SINGLEPLAYER
483             setGameMode(GameMode.SINGLEPLAYER);
484         }
485 
486         /**************************************************
487          * 1.1.3 - If the user highlights the "2 Player Mode" option, the
488          * "1 Player Mode" option is deselected, and two more options appear
489          * ("Human Opponent" and "Computer Opponent"). The "Human Opponent"
490          * option is highlighted by default.
491          **************************************************/
492         // Else if the two player game mode has been selected
493         else if (e.getComponent().getName().equals("2")) {
494 
495             // Deselect the singlePlayerLabel
496             singlePlayerLabel.setBackground(greenBackground);
497             singlePlayerLabel.setForeground(Color.WHITE);
498 
499             // Highlight the multiPlayerLabel
500             multiplayerLabel.setBackground(Color.YELLOW);
501             multiplayerLabel.setForeground(Color.BLACK);
502 
503             // Highlight the humanPlayerLabel
504             humanPlayerLabel.setForeground(Color.BLACK);
505             humanPlayerLabel.setBackground(Color.YELLOW);
506 
507             // Deselect the computerPlayerLabel
508             computerPlayerLabel.setForeground(Color.WHITE);
509             computerPlayerLabel.setBackground(greenBackground);
510 
511             // Make the playerTypeSelectionPanel visible
512             playerTypeSelectionPanel.setVisible(true);
513 
```

```
514             // Make the playerTwoNamePanel visible, enabled, and clear the name
515             playerTwoNamePanel.setVisible(true);
516             playerTwoNameLabel.setEnabled(true);
517             playerTwoNamePanel.setEnabled(true);
518             playerTwoName.setEnabled(true);
519             playerTwoName.setText("");
520 
521             // Set the game mode to GameMode.MULTIPLAYER
522             setGameMode(GameMode.MULTIPLAYER);
523         }
524 
525         /*************************************************
526          * 1.1.4 - When the "Human Opponent" option is highlighted, two text
527          * fields are displayed, labeled "Player One Name", and
528          * "Player Two Name".
529          *************************************************/
530         // Else if the human opponent player type has been selected
531         else if (e.getComponent().getName().equals("3")) {
532 
533             // Highlight the humanPlayerLabel
534             humanPlayerLabel.setForeground(Color.BLACK);
535             humanPlayerLabel.setBackground(Color.YELLOW);
536 
537             // Deselect the computerPlayerLabel
538             computerPlayerLabel.setForeground(Color.WHITE);
539             computerPlayerLabel.setBackground(greenBackground);
540 
541             // Clear the playerTwoName text field and enable it
542             playerTwoName.setText("");
543             playerTwoNameLabel.setEnabled(true);
544             playerTwoNamePanel.setEnabled(true);
545             playerTwoName.setEnabled(true);
546 
547             // Set the opponent player type to PlayerType.USER
548             setPlayerType(PlayerType.USER);
549         }
550 
551         /*************************************************
552          * 1.1.5 - When the "Computer Opponent" option is highlighted, the text
553          * fields for "Player One Name" and "Player Two Name" are displayed, but
554          * the "Player Two Name" field is disabled, and "Computer" is supplied
555          * for the "Player Two Name".
556          *************************************************/
557         // Else if the Computer opponent player type has been selected
558         else if (e.getComponent().getName().equals("4")) {
559 
560             // Deselect the humanPlayerLabel
561             humanPlayerLabel.setForeground(Color.WHITE);
562             humanPlayerLabel.setBackground(greenBackground);
563 
564             // Highlight the computerPlayerLabel
565             computerPlayerLabel.setForeground(Color.BLACK);
566             computerPlayerLabel.setBackground(Color.YELLOW);
567 
568             // Set the player two name text field to "Computer" and disable it
569             playerTwoName.setText("Computer");
570             playerTwoNameLabel.setEnabled(false);
```

```
571             playerTwoNamePanel.setEnabled(false);
572             playerTwoName.setEnabled(false);
573 
574             // Set the opponent player type to PlayerType.COMPUTER
575             setPlayerType(PlayerType.COMPUTER);
576         }
577 
578     }
579 
580     /**
581      * Not used
582      */
583     @Override
584     public void mouseEntered(MouseEvent e) {
585 
586     }
587 
588     /**
589      * Not used
590      */
591     @Override
592     public void mouseExited(MouseEvent e) {
593 
594     }
595 
596     /**
597      * Not used
598      */
599     @Override
600     public void mousePressed(MouseEvent e) {
601 
602     }
603 
604     /**
605      * Not used
606      */
607     @Override
608     public void mouseReleased(MouseEvent e) {
609 
610     }
611 
612     /**
613      * This method is used to get JLabels for testing
614      *
615      * @param labelNumber
616      *            Specifies the JLabel to return
617      * @return Component representing an option the user selected
618      */
619     public JLabel getJLabel(int labelNumber) {
620 
621         // Return the selected JLabel
622         switch (labelNumber) {
623         case 1:
624             return singlePlayerLabel;
625         case 2:
626             return multiplayerLabel;
627         case 3:
```

```
628             return humanPlayerLabel;
629         case 4:
630             return computerPlayerLabel;
631         case 5:
632             return playerTwoNameLabel;
633         default:
634             return null;
635         }
636 
637     }
638 
639     /**
640      * This method is used to get JPanels for testing
641      *
642      * @param panelNumber
643      *            Specifies the JPanel to return
644      * @return JPanel
645      */
646     public JPanel getJPanel(int panelNumber) {
647 
648         // Return the selected JPanel
649         switch (panelNumber) {
650         case 1:
651             return playerTypeSelectionPanel;
652         case 2:
653             return playerTwoNamePanel;
654         default:
655             return null;
656         }
657 
658     }
659 
660     /**
661      * This method is used to get JTextFields for testing
662      *
663      * @param textFieldNumber
664      *            Specifies the TextField to return
665      * @return JTextField
666      */
667     public JTextField getJTextField(int textFieldNumber) {
668 
669         // Return the selected JTextField
670         switch (textFieldNumber) {
671         case 1:
672             return playerTwoName;
673         default:
674             return null;
675         }
676 
677     }
678 }
679 
```

```java
package com.lotsofun.farkle;

import java.awt.Color;
import java.awt.Dialog.ModalityType;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Image;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.imageio.ImageIO;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollBar;
import javax.swing.JScrollPane;
import javax.swing.ScrollPaneConstants;
import javax.swing.SwingConstants;
import javax.swing.event.MenuEvent;
import javax.swing.event.MenuListener;

/**
 * The primary graphic user interface class for the Farkle application,
 * representing the view in a model view controller architecture.
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public class FarkleUI extends JFrame {

    /** Serialize this to comply with component conventions */
    private static final long serialVersionUID = 1L;

    /** The array of Die objects */
    private Die[] dice = new Die[6];

    /** The "Roll Dice" button */
```

```
58      private JButton rollBtn = new JButton("Roll Dice");
59
60      /** The "Bank Score" button */
61      private JButton bankBtn = new JButton("Bank Score");
62
63      /** The "Select All" button */
64      private JButton selectAllBtn = new JButton("Select All");
65
66      /** The controller for the Farkle application */
67      private FarkleController controller;
68
69      /** Player one label */
70      private JLabel player1NameLabel = new JLabel("Player 1: ");
71
72      /** Player one's name */
73      private JLabel player1Name = new JLabel("");
74
75      /** Player two label */
76      private JLabel player2NameLabel = new JLabel("Player 2: ");
77
78      /** Player two's name */
79      private JLabel player2Name = new JLabel("");
80
81      /** Player one turn score number labels */
82      private ArrayList<JLabel> player1ScoreLabels = new ArrayList<JLabel>();
83
84      /** Player two turn score number labels */
85      private ArrayList<JLabel> player2ScoreLabels = new ArrayList<JLabel>();
86
87      /** Player one turn scores */
88      private ArrayList<JLabel> player1Scores = new ArrayList<JLabel>();
89
90      /** Player two turn scores */
91      private ArrayList<JLabel> player2Scores = new ArrayList<JLabel>();
92
93      /** Player one game score label */
94      private JLabel player1GameScoreLabel = new JLabel("Total Score: ");
95
96      /** Player one game score */
97      private JLabel player1GameScore = new JLabel("0");
98
99      /** Player two game score label */
100     private JLabel player2GameScoreLabel = new JLabel("Total Score: ");
101
102     /** Player two game score */
103     private JLabel player2GameScore = new JLabel("0");
104
105     /** High score label */
106     private JLabel highScoreTitle = new JLabel("High Score: ");
107
108     /** High score */
109     private JLabel highScore = new JLabel();
110
111     /** Score for selected dice */
112     private JLabel runningScore = new JLabel("0");
113
114     /** Score for the current roll */
```

```
115     private JLabel rollScore = new JLabel("0");
116
117     /** Sounds used during the animated roll */
118     private ArrayList<URL> rollSounds = new ArrayList<URL>();
119
120     /** Sounds used during the banking action */
121     private ArrayList<URL> bankSounds = new ArrayList<URL>();
122
123     /** Sound used indicating a bonus turn was earned */
124     private URL bonusSound;
125
126     /** AudioStream for played sounds */
127     private AudioInputStream audioStream = null;
128
129     /** Standard background color used */
130     private Color greenBackground = new Color(35, 119, 34);
131
132     /** Farkle message */
133     private JDialog farkleMessage = new FarkleMessage();
134
135     /** Player one scores panel */
136     private JPanel player1ScorePanel = null;
137
138     /** Player one scores scroll bar */
139     private JScrollBar player1ScrollBar = null;
140
141     /** Player two scores panel */
142     private JPanel player2ScorePanel = null;
143
144     /** Player two scores scroll bar */
145     private JScrollBar player2ScrollBar = null;
146
147     /** Flag set to indicate this is the first time the game is initialized */
148     private boolean isFirstRun = true;
149
150     /** Flag used for testing only */
151     private boolean isTest;
152
153     /**
154      * Constructor: Get a reference to the controller object and fire up the UI
155      *
156      * @param controller
157      *            the farkle controller used for the application
158      */
159     public FarkleUI(FarkleController controller) {
160
161         // Set the reference to the farkle controller
162         this.controller = controller;
163
164         // Determine if this instance is used for testing
165         isTest = this.controller.isTest;
166
167         // Initialize the user interface
168         initUI();
169     }
170
171     /**
```

```
172     * Build the UI base
173     */
174    public void initUI() {
175        /*
176         * If this is the first run, pass a reference to the controller and
177         * build the window.
178         */
179        if (isFirstRun) {
180
181            // Toggle the isFirstRun flag
182            isFirstRun = false;
183
184            // Pass the reference to the controller
185            controller.setUI(this);
186
187            // Instantiate the necessary sounds
188            getSounds();
189
190            // Create and set up the main Window
191            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
192            this.setPreferredSize(new Dimension(1024, 768));
193            this.setResizable(false);
194
195            // Pack the main window
196            this.pack();
197
198            // Center and display the window
199            this.setLocationRelativeTo(null);
200        }
201
202        /*
203         * Remove all components from this container which may exist if it is
204         * not the first init() call
205         */
206        this.getContentPane().removeAll();
207        if (null != player1ScoreLabels) {
208            player1ScoreLabels.clear();
209        }
210
211        if (null != player1Scores) {
212            player1Scores.clear();
213        }
214
215        if (null != player2ScoreLabels) {
216            player2ScoreLabels.clear();
217        }
218
219        if (null != player2Scores) {
220            player2Scores.clear();
221        }
222
223        // Create the layout for the window
224        GridLayout layout = new GridLayout(1, 3, 10, 10);
225
226        // Hide the grid lines
227        layout.setHgap(0);
228        layout.setVgap(0);
```

```
229
230         // Set the layout and background color
231         this.setLayout(layout);
232         this.getContentPane().setBackground(greenBackground);
233
234         // Enable the window
235         this.setEnabled(true);
236
237         // If this is not a test, make the window visible
238         if (isTest == false) {
239             this.setVisible(true);
240
241             // Else, this is a test instance so hide the window
242         } else {
243             this.setVisible(false);
244         }
245
246         // Add the menu bar
247         this.setJMenuBar(createFarkleMenuBar());
248
249         // If this is not a test, create a new game
250         if (isTest == false) {
251             controller.newGame(null);
252         }
253
254         // Pack the window
255         pack();
256     }
257
258     /**
259      * Create a JPanel that contains the "Roll Dice", "Select All", and
260      * "Bank Score" buttons and attach a Listener to each
261      *
262      * @return the created JPanel with the three buttons
263      */
264     public JPanel[] createButtonPanel() {
265
266         // An array of JPanels each of which holds a button
267         JPanel buttonPanels[] = { new JPanel(), new JPanel(), new JPanel() };
268
269         /*************************************************
270          * 1.2.4 - A "Roll Dice" button shall be displayed below the dice in the
271          * middle of the screen.
272          *************************************************/
273         // Add an action listener to the "Roll Dice" button
274         if (rollBtn.getActionListeners().length == 0) {
275             rollBtn.addActionListener(controller);
276         }
277
278         // Set the preferred size and center the button
279         rollBtn.setPreferredSize(bankBtn.getMinimumSize());
280         rollBtn.setAlignmentX(CENTER_ALIGNMENT);
281         rollBtn.setAlignmentY(CENTER_ALIGNMENT);
282
283         // Add the button and set the background color
284         buttonPanels[0].add(rollBtn);
285         buttonPanels[0].setBackground(greenBackground);
```

```
286
287         /**************************************************
288          * 1.2.8 - A "Select All" button shall be displayed below the dice in
289          * the middle of the screen, and shall be initially disabled.
290          **************************************************/
291         // Add an action listener to the "Select All" button
292         if (selectAllBtn.getActionListeners().length == 0) {
293             selectAllBtn.addActionListener(controller);
294         }
295
296         // Set the preferred size and center the button
297         selectAllBtn.setPreferredSize(bankBtn.getMinimumSize());
298         selectAllBtn.setAlignmentX(CENTER_ALIGNMENT);
299         selectAllBtn.setAlignmentY(CENTER_ALIGNMENT);
300
301         // Add the button and set the background color
302         buttonPanels[1].add(selectAllBtn);
303         buttonPanels[1].setBackground(greenBackground);
304
305         // Disable the button
306         selectAllBtn.setEnabled(false);
307
308         /**************************************************
309          * 1.2.5: A "Bank Score" button shall be displayed below the dice in the
310          * middle of the screen, (and shall initially be disabled).
311          **************************************************/
312         // Add an action listener to the "Bank Score" button
313         if (bankBtn.getActionListeners().length == 0) {
314             bankBtn.addActionListener(controller);
315         }
316
317         // Set the preferred size and center the button
318         bankBtn.setPreferredSize(bankBtn.getMinimumSize());
319         bankBtn.setAlignmentX(CENTER_ALIGNMENT);
320         bankBtn.setAlignmentY(CENTER_ALIGNMENT);
321
322         // Add the button and set the background color
323         buttonPanels[2].add(bankBtn);
324         buttonPanels[2].setBackground(greenBackground);
325
326         // Disable the button
327         getBankBtn().setEnabled(false);
328
329         // Return the buttons
330         return buttonPanels;
331     }
332
333     /**
334      * Combine the diceHeader and diceGrid panels in to a single panel which can
335      * be added to the frame
336      */
337     public void buildDicePanel() {
338
339         // Create the dice header panel
340         JPanel diceHeaderPanel = createDiceHeaderPanel();
341
342         // Create the dice grid panel
```

```
343         JPanel diceGridPanel = createDiceGridPanel();
344 
345         // Create a new panel that will combine the header and grid panels
346         JPanel dicePanel = new JPanel();
347 
348         // Set the layout of the panel
349         dicePanel.setLayout(new BoxLayout(dicePanel, BoxLayout.Y_AXIS));
350 
351         // Add a border
352         diceHeaderPanel.setBorder(BorderFactory
353                 .createLineBorder(Color.WHITE, 3));
354 
355         // Add the dice header
356         dicePanel.add(diceHeaderPanel);
357 
358         // Set the preferred size of the dice grid panel and add it
359         diceGridPanel.setPreferredSize(diceGridPanel.getMaximumSize());
360         dicePanel.add(diceGridPanel);
361 
362         // Add the dice panel to this frame
363         this.add(dicePanel);
364     }
365 
366     /**************************************************
367      * 1.2.2 - The total turn score shall be displayed above the dice in the
368      * center of the screen, and the score for the selected dice of the current
369      * roll shall be displayed directly below the turn score in the center of
370      * the screen. This score shall be updated as each die is selected.
371      **************************************************/
372     /**
373      * Create a JPanel to hold the Turn Score, and Roll Score labels and their
374      * corresponding values which will be displayed above the dice in the center
375      * of the screen.
376      *
377      * @return JPanel of the information displayed above the dice
378      */
379     public JPanel createDiceHeaderPanel() {
380 
381         // Instantiate the JPanel for the dice header
382         JPanel diceHeaderPanel = new JPanel(new GridLayout(0, 2, 0, 0));
383 
384         // JLabels for the turn score and roll score
385         JLabel turnScore = new JLabel("<html>Turn Score: </html>");
386         JLabel rollScoreLabel = new JLabel("<html>Roll Score: </html>");
387 
388         // Set the font for the turn score label
389         turnScore.setForeground(Color.WHITE);
390         turnScore.setFont(new Font("Arial Black", Font.BOLD, 14));
391 
392         // Create a border for the turn score label
393         turnScore.setBorder(BorderFactory.createCompoundBorder(
394                 BorderFactory.createMatteBorder(0, 0, 1, 0, Color.WHITE),
395                 BorderFactory.createEmptyBorder(15, 3, 15, 0)));
396 
397         // Add the turn score label to the panel
398         diceHeaderPanel.add(turnScore);
399 
```

```
400         // Set the font for the running turn score and center it
401         runningScore.setForeground(Color.WHITE);
402         runningScore.setFont(new Font("Arial Black", Font.BOLD, 14));
403         runningScore.setHorizontalAlignment(SwingConstants.CENTER);
404 
405         // Set the border for the running turn score
406         runningScore.setBorder(BorderFactory.createCompoundBorder(
407                 BorderFactory.createMatteBorder(0, 0, 1, 0, Color.WHITE),
408                 BorderFactory.createEmptyBorder(15, 0, 15, 3)));
409 
410         // Add the running turn score to the header panel
411         diceHeaderPanel.add(runningScore);
412 
413         // Set the font for the roll score label
414         rollScoreLabel.setForeground(Color.WHITE);
415         rollScoreLabel.setFont(new Font("Arial Black", Font.BOLD, 14));
416 
417         // Set the border for the roll score label
418         rollScoreLabel.setBorder(BorderFactory.createMatteBorder(0, 3, 0, 0,
419                 Color.WHITE));
420         rollScoreLabel.setBorder(BorderFactory.createEmptyBorder(15, 3, 15, 0));
421 
422         // Add the roll score label to the header panel
423         diceHeaderPanel.add(rollScoreLabel);
424 
425         // Set the font for the roll score and center it
426         rollScore.setForeground(Color.WHITE);
427         rollScore.setFont(new Font("Arial Black", Font.BOLD, 14));
428         rollScore.setHorizontalAlignment(SwingConstants.CENTER);
429 
430         // Set the border for the roll score
431         rollScore.setBorder(BorderFactory.createMatteBorder(0, 0, 0, 3,
432                 Color.WHITE));
433         rollScore.setBorder(BorderFactory.createEmptyBorder(15, 0, 15, 3));
434 
435         // Add the roll score to the header panel
436         diceHeaderPanel.add(rollScore);
437 
438         // Set the background color for the header panel and return it
439         diceHeaderPanel.setBackground(greenBackground);
440         return diceHeaderPanel;
441     }
442 
443     /*************************************************
444      * 1.2.1 The center of the screen shall display the six dice used during
445      * gameplay. These dice shall display the name of the game, Â“"Farkle"Â”,
446      * until the user selects the roll button for the first time.
447      *************************************************/
448     /**
449      * Create a JPanel with six Dice, the running score JLabels and the Roll and
450      * Bank buttons
451      *
452      * @return
453      */
454     public JPanel createDiceGridPanel() {
455 
456         /*************************************************
```

```
457          * 3.0.0: Dice
458          ***************************************************/
459         // Create the panel
460         JPanel dicePanel = new JPanel(new GridLayout(0, 3, 0, 0));
461 
462         /***************************************************
463          * 3.1.0: Farkle is played with six standard 6 sided dice with each side
464          * numbered from 1 through 6 (inclusive).
465          ***************************************************/
466         // Initialize the dice and add to panel
467         for (int i = 0; i < dice.length; i++) {
468             dice[i] = new Die(controller);
469             dicePanel.add(new JLabel(" "));
470             dicePanel.add(dice[i]);
471             dice[i].setHorizontalAlignment(SwingConstants.CENTER);
472             dicePanel.add(new JLabel(" "));
473         }
474 
475         // Add some spacing below the dice
476         dicePanel.add(new JLabel(" "));
477         dicePanel.add(new JLabel(" "));
478         dicePanel.add(new JLabel(" "));
479 
480         // Add the buttons below the dice
481         JPanel btns[] = createButtonPanel();
482         dicePanel.add(btns[0]);
483         dicePanel.add(btns[1]);
484         dicePanel.add(btns[2]);
485 
486         // Set the background color of the dice panel
487         dicePanel.setBackground(greenBackground);
488 
489         // Add a border to the panel
490         dicePanel.setBorder(BorderFactory.createCompoundBorder(
491                 BorderFactory.createLineBorder(Color.WHITE, 3),
492                 BorderFactory.createEmptyBorder(3, 10, 3, 10)));
493 
494         // Return the created panel
495         return dicePanel;
496     }
497 
498     /**
499      *
500      * @param gameMode
501      *            The selected game mode (GameMode.SINGLEPLAYER or
502      *            GameMode.MULTIPLAYER)
503      */
504     public void buildPlayerPanel(GameMode gameMode) {
505 
506         // Create the main player panel
507         JPanel playersPanel = new JPanel();
508 
509         // Set the layout and border for the panel
510         playersPanel.setLayout(new BoxLayout(playersPanel, BoxLayout.Y_AXIS));
511         playersPanel.setBorder(BorderFactory.createLineBorder(Color.WHITE, 3));
512 
513         /***************************************************
```

```
514          * 1.3.0 - One player mode graphic user interface
515          ************************************************/
516         // If the selected game mode is GameMode.SINGLEPLAYER
517         if (null != gameMode && gameMode == GameMode.SINGLEPLAYER) {
518
519             /************************************************
520              * 1.3.6 - The top of the left hand side of the screen shall display
521              * "Player: ", along with the provided name of the player.
522              ************************************************/
523             // Create the scroll pane for player 1 and add an empty border
524             JScrollPane player1ScrollPanel = createPlayerScorePanel(1, 10);
525             player1ScrollBar = player1ScrollPanel.getVerticalScrollBar();
526             player1ScrollPanel.setBorder(BorderFactory.createEmptyBorder(0, 0,
527                     0, 6));
528
529             // Create the player 1 name panel
530             JPanel player1NamePanel = createPlayerNamePanel(1);
531
532             // Set the border for the player 1 name panel
533             player1NamePanel.setBorder(BorderFactory.createMatteBorder(0, 0, 6,
534                     0, Color.WHITE));
535
536             // Add the player 1 name panel to the players panel
537             playersPanel.add(player1NamePanel);
538
539             // Set the preferred size and background for the player 1 scroll
540             // panel
541             player1ScrollPanel.setPreferredSize(new Dimension(350, 568));
542             player1ScrollPanel.setBackground(greenBackground);
543
544             // Add the player 1 scroll panel to the players panel
545             playersPanel.add(player1ScrollPanel);
546
547             /************************************************
548              * 1.3.5 - The current highest achieved score shall be displayed on
549              * the lower left hand corner of the screen. This score shall
550              * initially be set to 0 points.
551              ************************************************/
552             // Add the high score to the players panel
553             addHighScore(playersPanel);
554
555             /************************************************
556              * 1.4.0 - Two player mode graphic user interface
557              ************************************************/
558             // Else if the game mode is GameMode.MULTIPLAYER
559         } else if (null != gameMode && gameMode == GameMode.MULTIPLAYER) {
560
561             /************************************************
562              * 1.4.2.a - Each player shall be indicated in the following manner:
563              * "Player: " along with the provided playerâ€™s name, or "Computer"
564              * if a "Computer Opponent" has been selected, followed by the
565              * running point total for the current game for that player.
566              ************************************************/
567             /************************************************
568              * 1.4.5 - The turn totals for each player shall be displayed in a
569              * scroll pane below that playerâ€™s name and game score. This scroll
570              * pane shall initially display 5 turns, adding additional turns
```

```
571             * after they are taken. The scrolling ability shall be enabled at
572             * the beginning of the 11th turn.
573             ************************************************/
574            // Create the scroll pane for player 1 and add an empty border
575            JScrollPane player1ScrollPanel = createPlayerScorePanel(1, 5);
576            player1ScrollBar = player1ScrollPanel.getVerticalScrollBar();
577            player1ScrollPanel.setBorder(BorderFactory.createEmptyBorder(0, 0,
578                    0, 6));
579
580            // Create the player 1 name panel
581            JPanel player1NamePanel = createPlayerNamePanel(1);
582
583            // Set the border for the player 1 name panel
584            player1NamePanel.setBorder(BorderFactory.createMatteBorder(0, 0, 6,
585                    0, Color.WHITE));
586
587            // Add the player 1 name panel to the players panel
588            playersPanel.add(player1NamePanel);
589
590            // Set the preferred size and background for the player 1 scroll
591            // panel
592            player1ScrollPanel.setPreferredSize(new Dimension(350, 249));
593            player1ScrollPanel.setBackground(greenBackground);
594
595            // Add the player 1 scroll panel to the players panel
596            playersPanel.add(player1ScrollPanel);
597
598            // Create the scroll pane for player 2 and add an empty border
599            JScrollPane player2ScrollPanel = createPlayerScorePanel(2, 5);
600            player2ScrollBar = player2ScrollPanel.getVerticalScrollBar();
601            player2ScrollPanel.setBorder(BorderFactory.createEmptyBorder(0, 0,
602                    0, 6));
603
604            // Create the player 2 name panel
605            JPanel player2NamePanel = createPlayerNamePanel(2);
606
607            // Set the border for the player 2 name panel
608            player2NamePanel.setBorder(BorderFactory.createMatteBorder(6, 0, 6,
609                    0, Color.WHITE));
610
611            // Add the player 2 name panel to the players panel
612            playersPanel.add(player2NamePanel);
613
614            // Set the preferred size and background for the player 2 scroll
615            // panel
616            player2ScrollPanel.setPreferredSize(new Dimension(350, 249));
617            player2ScrollPanel.setBackground(greenBackground);
618
619            // Add the player 2 scroll panel to the players panel
620            playersPanel.add(player2ScrollPanel);
621
622            // Set the background for the players panel
623            playersPanel.setBackground(greenBackground);
624        }
625
626        // Set the background color
627        playersPanel.setBackground(greenBackground);
```

```
628
629         // Add this panel to the window
630         this.add(playersPanel, 0);
631     }
632
633     /**
634      * Create the panel that displays the player name and total game score for
635      * that player.
636      *
637      * @param playerNumber
638      *            int representing the player for which this panel is created
639      * @return constructed JPanel with player name and game score
640      */
641     private JPanel createPlayerNamePanel(int playerNumber) {
642
643         // Create the layout for the panel
644         GridLayout gridLayout = new GridLayout(0, 2, 0, 0);
645         gridLayout.setHgap(0);
646         gridLayout.setVgap(0);
647
648         // Create the panel with the layout
649         JPanel playerNamePanel = new JPanel(gridLayout);
650
651         // Add the player name label and the player name
652         JLabel playerNameLabel = (playerNumber == 1) ? player1NameLabel
653                 : player2NameLabel;
654         JLabel playerName = (playerNumber == 1) ? player1Name : player2Name;
655
656         /*****************************************************
657          * 1.3.2 - The overall point total for the current game shall be
658          * displayed on the upper left hand corner of the screen, just below the
659          * player's name.
660          *****************************************************/
661         /*****************************************************
662          * 1.4.2 - The left side of the screen shall have an area to display the
663          * overall accumulated point total for each player. This takes the place
664          * of the area displaying the point total for each turn in the one
665          * player mode graphic user interface.
666          *****************************************************/
667         // Add the player score label and the player score
668         JLabel playerGameScoreLabel = (playerNumber == 1) ? player1GameScoreLabel
669                 : player2GameScoreLabel;
670         JLabel playerGameScore = (playerNumber == 1) ? player1GameScore
671                 : player2GameScore;
672
673         // Set the font for the player name label
674         playerNameLabel.setForeground(Color.WHITE);
675         playerNameLabel.setFont(new Font("Arial Black", Font.BOLD, 14));
676
677         // Set the border for the player name label
678         playerNameLabel.setBorder(BorderFactory.createCompoundBorder(
679                 BorderFactory.createMatteBorder(0, 0, 1, 0, Color.WHITE),
680                 BorderFactory.createEmptyBorder(15, 3, 15, 0)));
681
682         // Add the player name label to the panel
683         playerNamePanel.add(playerNameLabel);
684
```

```
685         // Set the font for the player name
686         playerName.setForeground(Color.WHITE);
687         playerName.setFont(new Font("Arial Black", Font.BOLD, 14));
688 
689         // Set the border for the player name
690         playerName.setBorder(BorderFactory.createCompoundBorder(
691                 BorderFactory.createMatteBorder(0, 0, 1, 0, Color.WHITE),
692                 BorderFactory.createEmptyBorder(15, 13, 15, 3)));
693 
694         // Add the player name
695         playerNamePanel.add(playerName);
696 
697         /**************************************************
698          * 1.3.3 - The left side of the screen shall have an area to display the
699          * point total for each of the ten turns taken in single player mode.
700          **************************************************/
701         // Set the font for the game score label
702         playerGameScoreLabel.setForeground(Color.WHITE);
703         playerGameScoreLabel.setFont(new Font("Arial Black", Font.BOLD, 14));
704 
705         // Set the border for the game score label
706         playerGameScoreLabel.setBorder(BorderFactory.createEmptyBorder(15, 3,
707                 15, 3));
708 
709         // Add the game score label to the player name panel
710         playerNamePanel.add(playerGameScoreLabel);
711 
712         // Set the font for the game score
713         playerGameScore.setForeground(Color.WHITE);
714         playerGameScore.setFont(new Font("Arial Black", Font.BOLD, 14));
715 
716         // Set the border for the game score
717         playerGameScore.setBorder(BorderFactory
718                 .createEmptyBorder(15, 13, 15, 3));
719 
720         // Add the game score to the player name panel
721         playerNamePanel.add(playerGameScore);
722 
723         // Set the background for the player name panel
724         playerNamePanel.setBackground(greenBackground);
725 
726         // Set the preferred and maximum size for the player name panel
727         playerNamePanel.setPreferredSize(new Dimension(350, 110));
728         playerNamePanel.setMaximumSize(new Dimension(350, 110));
729 
730         // Return the created panel
731         return playerNamePanel;
732     }
733 
734     /**
735      * Create a new JPanel that contains all the JLabels necessary to represent
736      * a player with 10 turns
737      *
738      * @param playerNumber
739      *            int player position for which this Scroll Pane is created
740      * @param scoreLabelCount
741      *            int number of turn score labels to include
```

```
742     * @return JScrollPane populated with player turn information
743     */
744    private JScrollPane createPlayerScorePanel(int playerNumber,
745            int scoreLabelCount) {
746
747        // Create the player panel
748        JPanel playerPanel = new JPanel(new GridLayout(0, 2, 0, 2));
749
750        // Set the player panel according to the player number parameter
751        if (playerNumber == 1) {
752            player1ScorePanel = playerPanel;
753        } else {
754            player2ScorePanel = playerPanel;
755        }
756
757        /**************************************************
758         * 1.4.3: The left side of the screen shall have an area to display the
759         * point total for each of the ten turns taken in single player mode.
760         **************************************************/
761        // Add the requested number of turn score labels
762        for (int i = 0; i < scoreLabelCount; i++) {
763            addTurnScore(playerNumber, i + 1);
764        }
765
766        // Set the background for the player panel and add an empty border
767        playerPanel.setBackground(greenBackground);
768        playerPanel.setBorder(BorderFactory.createEmptyBorder(3, 6, 3, 17));
769
770        // Create the JScrollPane from the player panel
771        JScrollPane playerScrollPane = new JScrollPane(playerPanel,
772                ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
773                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
774
775        // Return the created scroll pane
776        return playerScrollPane;
777    }
778
779    /**
780     * Add the high score panel to a panel
781     *
782     * @param panel
783     *            JPanel to which the high score panel is to be added
784     */
785    public void addHighScore(JPanel panel) {
786
787        /**************************************************
788         * 1.4.5: The current highest achieved score shall be displayed. This
789         * score shall initially be set to 5000 points.
790         **************************************************/
791        // Create the high score panel
792        JPanel highScorePanel = new JPanel(new GridLayout(0, 2, 0, 0));
793
794        // Set the font for the high score title
795        highScoreTitle.setForeground(Color.WHITE);
796        highScoreTitle.setFont(new Font("Arial Black", Font.BOLD, 14));
797
798        // Add the high score title to the high score panel
```

```
799         highScorePanel.add(highScoreTitle);
800
801         // Set the font for the high score
802         highScore.setForeground(Color.WHITE);
803         highScore.setFont(new Font("Arial Black", Font.BOLD, 14));
804
805         // Add the high score to the high score panel
806         highScorePanel.add(highScore);
807         highScorePanel.setBackground(greenBackground);
808
809         // Set the maximum size of the panel
810         highScorePanel.setMaximumSize(new Dimension(350, 50));
811
812         // Add a border to the panel
813         highScorePanel.setBorder(BorderFactory.createCompoundBorder(
814                 BorderFactory.createMatteBorder(6, 0, 0, 0, Color.WHITE),
815                 BorderFactory.createEmptyBorder(0, 3, 0, 0)));
816
817         // Add the high score panel to the supplied panel
818         panel.add(highScorePanel);
819     }
820
821     /**
822      * Add a turn score to the player score panel for a given player
823      *
824      * @param playerNumber
825      *            Player number for which the turn score is added
826      * @param turnNumber
827      *            The turn number to include with this turn score
828      */
829     public void addTurnScore(int playerNumber, int turnNumber) {
830
831         // Get a reference to the player score labels for the provided player
832         ArrayList<JLabel> playerScoreLabels = (playerNumber == 1) ? player1ScoreLabels
833                 : player2ScoreLabels;
834
835         // Get a reference to the player scores for the provided player
836         ArrayList<JLabel> playerScores = (playerNumber == 1) ? player1Scores
837                 : player2Scores;
838
839         // Get a reference to the player score panel for the provided player
840         JPanel playerPanel = (playerNumber == 1) ? player1ScorePanel
841                 : player2ScorePanel;
842
843         // Add turn label for the provided turn
844         JLabel turnLabel = new JLabel("Turn " + (turnNumber) + ": ");
845
846         // Create an empty border for the label
847         turnLabel.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 0));
848
849         // Set the font for the turn label
850         turnLabel.setForeground(Color.WHITE);
851         turnLabel.setFont(new Font("Arial Black", Font.BOLD, 14));
852
853         // Set the maximum and preferred sizes for the turn label
854         turnLabel.setMaximumSize(new Dimension(175, 20));
855         turnLabel.setPreferredSize(new Dimension(175, 20));
```

```
856
857         // Add the turn label to the player score labels
858         playerScoreLabels.add(turnLabel);
859
860         // Add the turn number to the player panel
861         playerPanel.add(playerScoreLabels.get(turnNumber - 1));
862
863         // Add the score label for each turn
864         JLabel turnScoreLabel = new JLabel();
865
866         // Set the turn score label font
867         turnScoreLabel.setForeground(Color.WHITE);
868         turnScoreLabel.setFont(new Font("Arial Black", Font.BOLD, 14));
869
870         // Set the maximum and preferred sizes for the score label
871         turnLabel.setMaximumSize(new Dimension(175, 20));
872         turnLabel.setPreferredSize(new Dimension(175, 20));
873
874         // Add the turn score label to the panel
875         playerScores.add(turnScoreLabel);
876
877         // add the player scores to the player panel
878         playerPanel.add(playerScores.get(turnNumber - 1));
879     }
880
881     /*****************************************************
882      * 1.2.3 - Rules for the scoring combinations shall be displayed on the
883      * right side of the screen.
884      *****************************************************/
885     /**
886      * Create a new JPanel that displays the score guide image
887      */
888     public void createScoreGuidePanel() {
889
890         // Create the score panel
891         JPanel scorePanel = new JPanel();
892
893         // Add the FarkleScores.png image to the score panel
894         try {
895             Image scoreGuide = ImageIO.read(getClass().getResource(
896                     "/images/FarkleScores.png"));
897             scoreGuide = scoreGuide.getScaledInstance(200, 680,
898                     Image.SCALE_SMOOTH);
899             JLabel scoreLabel = new JLabel(new ImageIcon(scoreGuide));
900             scorePanel.add(scoreLabel);
901
902             // Set the background of the panel
903             scorePanel.setBackground(greenBackground);
904         } catch (IOException e) {
905             e.printStackTrace();
906             System.exit(0);
907         }
908
909         // Add a border to the score panel
910         scorePanel.setBorder(BorderFactory.createCompoundBorder(
911                 BorderFactory.createLineBorder(Color.WHITE, 3),
912                 BorderFactory.createEmptyBorder(3, 17, 3, 17)));
```

```
913
914         // Add the score panel to the window
915         this.add(scorePanel);
916     }
917
918     /**
919      * Get the "Roll Dice" button
920      *
921      * @return The "Roll Dice" button
922      */
923     public JButton getRollBtn() {
924         return rollBtn;
925     }
926
927     /**
928      * Get the "Bank Score" button
929      *
930      * @return The "Bank Score" button
931      */
932     public JButton getBankBtn() {
933         return bankBtn;
934     }
935
936     /**
937      * Get the "Select All" button
938      *
939      * @return The "Select All" button
940      */
941     public JButton getSelectAllBtn() {
942         return selectAllBtn;
943     }
944
945     /**
946      * Set the game score for a given player
947      *
948      * @param playerNumber
949      *             Player number for which to set the game score
950      * @param score
951      *             The score to set
952      */
953     public void setGameScore(int playerNumber, int score) {
954         JLabel gameScore = (playerNumber == 1) ? player1GameScore
955                 : player2GameScore;
956         gameScore.setText("" + score);
957     }
958
959     /**
960      * Get the game score for a chosen player
961      *
962      * @param playerNumber
963      *             Player number for which the game score is sought
964      * @return The game score for the chosen player
965      */
966     public int getGameScore(int playerNumber) {
967
968         // Get the game score JLabel for the chosen player
969         JLabel gameScore = (playerNumber == 1) ? player1GameScore
```

```
970                 : player2GameScore;
971
972         try {
973             // Parse the game score as an Integer
974             return Integer.parseInt(gameScore.getText());
975         } catch (NumberFormatException e) {
976             e.printStackTrace();
977             return -10000;
978         }
979     }
980
981     /**
982      * Set the high score label text
983      *
984      * @param score
985      *             High score for which to set
986      */
987     public void setHighScore(int score) {
988         this.highScore.setText("" + score);
989     }
990
991     /**
992      * Get the high score displayed in the high score label
993      *
994      * @return The high score
995      */
996     public int getHighScore() {
997         try {
998             // Parse the high score label text as an integer and return it
999             return Integer.parseInt(this.highScore.getText());
1000        } catch (NumberFormatException e) {
1001            e.printStackTrace();
1002            return -10000;
1003        }
1004    }
1005
1006    /**
1007     * Get the running score displayed in the running score label
1008     *
1009     * @return The running score
1010     */
1011    public int getRunningScore() {
1012        try {
1013            // Parse the running score label text as an integer and return it
1014            return Integer.valueOf(runningScore.getText());
1015        } catch (NumberFormatException e) {
1016            e.printStackTrace();
1017            return -10000;
1018        }
1019    }
1020
1021    /**
1022     * Randomly roll all of the dice
1023     */
1024    public void rollDice() {
1025
1026        // Roll all of the dice
```

```
1027        for (Die d : dice) {
1028            d.roll();
1029        }
1030    }
1031
1032    /**
1033     * Get the dice with the specified DieStates
1034     *
1035     * @param dieState
1036     *            One or more DieState types
1037     * @return the dice with the specified DieStates
1038     */
1039    public ArrayList<Die> getDice(DieState... dieState) {
1040
1041        // Create a list of the provided die states
1042        ArrayList<DieState> dieStates = new ArrayList<DieState>();
1043        for (DieState state : dieState) {
1044            dieStates.add(state);
1045        }
1046
1047        // Build a list of all dice that are in one of the specified die states
1048        ArrayList<Die> retDice = new ArrayList<Die>();
1049        for (Die d : dice) {
1050            if (dieStates.contains(d.getState())) {
1051                retDice.add(d);
1052            }
1053        }
1054
1055        // Return the list of dice in the specified states
1056        return retDice;
1057    }
1058
1059    /**
1060     * Get the values from the dice with the specified DieStates
1061     *
1062     * @param dieState
1063     *            One or more DieState types
1064     * @return List<Integer> values from the dice with the specified DieStates
1065     */
1066    public List<Integer> getDieValues(DieState... dieState) {
1067
1068        // Get the list of dice that are in one of the specified die states
1069        ArrayList<Die> diceToCheck = getDice(dieState);
1070
1071        // Build the list of die values that are in one of the specified die
1072        // states
1073        List<Integer> retVals = new ArrayList<Integer>();
1074        for (Die d : diceToCheck) {
1075            retVals.add(d.getValue());
1076        }
1077
1078        // Return the list of die values
1079        return retVals;
1080    }
1081
1082    /**
1083     * Set the state of all six dice to UNHELD
```

```
1084     */
1085    public void resetDice() {
1086        for (Die d : dice) {
1087            d.setState(DieState.UNHELD);
1088        }
1089    }
1090
1091    /**
1092     * Prevents dice from being selected
1093     */
1094    public void disableDice() {
1095        for (Die d : dice) {
1096            d.setState(DieState.DISABLED);
1097        }
1098    }
1099
1100    /**
1101     * Resets dice, enables the roll button, and disables the select all and
1102     * bank buttons
1103     */
1104    public void resetDicePanel() {
1105        resetDice();
1106        this.rollBtn.setEnabled(true);
1107        this.selectAllBtn.setEnabled(false);
1108        this.bankBtn.setEnabled(false);
1109    }
1110
1111    /**************************************************
1112     * 1.2.1 The center of the screen shall display the six dice used during
1113     * gameplay. These dice shall display the name of the game, Â“"Farkle"Â”,
1114     * until the user selects the roll button for the first time.
1115     **************************************************/
1116    /**
1117     * Initialize the dice to display the word "Farkle" at the beginning of a
1118     * game
1119     */
1120    public void diceFirstRun() {
1121        dice[0].setValue('f');
1122        dice[1].setValue('a');
1123        dice[2].setValue('r');
1124        dice[3].setValue('k');
1125        dice[4].setValue('l');
1126        dice[5].setValue('e');
1127    }
1128
1129    /**
1130     * Reset all scores for a specified player
1131     *
1132     * @param playerNumber
1133     *            The specified player number
1134     */
1135    public void resetScores(int playerNumber) {
1136
1137        // Get a reference to the scores
1138        ArrayList<JLabel> scores = getPlayerScores(playerNumber);
1139
1140        // Reset the scores
```

```
1141        for (JLabel score : scores) {
1142            score.setText("");
1143        }
1144    }
1145
1146    /**
1147     * Update all dice with a HELD state to a SCORED state
1148     */
1149    public void lockScoredDice() {
1150        for (Die d : dice) {
1151            if (d.getState() == DieState.HELD) {
1152                d.setState(DieState.SCORED);
1153            }
1154        }
1155    }
1156
1157    /**
1158     * Update the turn label for the specified player with the specified score
1159     *
1160     * @param int player Specified player number
1161     * @param int turn Specified turn number
1162     * @param int score Turn score to set
1163     */
1164    public void setTurnScore(int player, int turn, int score) {
1165        ArrayList<JLabel> scores = getPlayerScores(player);
1166        scores.get(turn - 1).setText("" + score);
1167    }
1168
1169    /**
1170     * Get the turn score for a specified player and specified turn number
1171     *
1172     * @param player
1173     *            int specified player's position
1174     * @param turn
1175     *            int specified turn number
1176     * @return The turn score for the specified player and turn number
1177     */
1178    public int getTurnScore(int player, int turn) {
1179
1180        // Get a reference to the turn scores
1181        ArrayList<JLabel> scores = getPlayerScores(player);
1182
1183        // Range check for the provided turn number
1184        if (scores.size() >= turn) {
1185
1186            // If the turn score for the provided turn number is empty, return 0
1187            if (null != scores.get(turn - 1).getText()
1188                    && "".equals(scores.get(turn - 1).getText())) {
1189                return 0;
1190
1191                // Else, return it as a parsed integer
1192            } else {
1193                try {
1194                    return Integer.parseInt(scores.get(turn - 1).getText());
1195                } catch (NumberFormatException e) {
1196                    e.printStackTrace();
1197                    return 0;
```

```
1198                }
1199            }
1200
1201            // Else, the range is out of bounds
1202        } else {
1203            return -10000;
1204        }
1205    }
1206
1207    /**
1208     * Update the running score label with the specified score
1209     *
1210     * @param score
1211     *             The score to set
1212     */
1213    public void setRunningScore(int score) {
1214        runningScore.setText("" + score);
1215    }
1216
1217    /**
1218     * Update the roll score label with the specified score
1219     *
1220     * @param score
1221     *             The score to set
1222     */
1223    public void setRollScore(int score) {
1224        rollScore.setText("" + score);
1225    }
1226
1227    /**
1228     * Get the roll score from the roll score label text
1229     *
1230     * @return The roll score
1231     */
1232    public int getRollScore() {
1233        try {
1234            // Return the roll score label text parsed as an integer
1235            return Integer.parseInt(this.rollScore.getText());
1236        } catch (NumberFormatException e) {
1237            e.printStackTrace();
1238            return -10000;
1239        }
1240    }
1241
1242    /**
1243     * Randomly play one of the four dice roll sounds
1244     */
1245    public void playRollSound() {
1246        try {
1247            // Randomly play one of the three roll sounds
1248            audioStream = AudioSystem.getAudioInputStream(rollSounds
1249                    .get(new Random().nextInt(3)));
1250            Clip clip = AudioSystem.getClip();
1251            clip.open(audioStream);
1252            clip.start();
1253        } catch (UnsupportedAudioFileException e) {
1254            e.printStackTrace();
```

```
1255        } catch (LineUnavailableException e) {
1256            e.printStackTrace();
1257        } catch (IOException e) {
1258            e.printStackTrace();
1259        }
1260    }
1261
1262    /**
1263     * Play the bonus roll sound
1264     */
1265    public void playBonusSound() {
1266        try {
1267            audioStream = AudioSystem.getAudioInputStream(bonusSound);
1268            Clip clip = AudioSystem.getClip();
1269            clip.open(audioStream);
1270            clip.start();
1271        } catch (UnsupportedAudioFileException e) {
1272            e.printStackTrace();
1273        } catch (LineUnavailableException e) {
1274            e.printStackTrace();
1275        } catch (IOException e) {
1276            e.printStackTrace();
1277        }
1278    }
1279
1280    /**
1281     * Play the bank sound
1282     */
1283    public void playBankSound() {
1284        try {
1285            audioStream = AudioSystem.getAudioInputStream(bankSounds
1286                    .get(new Random().nextInt(3)));
1287            Clip clip = AudioSystem.getClip();
1288            clip.open(audioStream);
1289            clip.start();
1290        } catch (UnsupportedAudioFileException e) {
1291            e.printStackTrace();
1292        } catch (LineUnavailableException e) {
1293            e.printStackTrace();
1294        } catch (IOException e) {
1295            e.printStackTrace();
1296        }
1297    }
1298
1299    /**
1300     * Display a message to the user via JOptionPane
1301     *
1302     * @param message
1303     *            Message to be displayed in the main box.
1304     * @param title
1305     *            Title of the JOptionPane window.
1306     */
1307    @SuppressWarnings("deprecation")
1308    public void displayMessage(String message, String title) {
1309
1310        // Set the message for the JOptionPane
1311        JOptionPane pane = new JOptionPane(message);
```

```
1312
1313        // Create a JDialog, and set the title
1314        JDialog dialog = pane.createDialog(title);
1315
1316        // Set the dialog to display in front of all other windows
1317        dialog.setModalityType(ModalityType.APPLICATION_MODAL);
1318
1319        // Show the dialog
1320        dialog.show();
1321    }
1322
1323    /**
1324     * Display a Yes/No JOption pane to the user
1325     *
1326     * @param message
1327     *            Message to be displayed in the main box
1328     * @param title
1329     *            Title of the JOptionPane window
1330     * @return boolean true if the user selected yes, false otherwise
1331     */
1332    public boolean displayYesNoMessage(String message, String title) {
1333        boolean retVal;
1334        int dialogResult = JOptionPane.showConfirmDialog(this, message, title,
1335                JOptionPane.YES_NO_OPTION);
1336        if (dialogResult == JOptionPane.YES_OPTION) {
1337            retVal = true;
1338        } else {
1339            retVal = false;
1340        }
1341        return retVal;
1342    }
1343
1344    /**
1345     * Dispose the window and close the application
1346     */
1347    @Override
1348    public void dispose() {
1349        super.dispose();
1350        System.exit(0);
1351    }
1352
1353    /**
1354     * Select dice animation used for the computer player's dice selection
1355     *
1356     * @param valuesNeeded
1357     *            List of int values to select
1358     * @throws InterruptedException
1359     */
1360    public void selectDice(List<Integer> valuesNeeded)
1361            throws InterruptedException {
1362
1363        // Retrieve a list of all unheld dice
1364        List<Die> dice = getDice(DieState.UNHELD);
1365
1366        // For each die in dice, select it if it is in the list of values needed
1367        for (Die d : dice) {
1368            for (Integer i : valuesNeeded) {
```

```
1369
1370                // If the current dice matches the current value needed
1371                if (d.getValue() == i) {
1372
1373                    // Pause the animation for a half second
1374                    Thread.sleep(500);
1375
1376                    // Select the die
1377                    d.dispatchEvent(new DieClickEvent(d,
1378                            MouseEvent.MOUSE_PRESSED, System
1379                                    .currentTimeMillis(), MouseEvent.BUTTON1, d
1380                                    .getX(), d.getY(), 1, false));
1381
1382                    // Remove the value needed
1383                    valuesNeeded.remove(i);
1384
1385                    // Break the inner for loop
1386                    break;
1387                }
1388            }
1389        }
1390
1391        // Pause for 1.5 seconds after all dice are selected
1392        Thread.sleep(1500);
1393    }
1394
1395    /**
1396     * Selects all unheld dice
1397     */
1398    public void selectAllDice() {
1399
1400        // Get the list of all unheld dice
1401        ArrayList<Die> dice = getDice(DieState.UNHELD);
1402
1403        // Simulate a mouse click selecting each unheld die
1404        for (Die d : dice) {
1405            d.dispatchEvent(new MouseEvent(d, MouseEvent.MOUSE_PRESSED, System
1406                    .currentTimeMillis(), MouseEvent.BUTTON1, d.getX(), d
1407                    .getY(), 1, false));
1408        }
1409    }
1410
1411    /**
1412     * Set the player name label text for a specified player
1413     *
1414     * @param playerNumber
1415     *            The number of the specified player
1416     * @param name
1417     *            The name to set
1418     */
1419    public void setPlayerName(int playerNumber, String name) {
1420        if (null != name && playerNumber >= 1 && playerNumber <= 2) {
1421            if (playerNumber == 1) {
1422                this.player1Name.setText(name);
1423            } else {
1424                this.player2Name.setText(name);
1425            }
```

```
1426        }
1427    }
1428
1429    /**
1430     * Unhighlight all player turn scores for a specified player
1431     *
1432     * @param playerNumber
1433     *            int player position
1434     */
1435    public void unHighlightAllTurnScores(int playerNumber) {
1436
1437        // Get the labels and their scores for the given player
1438        ArrayList<JLabel> scoreLabels = getPlayerScoreLabels(playerNumber);
1439        ArrayList<JLabel> scores = getPlayerScores(playerNumber);
1440
1441        // Make sure it's a 1:1 relationship
1442        assert (scoreLabels.size() == scores.size());
1443
1444        // Unhighlight each set
1445        for (int i = 0; i < scoreLabels.size(); i++) {
1446            unHighlightTurn(scoreLabels.get(i), scores.get(i));
1447        }
1448    }
1449
1450    /**
1451     * Highlight a specified turn score for a specified player
1452     *
1453     * @param playerNumber
1454     *            int player position
1455     * @param turn
1456     *            int turn number
1457     * @param isBonusTurn
1458     *            boolean is a bonus turn
1459     */
1460    public void highlightTurnScore(int playerNumber, int turn,
1461            boolean isBonusTurn) {
1462
1463        // Unhighlight all turns
1464        unHighlightAllTurnScores(playerNumber);
1465
1466        // Get the labels and their scores for the given player
1467        ArrayList<JLabel> scoreLabels = getPlayerScoreLabels(playerNumber);
1468        ArrayList<JLabel> scores = getPlayerScores(playerNumber);
1469
1470        // Add a turn score if it does not already exist in the turn scores
1471        // panel
1472        if (turn > 0 && turn > scoreLabels.size()) {
1473            addTurnScore(playerNumber, turn);
1474        }
1475
1476        // Highlight the desired turn
1477        highlightTurn(scoreLabels.get(turn - 1), scores.get(turn - 1),
1478                isBonusTurn);
1479
1480        // Get the scroll bar associated with the given player
1481        JScrollBar playerScrollBar = (playerNumber == 1) ? player1ScrollBar
1482                : player2ScrollBar;
```

```
1483
1484        if (null != playerScrollBar) {
1485
1486            // Revalidate the ScrollPane to get the updated dimensions
1487            // in case a new label has been added
1488            playerScrollBar.getParent().validate();
1489
1490            // Scroll to the bottom of the JScrollPane
1491            playerScrollBar.setValue(playerScrollBar.getMaximum());
1492        }
1493    }
1494
1495    /**
1496     * Unhighlight a specified score label and score
1497     *
1498     * @param scoreLabel
1499     *            JLabel to unhighlight
1500     * @param score
1501     *            JLabel to unhighlight
1502     */
1503    public void unHighlightTurn(JLabel scoreLabel, JLabel score) {
1504
1505        /*
1506         * Remove the white background, and set the font to white for each
1507         * JLabel component
1508         */
1509        scoreLabel.setOpaque(false);
1510        scoreLabel.setForeground(Color.WHITE);
1511        score.setOpaque(false);
1512        score.setForeground(Color.WHITE);
1513    }
1514
1515    /**
1516     * Highlight a specified score label and score
1517     *
1518     * @param scoreLabel
1519     *            JLabel to highlight
1520     * @param score
1521     *            JLabel to highlight
1522     * @param isBonusTurn
1523     *            boolean is to be highlighted as a bonus turn
1524     */
1525    public void highlightTurn(JLabel scoreLabel, JLabel score,
1526            boolean isBonusTurn) {
1527
1528        /************************************************
1529         * 1.3.4 - The current turn shall be indicated by highlighting that turn
1530         * on the left side of the screen. This turn shall be highlighted as
1531         * soon as the previous turn ends (which occurs after the player selects
1532         * the "Bank Score" button, or after the Farkle message animation
1533         * completes), and before the player selects the "Roll Dice" button for
1534         * the current turn.
1535         ************************************************/
1536        /*
1537         * Highlight given turn in white, and set font to black if it's not a
1538         * bonus turn
1539         */
```

```
1540        if (!isBonusTurn) {
1541            scoreLabel.setOpaque(true);
1542            scoreLabel.setBackground(Color.WHITE);
1543            scoreLabel.setForeground(Color.BLACK);
1544
1545            score.setOpaque(true);
1546            score.setBackground(Color.WHITE);
1547            score.setText("");
1548            score.setForeground(Color.BLACK);
1549
1550            /*
1551             * Else, this is a bonus turn. Highlight given turn in yellow, set
1552             * font to black, and display the bonus turn text.
1553             */
1554        } else {
1555            scoreLabel.setOpaque(true);
1556            scoreLabel.setBackground(Color.YELLOW);
1557            scoreLabel.setForeground(Color.BLACK);
1558
1559            score.setOpaque(true);
1560            score.setText("BONUS ROLL!");
1561            score.setBackground(Color.YELLOW);
1562            score.setForeground(Color.BLACK);
1563            score.setBorder(BorderFactory.createEmptyBorder(0, 0, 0, 3));
1564        }
1565    }
1566
1567    /**
1568     * Get all score labels for a specified player
1569     *
1570     * @param playerNumber
1571     *            The number of the specified player
1572     * @return An ArrayList<JLabel> of the score labels
1573     */
1574    public ArrayList<JLabel> getPlayerScoreLabels(int playerNumber) {
1575        ArrayList<JLabel> scoreLabels = (playerNumber == 1) ? player1ScoreLabels
1576                : player2ScoreLabels;
1577        return scoreLabels;
1578    }
1579
1580    /**
1581     * Get all turn scores for a specified player
1582     *
1583     * @param playerNumber
1584     *            The number of the specified player
1585     * @return An ArrayList<JLabel> of the player scores
1586     */
1587    public ArrayList<JLabel> getPlayerScores(int playerNumber) {
1588        ArrayList<JLabel> scores = (playerNumber == 1) ? player1Scores
1589                : player2Scores;
1590        return scores;
1591    }
1592
1593    /**
1594     * Initialize all sounds used for the roll, bank, and bonus roll actions.
1595     */
1596    private void getSounds() {
```

```
1597        // Initialize the roll sounds
1598        rollSounds.add(getClass().getResource("/sounds/roll1.wav"));
1599        rollSounds.add(getClass().getResource("/sounds/roll2.wav"));
1600        rollSounds.add(getClass().getResource("/sounds/roll3.wav"));
1601        rollSounds.add(getClass().getResource("/sounds/roll4.wav"));
1602
1603        // Initialize the bank sounds
1604        bankSounds.add(getClass().getResource("/sounds/bank1.wav"));
1605        bankSounds.add(getClass().getResource("/sounds/bank2.wav"));
1606        bankSounds.add(getClass().getResource("/sounds/bank3.wav"));
1607        bankSounds.add(getClass().getResource("/sounds/bank4.wav"));
1608
1609        // Initialize the bonus turn sound
1610        bonusSound = getClass().getResource("/sounds/bonus.wav");
1611    }
1612
1613    /**
1614     * Create a file menu with five sub menus:
1615     * Hint - displays the maximum
1616     * scoring dice combination for a given roll.
1617     * New Game - Resets the
1618     * application to the game mode options menu
1619     * Reset Game - Resets the game
1620     * with the current configuration
1621     * Reset High Score - Resets the stored high
1622     * score to 0
1623     * Exit - Closes the application
1624     *
1625     * @return The Farkle menu bar
1626     */
1627    public JMenuBar createFarkleMenuBar() {
1628
1629        /*************************************************
1630         * 1.2.10 - A menu shall be displayed at the top of the main GUI with
1631         * one main option, "File", and four five options: "Hint", "New Game",
1632         * "Restart Game", "Reset High Score", and "Quit".
1633         *************************************************/
1634        // Instantiate the JMenuBar
1635        JMenuBar menuBar = new JMenuBar();
1636        setJMenuBar(menuBar);
1637
1638        // Instantiate the JMenu and add it to the JMenu Bar
1639        JMenu fileMenu = new JMenu("File");
1640        menuBar.add(fileMenu);
1641
1642        // Create the JMenu Items and add them to the JMenu
1643        final JMenuItem hintAction = new JMenuItem("Hint");
1644        final JMenuItem newAction = new JMenuItem("New Game");
1645        final JMenuItem resetAction = new JMenuItem("Reset Game");
1646        final JMenuItem resetHighScoreAction = new JMenuItem("Reset High Score");
1647        final JMenuItem exitAction = new JMenuItem("Exit");
1648        fileMenu.add(hintAction);
1649        fileMenu.add(newAction);
1650        fileMenu.add(resetAction);
1651        fileMenu.add(resetHighScoreAction);
1652        fileMenu.add(exitAction);
1653
```

```
1654        /*
1655         * Add a menu listener that determines if the hint, new, and reset
1656         * actions are available whenever the file menu is opened
1657         */
1658        fileMenu.addMenuListener(new MenuListener() {
1659
1660            @Override
1661            public void menuSelected(MenuEvent e) {
1662
1663                /*************************************************
1664                 * 1.2.10.e - The "Hint" option shall only be available after a
1665                 * player has rolled, and before that player has selected any
1666                 * dice.
1667                 *************************************************/
1668                hintAction.setEnabled(controller.isHintAvailable());
1669
1670                /*************************************************
1671                 * 1.2.10.g - The "New Game" option and the "Reset Game" option
1672                 * shall be disabled while the computer is taking its turn.
1673                 *************************************************/
1674                newAction.setEnabled(controller.isResetOrNewGameAvailable());
1675                resetAction.setEnabled(controller.isResetOrNewGameAvailable());
1676
1677                // Determine if the reset high score menu option is available
1678                resetHighScoreAction.setEnabled(controller.isResetHighScoreAvailable());
1679            }
1680
1681            @Override
1682            public void menuDeselected(MenuEvent e) {
1683            }
1684
1685            @Override
1686            public void menuCanceled(MenuEvent e) {
1687            }
1688
1689        });
1690
1691        /*************************************************
1692         * 1.2.10.a If the user selects the Â“New GameÂ” option, the select game
1693         * mode option box is displayed.
1694         *************************************************/
1695        if (newAction.getActionListeners().length == 0) {
1696            newAction.addActionListener(new ActionListener() {
1697                @Override
1698                public void actionPerformed(ActionEvent arg0) {
1699                    controller.endGame(false, true, false);
1700                }
1701            });
1702        }
1703
1704        /*************************************************
1705         * 1.2.10.b If the user selects the Â“Restart GameÂ” option, the current
1706         * game with all current configurations (player mode, player names, and
1707         * player types) is restarted.
1708         *************************************************/
1709        if (resetAction.getActionListeners().length == 0) {
1710            resetAction.addActionListener(new ActionListener() {
```

```
1711                @Override
1712                public void actionPerformed(ActionEvent arg0) {
1713                    controller.endGame(true, false, false);
1714                }
1715            });
1716        }
1717
1718        /*****************************************************
1719         * 1.2.10.f - If the user selects the "Reset High Score" option, the
1720         * high score is reset to 0.
1721         *****************************************************/
1722        if (resetHighScoreAction.getActionListeners().length == 0) {
1723            resetHighScoreAction.addActionListener(new ActionListener() {
1724                @Override
1725                public void actionPerformed(ActionEvent e) {
1726                    controller.resetHighScore();
1727                }
1728            });
1729        }
1730
1731        /*****************************************************
1732         * 1.2.10.c If the user selects the "Â“Quit"Â” option, the application is
1733         * closed.
1734         *****************************************************/
1735        if (exitAction.getActionListeners().length == 0) {
1736            exitAction.addActionListener(new ActionListener() {
1737                @Override
1738                public void actionPerformed(ActionEvent arg0) {
1739                    FarkleUI.this.dispose();
1740                }
1741            });
1742        }
1743
1744        /*****************************************************
1745         * 1.2.10.d - If the user selects the "Hint" option, the dice
1746         * combination for the highest possible score for the current roll is
1747         * displayed.
1748         *****************************************************/
1749        if (hintAction.getActionListeners().length == 0) {
1750            hintAction.addActionListener(new ActionListener() {
1751                @Override
1752                public void actionPerformed(ActionEvent arg0) {
1753                    List<Integer> highestScoringDieValues = controller
1754                            .getHighestScoringDieValues();
1755                    if (highestScoringDieValues.size() != 0) {
1756                        displayMessage(
1757                                "Select " + highestScoringDieValues
1758                                        + " to score "
1759                                        + controller.getHighestPossibleScore()
1760                                        + " points.", "Hint");
1761                    }
1762                }
1763            });
1764        }
1765
1766        return menuBar;
1767    }
```

```
1768
1769     /**
1770      * Get the farkle message
1771      *
1772      * @return The farkleMessage JDialog
1773      */
1774     public JDialog getFarkleMessage() {
1775         return farkleMessage;
1776     }
1777 }
1778
```

```java
package com.lotsofun.farkle;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.prefs.Preferences;

/**
 * The Game class serves as the Model for the Farkle application. This class is
 * responsible for tracking each player, setting high scores, storing the game
 * mode, calculating all scoring, and determining game outcomes.
 *
 * @author Brant Mullinix
 * @version 3.0.0
 */
public class Game {

    /** Stores the high score */
    private Preferences prefs;

    /** Tracks the current player during the game */
    private int currentPlayer = 1;

    /** Stores the game mode (single player or two player) */
    private GameMode gameMode;

    /** References to the players in a game */
    private Player[] players = new Player[2];

    /** Reference to the controller for the game */
    private FarkleController controller;

    /** Used to determine if a player is given a bonus turn */
    private boolean bonusTurn = false;

    /**
     * Constructor: Creates a new Game object, passes the GameMode and a
     * reference to the controller, and retrieves and sets the high score
     *
     * @param gMode
     *            The selected game mode for the game
     * @param controller
     *            The controller for the game that links the model to the view
     */
    public Game(GameMode gMode, FarkleController controller) {

        // Define the preferences file
        prefs = Preferences.userRoot().node(this.getClass().getName());

        // Set the game mode and controller fields
        this.gameMode = gMode;
        this.controller = controller;

        // Player 1 is instantiated regardless of the game mode
        players[0] = new Player(1);

        // If the game mode is set to two player, instantiate player 2.
```

```
58          if (gameMode == GameMode.MULTIPLAYER) {
59              players[1] = new Player(2);
60          }
61
62          // Set the high score label
63          controller.setUIHighScore(getHighScore());
64
65          // Initialize the turns
66          resetGame();
67      }
68
69      /**
70       * Calculate the score of a list of integers representing a roll of dice per
71       * the rules of this version of Farkle. This method calculates the scores in
72       * accordance with the following requirements:
73       *
74       * 6.0.0 - Scoring
75       * 6.1.0 - Each 1 rolled is worth 100 points
76       * 6.2.0 - Each 5 rolled is worth 50 points
77       * 6.3.0 - Three 1's are worth 1000 points
78       * 6.4.0 - Three of a kind of any value other than 1 is worth 100 times the
79       * value of the die (e.g. three 4's is worth 400 points).
80       * 6.5.0 - Four, five, or six of a kind is scored by doubling the three of a
81       * kind value for every additional matching die (e.g. five 3's would be
82       * scored as 300 X 2 X 2 = 1200.
83       * 6.6.0 - Three distinct doubles (e.g. 1-1-2-2-3-3) are worth 750 points.
84       * This scoring rule does not include the condition of rolling four of a
85       * kind along with a pair (e.g. 2-2-2-2-3-3 does not satisfy the three
86       * distinct doubles scoring rule).
87       * 6.7.0 - A straight (e.g. 1-2-3-4-5-6), which can only be achieved when
88       * all 6 dice are rolled, is worth 1500 points.
89       *
90       * @param roll
91       *            A List of Integers representing the roll of dice for which the
92       *            Farkle score will be calculated.
93       * @param checkHeldDice
94       *            A flag that is set to true when the calculated score should be
95       *            zero if any of the dice do not contribute to the score
96       * @return int - The calculated score for the list of dice
97       */
98      public int calculateScore(List<Integer> roll, boolean checkHeldDice) {
99
100         // Initialize the calculated score and set it to 0
101         Integer calculatedScore;
102
103         // If roll is not null, continue calculating the score
104         if (roll != null) {
105
106             // Set calculatedScore to 0
107             calculatedScore = 0;
108
109             // Calculate the roll score if roll is not empty
110             if (!roll.isEmpty()) {
111
112                 // Test to ensure all values are within the range of 1 to 6
113                 // (inclusive)
114                 boolean incorrectValuePresent = false;
```

```
115                 for (Integer value : roll) {
116                     if (!((value == 1) || (value == 2) || (value == 3)
117                             || (value == 4) || (value == 5) || (value == 6))) {
118                         incorrectValuePresent = true;
119                     }
120                 }
121 
122                 /*
123                  * If all values are within the accepted range of 1 to 6
124                  * (inclusive) calculate the score
125                  */
126                 if (!incorrectValuePresent) {
127 
128                     // Flag to check for one or two dice
129                     boolean oneOrTwoStrictDie = false;
130 
131                     // Determine the number of dice held
132                     int numberOfDieHeld = 0;
133                     numberOfDieHeld = roll.size();
134 
135                     // Flag to check for a straight
136                     boolean isStraight = true;
137 
138                     // Flags to check for three pair
139                     boolean isThreePair = true;
140                     int countOfPairs = 0;
141 
142                     /*
143                      * This array stores the count of each die in the roll.
144                      * Index 0 represents a die with value 1, etc.
145                      */
146                     int[] countedDie = new int[6];
147 
148                     /*
149                      * Determine the value of each die in the roll and add to
150                      * the total count for each die value in countedDie
151                      */
152                     for (int value : roll) {
153 
154                         /*
155                          * decrement value to get the proper die index and
156                          * increment the count for that value of die
157                          */
158                         countedDie[--value]++;
159                     }
160 
161                     /*
162                      * Calculate the score for the list of dice by looping
163                      * through the dice count for each value of die
164                      */
165                     for (int i = 0; i < countedDie.length; i++) {
166 
167                         // Get the count for the current die value
168                         Integer currentCount = countedDie[i];
169 
170                         // If the die is 2, 3, 4, or 6 (aka strict dice)
171                         if (i >= 1 && i != 4) {
```

```
172                         // If the die appears 1 or 2 times set the flag to
173                         // true
174                         if ((currentCount == 1) || (currentCount == 2)) {
175                             oneOrTwoStrictDie = true;
176                         }
177                     }
178
179                     /*
180                      * If the current count does not equal one, it can be
181                      * deduced that this roll does not include a straight.
182                      */
183                     if (currentCount != 1)
184                         isStraight = false;
185
186                     /*
187                      * If the current count does not equal 2 and does not
188                      * equal 0, it can be deduced that this roll does not
189                      * include three pair
190                      */
191                     if (currentCount != 2 && currentCount != 0)
192                         isThreePair = false;
193
194                     // Keep track of the total number of pairs
195                     if (currentCount == 2) {
196                         countOfPairs++;
197                     }
198
199                     // Temporary variable used for power calculations
200                     Double temp;
201
202                     // Add to the score based on the current die value
203                     switch (i) {
204
205                     /*
206                      * If the current die value is 1 (index 0), add 100 to
207                      * the score for every 1. If there are three or more,
208                      * add 1000 * 2 ^ (count - 3)
209                      */
210                     case 0:
211                         if (currentCount > 0 && currentCount < 3) {
212                             calculatedScore = calculatedScore
213                                     + currentCount * 100;
214                         } else if (currentCount >= 3) {
215                             temp = (Math.pow(2, (currentCount - 3)));
216                             calculatedScore = calculatedScore + 1000
217                                     * temp.intValue();
218                         }
219                         break;
220
221                     /*
222                      * If the current die value is 5 (index 4), add 50 to
223                      * the score for every 5. If there are three or more,
224                      * add 500 * 2 ^ (count - 3)
225                      */
226                     case 4:
227                         if (currentCount > 0 && currentCount < 3) {
228                             calculatedScore += currentCount * 50;
```

```
229                         } else if (currentCount >= 3) {
230                             temp = Math.pow(2, (currentCount - 3));
231                             calculatedScore = calculatedScore + 500
232                                     * temp.intValue();
233                         }
234                         break;
235 
236                     // The die value is 2, 3, 4, or 6.
237                     default:
238                         /*
239                          * If the count of the current die is greater than
240                          * 3, add the current die value * 100 * 2 ^ (count -
241                          * 3)
242                          */
243                         if (currentCount >= 3) {
244                             temp = Math.pow(2, (currentCount - 3));
245                             calculatedScore += (i + 1) * 100
246                                     * temp.intValue();
247                         }
248 
249                         /*
250                          * Else if the is straight flag is true, and the die
251                          * value is 6 (index 5) and 6 dice were rolled, it
252                          * is confirmed that a straight was rolled, set the
253                          * calculated score to 1500
254                          */
255                         else if (isStraight == true && i == 5
256                                 && roll.size() == 6) {
257                             calculatedScore = 1500;
258                         }
259 
260                         /*
261                          * Else if the is three pair flag is true, and the
262                          * die value is 6 (index 5) and 6 dice were rolled,
263                          * it is confirmed that 3 pairs was rolled, set the
264                          * calculated score to 750
265                          */
266                         else if (isThreePair == true && i == 5
267                                 && roll.size() == 6) {
268                             calculatedScore = 750;
269                         }
270 
271                     }
272 
273                 }
274 
275                 /*
276                  * If the checkHeldDice flag is true, we need to return 0 if
277                  * any of the dice do not contribute to the score.
278                  */
279                 if (checkHeldDice) {
280 
281                     /*
282                      * if the number of counted pairs is less than 3, set
283                      * the isThreePair flag to false.
284                      */
285                     if (countOfPairs != 3) {
```

```
286                         isThreePair = false;
287                     }
288
289                     /*
290                      * If a 2, 3, 4, or 6 occurs once or twice and the roll
291                      * does not contain 3 pair or a straight, set
292                      * calculatedScore to 0
293                      */
294                     if (oneOrTwoStrictDie && !isStraight && !isThreePair) {
295                         calculatedScore = 0;
296                     }
297
298                     // If no dice are held, set calculatedScpre to 0.
299                     if (numberOfDieHeld == 0) {
300                         calculatedScore = 0;
301                     }
302
303                 }
304
305                 /*
306                  * Else, at least one die value was outside of the accepted
307                  * range, set calculatedScore to 0
308                  */
309             } else {
310                 calculatedScore = 0;
311             }
312         }
313
314         // The list of integers representing the roll was null. set
315         // calculatedScore to 0
316     } else {
317         calculatedScore = 0;
318     }
319
320     // Return the calculated score
321     return calculatedScore;
322 }
323
324 /**
325  * Calculate the highest possible score for a given roll of the dice, and
326  * details the dice that must be selected to achieve that score.
327  *
328  * @param dice
329  *            The list of integers representing the roll of dice
330  * @return Object[] array - the first index holds the calculated highest
331  *         possible score for a given roll, and the second index is a list
332  *         of dice that must be selected to achieve that score.
333  */
334 public Object[] calculateHighestScore(List<Integer> dice) {
335
336     // The calculated highest score
337     int highestScorePossible = 0;
338
339     // The Object[] array that will be returned
340     Object[] highestScore = new Object[2];
341
342     // The list of dice that must be selected to produce the highest score
```

```
343         List<Integer> highestScoringCombination = new ArrayList<Integer>();
344 
345         /*
346          * If the dice list is not empty or null, calculate the score for all
347          * permutations of selected dice, and determine that which results in
348          * the highest score
349          */
350         if ((dice != null) && dice.size() > 0) {
351 
352             // Determine the number of permutations of selected dice
353             Integer magicNumber = (int) Math.pow(2, dice.size()) - 1;
354 
355             // Calculate the score for each permutation
356             for (int i = 1; i <= magicNumber; i++) {
357 
358                 // The list of dice in this permutation
359                 List<Integer> tempList = new ArrayList<Integer>();
360 
361                 // Compute the binary number for this permutation
362                 char[] binaryNumber = new char[dice.size()];
363                 Arrays.fill(binaryNumber, '0');
364                 char[] ch = Integer.toBinaryString(i).toCharArray();
365                 for (int c = 0; c < ch.length; c++) {
366                     binaryNumber[(ch.length) - (c + 1)] = ch[c];
367                 }
368 
369                 // Add the appropriate die from the roll for each 1 in
370                 // binaryNumber
371                 for (int j = 0; j < binaryNumber.length; j++) {
372                     if (binaryNumber[j] == '1') {
373                         tempList.add(dice.get(j));
374                     }
375                 }
376 
377                 // Calculate the score for this permutation
378                 int tempScore = calculateScore(tempList, false);
379 
380                 // Compare tempScore to the previously set highest calculated
381                 // score
382                 if (tempScore > highestScorePossible) {
383                     highestScorePossible = tempScore;
384                     highestScoringCombination.clear();
385                     highestScoringCombination = tempList;
386                 }
387             }
388         }
389 
390         /*
391          * Add the highest calculated score and that scores combination to the
392          * highest score array, and return it
393          */
394         highestScore[0] = highestScorePossible;
395         highestScore[1] = highestScoringCombination;
396         return highestScore;
397     }
398 
399     /**
```

```
400      * Set this turn's score to 0 and end the current player's turn
401      */
402     public void farkle() {
403
404         // Set the turn score to 0 for the current player
405         controller.setTurnScore(currentPlayer, getTurnNumberForCurrentPlayer(),
406                 0);
407
408         // End the current player's turn
409         getCurrentPlayer().endTurn(true);
410
411         // Get the next player
412         currentPlayer = getNextPlayer();
413
414         // Set the next player's roll score to 0
415         processHold(0);
416     }
417
418     /**
419      * Set this turn's score to the total of all rolls and end the current
420      * player's turn.
421      *
422      * @return int - Game score for the current player when this method was
423      *         called
424      */
425     public int bank() {
426
427         // The game score for the current player
428         int retVal = 0;
429
430         // Set the turn score for the current player in the controller
431         controller.setTurnScore(currentPlayer, getTurnNumberForCurrentPlayer(),
432                 getRollScores());
433
434         // End the current player's turn
435         getCurrentPlayer().endTurn(false);
436
437         // Retrieve the game score for the current player
438         retVal = getCurrentPlayer().getGameScore();
439
440         // Set the new current player
441         currentPlayer = getNextPlayer();
442
443         // Return the game score
444         return retVal;
445     }
446
447     /**
448      * Get the integer index of the next player in the players array
449      *
450      * @return int - index of the next player
451      */
452     public int getNextPlayer() {
453
454         /*
455          * If gameMode is Multiplayer, return the index of the player that is
456          * not current
```

```
457          */
458         if (gameMode == GameMode.MULTIPLAYER) {
459             return (currentPlayer == 1) ? 2 : 1;
460         } else {
461 
462             // gameMode is Singleplayer return the index for player 1
463             return 1;
464         }
465     }
466 
467     /**
468      * Add the provided score to the map of roll scores for the current turn
469      *
470      * @param score
471      *             - The roll score to add
472      */
473     public void processHold(int score) {
474 
475         Player player = getCurrentPlayer();
476         player.scoreRoll(score);
477     }
478 
479     /**
480      * Increment the current roll number of the current player
481      */
482     public void processRoll() {
483 
484         getCurrentPlayer()
485                 .setRollNumber(getCurrentPlayer().getRollNumber() + 1);
486     }
487 
488     /**
489      * Get the total score of all the rolls for this turn
490      *
491      * @return int the total score of all rolls for this turn
492      */
493     public int getRollScores() {
494         return getCurrentPlayer().getRollScores();
495     }
496 
497     /**
498      * Get the current player
499      *
500      * @return the current player
501      */
502     public Player getCurrentPlayer() {
503         return players[currentPlayer - 1];
504     }
505 
506     /**
507      * Set the current player
508      *
509      * @param currentPlayer
510      *             The index of the player to set as current (1 or 2)
511      */
512     public void setCurrentPlayer(int currentPlayer) {
513         this.currentPlayer = currentPlayer;
```

```
514     }
515
516     /**
517      * Get the game mode of the current game
518      *
519      * @return GameMode.SINGLEPLAYER or GameMode.MULTIPLAYER
520      */
521     public GameMode getGameMode() {
522         return gameMode;
523     }
524
525     /**
526      * Get the array of players for the current game
527      *
528      * @return Player[] array of players for the current game
529      */
530     public Player[] getPlayers() {
531         return players;
532     }
533
534     /**
535      * Get the high score
536      *
537      * @return int high score saved in Preferences
538      */
539     public int getHighScore() {
540         return prefs.getInt("HighScore", 0);
541     }
542
543     /**
544      * Set the high score and save it in preferences
545      *
546      * @param highScore
547      *            integer to save as the high score in preferences
548      */
549     public void setHighScore(int highScore) {
550         prefs.putInt("HighScore", highScore);
551     }
552
553     /**
554      * Reset the stored high score to 0.
555      */
556     public void resetHighScore() {
557         prefs.putInt("HighScore", 0);
558     }
559
560     /**
561      * Set all the turn scores, roll scores, and game scores for all players in
562      * the Game object to 0, and set the first player as the current player
563      */
564     public void resetGame() {
565
566         // For each player, reset the roll scores, turn scores, and game scores
567         for (Player player : players) {
568             if (null != player) {
569                 player.setTurnNumber(1);
570                 player.resetTurnScores();
```

```
571                 player.resetRollScores();
572                 player.setGameScore(0);
573                 player.setRollNumber(0);
574             }
575         }
576 
577         // Set player 1 as the current player
578         this.setCurrentPlayer(1);
579     }
580 
581     /**
582      * Set the name of a chosen player
583      *
584      * @param playerNumber
585      *            Player's position - NOT AN INDEX (1 or 2)
586      * @param name
587      *            The string used to set the player's name
588      */
589     public void setPlayerName(int playerNumber, String name) {
590         if (null != name && playerNumber >= 1 && playerNumber <= 2) {
591             this.players[playerNumber - 1].setPlayerName(name);
592         }
593     }
594 
595     /**
596      * Get the turn number for the current player
597      *
598      * @return int - Turn number for the current player
599      */
600     public int getTurnNumberForCurrentPlayer() {
601         return this.getCurrentPlayer().getTurnNumber();
602     }
603 
604     /**
605      * Get the game score for the current player
606      *
607      * @return int - Total game score for the current player
608      */
609     public int getGameScoreForCurrentPlayer() {
610         return this.getCurrentPlayer().getGameScore();
611     }
612 
613     /**
614      * Get the player number for the current player
615      *
616      * @return int - Player number for the current player
617      */
618     public int getPlayerNumberForCurrentPlayer() {
619         return this.getCurrentPlayer().getPlayerNumber();
620     }
621 
622     /**
623      * Get the game score for a chosen player
624      *
625      * @param playerNumber
626      *            The player for which the game score is sought
627      * @return int - Total game score for the chosen player
```

```
628      */
629     public int getGameScoreForPlayer(int playerNumber) {
630         return players[playerNumber - 1].getGameScore();
631     }
632 
633     /**
634      * Get the name of a chosen player
635      *
636      * @param playerNumber
637      *            The player for which the name is sought
638      * @return String - The name of the chosen player
639      */
640     public String getPlayerName(int playerNumber) {
641         return players[playerNumber - 1].getPlayerName();
642     }
643 
644     /**
645      * Get the player type for the current player
646      *
647      * @return The player type for the current player (PlayerType.USER or
648      *         PlayerType.COMPUTER)
649      */
650     public PlayerType getPlayerTypeForCurrentPlayer() {
651         return this.getCurrentPlayer().getType();
652     }
653 
654     public PlayerType getPlayerTypeForPlayer(int playerNumber) {
655         return players[playerNumber - 1].getType();
656     }
657 
658     /**
659      * Set the player type for a chosen player
660      *
661      * @param playerNumber
662      *            The player for which the type is to be set
663      * @param playerType
664      *            The type to set the chose player to (PlayerType.USER or
665      *            PlayerType.COMPUTER)
666      */
667     public void setPlayerType(int playerNumber, PlayerType playerType) {
668         players[playerNumber - 1].setType(playerType);
669     }
670 
671     /**
672      * Get the bonusTurn flag
673      *
674      * @return boolean - True if the current turn is a bonus turn, false
675      *         otherwise
676      */
677     public boolean isBonusTurn() {
678         return bonusTurn;
679     }
680 
681     /**
682      * Set the bonustTurn flag
683      *
684      * @param isBonusTurn
```

```
685      *            - True if the current turn is a bonus turn, false otherwise
686      */
687     public void setBonusTurn(boolean isBonusTurn) {
688         this.bonusTurn = isBonusTurn;
689     }
690 
691     /**
692      * Gets a String array with the winning player's name and score. Array will
693      * be of length 3 should there be a tie: player1, player2, score
694      *
695      * @return String[] array with the winning player's information
696      */
697     public String[] getWinningPlayerInfo() {
698 
699         // If this is sinle player mode, return the player's name and game score
700         if (this.gameMode != GameMode.MULTIPLAYER) {
701             return new String[] { getPlayerName(1),
702                     "" + getGameScoreForPlayer(1) };
703 
704             // Else determine the winner and return that player's name and score
705         } else {
706 
707             // Get the score for each player
708             int player1Score = getGameScoreForPlayer(1);
709             int player2Score = getGameScoreForPlayer(2);
710 
711             // If player 1 wins, return player 1's name and score
712             if (player1Score > player2Score) {
713                 return new String[] { getPlayerName(1),
714                         "" + getGameScoreForPlayer(1) };
715 
716                 // Else if player 2 wins, return player 2's name and score
717             } else if (player1Score < player2Score) {
718                 return new String[] { getPlayerName(2),
719                         "" + getGameScoreForPlayer(2) };
720 
721                 // Else there was a tie, return both players' names and the
722                 // score
723             } else {
724                 return new String[] { getPlayerName(1), getPlayerName(2),
725                         "" + getGameScoreForPlayer(1) };
726             }
727         }
728     }
729 }
730 
```

GameMode.java

```java
package com.lotsofun.farkle;

/**
 * Simple Enum for Game Modes
 * SINGLEPLAYER - Single player game
 * MULTIPLAYER - Two player game
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public enum GameMode {
    SINGLEPLAYER, MULTIPLAYER;
}

```

```java
package com.lotsofun.farkle;

import java.util.ArrayList;
import java.util.HashMap;

/**
 * The Player class creates a player object that tracks the roll scores, game
 * score, player number, player type, player name, turn number, roll number,
 * turn scores, and game score for this player.
 *
 * @author Brant Mullinix
 * @version 3.0.0
 */
public class Player {

    /** The hash map of roll scores for the player */
    private HashMap<Integer, Integer> rollScore = new HashMap<Integer, Integer>();

    /** This player's number */
    private int playerNumber;

    /** The turn number for this player */
    private int turnNumber = 1;

    /** The roll number for this player */
    private int rollNumber;

    /** The game score for this player */
    private int gameScore = 0;

    /** The list of turn scores for this player */
    private ArrayList<Integer> turnScores;

    /** The PlayerType for this player */
    private PlayerType type;

    /** The name for this player */
    private String playerName;

    /**
     * Constructor: Takes player number and initializes the turnScores
     *
     * @param playerNumber
     *            The player number for this player.
     */
    public Player(int playerNumber) {
        // Set this player's number
        this.playerNumber = playerNumber;

        // Initialize the type to PlayerType.USER
        type = PlayerType.USER;

        // Initialize the turn scores
        turnScores = new ArrayList<Integer>();
    }

    /**
```

```
58      * Puts the score in to the map for the given roll
59      *
60      * @param score
61      *            The integer score to enter into the rollScore map
62      */
63     public void scoreRoll(int score) {
64         rollScore.put(rollNumber, score);
65     }
66
67     /**
68      * Get the sum of all the roll scores
69      *
70      * @return integer sum of all the roll scores
71      */
72     public int getRollScores() {
73
74         // Variable used to sum the roll scores
75         int currentTurnScore = 0;
76
77         // Sum the roll scores, and return the sum
78         for (int i : this.getRollScore().values()) {
79             currentTurnScore += i;
80         }
81         return currentTurnScore;
82     }
83
84     /**
85      * Sets the appropriate turn's score, increments turnNumber, resets
86      * rollNumber, currentPlayer, and rollSCore
87      *
88      * @param didFarkle
89      *            Set to true if the end of the turn was caused by a Farkle
90      */
91     public void endTurn(boolean didFarkle) {
92         // If the turn ended on a Farkle, set add 0 to the turn scores list
93         if (didFarkle) {
94             turnScores.add(0);
95         }
96
97         // Else, add the sum of the roll scores to the list
98         else {
99             turnScores.add(getRollScores());
100        }
101
102        // Add the turn score to this player's game score
103        gameScore += turnScores.get(turnNumber - 1);
104
105        // Increment the turn number
106        turnNumber++;
107
108        // Reset the roll number to 0
109        rollNumber = 0;
110
111        // Clear the map of roll scores
112        rollScore.clear();
113    }
114
```

```
115     /**
116      * Get this player's player number
117      *
118      * @return integer representing the player number
119      */
120     public int getPlayerNumber() {
121         return playerNumber;
122     }
123
124     /**
125      * Get this player's current turn number
126      *
127      * @return integer representing the current turn number
128      */
129     public int getTurnNumber() {
130         return turnNumber;
131     }
132
133     /**
134      * Set the turn number for this player
135      *
136      * @param turnNumber
137      *             integer turn number to set
138      */
139     public void setTurnNumber(int turnNumber) {
140         this.turnNumber = turnNumber;
141     }
142
143     /**
144      * Get this player's current roll number
145      *
146      * @return int representing the current roll number
147      */
148     public int getRollNumber() {
149         return rollNumber;
150     }
151
152     /**
153      * Set this player's roll number
154      *
155      * @param rollNumber
156      *             integer roll number to set
157      */
158     public void setRollNumber(int rollNumber) {
159         this.rollNumber = rollNumber;
160     }
161
162     /**
163      * Get this player's total game score for the current game
164      *
165      * @return integer representing the current total game score
166      */
167     public int getGameScore() {
168         return gameScore;
169     }
170
171     /**
```

```
172      * Set the game score for the current player
173      *
174      * @param gameScore
175      *            integer game score to set
176      */
177     public void setGameScore(int gameScore) {
178         this.gameScore = gameScore;
179     }
180
181     /**
182      * Get the list of turn scores for this player
183      *
184      * @return ArrayList<Integer> of the turn scores
185      */
186     public ArrayList<Integer> getTurnScores() {
187         return turnScores;
188     }
189
190     /**
191      * Set the list of turn scores for this player
192      *
193      * @param turnScores
194      *            ArrayList<Integer> of turn scores to set
195      */
196     public void setTurnScores(ArrayList<Integer> turnScores) {
197         this.turnScores = turnScores;
198     }
199
200     /**
201      * Clear the list of turn scores for this player.
202      */
203     public void resetTurnScores() {
204         this.turnScores.clear();
205     }
206
207     /**
208      * Get the player type of this player
209      *
210      * @return PlayerType for this player
211      */
212     public PlayerType getType() {
213         return type;
214     }
215
216     /**
217      * Set the player type for this player
218      *
219      * @param type
220      *            PlayerType.USER or PlayerType.COMPUTER
221      */
222     public void setType(PlayerType type) {
223         this.type = type;
224     }
225
226     /**
227      * Get the map of roll scores for this player
228      *
```

```
229      * @return HashMap<Integer, Integer> of roll scores for this player
230      */
231     public HashMap<Integer, Integer> getRollScore() {
232         return rollScore;
233     }
234 
235     /**
236      * Set the map of roll scores for this player
237      *
238      * @param rollScore
239      *            roll scores to set
240      */
241     public void setRollScore(HashMap<Integer, Integer> rollScore) {
242         this.rollScore = rollScore;
243     }
244 
245     /**
246      * Clear the roll scores for this player
247      */
248     public void resetRollScores() {
249         this.rollScore.clear();
250     }
251 
252     /**
253      * Get the name for this player
254      *
255      * @return String player name
256      */
257     public String getPlayerName() {
258         return playerName;
259     }
260 
261     /**
262      * Set the name for this player
263      *
264      * @param playerName
265      *            name to set
266      */
267     public void setPlayerName(String playerName) {
268         this.playerName = playerName;
269     }
270 }
271 
```

PlayerType.java

```java
package com.lotsofun.farkle;

/**
 * Simple enum for player types
 * USER - live player
 * COMPUTER - computer player
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public enum PlayerType {
    USER, COMPUTER;
}

```

Testing Documentation

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
1	1.1.0	Black Box	Upon opening the application, the user is greeted with an option box that includes all configuration options for gameplay. These options include “1 Player Mode”, “2 Player Mode”, “Human Opponent” (if two player mode is selected), “Computer Opponent” (if two player mode is selected), and text fields to enter the associated player names. Also included is a “Start” button and a “Close” button (both of which are always enabled). This option dialog box should pop up over the main GUI set to a solid green color.	User opens the application.	An option box pops up that includes all configuration options for gameplay (“1 Player Mode”, “2 Player Mode”, “Human Opponent” (if two player mode is selected), “Computer Opponent” (if two player mode is selected), and text fields to enter the associated player names.) Also included is a “Start” button and a “Close” button (both of which are always enabled).	yes
2	1.1.1	Black Box	If the user selects the “Close” button at any time, the application closes.	User selects"Close" from the game mode option box.	The application immediately closes.	yes
3	1.1.2	Black Box	The “1 Player Mode” is highlighted by default when the application is first opened, and a blank text field for player one’s name is displayed.	User opens the application.	The "1 Player Mode" option is highlighted and a blank text field for player one's name is displayed.	yes
4	1.1.2.a	Black Box	If the user selects the “Start” button with “1 Player Mode” highlighted and the “Player One Name” field empty, the one player GUI opens with the name "Jacob" assigned to player one.	User selects the "Start" button from the game mode options menu without entering a name in the "Player One" name text box.	The single player mode GUI opens with the name "Jacob" assigned to player one.	yes
5	1.1.2.b	Black Box	If the user selects the “Start” button with “1 Player Mode” highlighted and a name supplied in the “Player One Name” text field, the one player GUI opens with the provided name assigned to player one.	User selects the "Start" button from the game mode options menu with a name entered in the "Player One" name text box.	The single player mode GUI opens with the supplied name assigned to player one.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
6	1.1.3	Black Box & White Box	If the user highlights the “2 Player Mode” option, the “1 Player Mode” option is deselected, and two more options appear (“Human Opponent” and “Computer Opponent”). The “Human Opponent” option is highlighted by default.	User selects the "2 Player Mode" option from the game mode options menu.	"1 Player Mode" is no longer highlighted, and options for the opponent type are displayed ("Human Opponent" and "Computer looibebt"). "Human Opponent" is selected by default.	yes
7	1.1.4	Black Box & White Box	When the “Human Opponent” option is highlighted, two text fields are displayed, labeled “Player One Name”, and “Player Two Name”.	User selects "2 Player Mode" and "Human Opponent" from the game mode option menu.	Two text fields are displayed, "Player One Name", and "Player Two Name".	yes
8	1.1.4.a	Black Box	If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field is empty, the “Player Two Name” field is empty, and the user selects the “Start” button, the two player mode GUI is opened, and the names "Jacob" and "Brant" are assigned to player 1 and player 2, respectively.	User selects "2 Player Mode" and "Human Opponent" from the game mode option menu, and selects the "Start" button without entering names for player one or player two.	The two player mode GUI opens with the names "Jacob" and "Brant" assigned to player one and player two, respectively.	yes
9	1.1.4.b	Black Box	If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field is empty, the “Player Two Name” field contains a name, and the user selects the “Start” button, the two player mode GUI is opened, the name "Jacob" is assigned to player 1, and the supplied name is assigned to player 2.	User selects "2 Player Mode" and "Human Opponent" from the game mode option menu, provides a name for player two, and selects the "Start" button without entering a name for player one.	The two player mode GUI opens with the the supplied name assigned to player two, and the name "Jacob" assigned to player one.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
10	1.1.4.c	Black Box	If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field contains a name, the “Player Two Name” field is empty, and the user selects the “Start” button, the two player mode GUI is opened, the name "Brant" is assigned to player 2, and the supplied name is assigned to player 1.	User selects "2 Player Mode" and "Human Opponent" from the game mode option menu, provides a name for player one, and selects the "Start" button without entering a name for player two.	The two player mode GUI opens with the the supplied name assigned to player one, and the name "Brant" assigned to player two.	yes
11	1.1.4.d	Black Box	If “Two Player Mode” is highlighted, “Human Opponent” is highlighted, the “Player One Name” field contains a name, the “Player Two Name” field contains a name, and the user selects the “Start” button, the two player mode GUI is opened, and the supplied names are assigned to player 1 and player 2 accordingly.	User selects "2 Player Mode" and "Human Opponent" from the game mode option menu, supplies names for player one and player two, and selects the "Start" button without entering names for player one or player two.	The two player mode GUI opens with the supplied names assigned to player one and player two, respectively.	yes
12	1.1.5	Black Box & White Box	When the “Computer Opponent” option is highlighted, the text fields for “Player One Name” and “Player Two Name” are displayed, but the “Player Two Name” field is disabled, and “Computer” is supplied for the “Player Two Name”.	User selects "2 Player Mode" and "Computer Opponent" from the game mode options menu.	Two text fields are displayed, "Player One Name", and "Player Two Name". The "Player Two Name" text field is filled in with the name "Computer" and is disabled.	yes
13	1.1.5.a	Black Box	If “Two Player Mode” is highlighted, “Computer Opponent” is highlighted, the “Player One Name” field is empty, and the user selects the “Start” button, the two player mode GUI is opened, the name "Jacob" is assigned to player 1, and “Computer” is assigned to player 2.	User selects "2 Player Mode" and "Computer Opponent", and the "Start" button from the game mode options menu without supplying a name for player one.	The two player GUI opens with the name "Jacob" assigned to player one, and the name "Computer" assigned to player two.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
14	1.1.5.b	Black Box	If “Two Player Mode” is highlighted, “Computer Opponent” is highlighted, the “Player One Name” field contains a name, and the user selects the “Start” button, the two player mode GUI is opened, the supplied name is assigned to player 1, and “Computer” is assigned to player 2.	User selects "2 Player Mode" and "Computer Opponent", and the "Start" button from the game mode options menu and supplies a name for player one.	The two player GUI opens with the supplied name assigned to player one, and the name "Computer" assigned to player two.	yes
15	1.2.1	Black Box	The center of the screen shall display the six dice used during gameplay. These dice shall display the name of the game, “Farkle”, until the user selects the roll button for the first time.	User starts a one player mode, or a two player mode game.	The dice display the name "Farkle" until the user selects the roll button for the first time.	yes
16	1.2.2	Black Box	The total turn score shall be displayed above the dice in the center of the screen, and the score for the selected dice of the current roll shall be displayed directly below the turn score in the center of the screen. This score shall be updated as each die is selected.	After a given roll, the user selects any combination of scoring dice.	If all selected dice contribute to the score, the calculated score for the selected dice is displayed just above the dice. If any of the selected dice are not scorable, 0 is displayed just above the dice.	yes
17	1.2.3	Black Box	Rules for the scoring combinations shall be displayed on the right side of the screen.	User starts a one player mode, or a two player mode game.	An image summarizing scoring rules is displayed on the right side of the screen.	yes
18	1.2.4	Black Box	A “Roll Dice” button shall be displayed below the dice in the middle of the screen.	User starts a one player mode, or a two player mode game.	A "Roll Dice" button is displayed below the dice in the middle of the screen.	yes
19	1.2.5	Black Box	A “Bank Score” button shall be displayed below the dice in the middle of the screen, and shall initially be disabled.	User starts a one player mode, or a two player mode game.	A "Bank Score" button is displayed below the dice in the middle of the screen, and is initially disabled.	yes
20	1.2.6	Black Box	After each roll, dice that have previously been selected, scored, and locked shall be shaded to indicate they will not be available on the next roll, and the turn accumulated score shall be displayed above the dice.	At the conclusion of a roll, the user selects scoring dice and subsequently selects the "Roll Dice" button.	All selected dice are shaded to indicate that they are locked, and the remaining dice are rolled.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
21	1.2.7	Black Box	If any roll results in 0 points, the word “Farkle” is prominently displayed, and 0 points is displayed in the accumulated turn score above the dice. The dice retain their current values that resulted in the Farkle. After the Farkle message is displayed, the dice still retain the values that resulted in the Farkle, but all dice are unlocked and play passes to the next player (in two player mode), or to the next turn (in one player mode).	The user selects "Roll Dice", and the resulting roll results in a score of 0 for every possible combination of the rolled dice.	A message dialog box pops up displaying the word "Farkle". The accumulated turn score displayed above the dice is set to 0. The dice retain the values that resulted in the farkle until the next turn is taken.	yes
22	1.2.8	Black Box	A “Select All” button shall be displayed below the dice in the middle of the screen, and shall be initially disabled.	User starts a one player mode, or a two player mode game.	A "Select All" button is displayed below the dice in the middle of the screen, and is disabled at the start of the game.	yes
23	1.2.9	Black Box	During a current roll, current dice selected by the user shall be indicated with a yellow border around each selected die, and the score for the currently selected dice shall be updated above the dice.	User rolls the dice, and selects at least one die after the roll.	All currently selected dice are indicated by being boxed in a yellow border. The score of the selected dice is displayed above the dice.	yes
24	1.2.10	Black Box	A menu shall be displayed at the top of the main GUI with one main option, “File”, and five sub options: “Hint”, “New Game”, “Restart Game”, “Reset High Score”, and “Quit”.	User starts a one player mode, or a two player mode game.	A menu is displayed with "File" as the main option, and four sub options: “Hint”, “New Game”, “Restart Game”, and “Quit”.	yes
25	1.2.10.a	Black Box	If the user selects the “New Game” option, the select game mode option box is displayed.	User selects the "New Game" option under the "File" main menu.	The game mode option box is displayed.	yes
26	1.2.10.b	Black Box & White Box	If the user selects the “Restart Game” option, the current game with all current configurations (player mode, player names, and player types) is restarted.	User selects the Restart Game" option under the "File" main menu.	The game is restarted with the current game configuration for player mode, player names, player types, etc.	yes
27	1.2.10.c	Black Box	If the user selects the “Quit” option, the application is closed.	User selects the "Quit" option under the "File" main menu.	The application immediately closes	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
28	1.2.10.d	Black Box	If the user selects the “Hint” option, the dice combination for the highest possible score for the current roll is displayed.	User selects the "Hint" option under the "File" main menu after rolling the dice.	The combination of dice that results in the maximum score for the current roll is displayed in a message box.	yes
29	1.2.10.e	Black Box	The “Hint” option shall only be available after a player has rolled, and before that player has selected any dice.	The user has yet to roll the dice, or the user has rolled the dice and selected one or more dice	The hint menu option is disabled	yes
30	1.2.10.f	Black Box & White Box	If the user selects the “Reset High Score” option, the high score is reset to 0.	The user selects the "Reset High Score" menu option	The high score is reset to 0 points	yes
31	1.2.10.g	Black Box	The “New Game” option and the “Reset Game” option shall be disabled while the computer is taking its turn.	The computer takes a turn in a two player mode against a computer opponent game.	The "New Game" and "Reset Game" options are disabled	yes
32	1.3.1	Black Box	The title of the window shall display: “Farkle – Single Player Mode”.	One player mode is started.	The title of the one player GUI window displays "Farkle - Single Player Mode".	yes
33	1.3.2	Black Box	The overall point total for the current game shall be displayed on the upper left hand corner of the screen, just below the player’s name.	One player mode is started.	The point total for the current game is initially displayed as 0. This total is updated after each turn.	yes
34	1.3.3	Black Box	The left side of the screen shall have an area to display the point total for each of the ten turns taken in single player mode.	One player mode is started.	The left hand of the screen displays ten separate turns, each of which are initially empty. After a turn, that turn is updated with the points accumulated during the turn.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
35	1.3.4	Black Box	The current turn shall be indicated by highlighting that turn on the left side of the screen. This turn shall be highlighted as soon as the previous turn ends (which occurs after the player selects the “Bank Score” button, or the Farkle message dialog box animation concludes), and before the player selects the “Roll Dice” button for the current turn.	One player mode is started.	Turn 1 is initially highlighted indicating it is the current turn. After the turn is complete, the next turn is highlighted.	yes
36	1.3.5	Black Box	The current highest achieved score shall be displayed on the lower left hand corner of the screen . This score shall initially be set to 0 points.	One player mode is started.	The current highest achieved overall score in one player mode is initially set to 0 points. If any game results in a higher overall score, the highest achieved score is set to that score.	yes
37	1.3.6	Black Box	The top of the left hand side of the screen shall display “Player: “, along with the provided name of the player.	One player mode is started.	The top of the left hand side of the screen displays "Player: ", along with the provided name of the player.	yes
38	1.4.1	Black Box	The title of the window shall display, “Farkle – Two Player Mode”.	Two player mode is started.	The title of the window shall display, “Farkle – Two Player Mode”.	yes
39	1.4.2	Black Box	The left side of the screen shall have an area to display the overall accumulated point total for each player. This takes the place of the area displaying the point total for each turn in the one player mode graphic user interface.	Two player mode is started.	Left hand side of screen displays overall accumulated point total for each player for the current game	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
40	1.4.2.a	Black Box	Each player shall be indicated in the following manner: “Player: “ along with the provided player’s name, or “Computer” if a “Computer Opponent” has been selected, followed by the running point total for the current game for that player.	Two player mode is started.	Each player shall be indicated in the following manner: “Player: “ along with the provided player’s name, or “Computer” if a “Computer Opponent” has been selected, followed by the running point total for the current game for that player.	yes
41	1.4.3	Black Box	The player whose turn it is shall be indicated by highlighting that player’s current turn on the left side of the screen. The turn shall be highlighted after the previous player’s roll ends and before the player selects the “Roll Dice” button for the first time for the current turn.	Two player mode is started.	The current player's turn is indicated by highlighting that player.	yes
42	1.4.4	Black Box	The first player to meet the minimum total point threshold required to win the game (equal to 10,000 points) shall be highlighted in a different color to indicate each subsequent player has one more turn to try and beat that player’s score.	A player in two player mode reaches 10,000 points.	The player to reach 10,000 points is indicated so by highlighting that player in a different color.	yes
43	1.4.5	Black Box	The turn totals for each player shall be displayed in a scroll pane below that player’s name and game score. This scroll pane shall initially display 5 turns, adding additional turns after they are taken. The scrolling ability shall be enabled at the beginning of the 11th turn.	A given player reaches the 11th turn during a game	Scrolling ability is enabled for that players turn panel, and is automatically scrolled to the bottom	yes
44	1.4.6	Black Box	After a player has surpassed the 10,000 point threshold, a message dialog box is displayed indicating the other player has one last turn to try and beat that player’s total score.	A player surpasses the 10,000 point threshold during a 2 player game	A message dialog box is displayed indicating the other player has one last turn to try and beat that player’s total score.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
45	1.5.1	Black Box	At the conclusion of the game, an option box shall be displayed with the player's overall score for the completed game (in one player mode), or the winner of the current game (in two player mode). This option box shall include three options: "Play Again?", "Main Menu", and "Quit".	A game in either one player mode or two player mode ends.	An option box shall be displayed with the player's overall score for the completed game (in one player mode), or the winner of the current game (in two player mode) that includes three options: "Play Again?", "Main Menu", and "Quit".	yes
46	1.5.1.a	Black Box & White Box	If the user selects the "Play Again?" button, the game will be restarted with all of the same configuration options of the previous game (player mode, player's names, and player types).	The user selects the "Play Again?" button from the conclusion of game option box.	The game is restarted with the current game configuration for player mode, player names, player types, etc.	yes
47	1.5.1.b	Black Box & White Box	If the user selects the "Main Menu" button, the select game mode option box will be displayed.	The user selects the "Main Menu" button from the conclusion of game option box.	The game mode option box is displayed.	yes
48	1.5.1.c	Black Box	If the user selects the "Quit" button, the application will immediately close.	The user selects the "Quit" button from the conclusion of game option box.	The application closes.	yes
49	1.5.1.d	Black Box	If the user selects the close button in the upper portion of the conclusion of game option box, the application will immediately close.	The user selects the close button from the conclusion of game option box.	The application closes.	yes
50	2.1.0	Black Box	When one player mode is selected, the one player mode graphic user interface is displayed with the name "Farkle" displayed on the dice, the "Bank Score" button disabled, the "Select All" button disabled, and turn number one highlighted. The user will have ten turns to try and get as many points as possible.	One player mode is started.	The one player mode GUI opens with the "Bank Score" and "Select All" buttons disabled, and "Farkle" displayed on the dice.	yes
51	2.1.1	Black Box	Each turn is taken according to the requirements of section 4.0.0.	One player mode is started.	Each turn follows the requirements of requirement 4.0.0	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
52	2.1.2	Black Box & White Box	The game ends at the conclusion of the tenth turn, and the player's score is compared to the current high score.	The tenth turn is completed in one player mode.	The conclusion of game option box is displayed.	yes
53	2.1.3	Black Box	If the player's score is greater than the current high score, a congratulatory message is displayed, and the player's score replaces the previous high score.	The tenth turn is completed in one player mode, and the player achieves a high score.	A congratulatory message is displayed, and the player's score replaces the previous high score.	yes
54	2.1.4	Black Box	The conclusion of game option box is displayed at the completion of the tenth turn.	The tenth turn is completed in one player mode.	The conlcusion of game option box is displayed.	yes
55	2.2.0	Black Box	When two player mode against a live person is selected, the two player mode graphic user interface is displayed with the name "Farkle" displayed on the dice, the "Bank Score" button disabled, the "Select All" button disabled, and player one highlighted indicating it is his or her turn.	Two player mode against a human opponent is started.	The two player mode GUI opens with the "Bank Score" and "Select All" buttons disabled, and "Farkle" displayed on the dice.	yes
56	2.2.1	Black Box	Each turn is taken according to the requirements of section 4.0.0. The current player for each turn is highlighted during that player's turn.	Two player mode against a human opponent is started.	Each turn follows the requirements of requirement 4.0.0, and the current player's turn is indicated by highlighting that player.	yes
57	2.2.2	Black Box	The first player to surpass 10,000 total points at the end of a given turn is highlighted in a different color.	A player surpases 10,000 points in two player mode agains a human opponent.	A player surpases 10,000 points and that player's turn ends.	yes
58	2.2.3	Black Box & White Box	The other player has one more turn to try and surpass the point total of the first player to surpass 10,000 points.	A player surpases 10,000 points in two player mode against a human opponent, and the turn ends passing control to the other player.	A player surpases 10,000 points and that player's turn ends.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
59	2.2.4	Black Box	The conclusion of game option box is displayed after a player wins.	A two player mode against a human opponent game ends with a winner.	The conclusion of game option box is displayed.	yes
60	2.3.0	Black Box	When two player mode against the computer is selected, the two player mode graphic user interface is displayed with the name “Farkle” displayed on the dice, the “Bank Score” button disabled, the “Select All” button disabled, and player one highlighted indicating it is his turn.	Two player mode against a computer opponent is started.	The two player mode GUI opens with the "Bank Score" and "Select All" buttons disabled, and "Farkle" displayed on the dice.	yes
61	2.3.1	Black Box	Each turn is taken according to the requirements of section 4.0.0. The current player for each turn is highlighted during that players turn.	Two player mode against a computer opponent is started.	Each turn follows the requirements of requirement 4.0.0, and the current player's turn is indicated by highlighting that player.	yes
62	2.3.2	Black Box	Decisions made during the computer player’s turn are chosen in accordance with requirements section 5.0.0.	The computer takes a turn in a two player mode against a computer opponent game.	The computer takes a turn in two player mode against a computer opponent.	yes
63	2.3.3	Black Box	The first player to surpass 10,000 total points at the end of a given turn is highlighted in a different color.	A player surpases 10,000 points in two player mode agains a computer opponent.	A player surpases 10,000 points and that player's turn ends.	yes
64	2.3.4	Black Box & White Box	The other player has one more turn to try and surpass the point total of the first player to surpass 10,000 points.	A player surpases 10,000 points in two player mode against a computer opponent, and the turn ends passing control to the other player.	A player surpases 10,000 points and that player's turn ends.	yes
65	2.3.5	Black Box	The conclusion of game option box is displayed after a player wins.	A two player mode against a computer opponent game ends with a winner.	The conclusion of game option box is displayed.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
66	3.1.0	Black Box & White Box	Farkle is played with six standard 6 sided dice with each side numbered from 1 through 6 (inclusive).	One player mode or two player mode is started.	Six dice are displayed in the center of the screen.	yes
67	3.2.0	Black Box & White Box	Each die that is rolled shall be assigned a random value from 1 to 6 (inclusive) at the conclusion of the roll.	A roll occurs during one player mode or two player mode.	The dice values after the roll are randomly determined, and fall within the range of 1 to 6 (inclusive).	yes
68	4.1.0	Black Box	At the beginning of the turn, the turn score is set to 0. The player selects the “Roll Dice” button, and all 6 dice are rolled in accordance with requirement 3.2.0. The “Select All” button is enabled after the initial roll takes place.	A previous turn ends and a new turn begins, and the player selects the "Roll Dice" button.	The turn score is 0, and all 6 dice are rolled. The "Select All" button is enabled.	yes
69	4.2.0	White Box	The resulting roll is analyzed according to requirement 6.0.0 to determine if the player farkled. A farkle occurs if the roll results in 0 points.	A player completes a roll of the dice.	If all combinations of the rolled dice result in 0 points, the player Farkles.	yes
70	4.3.0	Black Box	If the player did not farkle, he or she must select at least one scoring die. The score for the selected dice is calculated according to requirement 5.0.0, and is updated after each die selection. The score for the selected dice is displayed in accordance with section 1.2.9. If any of the selected dice are not scorable, a selected dice score of 0 is displayed and the “Roll Dice” and “Bank Score” buttons are disabled.	A player completes a roll of the dice, in which at least one die is scorable. The player then selects dice he or she wants scored for that roll.	If all selected dice contribute to the score, the calculated score for the selected dice is displayed below the dice, and the "Roll Dice" button is enabled. If any of the selected dice are not scorable, 0 is displayed below the dice and the "Roll Dice" button is disabled. If the turn score is greater than or equal to 300, the "Bank Score" button is enabled.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
71	4.4.0	Black Box	When all of the selected dice contribute to the point total for the roll, the “Roll Dice” button is enabled, and the roll point total is added to the turn score.	A player completes a roll of the dice, in which at least one die is scorable. The player then selects dice he or she wants scored for that roll. All selected dice are scorable.	The "Roll Dice" button is enabled, and the roll point total is added to the current turn score.	yes
72	4.5.0	Black Box	If the current turn score is greater than or equal to 300, the bank button is enabled.	A player completes a roll of the dice, in which at least one die is scorable. The player then selects dice he or she wants scored for that roll. All selected dice are scorable, and the turn score is greater than or equal to 300.	The "Bank Score" button is enabled.	yes
73	4.6.0	Black Box	If the player elects to roll again the selected dice are locked, the remaining dice are rolled, and the process returns to requirement 4.2.0.	A player completes a roll of the dice, in which at least one die is scorable. The player then selects dice he or she wants scored for that roll. All selected dice are scorable, and the player selects the "Roll Dice" button again.	The selected dice are locked (indicated by shading them), and the remaining dice are rolled.	yes
74	4.7.0	Black Box	If all six dice have been selected, and they all contribute to the turns point total, the player is issued a bonus roll indicated with a message in the current turn box. All selected and locked dice are deselected and unlocked, and the process returns to requirement 4.1.0.	A player completes a roll of the dice, and all rolled dice are scorable and selected.	A "Bonus Roll" message is displayed in the current turn for the current player, all selected and locked dice are deselected and unlocked, and the "Roll Dice" button is enabled.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
75	4.8.0	Black Box & White Box	If the player selects the bank button, the current turn point total is added to the player's game point total, and the turn is over.	A player's turn score is greater than or equal to 300, and that player selects the "Bank Score" button.	The turn score is displayed in the point total for that turn, and is added to the player's game total.	yes
76	4.9.0	Black Box	If the player farkles on any roll during the current turn, that player loses all points accumulated during the current turn, the farkle message is displayed, and the turn is over.	All combinations of dice after a player's roll result in 0 points.	A farkle message is displayed, the player is awarded 0 points for the turn, and play passes to the next turn (one player mode), or the next player (two player mode).	yes
77	4.10.0	Black Box & White Box	When the turn is over all dice are unlocked while continuing to display their last value , the "Roll Dice" button is enabled, the "Bank Score" button is disabled, the "Select All" button is disabled, the current turn point total is set to 0, the current roll point total is set to 0, and play passes to the next player (two player mode) or the next turn (single player mode) by highlighting the appropriate player or turn on the left hand side of the screen.	A turn ends via Farkle or selection of the "Bank Score" button.	All dice are unlocked and unselected while continuing to display their last value. Play passes to the next turn (one player mode) or the next player (two player mode) indicated via highlighting. The "Bank Score" and "Select All" buttons are disabled. The "Roll Dice" button is enabled. The current turn point total, and current roll point totals are set to 0.	yes
78	5.1.0	Black Box	The computer player takes its turn in accordance with requirement 4.0.0, and the dice selection as well as the decision to continue rolling the dice are made in accordance with the following requirements.	The computer takes a turn in a two player mode against a computer opponent game.	The computer decides to roll again based on the following requirements.	yes
79	5.2.0	Black Box	After each roll, the computer player always selects the maximum scoring combination of dice.	The computer takes a turn in a two player mode against a computer opponent game.	The computer selects all scorable dice after a roll.	yes
80	5.3.0	Black Box	If the current turn point total is less than the goal calculated in section 5.5.0, the computer always rolls again.	The computer's turn score after a given roll is less than 300.	The computer always rolls again.	yes

Testing Documentation

Test Case #	Requirement Tested	Method Used	Rationale	Input(s)	Expected Output	Passed?
81	5.4.0	Black Box	If the previous roll resulted in a bonus roll, the computer always rolls again.	The computer's roll results in a bonus roll	The computer always rolls again.	yes
82	5.5.0	Black Box	The computer's goal is calculated after each roll. This goal is pseudo-randomly selected as 300 fifty percent of the time, 600 thirty percent of the time, and 1000 twenty percent of the time.	The computer rolls the dice	The computer will elect to bank the dice if the score is greater than 300 points 50% of the time, if the score is greater than 600 80% of the time, and if the score is greater than 1000 100% of the time. Provided a bonus roll was not achievied on the previous roll.	yes
83	6.1.0	White Box	Each 1 rolled is worth 100 points.	A single 1 die is selected.	The player is given 100 points.	yes
84	6.2.0	White Box	Each 5 rolled is worth 50 points.	A single 5 die is selected.	The player is given 50 points.	yes
85	6.3.0	White Box	Three 1's are worth 1000 points.	Three 1 dice are selected	The player is given 1000 points.	yes
86	6.4.0	White Box	Three of a kind of any value other than 1 is worth 100 times the value of the die (e.g. three 4's is worth 400 points).	Three same value dice are selected (other than 1's).	The player is given the die value * 100 points.	yes
87	6.5.0	White Box	Four, five, or six of a kind is scored by doubling the three of a kind value for every additional matching die (e.g. five 3's would be scored as 300 X 2 X 2 = 1200).	Four, five, or six same value dice are selected.	The player is given the three of a kind value * 2^(number of mathing dice - 3).	yes
88	6.6.0	White Box	Three distinct doubles (e.g. 1-1-2-2-3-3) is worth 750 points. This scoring rule does not include the condition of rolling four of a kind along with a pair (e.g. 2-2-2-2-3-3 is worth does not satisfy the three distinct doubles scoring rule).	Three distinct pairs of dice are selected.	The player is given 750 points.	yes
89	6.7.0	White Box	A straight (e.g. 1-2-3-4-5-6), which can only be achieved when all 6 dice are rolled, is worth 1500 points.	A six dice straight is selected.	The player is given 1500 points.	yes

```java
package com.lotsofun.farkle;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import java.util.ArrayList;
import java.util.HashMap;
import org.junit.Test;

/**
 * Test the Farkle Controller class
 *
 * @author Curtis Brown
 * @version 3.0.0
 */
public class FarkleControllerTest {

    /**
     * Test the setUI method
     */
    @Test
    public void testSetUI() {
        // Create a new controller
        FarkleController controller = new FarkleController(true);

        // It's ui should be null
        assertEquals(controller.farkleUI, null);

        // setUI is called in by the FarklUI constructor
        FarkleUI ui = new FarkleUI(controller);
        FarkleUI currentUI = controller.farkleUI;

        // The ui instance ids should match
        assertEquals(ui, currentUI);

    }

    /**************************************************************************
     * 4.2.0 - The resulting roll is analyzed according to requirement 6.0.0 to
     * determine if the player farkled. A farkle occurs if the roll results in 0
     * points.
     *************************************************************************/
    /**
     * Test the farkle() method
     */
    @Test
    public void testFarkle() {

        // Single player test
        FarkleOptionsDialog options = new FarkleOptionsDialog(null);
        options.setGameMode(GameMode.SINGLEPLAYER);
        options.setPlayer1Name("player1");

        FarkleController controller = new FarkleController(true);
        FarkleUI ui = new FarkleUI(controller);
        controller.newGame(options);
        Game game = controller.farkleGame;
```

```
58
59          /**************************************************************************
60           * 3.1.0 - Farkle is played with six standard 6 sided dice with each
61           * side numbered from 1 through 6 (inclusive).
62           *************************************************************************/
63          ArrayList<Die> dice = ui.getDice(DieState.UNHELD);
64          assertEquals(dice.size(), 6);
65
66          // Set up random game conditions
67          ui.setRunningScore(50);
68          assertEquals(ui.getRunningScore(), 50);
69          ui.setRollScore(50);
70          assertEquals(ui.getRollScore(), 50);
71          assertEquals(ui.getDice(DieState.DISABLED).size(), 0);
72          assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
73
74          // Call farkle
75          controller.farkle();
76
77          // Check for farkle conditions
78          assertEquals(ui.getRunningScore(), 0);
79          assertEquals(ui.getRollScore(), 0);
80          assertEquals(ui.getDice(DieState.DISABLED).size(), 6);
81          assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
82
83          // Multiplayer test
84          options = new FarkleOptionsDialog(null);
85          options.setGameMode(GameMode.MULTIPLAYER);
86          options.setPlayer1Name("player1");
87          options.setPlayer2Name("player2");
88
89          controller = new FarkleController(true);
90          ui = new FarkleUI(controller);
91          controller.newGame(options);
92          game = controller.farkleGame;
93
94          // Set up random game conditions
95          ui.setRunningScore(50);
96          assertEquals(ui.getRunningScore(), 50);
97          ui.setRollScore(50);
98          assertEquals(ui.getRollScore(), 50);
99          assertEquals(ui.getDice(DieState.DISABLED).size(), 0);
100         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
101
102         // Call farkle
103         controller.farkle();
104
105         // Check for farkle conditions
106         assertEquals(ui.getRunningScore(), 0);
107         assertEquals(ui.getRollScore(), 0);
108         assertEquals(ui.getDice(DieState.DISABLED).size(), 6);
109         assertEquals(game.getPlayerNumberForCurrentPlayer(), 2);
110     }
111
112
113     /**************************************************************************
114      * 4.8.0 - If the player selects the bank button, the current turn point
```

```
115      * total is added to the player's game point total, and the turn is over.
116      ************************************************************************/
117     /**
118      * Test the bank() method
119      */
120     @Test
121     public void testBank() {
122
123         // Single player test
124         FarkleOptionsDialog options = new FarkleOptionsDialog(null);
125         options.setGameMode(GameMode.SINGLEPLAYER);
126         options.setPlayer1Name("player1");
127
128         FarkleController controller = new FarkleController(true);
129         FarkleUI ui = new FarkleUI(controller);
130         controller.newGame(options);
131         Game game = controller.farkleGame;
132
133         // Set up random game conditions
134         ui.setRunningScore(50);
135         assertEquals(ui.getRunningScore(), 50);
136         ui.setRollScore(50);
137         assertEquals(ui.getRollScore(), 50);
138         game.processHold(50);
139         assertEquals(ui.getGameScore(1), 0);
140         assertEquals(ui.getDice(DieState.DISABLED).size(), 0);
141         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
142
143         // Call bank
144         controller.bank();
145
146         // Check for bank conditions
147         assertEquals(ui.getRunningScore(), 0);
148         assertEquals(ui.getRollScore(), 0);
149         assertEquals(ui.getGameScore(1), 50);
150         assertEquals(ui.getDice(DieState.DISABLED).size(), 6);
151         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
152
153         // Multiplayer test
154         options = new FarkleOptionsDialog(null);
155         options.setGameMode(GameMode.MULTIPLAYER);
156         options.setPlayer1Name("player2");
157
158         controller = new FarkleController(true);
159         ui = new FarkleUI(controller);
160         controller.newGame(options);
161         game = controller.farkleGame;
162
163         // Set up random game conditions
164         ui.setRunningScore(50);
165         assertEquals(ui.getRunningScore(), 50);
166         ui.setRollScore(50);
167         assertEquals(ui.getRollScore(), 50);
168         game.processHold(50);
169         assertEquals(ui.getGameScore(1), 0);
170         assertEquals(ui.getDice(DieState.DISABLED).size(), 0);
171         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
```

```
172
173         // Call bank
174         controller.bank();
175
176         // Check for bank conditions
177         assertEquals(ui.getRunningScore(), 0);
178         assertEquals(ui.getRollScore(), 0);
179         assertEquals(ui.getGameScore(1), 50);
180         assertEquals(ui.getDice(DieState.DISABLED).size(), 6);
181         assertEquals(game.getPlayerNumberForCurrentPlayer(), 2);
182     }
183
184     /**************************************************************************
185      * 2.1.2 - The game ends at the conclusion of the tenth turn, and the
186      * player's score is compared to the current high score.
187      *************************************************************************/
188     /**
189      * Test the checkHighScore() method
190      */
191     @Test
192     public void testCheckHighScore() {
193
194         // Single player test
195         FarkleOptionsDialog options = new FarkleOptionsDialog(null);
196         options.setGameMode(GameMode.SINGLEPLAYER);
197         options.setPlayer1Name("player1");
198
199         FarkleController controller = new FarkleController(true);
200         FarkleUI ui = new FarkleUI(controller);
201         controller.newGame(options);
202         Game game = controller.farkleGame;
203
204         // Set up random game conditions
205         ui.setRunningScore(5000);
206         assertEquals(ui.getRunningScore(), 5000);
207         ui.setRollScore(5000);
208         assertEquals(ui.getRollScore(), 5000);
209         game.processHold(5000);
210         assertEquals(ui.getGameScore(1), 0);
211         game.setHighScore(5000);
212         assertEquals(game.getHighScore(), 5000);
213         assertEquals(ui.getDice(DieState.DISABLED).size(), 0);
214         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
215         assertFalse(controller.checkHighScore(game
216                 .getPlayerNumberForCurrentPlayer()));
217
218         // Call Bank
219         controller.bank();
220         assertEquals(ui.getRunningScore(), 0);
221         assertEquals(ui.getRollScore(), 0);
222         assertEquals(ui.getGameScore(1), 5000);
223         assertEquals(ui.getDice(DieState.DISABLED).size(), 6);
224         assertEquals(game.getPlayerNumberForCurrentPlayer(), 1);
225
226         // Set up another roll
227         game.processHold(50);
228         assertEquals(ui.getGameScore(1), 5000);
```

```
229         assertEquals(game.getHighScore(), 5000);
230         assertTrue(controller.checkHighScore(game
231                 .getPlayerNumberForCurrentPlayer()));
232     }
233 
234     /**********************************************************************
235      * 1.2.10.f If the user selects the “Reset High Score” option, the high
236      * score is reset to 0.
237      **********************************************************************/
238     /**
239      * Test the resetHighScore() method
240      */
241     @Test
242     public void testResetHighScore() {
243 
244         // Single player test
245         FarkleOptionsDialog options = new FarkleOptionsDialog(null);
246         options.setGameMode(GameMode.SINGLEPLAYER);
247         options.setPlayer1Name("player1");
248 
249         FarkleController controller = new FarkleController(true);
250         FarkleUI ui = new FarkleUI(controller);
251         controller.newGame(options);
252         Game game = controller.farkleGame;
253 
254         // Set the high score to 5000
255         controller.setUIHighScore(5000);
256         game.setHighScore(5000);
257         assertEquals(game.getHighScore(), 5000);
258 
259         // Call resetHighScore
260         controller.resetHighScore();
261 
262         // High score should be 0
263         assertEquals(ui.getHighScore(), 0);
264         assertEquals(game.getHighScore(), 0);
265     }
266 
267     /**********************************************************************
268      * 2.1.2 - The game ends at the conclusion of the tenth turn, and the
269      * player’s score is compared to the current high score.
270      **********************************************************************/
271     /**********************************************************************
272      * 2.2.3 - The other player has one more turn to try and surpass the point
273      * total of the first player to surpass 10,000 points.
274      **********************************************************************/
275     /**********************************************************************
276      * 2.3.4 - The other player has one more turn to try and surpass the point
277      * total of the first player to surpass 10,000 points.
278      **********************************************************************/
279     /**
280      * Test the tryToEndGame() method
281      */
282     @Test
283     public void testTryToEndGame() {
284 
285         // Single player test
```

```
286         FarkleOptionsDialog options = new FarkleOptionsDialog(null);
287         options.setGameMode(GameMode.SINGLEPLAYER);
288         options.setPlayer1Name("player1");
289 
290         FarkleController controller = new FarkleController(true);
291         @SuppressWarnings("unused")
292         FarkleUI ui = new FarkleUI(controller);
293         controller.newGame(options);
294         Game game = controller.farkleGame;
295 
296         // Test the isLastTurn flag for all turns less than 10
297         for (int i = 1; i < 10; i++) {
298             game.getPlayers()[0].setTurnNumber(i);
299             assertEquals(game.getTurnNumberForCurrentPlayer(), i);
300             controller.tryToEndGame();
301             assertEquals(controller.isLastTurn, false);
302         }
303 
304         // Test isLastTurn flag on the tenth turn
305         game.getPlayers()[0].setTurnNumber(10);
306         controller.tryToEndGame();
307         assertEquals(controller.isLastTurn, true);
308 
309         // Multiplayer test
310         options = new FarkleOptionsDialog(null);
311         options.setGameMode(GameMode.MULTIPLAYER);
312         options.setPlayer1Name("player1");
313         options.setPlayer2Name("player2");
314 
315         controller = new FarkleController(true);
316         ui = new FarkleUI(controller);
317         controller.newGame(options);
318         game = controller.farkleGame;
319 
320         // Test that the point threshold is 10000
321         assertEquals(controller.POINT_THRESHOLD, 10000);
322 
323         // Test the isLastTurn flag for all scores less than point threshold
324         for (int i = 0; i < 10000; i++) {
325             game.getPlayers()[0].setGameScore(i);
326             assertEquals(game.getGameScoreForPlayer(1), i);
327             controller.tryToEndGame();
328             assertFalse(controller.isLastTurn);
329         }
330 
331         // Check isLastTurn flag for a score equal to point threshold
332         game.getPlayers()[0].setGameScore(10000);
333         assertEquals(game.getGameScoreForPlayer(1), 10000);
334         controller.tryToEndGame();
335         assertTrue(controller.isLastTurn);
336     }
337 
338     /**************************************************************************
339      * 1.5.1.a - If the user selects the "Play Again?" button, the game will be
340      * restarted with all of the same configuration options of the previous game
341      * (player mode, player's names, and player types).
342      **************************************************************************/
```

```
343     /*************************************************************************
344      * 1.5.1.b - If the user selects the “Main Menu” button, the select game
345      * mode option box will be displayed (see section 1.1.0).
346      *************************************************************************/
347     /**
348      * Test the endGame() method
349      */
350     @Test
351     public void testEndGame() {
352
353         int testNumber = 0;
354         FarkleOptionsDialog options = null;
355         FarkleController controller = null;
356         FarkleUI ui = null;
357         Game game = null;
358
359         do {
360
361             // Single player test
362             options = new FarkleOptionsDialog(null);
363             options.setGameMode(GameMode.SINGLEPLAYER);
364             options.setPlayer1Name("player1");
365
366             controller = new FarkleController(true);
367             ui = new FarkleUI(controller);
368             controller.newGame(options);
369             game = controller.farkleGame;
370
371             // Set preconditions for game.resetGame test
372             game.processHold(50);
373             controller.bank();
374             assertEquals(game.getGameScoreForPlayer(1), 50);
375             game.processHold(100);
376             controller.bank();
377             assertEquals(game.getGameScoreForPlayer(1), 150);
378             game.processHold(150);
379             controller.bank();
380             assertEquals(game.getGameScoreForPlayer(1), 300);
381             assertEquals(game.getTurnNumberForCurrentPlayer(), 4);
382             assertEquals(game.getPlayers()[0].getTurnScores().size(), 3);
383
384             HashMap<Integer, Integer> rollScores = new HashMap<Integer, Integer>();
385             rollScores.put(1, 50);
386             rollScores.put(2, 100);
387             game.getPlayers()[0].setRollScore(rollScores);
388             assertEquals(game.getPlayers()[0].getRollScores(), 150);
389             assertEquals(ui.getTurnScore(1, 1), 50);
390
391             ArrayList<Die> dice = ui.getDice(DieState.UNHELD);
392             assertEquals(dice.size(), 0);
393             dice = ui.getDice(DieState.DISABLED);
394             assertEquals(dice.size(), 6);
395             for (Die d : dice) {
396                 d.setValue('3');
397                 assertEquals(d.getValue(), 3);
398             }
399
```

```
400             if (testNumber == 0) {
401
402                 // Call EndGame with replay = true
403                 controller.endGame(false, false, true);
404
405                 assertEquals(game.getGameScoreForPlayer(1), 0);
406                 assertEquals(game.getTurnNumberForCurrentPlayer(), 1);
407                 assertEquals(game.getPlayers()[0].getTurnScores().size(), 0);
408                 assertEquals(game.getPlayers()[0].getRollScores(), 0);
409                 assertEquals(ui.getTurnScore(1, 1), 0);
410
411                 dice = ui.getDice(DieState.UNHELD);
412                 assertEquals(dice.size(), 6);
413                 dice = ui.getDice(DieState.DISABLED);
414                 assertEquals(dice.size(), 0);
415                 for (Die d : dice) {
416                     assertEquals(d.getValue(), 0);
417                 }
418             }
419
420             if (testNumber == 1) {
421
422                 // Call EndGame with replay = false
423                 controller.endGame(false, false, false);
424                 controller.newGame(options);
425                 assertFalse(null == options);
426
427                 assertEquals(game.getGameScoreForPlayer(1), 0);
428                 assertEquals(game.getTurnNumberForCurrentPlayer(), 1);
429                 assertEquals(game.getPlayers()[0].getTurnScores().size(), 0);
430                 assertEquals(game.getPlayers()[0].getRollScores(), 0);
431                 assertEquals(ui.getTurnScore(1, 1), 0);
432
433                 dice = ui.getDice(DieState.UNHELD);
434                 assertEquals(dice.size(), 6);
435                 dice = ui.getDice(DieState.DISABLED);
436                 assertEquals(dice.size(), 0);
437                 for (Die d : dice) {
438                     assertEquals(d.getValue(), 0);
439                 }
440             }
441
442             testNumber++;
443         } while (testNumber < 2);
444     }
445 }
446
```

```java
package com.lotsofun.farkle;

import static org.junit.Assert.*;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Dialog.ModalityType;
import java.awt.event.MouseEvent;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * Complete JUnit testing of the FarkleOptionsDialog class
 *
 * @author Jacob Davidson
 * @version 3.0.0
 */
public class FarkleOptionsDialogTest {

    /** The FarkleOptionsDialog object to test */
    FarkleOptionsDialog farkleOptionsDialog;

    /** The primary background color */
    private Color greenBackground = new Color(35, 119, 34);

    /**
     * Instantiate the objects before testing begins
     *
     * @throws Exception
     */
    @Before
    public void setUp() throws Exception {

        // Instantiate the farkleOptionsDialog passing null as the parent frame
        farkleOptionsDialog = new FarkleOptionsDialog(null);

    }

    /**
     * Dispose the window at the completion of testing
     *
     * @throws Exception
     */
    @After
    public void tearDown() throws Exception {

        // Dispose of the farkleOptionsDialog box
        farkleOptionsDialog.dispose();

    }

    /**
     * Test the FarkleOptionsDialog(JFrame) constructor
     */
```

```
58      @Test
59      public void testConstructor() {
60
61          // Assert that the farkleOptionsDialog is not null
62          assertNotNull(farkleOptionsDialog);
63
64          // Assert that the player1Name and player2Name Strings are empty
65          assertTrue(farkleOptionsDialog.getPlayer1Name().isEmpty());
66          assertTrue(farkleOptionsDialog.getPlayer2Name().isEmpty());
67
68          // Assert that the gameMode is set to GameMode.SINGLEPLAYER
69          assertTrue(farkleOptionsDialog.getGameMode() == GameMode.SINGLEPLAYER);
70
71          // Assert that the opponent player type is PlayerType.USER
72          assertTrue(farkleOptionsDialog.getPlayerType() == PlayerType.USER);
73
74          // Assert that the preferred size should is set to Dimension(750, 250)
75          assertTrue(farkleOptionsDialog.getPreferredSize().equals(
76                  new Dimension(750, 250)));
77
78          // Assert that the modality is set to ModalityType.APPLICATION_MODAL
79          assertTrue(farkleOptionsDialog.getModalityType() == ModalityType.APPLICATION_MODAL);
80
81          // Assert that the window is undecorated
82          assertTrue(farkleOptionsDialog.isUndecorated());
83
84          // Assert that the window is not resizeable
85          assertTrue(!farkleOptionsDialog.isResizable());
86
87          // Assert that the JDialog box is not visible
88          assertTrue(!farkleOptionsDialog.isVisible());
89
90      }
91
92      /**
93       * Test the getPlayerType() and setPlayerType(PlayerType) methods
94       */
95      @Test
96      public void testGetAndSetPlayerType() {
97
98          // The player type should be initially set to PlayerType.USER
99          assertEquals(farkleOptionsDialog.getPlayerType(), PlayerType.USER);
100
101         // Set the player type to PlayerType.COMPUTER and test again
102         farkleOptionsDialog.setPlayerType(PlayerType.COMPUTER);
103         assertEquals(farkleOptionsDialog.getPlayerType(), PlayerType.COMPUTER);
104
105         // Set the player type to null and test again
106         farkleOptionsDialog.setPlayerType(null);
107         assertNull(farkleOptionsDialog.getPlayerType());
108
109         // Set the player type to PlayerType.USER and test again
110         farkleOptionsDialog.setPlayerType(PlayerType.USER);
111         assertEquals(farkleOptionsDialog.getPlayerType(), PlayerType.USER);
112
113     }
114
```

```
115     /**
116      * Test the getPlayer1Name() and setPlayer1Name(String) methods
117      */
118     @Test
119     public void testGetAndSetPlayer1Name() {
120
121         // Player 1 Name should initially be empty
122         assertTrue(farkleOptionsDialog.getPlayer1Name().isEmpty());
123
124         // Set the player 1 name to null and test again
125         farkleOptionsDialog.setPlayer1Name(null);
126         assertNull(farkleOptionsDialog.getPlayer1Name());
127
128         // Set the player 1 name to a string and test again
129         farkleOptionsDialog.setPlayer1Name("Jake");
130         assertTrue(farkleOptionsDialog.getPlayer1Name().equals("Jake"));
131
132     }
133
134     /**
135      * Test the getPlayer2Name() and setPlayer2Name(String) methods
136      */
137     @Test
138     public void testGetAndSetPlayer2Name() {
139
140         // Player 2 Name should initially be empty
141         assertTrue(farkleOptionsDialog.getPlayer2Name().isEmpty());
142
143         // Set the player 2 name to null and test again
144         farkleOptionsDialog.setPlayer2Name(null);
145         assertNull(farkleOptionsDialog.getPlayer2Name());
146
147         // Set the player 2 name to a string and test again
148         farkleOptionsDialog.setPlayer2Name("Brant");
149         assertTrue(farkleOptionsDialog.getPlayer2Name().equals("Brant"));
150
151     }
152
153     /**
154      * Test the getGameMode() and setGameMode(GameMode) methods
155      */
156     @Test
157     public void testGetAndSetGameMode() {
158
159         // The game mode should initially be set to GameMode.SINGLEPLAYER
160         assertEquals(farkleOptionsDialog.getGameMode(), GameMode.SINGLEPLAYER);
161
162         // Set the game mode to GameMode.MULTIPLAYER and test again
163         farkleOptionsDialog.setGameMode(GameMode.MULTIPLAYER);
164         assertEquals(farkleOptionsDialog.getGameMode(), GameMode.MULTIPLAYER);
165
166         // Set the game mode to null and test again
167         farkleOptionsDialog.setGameMode(null);
168         assertNull(farkleOptionsDialog.getGameMode());
169
170         // Set the game mode to GameMode.SINGLEPLAYER and test again
171         farkleOptionsDialog.setGameMode(GameMode.SINGLEPLAYER);
```

```
172         assertEquals(farkleOptionsDialog.getGameMode(), GameMode.SINGLEPLAYER);
173     }
174
175     /**
176      * Test the mouseClicked(MouseEvent e) method
177      */
178     @Test
179     public void testMouseClicked() {
180
181         // Get a reference to each of the JLabel objects used for selection
182         JLabel singlePlayerLabel = farkleOptionsDialog.getJLabel(1);
183         JLabel multiPlayerLabel = farkleOptionsDialog.getJLabel(2);
184         JLabel humanPlayerLabel = farkleOptionsDialog.getJLabel(3);
185         JLabel computerPlayerLabel = farkleOptionsDialog.getJLabel(4);
186
187         // Get the playerTypeSelectionPanel
188         JPanel playerTypeSelectionPanel = farkleOptionsDialog.getJPanel(1);
189
190         // Get the player two panel information
191         JPanel playerTwoNamePanel = farkleOptionsDialog.getJPanel(2);
192         JLabel playerTwoNameLabel = farkleOptionsDialog.getJLabel(5);
193         JTextField playerTwoName = farkleOptionsDialog.getJTextField(1);
194
195         // Instantiate MouseEvent objects that simulate the user clicking on the
196         // above JLabels
197         MouseEvent singlePlayerLabelClicked = new MouseEvent(singlePlayerLabel,
198                 1, 1, 0, 0, 0, 1, false);
199         MouseEvent multiPlayerLabelClicked = new MouseEvent(multiPlayerLabel,
200                 1, 1, 0, 0, 0, 1, false);
201         MouseEvent humanPlayerLabelClicked = new MouseEvent(humanPlayerLabel,
202                 1, 1, 0, 0, 0, 1, false);
203         MouseEvent computerPlayerLabelClicked = new MouseEvent(
204                 computerPlayerLabel, 1, 1, 0, 0, 0, 1, false);
205
206         /*************************************************
207          * 1.1.3 - If the user highlights the "2 Player Mode" option, the "1
208          * Player Mode" option is deselected, and two more options appear
209          * ("Human Opponent" and "Computer Opponent"). The "Human Opponent"
210          * option is highlighted by default.
211          *************************************************/
212         /*
213          * When the dialog box opens, the singlePlayerLabel is highlighted, test
214          * the click on the multiPlayerLabel.
215          */
216         farkleOptionsDialog.mouseClicked(multiPlayerLabelClicked);
217
218         // Assert that singlePlayerLabel is deselected
219         assertEquals(singlePlayerLabel.getBackground(), greenBackground);
220         assertEquals(singlePlayerLabel.getForeground(), Color.WHITE);
221
222         // Assert that multiPlayer is selected
223         assertEquals(multiPlayerLabel.getBackground(), Color.YELLOW);
224         assertEquals(multiPlayerLabel.getForeground(), Color.BLACK);
225
226         // Assert that "Human Opponent" is selected
227         assertEquals(humanPlayerLabel.getBackground(), Color.YELLOW);
228         assertEquals(humanPlayerLabel.getForeground(), Color.BLACK);
```

```
229
230         // Assert that "Computer Opponent" is deselected
231         assertEquals(computerPlayerLabel.getBackground(), greenBackground);
232         assertEquals(computerPlayerLabel.getForeground(), Color.WHITE);
233
234         // Assert that the playerTypeSelectionPanel is visible
235         assertTrue(playerTypeSelectionPanel.isVisible());
236
237         // Assert that the playerTwoNamePanel is visible
238         assertTrue(playerTwoNamePanel.isVisible());
239
240         // Assert that the playerTwoNameLabel and playerTwoName are enabled
241         assertTrue(playerTwoNameLabel.isEnabled());
242         assertTrue(playerTwoName.isEnabled());
243
244         // Assert that the playerTwoName is empty
245         assertTrue(playerTwoName.getText().isEmpty());
246
247         /*************************************************
248          * 1.1.5 - When the "Computer Opponent" option is highlighted, the text
249          * fields for "Player One Name" and "Player Two Name" are displayed, but
250          * the "Player Two Name" field is disabled, and "Computer" is supplied
251          * for the "Player Two Name".
252          *************************************************/
253         /*
254          * Two Player Mode is highlighted, simulate a click on
255          * "Computer Opponent"
256          */
257         farkleOptionsDialog.mouseClicked(computerPlayerLabelClicked);
258
259         // Assert that singlePlayerLabel is deselected
260         assertEquals(singlePlayerLabel.getBackground(), greenBackground);
261         assertEquals(singlePlayerLabel.getForeground(), Color.WHITE);
262
263         // Assert that multiPlayer is selected
264         assertEquals(multiPlayerLabel.getBackground(), Color.YELLOW);
265         assertEquals(multiPlayerLabel.getForeground(), Color.BLACK);
266
267         // Assert that "Human Opponent" is deselected
268         assertEquals(humanPlayerLabel.getBackground(), greenBackground);
269         assertEquals(humanPlayerLabel.getForeground(), Color.WHITE);
270
271         // Assert that "Computer Opponent" is selected
272         assertEquals(computerPlayerLabel.getBackground(), Color.YELLOW);
273         assertEquals(computerPlayerLabel.getForeground(), Color.BLACK);
274
275         // Assert that the playerTypeSelectionPanel is visible
276         assertTrue(playerTypeSelectionPanel.isVisible());
277
278         // Assert that the playerTwoNamePanel is visible
279         assertTrue(playerTwoNamePanel.isVisible());
280
281         // Assert that the playerTwoNameLabel and playerTwoName are disabled
282         assertTrue(!playerTwoNameLabel.isEnabled());
283         assertTrue(!playerTwoName.isEnabled());
284
285         // Assert that the playerTwoName is set to "Computer"
```

```
286         assertEquals(playerTwoName.getText(), "Computer");
287 
288         /*
289          * Two Player Mode is highlighted, simulate a click on "One Player Mode"
290          */
291         farkleOptionsDialog.mouseClicked(singlePlayerLabelClicked);
292 
293         // Assert that singlePlayerLabel is selected
294         assertEquals(singlePlayerLabel.getBackground(), Color.YELLOW);
295         assertEquals(singlePlayerLabel.getForeground(), Color.BLACK);
296 
297         // Assert that multiPlayer is deselected
298         assertEquals(multiPlayerLabel.getBackground(), greenBackground);
299         assertEquals(multiPlayerLabel.getForeground(), Color.WHITE);
300 
301         // Assert that "Human Opponent" is deselected
302         assertEquals(humanPlayerLabel.getBackground(), greenBackground);
303 
304         // Assert that "Computer Opponent" is deselected
305         assertEquals(computerPlayerLabel.getBackground(), greenBackground);
306 
307         // Assert that the playerTypeSelectionPanel is not visible
308         assertTrue(!playerTypeSelectionPanel.isVisible());
309 
310         // Assert that the playerTwoNamePanel is not visible
311         assertTrue(!playerTwoNamePanel.isVisible());
312 
313         // Assert that the playerTwoName is empty
314         assertTrue(playerTwoName.getText().isEmpty());
315 
316         /*************************************************
317          * 1.1.4 - When the "Human Opponent" option is highlighted, two text
318          * fields are displayed, labeled "Player One Name", and "Player Two
319          * Name".
320          *************************************************/
321         /*
322          * One player mode is highlighted. The user clicks on multiPlayerMode,
323          * computer, then human opponent
324          */
325         farkleOptionsDialog.mouseClicked(multiPlayerLabelClicked);
326         farkleOptionsDialog.mouseClicked(computerPlayerLabelClicked);
327         farkleOptionsDialog.mouseClicked(humanPlayerLabelClicked);
328 
329         // Assert that singlePlayerLabel is deselected
330         assertEquals(singlePlayerLabel.getBackground(), greenBackground);
331         assertEquals(singlePlayerLabel.getForeground(), Color.WHITE);
332 
333         // Assert that multiPlayer is selected
334         assertEquals(multiPlayerLabel.getBackground(), Color.YELLOW);
335         assertEquals(multiPlayerLabel.getForeground(), Color.BLACK);
336 
337         // Assert that "Human Opponent" is selected
338         assertEquals(humanPlayerLabel.getBackground(), Color.YELLOW);
339         assertEquals(humanPlayerLabel.getForeground(), Color.BLACK);
340 
341         // Assert that "Computer Opponent" is deselected
342         assertEquals(computerPlayerLabel.getBackground(), greenBackground);
```

```
343         assertEquals(computerPlayerLabel.getForeground(), Color.WHITE);
344
345         // Assert that the playerTypeSelectionPanel is visible
346         assertTrue(playerTypeSelectionPanel.isVisible());
347
348         // Assert that the playerTwoNamePanel is visible
349         assertTrue(playerTwoNamePanel.isVisible());
350
351         // Assert that the playerTwoNameLabel and playerTwoName are enabled
352         assertTrue(playerTwoNameLabel.isEnabled());
353         assertTrue(playerTwoName.isEnabled());
354
355         // Assert that the playerTwoName is empty
356         assertTrue(playerTwoName.getText().isEmpty());
357
358     }
359
360 }
361
```

```java
package com.lotsofun.farkle;

import static org.junit.Assert.*;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import org.junit.Before;
import org.junit.Test;

/**
 * The GameCalculateScoreTest class tests the calculateScore(List<Integer>,
 * boolean) and the calculateHighestScore(List<Integer>) methods and the of the
 * Game class for every possible combination of dice to ensure they comply with
 * the requirement 6.0.0, scoring, of the specific requirements in the
 * requirements document. This includes rolls of 1, 2, 3, 4, 5 or 6 dice.
 *
 * @author Jacob Davidson
 * @version 3.0.0
 */
public class GameCalculateScoreTest {

    /** The game object used for testing */
    private Game game;

    /** The FarkleController object used for testing */
    private FarkleController farkleController;

    /** The list of lists of permutations for a given roll to be tested */
    List<List<Integer>> myPermutations;

    /**
     * List of strings representing rolls for permutations of die that are not 1
     * or 5
     */
    List<String> automatedPermutations;

    /** Retrieved score for a given roll */
    int score = 0;

    /** Three of kind die value */
    int dieOneValue = 0;

    /**
     * The value of the second three of a kind die if two three of a kinds
     * rolled at once
     */
    int dieTwoValue = 0;

    /**
     * The object array used to store the returned array from the
     * calculateHighestScore method
     */
    Object[] highestScore;

    /** List of integers indicating the highest roll */
    List<Integer> highestRoll;

```

```
58      /**
59       * Instantiate the objects before testing begins
60       *
61       * @throws Exception
62       */
63      @Before
64      public void setUp() throws Exception {
65
66          // Instantiate the FarkleController object
67          farkleController = new FarkleController(true);
68
69          // Instantiate the game option. The Game Mode used does not change the
70          // results of this test
71          game = new Game(GameMode.SINGLEPLAYER, farkleController);
72
73      }
74
75      /**
76       * Test the helper methods that calculate the dice roll permutations used to
77       * check the calculateScore method
78       */
79      @Test
80      public void testPermutationHelperMethods() {
81
82          /* Test the permutations method */
83
84          // If null is passed to permutations, null should be returned
85          assertEquals(null, permutations(null));
86
87          // If letters anything but digits are in the string, null should be
88          // returned
89          assertEquals(null, permutations("ABC"));
90
91          // Check permutations returned from a string of integers
92          List<List<Integer>> permutationsToTest = permutations("123");
93          assertEquals(6, permutationsToTest.size());
94          List<Integer> numbers = new LinkedList<Integer>();
95          numbers.add(1);
96          numbers.add(2);
97          numbers.add(3);
98          assertTrue(permutationsToTest.contains(numbers));
99          numbers.clear();
100         numbers.add(1);
101         numbers.add(3);
102         numbers.add(2);
103         assertTrue(permutationsToTest.contains(numbers));
104         numbers.clear();
105         numbers.add(2);
106         numbers.add(1);
107         numbers.add(3);
108         assertTrue(permutationsToTest.contains(numbers));
109         numbers.clear();
110         numbers.add(2);
111         numbers.add(3);
112         numbers.add(1);
113         assertTrue(permutationsToTest.contains(numbers));
114         numbers.clear();
```

```
115         numbers.add(3);
116         numbers.add(1);
117         numbers.add(2);
118         assertTrue(permutationsToTest.contains(numbers));
119         numbers.clear();
120         numbers.add(3);
121         numbers.add(2);
122         numbers.add(1);
123         assertTrue(permutationsToTest.contains(numbers));
124 
125         /* Test the stringPermutations method */
126 
127         // If null is passed to stringPermutations, null should be returned
128         assertEquals(null, stringPermutations(null));
129 
130         // Check permutations returned from a string
131         List<String> stringPermutationsToTest = stringPermutations("123");
132         assertEquals(6, stringPermutationsToTest.size());
133         assertTrue(stringPermutationsToTest.contains("123"));
134         assertTrue(stringPermutationsToTest.contains("132"));
135         assertTrue(stringPermutationsToTest.contains("213"));
136         assertTrue(stringPermutationsToTest.contains("231"));
137         assertTrue(stringPermutationsToTest.contains("321"));
138         assertTrue(stringPermutationsToTest.contains("312"));
139 
140         /* Test the integerPermutations method */
141 
142         // If null is passed to integerPermutations, null should be returned
143         assertEquals(null, integerPermutations(null));
144 
145         // If a string with no variables is passed to integerPermutations,
146         // return that
147         // String should be the only string in the returned list
148         assertEquals(1, integerPermutations("123").size());
149         assertTrue(integerPermutations("123").contains("123"));
150 
151         // Check the permutations returned from a string with variables included
152         List<String> integerPermutationsToTest = integerPermutations("AB");
153         assertEquals(12, integerPermutationsToTest.size());
154         assertTrue(integerPermutationsToTest.contains("23"));
155         assertTrue(integerPermutationsToTest.contains("24"));
156         assertTrue(integerPermutationsToTest.contains("26"));
157         assertTrue(integerPermutationsToTest.contains("32"));
158         assertTrue(integerPermutationsToTest.contains("34"));
159         assertTrue(integerPermutationsToTest.contains("36"));
160         assertTrue(integerPermutationsToTest.contains("42"));
161         assertTrue(integerPermutationsToTest.contains("43"));
162         assertTrue(integerPermutationsToTest.contains("46"));
163         assertTrue(integerPermutationsToTest.contains("62"));
164         assertTrue(integerPermutationsToTest.contains("63"));
165         assertTrue(integerPermutationsToTest.contains("64"));
166 
167     }
168 
169     /**
170      * This method tests the fringe conditions of the calculateScore method of
171      * the Game class
```

```
172      */
173     @SuppressWarnings("unchecked")
174     @Test
175     public void testCalculateScoreFringe() {
176
177         // The list used for testing in this method
178         List<Integer> roll = new LinkedList<Integer>();
179
180         // Returned score
181         int score = 0;
182
183         // Test for a roll with an empty list
184         score = game.calculateScore(roll, true);
185         assertEquals(0, score);
186         score = game.calculateScore(roll, false);
187         assertEquals(0, score);
188
189         // Test calculateHighestScore with an empty list
190         highestScore = game.calculateHighestScore(roll);
191         assertEquals(0, highestScore[0]);
192         highestRoll = (List<Integer>) highestScore[1];
193         assertTrue(highestRoll.isEmpty());
194
195         // Test that a roll with negative number integers returns 0
196         roll.add(-1);
197         roll.add(-5);
198         score = game.calculateScore(roll, true);
199         assertEquals(0, score);
200         score = game.calculateScore(roll, false);
201         assertEquals(0, score);
202
203         // Test calculateHighestScore with the negative number
204         highestScore = game.calculateHighestScore(roll);
205         assertEquals(0, highestScore[0]);
206         highestRoll = (List<Integer>) highestScore[1];
207         assertTrue(highestRoll.isEmpty());
208
209         // Test that passing null to the calculateScore method returns 0
210         assertEquals(0, game.calculateScore(null, true));
211         assertEquals(0, game.calculateScore(null, false));
212
213         // Test calculateHighestScore with null
214         highestScore = game.calculateHighestScore(null);
215         assertEquals(0, highestScore[0]);
216         highestRoll = (List<Integer>) highestScore[1];
217         assertTrue(highestRoll.isEmpty());
218
219     }
220
221     /**
222      * This method tests the calculateScore method of the Game class for all
223      * permutations of 1 die, requirements 6.1.0 and 6.2.0: <br /><br />
224      * 6.1.0 Each 1 rolled is worth 100 points<br />
225      * 6.2.0 Each 5 rolled is worth 50 points
226      */
227     @SuppressWarnings("unchecked")
228     @Test
```

```
229     public void testCalculateScore1() {
230
231         /*
232          * Test 1: Check scoring for a die roll of A (A represents all die
233          * values other than 1 or 5)
234          */
235
236         // Generate all permutations of die not including 1 or 5
237         automatedPermutations = integerPermutations("A");
238
239         // Convert the list of Strings to a list of lists of integers
240         for (String numbers : automatedPermutations) {
241             myPermutations = permutations(numbers);
242
243             // Check each list of integers for proper scoring
244             for (List<Integer> currentNumbers : myPermutations) {
245                 // Test the total score
246                 score = game.calculateScore(currentNumbers, false);
247                 assertEquals(0, score);
248
249                 // Test the held score
250                 score = game.calculateScore(currentNumbers, true);
251                 assertEquals(0, score);
252
253                 // Test calculateHighestScore
254                 highestScore = game.calculateHighestScore(currentNumbers);
255                 assertEquals(0, highestScore[0]);
256                 highestRoll = (List<Integer>) highestScore[1];
257                 assertTrue(highestRoll.isEmpty());
258             }
259         }
260
261         /* Test 2: Check scoring for a die roll of 5 (requirement 6.2.0) */
262
263         // Generate all permutations of the roll and store in a list of lists of
264         // integers
265         myPermutations = permutations("5");
266
267         // Check each list of integers for correct scoring
268         for (List<Integer> numbers : myPermutations) {
269             // Test the total score
270             score = game.calculateScore(numbers, false);
271             assertEquals(50, score);
272
273             // Test the held score
274             score = game.calculateScore(numbers, true);
275             assertEquals(50, score);
276
277             // Test calculateHighestScore
278             highestScore = game.calculateHighestScore(numbers);
279             assertEquals(50, highestScore[0]);
280             highestRoll = (List<Integer>) highestScore[1];
281             assertEquals(highestRoll.size(), 1);
282             assertTrue(highestRoll.get(0).equals(5));
283         }
284
285         /* Test 3: Check scoring for a die roll of 1 (requirement 6.1.0) */
```

```
286
287         // Generate all permutations of the roll and store in a list of lists of
288         // integers
289         myPermutations = permutations("1");
290
291         // Check each list of integers for correct scoring
292         for (List<Integer> numbers : myPermutations) {
293             // Test the total score
294             score = game.calculateScore(numbers, false);
295             assertEquals(100, score);
296
297             // Test the held score
298             score = game.calculateScore(numbers, true);
299             assertEquals(100, score);
300
301             // Test calculateHighestScore
302             highestScore = game.calculateHighestScore(numbers);
303             assertEquals(100, highestScore[0]);
304             highestRoll = (List<Integer>) highestScore[1];
305             assertEquals(highestRoll.size(), 1);
306             assertTrue(highestRoll.get(0).equals(1));
307         }
308     }
309
310     /**
311      * This method tests the calculateScore method of the Game class for all
312      * permutations of 2 dice, requirement 6.1.0 and 6.2.0. <br />
313      * <br />
314      * 6.1.0 Each 1 rolled is worth 100 points<br />
315      * 6.2.0 Each 5 rolled is worth 50 points
316      */
317     @SuppressWarnings("unchecked")
318     @Test
319     public void testCalculateScore2() {
320
321         /*
322          * Test 4: Check scoring for a die roll of AA (A represents all die
323          * values other than 1 or 5)
324          */
325
326         // Generate all permutations of the roll
327         automatedPermutations = integerPermutations("AA");
328
329         // Convert the list of Strings to a list of lists of integers
330         for (String numbers : automatedPermutations) {
331             myPermutations = permutations(numbers);
332
333             // Check each list of integers for proper scoring
334             for (List<Integer> currentNumbers : myPermutations) {
335                 // Test the total score
336                 score = game.calculateScore(currentNumbers, false);
337                 assertEquals(0, score);
338
339                 // Test the held score
340                 score = game.calculateScore(currentNumbers, true);
341                 assertEquals(0, score);
342
```

```
343                 // Test calculateHighestScore
344                 highestScore = game.calculateHighestScore(currentNumbers);
345                 assertEquals(0, highestScore[0]);
346                 highestRoll = (List<Integer>) highestScore[1];
347                 assertTrue(highestRoll.isEmpty());
348             }
349         }
350 
351         /*
352          * Test 5: Check scoring for a die roll of AX (A and X represent all die
353          * values other than 1 or 5)
354          */
355 
356         // Generate all permutations of the roll
357         automatedPermutations = integerPermutations("AX");
358 
359         // Convert the list of Strings to a list of lists of integers
360         for (String numbers : automatedPermutations) {
361             myPermutations = permutations(numbers);
362 
363             // Check each list of integers for proper scoring
364             for (List<Integer> currentNumbers : myPermutations) {
365                 // Test the total score
366                 score = game.calculateScore(currentNumbers, false);
367                 assertEquals(0, score);
368 
369                 // Test the held score
370                 score = game.calculateScore(currentNumbers, true);
371                 assertEquals(0, score);
372 
373                 // Test calculateHighestScore
374                 highestScore = game.calculateHighestScore(currentNumbers);
375                 assertEquals(0, highestScore[0]);
376                 highestRoll = (List<Integer>) highestScore[1];
377                 assertTrue(highestRoll.isEmpty());
378             }
379         }
380 
381         /*
382          * Test 6: Check scoring for a die roll of 5X (X represent all die
383          * values other than 1 or 5) Requirement 6.2.0
384          */
385 
386         // Generate all permutations of the roll
387         automatedPermutations = integerPermutations("5X");
388 
389         // Convert the list of Strings to a list of lists of integers
390         for (String numbers : automatedPermutations) {
391             myPermutations = permutations(numbers);
392 
393             // Check each list of integers for proper scoring
394             for (List<Integer> currentNumbers : myPermutations) {
395                 // Test the total score
396                 score = game.calculateScore(currentNumbers, false);
397                 assertEquals(50, score);
398 
399                 // Test the held score
```

```
400                 score = game.calculateScore(currentNumbers, true);
401                 assertEquals(0, score);
402
403                 // Test calculateHighestScore
404                 highestScore = game.calculateHighestScore(currentNumbers);
405                 assertEquals(50, highestScore[0]);
406                 highestRoll = (List<Integer>) highestScore[1];
407                 assertEquals(highestRoll.size(), 1);
408                 assertTrue(highestRoll.get(0).equals(5));
409             }
410         }
411
412         /*
413          * Test 7: Check scoring for a die roll of 1X (X represent all die
414          * values other than 1 or 5) Requirement 6.1.0
415          */
416
417         // Generate all permutations of the roll
418         automatedPermutations = integerPermutations("1X");
419
420         // Convert the list of Strings to a list of lists of integers
421         for (String numbers : automatedPermutations) {
422             myPermutations = permutations(numbers);
423
424             // Check each list of integers for proper scoring
425             for (List<Integer> currentNumbers : myPermutations) {
426                 // Test the total score
427                 score = game.calculateScore(currentNumbers, false);
428                 assertEquals(100, score);
429
430                 // Test the held score
431                 score = game.calculateScore(currentNumbers, true);
432                 assertEquals(0, score);
433
434                 // Test calculateHighestScore
435                 highestScore = game.calculateHighestScore(currentNumbers);
436                 assertEquals(100, highestScore[0]);
437                 highestRoll = (List<Integer>) highestScore[1];
438                 assertEquals(highestRoll.size(), 1);
439                 assertTrue(highestRoll.get(0).equals(1));
440             }
441         }
442
443         /*
444          * Test 8: Check scoring for a die roll of 55 Requirement 6.2.0
445          */
446
447         // Generate all permutations of the roll and store in a list of lists of
448         // integers
449         myPermutations = permutations("55");
450
451         // Check each list of integers for correct scoring
452         for (List<Integer> numbers : myPermutations) {
453             // Test the total score
454             score = game.calculateScore(numbers, false);
455             assertEquals(100, score);
456
```

```
457             // Test the held score
458             score = game.calculateScore(numbers, true);
459             assertEquals(100, score);
460 
461             // Test calculateHighestScore
462             highestScore = game.calculateHighestScore(numbers);
463             assertEquals(100, highestScore[0]);
464             highestRoll = (List<Integer>) highestScore[1];
465             assertEquals(highestRoll.size(), 2);
466             assertTrue(highestRoll.get(0).equals(5));
467             assertTrue(highestRoll.get(1).equals(5));
468         }
469 
470         /*
471          * Test 9: Check scoring for a die roll of 15 Requirement 6.1.0 and
472          * 6.2.0
473          */
474 
475         // Generate all permutations of the roll and store in a list of lists of
476         // integers
477         myPermutations = permutations("15");
478 
479         // Check each list of integers for correct scoring
480         for (List<Integer> numbers : myPermutations) {
481             // Test the total score
482             score = game.calculateScore(numbers, false);
483             assertEquals(150, score);
484 
485             // Test the held score
486             score = game.calculateScore(numbers, true);
487             assertEquals(150, score);
488 
489             // Test calculateHighestScore
490             highestScore = game.calculateHighestScore(numbers);
491             assertEquals(150, highestScore[0]);
492             highestRoll = (List<Integer>) highestScore[1];
493             assertEquals(highestRoll.size(), 2);
494             assertTrue(highestRoll.contains(1));
495             assertTrue(highestRoll.contains(5));
496         }
497 
498         /*
499          * Test 10: Check scoring for a die roll of 11 Requirement 6.1.0
500          */
501 
502         // Generate all permutations of the roll and store in a list of lists of
503         // integers
504         myPermutations = permutations("11");
505 
506         // Check each list of integers for correct scoring
507         for (List<Integer> numbers : myPermutations) {
508             // Test the total score
509             score = game.calculateScore(numbers, false);
510             assertEquals(200, score);
511 
512             // Test the held score
513             score = game.calculateScore(numbers, true);
```

```
514             assertEquals(200, score);
515
516             // Test the calculateHighestScore method
517             highestScore = game.calculateHighestScore(numbers);
518             assertEquals(200, highestScore[0]);
519             highestRoll = (List<Integer>) highestScore[1];
520             assertEquals(highestRoll.size(), 2);
521             assertTrue(highestRoll.contains(1));
522             assertTrue(!highestRoll.contains(2));
523             assertTrue(!highestRoll.contains(3));
524             assertTrue(!highestRoll.contains(4));
525             assertTrue(!highestRoll.contains(5));
526             assertTrue(!highestRoll.contains(6));
527         }
528
529     }
530
531     /**
532      * This method tests the calculateScore method of the Game class for all
533      * permutations of 3 dice, requirements 6.1.0, 6.2.0, 6.3.0, and 6.4.0. <br />
534      * <br />
535      * 6.1.0 Each 1 rolled is worth 100 points<br />
536      * 6.2.0 Each 5 rolled is worth 50 points<br />
537      * 6.3.0 Three 1â€™s are worth 1000 points<br />
538      * 6.4.0 Three of a kind of any value other than 1 is worth 100 times the
539      * value of the die (e.g. three 4â€™s is worth 400 points).
540      */
541     @SuppressWarnings("unchecked")
542     @Test
543     public void testCalculateScore3() {
544
545         /*
546          * Test 11: Check scoring for a die roll of AXY (A, X and Y each
547          * represent all die values other than 1 or 5)
548          */
549
550         // Generate all permutations of the roll
551         automatedPermutations = integerPermutations("AXY");
552
553         // Convert the list of Strings to a list of lists of integers
554         for (String numbers : automatedPermutations) {
555             myPermutations = permutations(numbers);
556
557             // Check each list of integers for proper scoring
558             for (List<Integer> currentNumbers : myPermutations) {
559                 // Test the total score
560                 score = game.calculateScore(currentNumbers, false);
561                 assertEquals(0, score);
562
563                 // Test the held score
564                 score = game.calculateScore(currentNumbers, true);
565                 assertEquals(0, score);
566
567                 // Test the calculateHighestScore method
568                 highestScore = game.calculateHighestScore(currentNumbers);
569                 assertEquals(0, highestScore[0]);
570                 highestRoll = (List<Integer>) highestScore[1];
```

```
571                 assertTrue(highestRoll.isEmpty());
572             }
573         }
574 
575         /*
576          * Test 12: Check scoring for a die roll of AAX (A and X each represent
577          * all die values other than 1 or 5)
578          */
579 
580         // Generate all permutations of the roll
581         automatedPermutations = integerPermutations("AX");
582 
583         // Convert the list of Strings to a list of lists of integers
584         for (String numbers : automatedPermutations) {
585             myPermutations = permutations(numbers);
586 
587             // Check each list of integers for proper scoring
588             for (List<Integer> currentNumbers : myPermutations) {
589                 // Test the total score
590                 score = game.calculateScore(currentNumbers, false);
591                 assertEquals(0, score);
592 
593                 // Test the held score
594                 score = game.calculateScore(currentNumbers, true);
595                 assertEquals(0, score);
596 
597                 // Test the calculateHighestScore method
598                 highestScore = game.calculateHighestScore(currentNumbers);
599                 assertEquals(0, highestScore[0]);
600                 highestRoll = (List<Integer>) highestScore[1];
601                 assertTrue(highestRoll.isEmpty());
602             }
603         }
604 
605         /*
606          * Test 13: Check scoring for a die roll of 5AA (A represents all die
607          * values other than 1 or 5) Requirement 6.2.0
608          */
609 
610         // Generate all permutations of the roll
611         automatedPermutations = integerPermutations("5AA");
612 
613         // Convert the list of Strings to a list of lists of integers
614         for (String numbers : automatedPermutations) {
615             myPermutations = permutations(numbers);
616 
617             // Check each list of integers for proper scoring
618             for (List<Integer> currentNumbers : myPermutations) {
619                 // Test the total score
620                 score = game.calculateScore(currentNumbers, false);
621                 assertEquals(50, score);
622 
623                 // Test the held score
624                 score = game.calculateScore(currentNumbers, true);
625                 assertEquals(0, score);
626 
627                 // Test the calculateHighestScore method
```

```
628                 highestScore = game.calculateHighestScore(currentNumbers);
629                 assertEquals(50, highestScore[0]);
630                 highestRoll = (List<Integer>) highestScore[1];
631                 assertEquals(highestRoll.size(), 1);
632                 assertTrue(!highestRoll.contains(1));
633                 assertTrue(!highestRoll.contains(2));
634                 assertTrue(!highestRoll.contains(3));
635                 assertTrue(!highestRoll.contains(4));
636                 assertTrue(highestRoll.contains(5));
637                 assertTrue(!highestRoll.contains(6));
638             }
639         }
640 
641         /*
642          * Test 14: Check scoring for a die roll of 5AX (A and X represent all
643          * die values other than 1 or 5) Requirement 6.2.0
644          */
645 
646         // Generate all permutations of the roll
647         automatedPermutations = integerPermutations("5AX");
648 
649         // Convert the list of Strings to a list of lists of integers
650         for (String numbers : automatedPermutations) {
651             myPermutations = permutations(numbers);
652 
653             // Check each list of integers for proper scoring
654             for (List<Integer> currentNumbers : myPermutations) {
655                 // Test the total score
656                 score = game.calculateScore(currentNumbers, false);
657                 assertEquals(50, score);
658 
659                 // Test the held score
660                 score = game.calculateScore(currentNumbers, true);
661                 assertEquals(0, score);
662 
663                 // Test the calculateHighestScore method
664                 highestScore = game.calculateHighestScore(currentNumbers);
665                 assertEquals(50, highestScore[0]);
666                 highestRoll = (List<Integer>) highestScore[1];
667                 assertEquals(highestRoll.size(), 1);
668                 assertTrue(!highestRoll.contains(1));
669                 assertTrue(!highestRoll.contains(2));
670                 assertTrue(!highestRoll.contains(3));
671                 assertTrue(!highestRoll.contains(4));
672                 assertTrue(highestRoll.contains(5));
673                 assertTrue(!highestRoll.contains(6));
674             }
675         }
676 
677         /*
678          * Test 15: Check scoring for a die roll of 1AA (A represents all die
679          * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
680          */
681 
682         // Generate all permutations of the roll
683         automatedPermutations = integerPermutations("1AA");
684 
```

```
685         // Convert the list of Strings to a list of lists of integers
686         for (String numbers : automatedPermutations) {
687             myPermutations = permutations(numbers);
688 
689             // Check each list of integers for proper scoring
690             for (List<Integer> currentNumbers : myPermutations) {
691                 // Test the total score
692                 score = game.calculateScore(currentNumbers, false);
693                 assertEquals(100, score);
694 
695                 // Test the held score
696                 score = game.calculateScore(currentNumbers, true);
697                 assertEquals(0, score);
698 
699                 // Test the calculateHighestScore method
700                 highestScore = game.calculateHighestScore(currentNumbers);
701                 assertEquals(100, highestScore[0]);
702                 highestRoll = (List<Integer>) highestScore[1];
703                 assertEquals(highestRoll.size(), 1);
704                 assertTrue(highestRoll.contains(1));
705                 assertTrue(!highestRoll.contains(2));
706                 assertTrue(!highestRoll.contains(3));
707                 assertTrue(!highestRoll.contains(4));
708                 assertTrue(!highestRoll.contains(5));
709                 assertTrue(!highestRoll.contains(6));
710             }
711         }
712 
713         /*
714          * Test 16: Check scoring for a die roll of 1AX (A and X represent all
715          * die values other than 1 or 5) Requirement 6.1.0
716          */
717 
718         // Generate all permutations of the roll
719         automatedPermutations = integerPermutations("1AX");
720 
721         // Convert the list of Strings to a list of lists of integers
722         for (String numbers : automatedPermutations) {
723             myPermutations = permutations(numbers);
724 
725             // Check each list of integers for proper scoring
726             for (List<Integer> currentNumbers : myPermutations) {
727                 // Test the total score
728                 score = game.calculateScore(currentNumbers, false);
729                 assertEquals(100, score);
730 
731                 // Test the held score
732                 score = game.calculateScore(currentNumbers, true);
733                 assertEquals(0, score);
734 
735                 // Test the calculateHighestScore method
736                 highestScore = game.calculateHighestScore(currentNumbers);
737                 assertEquals(100, highestScore[0]);
738                 highestRoll = (List<Integer>) highestScore[1];
739                 assertEquals(highestRoll.size(), 1);
740                 assertTrue(highestRoll.contains(1));
741                 assertTrue(!highestRoll.contains(2));
```

```
742                 assertTrue(!highestRoll.contains(3));
743                 assertTrue(!highestRoll.contains(4));
744                 assertTrue(!highestRoll.contains(5));
745                 assertTrue(!highestRoll.contains(6));
746             }
747         }
748 
749         /*
750          * Test 17: Check scoring for a die roll of 55A (A represents all die
751          * values other than 1 or 5) Requirement 6.2.0
752          */
753 
754         // Generate all permutations of the roll
755         automatedPermutations = integerPermutations("55A");
756 
757         // Convert the list of Strings to a list of lists of integers
758         for (String numbers : automatedPermutations) {
759             myPermutations = permutations(numbers);
760 
761             // Check each list of integers for proper scoring
762             for (List<Integer> currentNumbers : myPermutations) {
763                 // Test the total score
764                 score = game.calculateScore(currentNumbers, false);
765                 assertEquals(100, score);
766 
767                 // Test the held score
768                 score = game.calculateScore(currentNumbers, true);
769                 assertEquals(0, score);
770 
771                 // Test the calculateHighestScore method
772                 highestScore = game.calculateHighestScore(currentNumbers);
773                 assertEquals(100, highestScore[0]);
774                 highestRoll = (List<Integer>) highestScore[1];
775                 assertEquals(highestRoll.size(), 2);
776                 assertTrue(!highestRoll.contains(1));
777                 assertTrue(!highestRoll.contains(2));
778                 assertTrue(!highestRoll.contains(3));
779                 assertTrue(!highestRoll.contains(4));
780                 assertTrue(highestRoll.contains(5));
781                 assertTrue(!highestRoll.contains(6));
782             }
783         }
784 
785         /*
786          * Test 18: Check scoring for a die roll of 15A (A represents all die
787          * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
788          */
789 
790         // Generate all permutations of the roll
791         automatedPermutations = integerPermutations("15A");
792 
793         // Convert the list of Strings to a list of lists of integers
794         for (String numbers : automatedPermutations) {
795             myPermutations = permutations(numbers);
796 
797             // Check each list of integers for proper scoring
798             for (List<Integer> currentNumbers : myPermutations) {
```

```
799                 // Test the total score
800                 score = game.calculateScore(currentNumbers, false);
801                 assertEquals(150, score);
802 
803                 // Test the held score
804                 score = game.calculateScore(currentNumbers, true);
805                 assertEquals(0, score);
806 
807                 // Test the calculateHighestScore method
808                 highestScore = game.calculateHighestScore(currentNumbers);
809                 assertEquals(150, highestScore[0]);
810                 highestRoll = (List<Integer>) highestScore[1];
811                 assertEquals(highestRoll.size(), 2);
812                 assertTrue(highestRoll.contains(1));
813                 assertTrue(!highestRoll.contains(2));
814                 assertTrue(!highestRoll.contains(3));
815                 assertTrue(!highestRoll.contains(4));
816                 assertTrue(highestRoll.contains(5));
817                 assertTrue(!highestRoll.contains(6));
818             }
819         }
820 
821         /*
822          * Test 19: Check scoring for a die roll of 11A (A represents all die
823          * values other than 1 or 5) Requirement 6.1.0
824          */
825 
826         // Generate all permutations of the roll
827         automatedPermutations = integerPermutations("11A");
828 
829         // Convert the list of Strings to a list of lists of integers
830         for (String numbers : automatedPermutations) {
831             myPermutations = permutations(numbers);
832 
833             // Check each list of integers for proper scoring
834             for (List<Integer> currentNumbers : myPermutations) {
835                 // Test the total score
836                 score = game.calculateScore(currentNumbers, false);
837                 assertEquals(200, score);
838 
839                 // Test the held score
840                 score = game.calculateScore(currentNumbers, true);
841                 assertEquals(0, score);
842 
843                 // Test the calculateHighestScore method
844                 highestScore = game.calculateHighestScore(currentNumbers);
845                 assertEquals(200, highestScore[0]);
846                 highestRoll = (List<Integer>) highestScore[1];
847                 assertEquals(highestRoll.size(), 2);
848                 assertTrue(highestRoll.contains(1));
849                 assertTrue(!highestRoll.contains(2));
850                 assertTrue(!highestRoll.contains(3));
851                 assertTrue(!highestRoll.contains(4));
852                 assertTrue(!highestRoll.contains(5));
853                 assertTrue(!highestRoll.contains(6));
854             }
855         }
```

```
856
857         /*
858          * Test 20: Check scoring for a die roll of 155 Requirement 6.1.0 and
859          * 6.2.0
860          */
861
862         // Generate all permutations of the roll and store in a list of lists of
863         // integers
864         myPermutations = permutations("155");
865
866         // Check each list of integers for correct scoring
867         for (List<Integer> numbers : myPermutations) {
868             // Test the total score
869             score = game.calculateScore(numbers, false);
870             assertEquals(200, score);
871
872             // Test the held score
873             score = game.calculateScore(numbers, true);
874             assertEquals(200, score);
875
876             // Test the calculateHighestScore method
877             highestScore = game.calculateHighestScore(numbers);
878             assertEquals(200, highestScore[0]);
879             highestRoll = (List<Integer>) highestScore[1];
880             assertEquals(highestRoll.size(), 3);
881             assertTrue(highestRoll.contains(1));
882             assertTrue(!highestRoll.contains(2));
883             assertTrue(!highestRoll.contains(3));
884             assertTrue(!highestRoll.contains(4));
885             assertTrue(highestRoll.contains(5));
886             assertTrue(!highestRoll.contains(6));
887         }
888
889         /*
890          * Test 21: Check scoring for a die roll of 115 Requirement 6.1.0 and
891          * 6.2.0
892          */
893
894         // Generate all permutations of the roll and store in a list of lists of
895         // integers
896         myPermutations = permutations("115");
897
898         // Check each list of integers for correct scoring
899         for (List<Integer> numbers : myPermutations) {
900             // Test the total score
901             score = game.calculateScore(numbers, false);
902             assertEquals(250, score);
903
904             // Test the held score
905             score = game.calculateScore(numbers, true);
906             assertEquals(250, score);
907
908             // Test the calculateHighestScore method
909             highestScore = game.calculateHighestScore(numbers);
910             assertEquals(250, highestScore[0]);
911             highestRoll = (List<Integer>) highestScore[1];
912             assertEquals(highestRoll.size(), 3);
```

```
913             assertTrue(highestRoll.contains(1));
914             assertTrue(!highestRoll.contains(2));
915             assertTrue(!highestRoll.contains(3));
916             assertTrue(!highestRoll.contains(4));
917             assertTrue(highestRoll.contains(5));
918             assertTrue(!highestRoll.contains(6));
919         }
920 
921         /*
922          * Test 22: Check scoring for a die roll of AAA (A represents all die
923          * values other than 1 or 5) Requirement 6.4.0
924          */
925 
926         // Generate all permutations of the roll
927         automatedPermutations = integerPermutations("AAA");
928 
929         // Convert the list of Strings to a list of lists of integers
930         for (String numbers : automatedPermutations) {
931             myPermutations = permutations(numbers);
932 
933             // Check each list of integers for proper scoring
934             for (List<Integer> currentNumbers : myPermutations) {
935                 // Get the die value
936                 dieOneValue = currentNumbers.get(0);
937 
938                 // Test the total score
939                 score = game.calculateScore(currentNumbers, false);
940                 assertEquals(100 * dieOneValue, score);
941 
942                 // Test the held score
943                 score = game.calculateScore(currentNumbers, true);
944                 assertEquals(100 * dieOneValue, score);
945 
946                 // Test the calculateHighestScore method
947                 highestScore = game.calculateHighestScore(currentNumbers);
948                 assertEquals(100 * dieOneValue, highestScore[0]);
949                 highestRoll = (List<Integer>) highestScore[1];
950                 assertEquals(highestRoll.size(), 3);
951                 assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
952             }
953         }
954 
955         /*
956          * Test 23: Check scoring for a die roll of 555 Requirement 6.4.0
957          */
958 
959         // Generate all permutations of the roll and store in a list of lists of
960         // integers
961         myPermutations = permutations("555");
962 
963         // Check each list of integers for correct scoring
964         for (List<Integer> numbers : myPermutations) {
965             // Test the total score
966             score = game.calculateScore(numbers, false);
967             assertEquals(500, score);
968 
969             // Test the held score
```

```
970             score = game.calculateScore(numbers, true);
971             assertEquals(500, score);
972 
973             // Test the calculateHighestScore method
974             highestScore = game.calculateHighestScore(numbers);
975             assertEquals(500, highestScore[0]);
976             highestRoll = (List<Integer>) highestScore[1];
977             assertEquals(highestRoll.size(), 3);
978             assertTrue(Collections.frequency(highestRoll, 5) == 3);
979         }
980 
981         /*
982          * Test 24: Check scoring for a die roll of 111 Requirement 6.3.0
983          */
984 
985         // Generate all permutations of the roll and store in a list of lists of
986         // integers
987         myPermutations = permutations("111");
988 
989         // Check each list of integers for correct scoring
990         for (List<Integer> numbers : myPermutations) {
991             // Test the total score
992             score = game.calculateScore(numbers, false);
993             assertEquals(1000, score);
994 
995             // Test the held score
996             score = game.calculateScore(numbers, true);
997             assertEquals(1000, score);
998 
999             // Test the calculateHighestScore method
1000            highestScore = game.calculateHighestScore(numbers);
1001            assertEquals(1000, highestScore[0]);
1002            highestRoll = (List<Integer>) highestScore[1];
1003            assertEquals(highestRoll.size(), 3);
1004            assertTrue(Collections.frequency(highestRoll, 1) == 3);
1005        }
1006    }
1007 
1008    /**
1009     * This method tests the calculateScore method of the Game class for all
1010     * permutations of 4 dice, requirements 6.1.0, 6.2.0, 6.3.0, 6.4.0, and
1011     * 6.5.0: <br />
1012     * <br />
1013     * 6.1.0 Each 1 rolled is worth 100 points<br />
1014     * 6.2.0 Each 5 rolled is worth 50 points<br />
1015     * 6.3.0 Three 1â€™s are worth 1000 points<br />
1016     * 6.4.0 Three of a kind of any value other than 1 is worth 100 times the
1017     * value of the die (e.g. three 4â€™s is worth 400 points).<br />
1018     * 6.5.0 Four, five, or six of a kind is scored by doubling the three of a
1019     * kind value for every additional matching die (e.g. five 3â€™s would be
1020     * scored as 300 X 2 X 2 = 1200.
1021     */
1022    @SuppressWarnings("unchecked")
1023    @Test
1024    public void testCalculateScore4() {
1025 
1026        /*
```

```
1027         * Test 25: Check scoring for a die roll of AXYZ (A, X, Y and Z each
1028         * represent all die values other than 1 or 5)
1029         */
1030
1031        // Generate all permutations of the roll
1032        automatedPermutations = integerPermutations("AXYZ");
1033
1034        // Convert the list of Strings to a list of lists of integers
1035        for (String numbers : automatedPermutations) {
1036            myPermutations = permutations(numbers);
1037
1038            // Check each list of integers for proper scoring
1039            for (List<Integer> currentNumbers : myPermutations) {
1040                // Test the total score
1041                score = game.calculateScore(currentNumbers, false);
1042                assertEquals(0, score);
1043
1044                // Test the held score
1045                score = game.calculateScore(currentNumbers, true);
1046                assertEquals(0, score);
1047
1048                // Test the calculateHighestScore method
1049                highestScore = game.calculateHighestScore(currentNumbers);
1050                assertEquals(0, highestScore[0]);
1051                highestRoll = (List<Integer>) highestScore[1];
1052                assertTrue(highestRoll.isEmpty());
1053            }
1054        }
1055
1056        /*
1057         * Test 26: Check scoring for a die roll of AAXY (A, X and Y each
1058         * represent all die values other than 1 or 5)
1059         */
1060
1061        // Generate all permutations of the roll
1062        automatedPermutations = integerPermutations("AAXY");
1063
1064        // Convert the list of Strings to a list of lists of integers
1065        for (String numbers : automatedPermutations) {
1066            myPermutations = permutations(numbers);
1067
1068            // Check each list of integers for proper scoring
1069            for (List<Integer> currentNumbers : myPermutations) {
1070                // Test the total score
1071                score = game.calculateScore(currentNumbers, false);
1072                assertEquals(0, score);
1073
1074                // Test the held score
1075                score = game.calculateScore(currentNumbers, true);
1076                assertEquals(0, score);
1077
1078                // Test the calculateHighestScore method
1079                highestScore = game.calculateHighestScore(currentNumbers);
1080                assertEquals(0, highestScore[0]);
1081                highestRoll = (List<Integer>) highestScore[1];
1082                assertTrue(highestRoll.isEmpty());
1083            }
```

```
1084        }
1085
1086        /*
1087         * Test 27: Check scoring for a die roll of AAXX (A and X each represent
1088         * all die values other than 1 or 5)
1089         */
1090
1091        // Generate all permutations of the roll
1092        automatedPermutations = integerPermutations("AAXX");
1093
1094        // Convert the list of Strings to a list of lists of integers
1095        for (String numbers : automatedPermutations) {
1096            myPermutations = permutations(numbers);
1097
1098            // Check each list of integers for proper scoring
1099            for (List<Integer> currentNumbers : myPermutations) {
1100                // Test the total score
1101                score = game.calculateScore(currentNumbers, false);
1102                assertEquals(0, score);
1103
1104                // Test the held score
1105                score = game.calculateScore(currentNumbers, true);
1106                assertEquals(0, score);
1107
1108                // Test the calculateHighestScore method
1109                highestScore = game.calculateHighestScore(currentNumbers);
1110                assertEquals(0, highestScore[0]);
1111                highestRoll = (List<Integer>) highestScore[1];
1112                assertTrue(highestRoll.isEmpty());
1113            }
1114        }
1115
1116        /*
1117         * Test 28: Check scoring for a die roll of 5AXY (A, X and Y each
1118         * represent all die values other than 1 or 5) Requirement 6.2.0
1119         */
1120
1121        // Generate all permutations of the roll
1122        automatedPermutations = integerPermutations("5AXY");
1123
1124        // Convert the list of Strings to a list of lists of integers
1125        for (String numbers : automatedPermutations) {
1126            myPermutations = permutations(numbers);
1127
1128            // Check each list of integers for proper scoring
1129            for (List<Integer> currentNumbers : myPermutations) {
1130                // Test the total score
1131                score = game.calculateScore(currentNumbers, false);
1132                assertEquals(50, score);
1133
1134                // Test the held score
1135                score = game.calculateScore(currentNumbers, true);
1136                assertEquals(0, score);
1137
1138                // Test the calculateHighestScore method
1139                highestScore = game.calculateHighestScore(currentNumbers);
1140                assertEquals(50, highestScore[0]);
```

```
            highestRoll = (List<Integer>) highestScore[1];
            assertEquals(highestRoll.size(), 1);
            assertTrue(Collections.frequency(highestRoll, 5) == 1);
        }
    }

    /*
     * Test 29: Check scoring for a die roll of 5AAX (A and X each represent
     * all die values other than 1 or 5) Requirement 6.2.0
     */

    // Generate all permutations of the roll
    automatedPermutations = integerPermutations("5AAX");

    // Convert the list of Strings to a list of lists of integers
    for (String numbers : automatedPermutations) {
        myPermutations = permutations(numbers);

        // Check each list of integers for proper scoring
        for (List<Integer> currentNumbers : myPermutations) {
            // Test the total score
            score = game.calculateScore(currentNumbers, false);
            assertEquals(50, score);

            // Test the held score
            score = game.calculateScore(currentNumbers, true);
            assertEquals(0, score);

            // Test the calculateHighestScore method
            highestScore = game.calculateHighestScore(currentNumbers);
            assertEquals(50, highestScore[0]);
            highestRoll = (List<Integer>) highestScore[1];
            assertEquals(highestRoll.size(), 1);
            assertTrue(Collections.frequency(highestRoll, 5) == 1);
        }
    }

    /*
     * Test 30: Check scoring for a die roll of 1AXY (A, X and Y each
     * represent all die values other than 1 or 5) Requirement 6.1.0
     */

    // Generate all permutations of the roll
    automatedPermutations = integerPermutations("1AXY");

    // Convert the list of Strings to a list of lists of integers
    for (String numbers : automatedPermutations) {
        myPermutations = permutations(numbers);

        // Check each list of integers for proper scoring
        for (List<Integer> currentNumbers : myPermutations) {
            // Test the total score
            score = game.calculateScore(currentNumbers, false);
            assertEquals(100, score);

            // Test the held score
            score = game.calculateScore(currentNumbers, true);
```

```
1198                assertEquals(0, score);
1199
1200                // Test the calculateHighestScore method
1201                highestScore = game.calculateHighestScore(currentNumbers);
1202                assertEquals(100, highestScore[0]);
1203                highestRoll = (List<Integer>) highestScore[1];
1204                assertEquals(highestRoll.size(), 1);
1205                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1206            }
1207        }
1208
1209        /*
1210         * Test 31: Check scoring for a die roll of 1AAX (A and X each represent
1211         * all die values other than 1 or 5) Requirement 6.1.0
1212         */
1213
1214        // Generate all permutations of the roll
1215        automatedPermutations = integerPermutations("1AAX");
1216
1217        // Convert the list of Strings to a list of lists of integers
1218        for (String numbers : automatedPermutations) {
1219            myPermutations = permutations(numbers);
1220
1221            // Check each list of integers for proper scoring
1222            for (List<Integer> currentNumbers : myPermutations) {
1223                // Test the total score
1224                score = game.calculateScore(currentNumbers, false);
1225                assertEquals(100, score);
1226
1227                // Test the held score
1228                score = game.calculateScore(currentNumbers, true);
1229                assertEquals(0, score);
1230
1231                // Test the calculateHighestScore method
1232                highestScore = game.calculateHighestScore(currentNumbers);
1233                assertEquals(100, highestScore[0]);
1234                highestRoll = (List<Integer>) highestScore[1];
1235                assertEquals(highestRoll.size(), 1);
1236                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1237            }
1238        }
1239
1240        /*
1241         * Test 32: Check scoring for a die roll of 55AX (A and X each represent
1242         * all die values other than 1 or 5) Requirement 6.2.0
1243         */
1244
1245        // Generate all permutations of the roll
1246        automatedPermutations = integerPermutations("55AX");
1247
1248        // Convert the list of Strings to a list of lists of integers
1249        for (String numbers : automatedPermutations) {
1250            myPermutations = permutations(numbers);
1251
1252            // Check each list of integers for proper scoring
1253            for (List<Integer> currentNumbers : myPermutations) {
1254                // Test the total score
```

```
1255                score = game.calculateScore(currentNumbers, false);
1256                assertEquals(100, score);
1257
1258                // Test the held score
1259                score = game.calculateScore(currentNumbers, true);
1260                assertEquals(0, score);
1261
1262                // Test the calculateHighestScore method
1263                highestScore = game.calculateHighestScore(currentNumbers);
1264                assertEquals(100, highestScore[0]);
1265                highestRoll = (List<Integer>) highestScore[1];
1266                assertEquals(highestRoll.size(), 2);
1267                assertTrue(Collections.frequency(highestRoll, 5) == 2);
1268            }
1269        }
1270
1271        /*
1272         * Test 33: Check scoring for a die roll of 55AA (A represents all die
1273         * values other than 1 or 5) Requirement 6.2.0
1274         */
1275
1276        // Generate all permutations of the roll
1277        automatedPermutations = integerPermutations("55AA");
1278
1279        // Convert the list of Strings to a list of lists of integers
1280        for (String numbers : automatedPermutations) {
1281            myPermutations = permutations(numbers);
1282
1283            // Check each list of integers for proper scoring
1284            for (List<Integer> currentNumbers : myPermutations) {
1285                // Test the total score
1286                score = game.calculateScore(currentNumbers, false);
1287                assertEquals(100, score);
1288
1289                // Test the held score
1290                score = game.calculateScore(currentNumbers, true);
1291                assertEquals(0, score);
1292
1293                // Test the calculateHighestScore method
1294                highestScore = game.calculateHighestScore(currentNumbers);
1295                assertEquals(100, highestScore[0]);
1296                highestRoll = (List<Integer>) highestScore[1];
1297                assertEquals(highestRoll.size(), 2);
1298                assertTrue(Collections.frequency(highestRoll, 5) == 2);
1299            }
1300        }
1301
1302        /*
1303         * Test 34: Check scoring for a die roll of 15AX (A and X each represent
1304         * all die values other than 1 or 5) Requirement 6.1.0 and 6.2.0
1305         */
1306
1307        // Generate all permutations of the roll
1308        automatedPermutations = integerPermutations("15AX");
1309
1310        // Convert the list of Strings to a list of lists of integers
1311        for (String numbers : automatedPermutations) {
```

```
1312            myPermutations = permutations(numbers);
1313
1314            // Check each list of integers for proper scoring
1315            for (List<Integer> currentNumbers : myPermutations) {
1316                // Test the total score
1317                score = game.calculateScore(currentNumbers, false);
1318                assertEquals(150, score);
1319
1320                // Test the held score
1321                score = game.calculateScore(currentNumbers, true);
1322                assertEquals(0, score);
1323
1324                // Test the calculateHighestScore method
1325                highestScore = game.calculateHighestScore(currentNumbers);
1326                assertEquals(150, highestScore[0]);
1327                highestRoll = (List<Integer>) highestScore[1];
1328                assertEquals(highestRoll.size(), 2);
1329                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1330                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1331            }
1332        }
1333
1334        /*
1335         * Test 35: Check scoring for a die roll of 15AA (A represents all die
1336         * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
1337         */
1338
1339        // Generate all permutations of the roll
1340        automatedPermutations = integerPermutations("15AA");
1341
1342        // Convert the list of Strings to a list of lists of integers
1343        for (String numbers : automatedPermutations) {
1344            myPermutations = permutations(numbers);
1345
1346            // Check each list of integers for proper scoring
1347            for (List<Integer> currentNumbers : myPermutations) {
1348                // Test the total score
1349                score = game.calculateScore(currentNumbers, false);
1350                assertEquals(150, score);
1351
1352                // Test the held score
1353                score = game.calculateScore(currentNumbers, true);
1354                assertEquals(0, score);
1355
1356                // Test the calculateHighestScore method
1357                highestScore = game.calculateHighestScore(currentNumbers);
1358                assertEquals(150, highestScore[0]);
1359                highestRoll = (List<Integer>) highestScore[1];
1360                assertEquals(highestRoll.size(), 2);
1361                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1362                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1363            }
1364        }
1365
1366        /*
1367         * Test 36: Check scoring for a die roll of 11AX (A and X each represent
1368         * all die values other than 1 or 5) Requirement 6.1.0
```

```
1369         */
1370
1371        // Generate all permutations of the roll
1372        automatedPermutations = integerPermutations("11AX");
1373
1374        // Convert the list of Strings to a list of lists of integers
1375        for (String numbers : automatedPermutations) {
1376            myPermutations = permutations(numbers);
1377
1378            // Check each list of integers for proper scoring
1379            for (List<Integer> currentNumbers : myPermutations) {
1380                // Test the total score
1381                score = game.calculateScore(currentNumbers, false);
1382                assertEquals(200, score);
1383
1384                // Test the held score
1385                score = game.calculateScore(currentNumbers, true);
1386                assertEquals(0, score);
1387
1388                // Test the calculateHighestScore method
1389                highestScore = game.calculateHighestScore(currentNumbers);
1390                assertEquals(200, highestScore[0]);
1391                highestRoll = (List<Integer>) highestScore[1];
1392                assertEquals(highestRoll.size(), 2);
1393                assertTrue(Collections.frequency(highestRoll, 1) == 2);
1394            }
1395        }
1396
1397        /*
1398         * Test 37: Check scoring for a die roll of 55AA (A represents all die
1399         * values other than 1 or 5) Requirement 6.2.0
1400         */
1401
1402        // Generate all permutations of the roll
1403        automatedPermutations = integerPermutations("11AA");
1404
1405        // Convert the list of Strings to a list of lists of integers
1406        for (String numbers : automatedPermutations) {
1407            myPermutations = permutations(numbers);
1408
1409            // Check each list of integers for proper scoring
1410            for (List<Integer> currentNumbers : myPermutations) {
1411                // Test the total score
1412                score = game.calculateScore(currentNumbers, false);
1413                assertEquals(200, score);
1414
1415                // Test the held score
1416                score = game.calculateScore(currentNumbers, true);
1417                assertEquals(0, score);
1418
1419                // Test the calculateHighestScore method
1420                highestScore = game.calculateHighestScore(currentNumbers);
1421                assertEquals(200, highestScore[0]);
1422                highestRoll = (List<Integer>) highestScore[1];
1423                assertEquals(highestRoll.size(), 2);
1424                assertTrue(Collections.frequency(highestRoll, 1) == 2);
1425            }
```

```
1426        }
1427
1428        /*
1429         * Test 38: Check scoring for a die roll of 155A (A represents all die
1430         * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
1431         */
1432
1433        // Generate all permutations of the roll
1434        automatedPermutations = integerPermutations("155A");
1435
1436        // Convert the list of Strings to a list of lists of integers
1437        for (String numbers : automatedPermutations) {
1438            myPermutations = permutations(numbers);
1439
1440            // Check each list of integers for proper scoring
1441            for (List<Integer> currentNumbers : myPermutations) {
1442                // Test the total score
1443                score = game.calculateScore(currentNumbers, false);
1444                assertEquals(200, score);
1445
1446                // Test the held score
1447                score = game.calculateScore(currentNumbers, true);
1448                assertEquals(0, score);
1449
1450                // Test the calculateHighestScore method
1451                highestScore = game.calculateHighestScore(currentNumbers);
1452                assertEquals(200, highestScore[0]);
1453                highestRoll = (List<Integer>) highestScore[1];
1454                assertEquals(highestRoll.size(), 3);
1455                assertTrue(Collections.frequency(highestRoll, 5) == 2);
1456                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1457            }
1458        }
1459
1460        /*
1461         * Test 39: Check scoring for a die roll of 115A (A represents all die
1462         * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
1463         */
1464
1465        // Generate all permutations of the roll
1466        automatedPermutations = integerPermutations("115A");
1467
1468        // Convert the list of Strings to a list of lists of integers
1469        for (String numbers : automatedPermutations) {
1470            myPermutations = permutations(numbers);
1471
1472            // Check each list of integers for proper scoring
1473            for (List<Integer> currentNumbers : myPermutations) {
1474                // Test the total score
1475                score = game.calculateScore(currentNumbers, false);
1476                assertEquals(250, score);
1477
1478                // Test the held score
1479                score = game.calculateScore(currentNumbers, true);
1480                assertEquals(0, score);
1481
1482                // Test the calculateHighestScore method
```

```
1483                highestScore = game.calculateHighestScore(currentNumbers);
1484                assertEquals(250, highestScore[0]);
1485                highestRoll = (List<Integer>) highestScore[1];
1486                assertEquals(highestRoll.size(), 3);
1487                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1488                assertTrue(Collections.frequency(highestRoll, 1) == 2);
1489            }
1490        }
1491
1492        /*
1493         * Test 40: Check scoring for a die roll of AAAX (A and X represent all
1494         * die values other than 1 or 5) Requirement 6.4.0
1495         */
1496
1497        // Generate all permutations of the roll
1498        automatedPermutations = integerPermutations("AAAX");
1499
1500        // Convert the list of Strings to a list of lists of integers
1501        for (String numbers : automatedPermutations) {
1502            myPermutations = permutations(numbers);
1503
1504            // Get the die value for calculating the score
1505            dieOneValue = Character.getNumericValue(numbers.charAt(0));
1506
1507            // Check each list of integers for proper scoring
1508            for (List<Integer> currentNumbers : myPermutations) {
1509                // Test the total score
1510                score = game.calculateScore(currentNumbers, false);
1511                assertEquals(100 * dieOneValue, score);
1512
1513                // Test the held score
1514                score = game.calculateScore(currentNumbers, true);
1515                assertEquals(0, score);
1516
1517                // Test the calculateHighestScore method
1518                highestScore = game.calculateHighestScore(currentNumbers);
1519                assertEquals(100 * dieOneValue, highestScore[0]);
1520                highestRoll = (List<Integer>) highestScore[1];
1521                assertEquals(highestRoll.size(), 3);
1522                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
1523            }
1524        }
1525
1526        /*
1527         * Test 41: Check scoring for a die roll of 555A (A represents all die
1528         * values other than 1 or 5) Requirement 6.4.0
1529         */
1530
1531        // Generate all permutations of the roll
1532        automatedPermutations = integerPermutations("555A");
1533
1534        // Convert the list of Strings to a list of lists of integers
1535        for (String numbers : automatedPermutations) {
1536            myPermutations = permutations(numbers);
1537
1538            // Check each list of integers for proper scoring
1539            for (List<Integer> currentNumbers : myPermutations) {
```

```
1540                // Test the total score
1541                score = game.calculateScore(currentNumbers, false);
1542                assertEquals(500, score);
1543
1544                // Test the held score
1545                score = game.calculateScore(currentNumbers, true);
1546                assertEquals(0, score);
1547
1548                // Test the calculateHighestScore method
1549                highestScore = game.calculateHighestScore(currentNumbers);
1550                assertEquals(500, highestScore[0]);
1551                highestRoll = (List<Integer>) highestScore[1];
1552                assertEquals(highestRoll.size(), 3);
1553                assertTrue(Collections.frequency(highestRoll, 5) == 3);
1554            }
1555        }
1556
1557        /*
1558         * Test 42: Check scoring for a die roll of 111A (A represents all die
1559         * values other than 1 or 5) Requirement 6.3.0
1560         */
1561
1562        // Generate all permutations of the roll
1563        automatedPermutations = integerPermutations("111A");
1564
1565        // Convert the list of Strings to a list of lists of integers
1566        for (String numbers : automatedPermutations) {
1567            myPermutations = permutations(numbers);
1568
1569            // Check each list of integers for proper scoring
1570            for (List<Integer> currentNumbers : myPermutations) {
1571                // Test the total score
1572                score = game.calculateScore(currentNumbers, false);
1573                assertEquals(1000, score);
1574
1575                // Test the held score
1576                score = game.calculateScore(currentNumbers, true);
1577                assertEquals(0, score);
1578
1579                // Test the calculateHighestScore method
1580                highestScore = game.calculateHighestScore(currentNumbers);
1581                assertEquals(1000, highestScore[0]);
1582                highestRoll = (List<Integer>) highestScore[1];
1583                assertEquals(highestRoll.size(), 3);
1584                assertTrue(Collections.frequency(highestRoll, 1) == 3);
1585            }
1586        }
1587
1588        /*
1589         * Test 43: Check scoring for a die roll of 5AAA (A represents all die
1590         * values other than 1 or 5) Requirement 6.2.0 and 6.4.0
1591         */
1592
1593        // Generate all permutations of the roll
1594        automatedPermutations = integerPermutations("5AAA");
1595
1596        // Convert the list of Strings to a list of lists of integers
```

```
1597        for (String numbers : automatedPermutations) {
1598            myPermutations = permutations(numbers);
1599
1600            // Get the die value for calculating the score
1601            dieOneValue = Character.getNumericValue(numbers.charAt(1));
1602
1603            // Check each list of integers for proper scoring
1604            for (List<Integer> currentNumbers : myPermutations) {
1605                // Test the total score
1606                score = game.calculateScore(currentNumbers, false);
1607                assertEquals(100 * dieOneValue + 50, score);
1608
1609                // Test the held score
1610                score = game.calculateScore(currentNumbers, true);
1611                assertEquals(100 * dieOneValue + 50, score);
1612
1613                // Test the calculateHighestScore method
1614                highestScore = game.calculateHighestScore(currentNumbers);
1615                assertEquals(100 * dieOneValue + 50, highestScore[0]);
1616                highestRoll = (List<Integer>) highestScore[1];
1617                assertEquals(highestRoll.size(), 4);
1618                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1619                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
1620            }
1621        }
1622
1623        /*
1624         * Test 44: Check scoring for a die roll of 1AAA (A represents all die
1625         * values other than 1 or 5) Requirement 6.1.0 and 6.4.0
1626         */
1627
1628        // Generate all permutations of the roll
1629        automatedPermutations = integerPermutations("1AAA");
1630
1631        // Convert the list of Strings to a list of lists of integers
1632        for (String numbers : automatedPermutations) {
1633            myPermutations = permutations(numbers);
1634
1635            // Get the die value for calculating the score
1636            dieOneValue = Character.getNumericValue(numbers.charAt(1));
1637
1638            // Check each list of integers for proper scoring
1639            for (List<Integer> currentNumbers : myPermutations) {
1640                // Test the total score
1641                score = game.calculateScore(currentNumbers, false);
1642                assertEquals(100 * dieOneValue + 100, score);
1643
1644                // Test the held score
1645                score = game.calculateScore(currentNumbers, true);
1646                assertEquals(100 * dieOneValue + 100, score);
1647
1648                // Test the calculateHighestScore method
1649                highestScore = game.calculateHighestScore(currentNumbers);
1650                assertEquals(100 * dieOneValue + 100, highestScore[0]);
1651                highestRoll = (List<Integer>) highestScore[1];
1652                assertEquals(highestRoll.size(), 4);
1653                assertTrue(Collections.frequency(highestRoll, 1) == 1);
```

```
1654                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
1655            }
1656        }
1657
1658        /*
1659         * Test 45: Check scoring for a die roll of 1555 Requirement 6.1.0 and
1660         * 6.4.0
1661         */
1662
1663        // Generate all permutations of the roll and store in a list of lists of
1664        // integers
1665        myPermutations = permutations("1555");
1666
1667        // Check each list of integers for correct scoring
1668        for (List<Integer> numbers : myPermutations) {
1669            // Test the total score
1670            score = game.calculateScore(numbers, false);
1671            assertEquals(600, score);
1672
1673            // Test the held score
1674            score = game.calculateScore(numbers, true);
1675            assertEquals(600, score);
1676
1677            // Test the calculateHighestScore method
1678            highestScore = game.calculateHighestScore(numbers);
1679            assertEquals(600, highestScore[0]);
1680            highestRoll = (List<Integer>) highestScore[1];
1681            assertEquals(highestRoll.size(), 4);
1682            assertTrue(Collections.frequency(highestRoll, 5) == 3);
1683            assertTrue(Collections.frequency(highestRoll, 1) == 1);
1684        }
1685
1686        /*
1687         * Test 46: Check scoring for a die roll of 5111 Requirement 6.2.0 and
1688         * 6.3.0
1689         */
1690
1691        // Generate all permutations of the roll and store in a list of lists of
1692        // integers
1693        myPermutations = permutations("5111");
1694
1695        // Check each list of integers for correct scoring
1696        for (List<Integer> numbers : myPermutations) {
1697            // Test the total score
1698            score = game.calculateScore(numbers, false);
1699            assertEquals(1050, score);
1700
1701            // Test the held score
1702            score = game.calculateScore(numbers, true);
1703            assertEquals(1050, score);
1704
1705            // Test the calculateHighestScore method
1706            highestScore = game.calculateHighestScore(numbers);
1707            assertEquals(1050, highestScore[0]);
1708            highestRoll = (List<Integer>) highestScore[1];
1709            assertEquals(highestRoll.size(), 4);
1710            assertTrue(Collections.frequency(highestRoll, 5) == 1);
```

```
1711            assertTrue(Collections.frequency(highestRoll, 1) == 3);
1712        }
1713
1714        /*
1715         * Test 47: Check scoring for a die roll of AAAA (A represents all die
1716         * values other than 1 or 5) Requirement 6.5.0
1717         */
1718
1719        // Generate all permutations of the roll
1720        automatedPermutations = integerPermutations("AAAA");
1721
1722        // Convert the list of Strings to a list of lists of integers
1723        for (String numbers : automatedPermutations) {
1724            myPermutations = permutations(numbers);
1725
1726            // Get the die value for calculating the score
1727            dieOneValue = Character.getNumericValue(numbers.charAt(1));
1728
1729            // Check each list of integers for proper scoring
1730            for (List<Integer> currentNumbers : myPermutations) {
1731                // Test the total score
1732                score = game.calculateScore(currentNumbers, false);
1733                assertEquals(100 * dieOneValue * 2, score);
1734
1735                // Test the held score
1736                score = game.calculateScore(currentNumbers, true);
1737                assertEquals(100 * dieOneValue * 2, score);
1738
1739                // Test the calculateHighestScore method
1740                highestScore = game.calculateHighestScore(currentNumbers);
1741                assertEquals(100 * dieOneValue * 2, highestScore[0]);
1742                highestRoll = (List<Integer>) highestScore[1];
1743                assertEquals(highestRoll.size(), 4);
1744                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
1745            }
1746        }
1747
1748        /*
1749         * Test 48: Check scoring for a die roll of 5555 Requirement 6.5.0
1750         */
1751
1752        // Generate all permutations of the roll and store in a list of lists of
1753        // integers
1754        myPermutations = permutations("5555");
1755
1756        // Check each list of integers for correct scoring
1757        for (List<Integer> numbers : myPermutations) {
1758            // Test the total score
1759            score = game.calculateScore(numbers, false);
1760            assertEquals(1000, score);
1761
1762            // Test the held score
1763            score = game.calculateScore(numbers, true);
1764            assertEquals(1000, score);
1765
1766            // Test the calculateHighestScore method
1767            highestScore = game.calculateHighestScore(numbers);
```

```
1768            assertEquals(1000, highestScore[0]);
1769            highestRoll = (List<Integer>) highestScore[1];
1770            assertEquals(highestRoll.size(), 4);
1771            assertTrue(Collections.frequency(highestRoll, 5) == 4);
1772        }
1773
1774        /*
1775         * Test 49: Check scoring for a die roll of 1111 Requirement 6.5.0
1776         */
1777
1778        // Generate all permutations of the roll and store in a list of lists of
1779        // integers
1780        myPermutations = permutations("1111");
1781
1782        // Check each list of integers for correct scoring
1783        for (List<Integer> numbers : myPermutations) {
1784            // Test the total score
1785            score = game.calculateScore(numbers, false);
1786            assertEquals(2000, score);
1787
1788            // Test the held score
1789            score = game.calculateScore(numbers, true);
1790            assertEquals(2000, score);
1791
1792            // Test the calculateHighestScore method
1793            highestScore = game.calculateHighestScore(numbers);
1794            assertEquals(2000, highestScore[0]);
1795            highestRoll = (List<Integer>) highestScore[1];
1796            assertEquals(highestRoll.size(), 4);
1797            assertTrue(Collections.frequency(highestRoll, 1) == 4);
1798        }
1799    }
1800
1801    /**
1802     * This method tests the calculateScore method of the Game class for all
1803     * permutations of 5 dice, requirements 6.1.0, 6.2.0, 6.3.0, 6.4.0, and
1804     * 6.5.0:<br />
1805     * <br />
1806     * 6.1.0 Each 1 rolled is worth 100 points<br />
1807     * 6.2.0 Each 5 rolled is worth 50 points<br />
1808     * 6.3.0 Three 1â€™s are worth 1000 points<br />
1809     * 6.4.0 Three of a kind of any value other than 1 is worth 100 times the
1810     * value of the die (e.g. three 4â€™s is worth 400 points).<br />
1811     * 6.5.0 Four, five, or six of a kind is scored by doubling the three of a
1812     * kind value for every additional matching die (e.g. five 3â€™s would be
1813     * scored as 300 X 2 X 2 = 1200.
1814     */
1815    @SuppressWarnings("unchecked")
1816    @Test
1817    public void testCalculateScore5() {
1818        /*
1819         * Test 50: Check scoring for a die roll of AAXYZ (A, X, Y and Z each
1820         * represent all die values other than 1 or 5)
1821         */
1822
1823        // Generate all permutations of the roll
1824        automatedPermutations = integerPermutations("AAXYZ");
```

```
1825
1826        // Convert the list of Strings to a list of lists of integers
1827        for (String numbers : automatedPermutations) {
1828            myPermutations = permutations(numbers);
1829
1830            // Check each list of integers for proper scoring
1831            for (List<Integer> currentNumbers : myPermutations) {
1832                // Test the total score
1833                score = game.calculateScore(currentNumbers, false);
1834                assertEquals(0, score);
1835
1836                // Test the held score
1837                score = game.calculateScore(currentNumbers, true);
1838                assertEquals(0, score);
1839
1840                // Test the calculateHighestScore method
1841                highestScore = game.calculateHighestScore(currentNumbers);
1842                assertEquals(0, highestScore[0]);
1843                highestRoll = (List<Integer>) highestScore[1];
1844                assertTrue(highestRoll.isEmpty());
1845            }
1846        }
1847
1848        /*
1849         * Test 51: Check scoring for a die roll of AAXXY (A, X and Y each
1850         * represent all die values other than 1 or 5)
1851         */
1852
1853        // Generate all permutations of the roll
1854        automatedPermutations = integerPermutations("AAXXY");
1855
1856        // Convert the list of Strings to a list of lists of integers
1857        for (String numbers : automatedPermutations) {
1858            myPermutations = permutations(numbers);
1859
1860            // Check each list of integers for proper scoring
1861            for (List<Integer> currentNumbers : myPermutations) {
1862                // Test the total score
1863                score = game.calculateScore(currentNumbers, false);
1864                assertEquals(0, score);
1865
1866                // Test the held score
1867                score = game.calculateScore(currentNumbers, true);
1868                assertEquals(0, score);
1869
1870                // Test the calculateHighestScore method
1871                highestScore = game.calculateHighestScore(currentNumbers);
1872                assertEquals(0, highestScore[0]);
1873                highestRoll = (List<Integer>) highestScore[1];
1874                assertTrue(highestRoll.isEmpty());
1875            }
1876        }
1877
1878        /*
1879         * Test 52: Check scoring for a die roll of 5AXYZ (A, X, Y and Z each
1880         * represent all die values other than 1 or 5) Requirement 6.2.0
1881         */
```

```
1882
1883        // Generate all permutations of the roll
1884        automatedPermutations = integerPermutations("5AXYZ");
1885
1886        // Convert the list of Strings to a list of lists of integers
1887        for (String numbers : automatedPermutations) {
1888            myPermutations = permutations(numbers);
1889
1890            // Check each list of integers for proper scoring
1891            for (List<Integer> currentNumbers : myPermutations) {
1892                // Test the total score
1893                score = game.calculateScore(currentNumbers, false);
1894                assertEquals(50, score);
1895
1896                // Test the held score
1897                score = game.calculateScore(currentNumbers, true);
1898                assertEquals(0, score);
1899
1900                // Test the calculateHighestScore method
1901                highestScore = game.calculateHighestScore(currentNumbers);
1902                assertEquals(50, highestScore[0]);
1903                highestRoll = (List<Integer>) highestScore[1];
1904                assertEquals(highestRoll.size(), 1);
1905                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1906            }
1907        }
1908
1909        /*
1910         * Test 53: Check scoring for a die roll of 5AAXY (A, X and Y each
1911         * represent all die values other than 1 or 5) Requirement 6.2.0
1912         */
1913
1914        // Generate all permutations of the roll
1915        automatedPermutations = integerPermutations("5AAXY");
1916
1917        // Convert the list of Strings to a list of lists of integers
1918        for (String numbers : automatedPermutations) {
1919            myPermutations = permutations(numbers);
1920
1921            // Check each list of integers for proper scoring
1922            for (List<Integer> currentNumbers : myPermutations) {
1923                // Test the total score
1924                score = game.calculateScore(currentNumbers, false);
1925                assertEquals(50, score);
1926
1927                // Test the held score
1928                score = game.calculateScore(currentNumbers, true);
1929                assertEquals(0, score);
1930
1931                // Test the calculateHighestScore method
1932                highestScore = game.calculateHighestScore(currentNumbers);
1933                assertEquals(50, highestScore[0]);
1934                highestRoll = (List<Integer>) highestScore[1];
1935                assertEquals(highestRoll.size(), 1);
1936                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1937            }
1938        }
```

```
1939
1940        /*
1941         * Test 54: Check scoring for a die roll of 5AAXX (A and X each
1942         * represent all die values other than 1 or 5) Requirement 6.2.0
1943         */
1944
1945        // Generate all permutations of the roll
1946        automatedPermutations = integerPermutations("5AAXX");
1947
1948        // Convert the list of Strings to a list of lists of integers
1949        for (String numbers : automatedPermutations) {
1950            myPermutations = permutations(numbers);
1951
1952            // Check each list of integers for proper scoring
1953            for (List<Integer> currentNumbers : myPermutations) {
1954                // Test the total score
1955                score = game.calculateScore(currentNumbers, false);
1956                assertEquals(50, score);
1957
1958                // Test the held score
1959                score = game.calculateScore(currentNumbers, true);
1960                assertEquals(0, score);
1961
1962                // Test the calculateHighestScore method
1963                highestScore = game.calculateHighestScore(currentNumbers);
1964                assertEquals(50, highestScore[0]);
1965                highestRoll = (List<Integer>) highestScore[1];
1966                assertEquals(highestRoll.size(), 1);
1967                assertTrue(Collections.frequency(highestRoll, 5) == 1);
1968            }
1969        }
1970
1971        /*
1972         * Test 55: Check scoring for a die roll of 1AXYZ (A, X, Y and Z each
1973         * represent all die values other than 1 or 5) Requirement 6.1.0
1974         */
1975
1976        // Generate all permutations of the roll
1977        automatedPermutations = integerPermutations("1AXYZ");
1978
1979        // Convert the list of Strings to a list of lists of integers
1980        for (String numbers : automatedPermutations) {
1981            myPermutations = permutations(numbers);
1982
1983            // Check each list of integers for proper scoring
1984            for (List<Integer> currentNumbers : myPermutations) {
1985                // Test the total score
1986                score = game.calculateScore(currentNumbers, false);
1987                assertEquals(100, score);
1988
1989                // Test the held score
1990                score = game.calculateScore(currentNumbers, true);
1991                assertEquals(0, score);
1992
1993                // Test the calculateHighestScore method
1994                highestScore = game.calculateHighestScore(currentNumbers);
1995                assertEquals(100, highestScore[0]);
```

```
1996                highestRoll = (List<Integer>) highestScore[1];
1997                assertEquals(highestRoll.size(), 1);
1998                assertTrue(Collections.frequency(highestRoll, 1) == 1);
1999            }
2000        }
2001
2002        /*
2003         * Test 56: Check scoring for a die roll of 1AAXY (A, X and Y each
2004         * represent all die values other than 1 or 5) Requirement 6.1.0
2005         */
2006
2007        // Generate all permutations of the roll
2008        automatedPermutations = integerPermutations("1AAXY");
2009
2010        // Convert the list of Strings to a list of lists of integers
2011        for (String numbers : automatedPermutations) {
2012            myPermutations = permutations(numbers);
2013
2014            // Check each list of integers for proper scoring
2015            for (List<Integer> currentNumbers : myPermutations) {
2016                // Test the total score
2017                score = game.calculateScore(currentNumbers, false);
2018                assertEquals(100, score);
2019
2020                // Test the held score
2021                score = game.calculateScore(currentNumbers, true);
2022                assertEquals(0, score);
2023
2024                // Test the calculateHighestScore method
2025                highestScore = game.calculateHighestScore(currentNumbers);
2026                assertEquals(100, highestScore[0]);
2027                highestRoll = (List<Integer>) highestScore[1];
2028                assertEquals(highestRoll.size(), 1);
2029                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2030            }
2031        }
2032
2033        /*
2034         * Test 57: Check scoring for a die roll of 1AAXX (A and X each
2035         * represent all die values other than 1 or 5) Requirement 6.1.0
2036         */
2037
2038        // Generate all permutations of the roll
2039        automatedPermutations = integerPermutations("1AAXX");
2040
2041        // Convert the list of Strings to a list of lists of integers
2042        for (String numbers : automatedPermutations) {
2043            myPermutations = permutations(numbers);
2044
2045            // Check each list of integers for proper scoring
2046            for (List<Integer> currentNumbers : myPermutations) {
2047                // Test the total score
2048                score = game.calculateScore(currentNumbers, false);
2049                assertEquals(100, score);
2050
2051                // Test the held score
2052                score = game.calculateScore(currentNumbers, true);
```

```
2053                assertEquals(0, score);
2054
2055                // Test the calculateHighestScore method
2056                highestScore = game.calculateHighestScore(currentNumbers);
2057                assertEquals(100, highestScore[0]);
2058                highestRoll = (List<Integer>) highestScore[1];
2059                assertEquals(highestRoll.size(), 1);
2060                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2061            }
2062        }
2063
2064        /*
2065         * Test 58: Check scoring for a die roll of 55AXY (A, X and Y each
2066         * represent all die values other than 1 or 5) Requirement 6.2.0
2067         */
2068
2069        // Generate all permutations of the roll
2070        automatedPermutations = integerPermutations("55AXY");
2071
2072        // Convert the list of Strings to a list of lists of integers
2073        for (String numbers : automatedPermutations) {
2074            myPermutations = permutations(numbers);
2075
2076            // Check each list of integers for proper scoring
2077            for (List<Integer> currentNumbers : myPermutations) {
2078                // Test the total score
2079                score = game.calculateScore(currentNumbers, false);
2080                assertEquals(100, score);
2081
2082                // Test the held score
2083                score = game.calculateScore(currentNumbers, true);
2084                assertEquals(0, score);
2085
2086                // Test the calculateHighestScore method
2087                highestScore = game.calculateHighestScore(currentNumbers);
2088                assertEquals(100, highestScore[0]);
2089                highestRoll = (List<Integer>) highestScore[1];
2090                assertEquals(highestRoll.size(), 2);
2091                assertTrue(Collections.frequency(highestRoll, 5) == 2);
2092            }
2093        }
2094
2095        /*
2096         * Test 59: Check scoring for a die roll of 55AAX (A and X each
2097         * represent all die values other than 1 or 5) Requirement 6.2.0
2098         */
2099
2100        // Generate all permutations of the roll
2101        automatedPermutations = integerPermutations("55AAX");
2102
2103        // Convert the list of Strings to a list of lists of integers
2104        for (String numbers : automatedPermutations) {
2105            myPermutations = permutations(numbers);
2106
2107            // Check each list of integers for proper scoring
2108            for (List<Integer> currentNumbers : myPermutations) {
2109                // Test the total score
```

```
2110                score = game.calculateScore(currentNumbers, false);
2111                assertEquals(100, score);
2112
2113                // Test the held score
2114                score = game.calculateScore(currentNumbers, true);
2115                assertEquals(0, score);
2116
2117                // Test the calculateHighestScore method
2118                highestScore = game.calculateHighestScore(currentNumbers);
2119                assertEquals(100, highestScore[0]);
2120                highestRoll = (List<Integer>) highestScore[1];
2121                assertEquals(highestRoll.size(), 2);
2122                assertTrue(Collections.frequency(highestRoll, 5) == 2);
2123            }
2124        }
2125
2126        /*
2127         * Test 60: Check scoring for a die roll of 15AXY (A, X and Y each
2128         * represent all die values other than 1 or 5) Requirement 6.1.0 and
2129         * 6.2.0
2130         */
2131
2132        // Generate all permutations of the roll
2133        automatedPermutations = integerPermutations("15AXY");
2134
2135        // Convert the list of Strings to a list of lists of integers
2136        for (String numbers : automatedPermutations) {
2137            myPermutations = permutations(numbers);
2138
2139            // Check each list of integers for proper scoring
2140            for (List<Integer> currentNumbers : myPermutations) {
2141                // Test the total score
2142                score = game.calculateScore(currentNumbers, false);
2143                assertEquals(150, score);
2144
2145                // Test the held score
2146                score = game.calculateScore(currentNumbers, true);
2147                assertEquals(0, score);
2148
2149                // Test the calculateHighestScore method
2150                highestScore = game.calculateHighestScore(currentNumbers);
2151                assertEquals(150, highestScore[0]);
2152                highestRoll = (List<Integer>) highestScore[1];
2153                assertEquals(highestRoll.size(), 2);
2154                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2155                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2156            }
2157        }
2158
2159        /*
2160         * Test 61: Check scoring for a die roll of 15AAX (A and X each
2161         * represent all die values other than 1 or 5) Requirement 6.1.0 and
2162         * 6.2.0
2163         */
2164
2165        // Generate all permutations of the roll
2166        automatedPermutations = integerPermutations("15AAX");
```

```
2167
2168        // Convert the list of Strings to a list of lists of integers
2169        for (String numbers : automatedPermutations) {
2170            myPermutations = permutations(numbers);
2171
2172            // Check each list of integers for proper scoring
2173            for (List<Integer> currentNumbers : myPermutations) {
2174                // Test the total score
2175                score = game.calculateScore(currentNumbers, false);
2176                assertEquals(150, score);
2177
2178                // Test the held score
2179                score = game.calculateScore(currentNumbers, true);
2180                assertEquals(0, score);
2181
2182                // Test the calculateHighestScore method
2183                highestScore = game.calculateHighestScore(currentNumbers);
2184                assertEquals(150, highestScore[0]);
2185                highestRoll = (List<Integer>) highestScore[1];
2186                assertEquals(highestRoll.size(), 2);
2187                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2188                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2189            }
2190        }
2191
2192        /*
2193         * Test 62: Check scoring for a die roll of 11AXY (A, X and Y each
2194         * represent all die values other than 1 or 5) Requirement 6.1.0
2195         */
2196
2197        // Generate all permutations of the roll
2198        automatedPermutations = integerPermutations("11AXY");
2199
2200        // Convert the list of Strings to a list of lists of integers
2201        for (String numbers : automatedPermutations) {
2202            myPermutations = permutations(numbers);
2203
2204            // Check each list of integers for proper scoring
2205            for (List<Integer> currentNumbers : myPermutations) {
2206                // Test the total score
2207                score = game.calculateScore(currentNumbers, false);
2208                assertEquals(200, score);
2209
2210                // Test the held score
2211                score = game.calculateScore(currentNumbers, true);
2212                assertEquals(0, score);
2213
2214                // Test the calculateHighestScore method
2215                highestScore = game.calculateHighestScore(currentNumbers);
2216                assertEquals(200, highestScore[0]);
2217                highestRoll = (List<Integer>) highestScore[1];
2218                assertEquals(highestRoll.size(), 2);
2219                assertTrue(Collections.frequency(highestRoll, 1) == 2);
2220            }
2221        }
2222
2223        /*
```

```
2224         * Test 63: Check scoring for a die roll of 11AAX (A and X each
2225         * represent all die values other than 1 or 5) Requirement 6.1.0
2226         */
2227
2228        // Generate all permutations of the roll
2229        automatedPermutations = integerPermutations("11AAX");
2230
2231        // Convert the list of Strings to a list of lists of integers
2232        for (String numbers : automatedPermutations) {
2233            myPermutations = permutations(numbers);
2234
2235            // Check each list of integers for proper scoring
2236            for (List<Integer> currentNumbers : myPermutations) {
2237                // Test the total score
2238                score = game.calculateScore(currentNumbers, false);
2239                assertEquals(200, score);
2240
2241                // Test the held score
2242                score = game.calculateScore(currentNumbers, true);
2243                assertEquals(0, score);
2244
2245                // Test the calculateHighestScore method
2246                highestScore = game.calculateHighestScore(currentNumbers);
2247                assertEquals(200, highestScore[0]);
2248                highestRoll = (List<Integer>) highestScore[1];
2249                assertEquals(highestRoll.size(), 2);
2250                assertTrue(Collections.frequency(highestRoll, 1) == 2);
2251            }
2252        }
2253
2254        /*
2255         * Test 64: Check scoring for a die roll of 155AX (A and X each
2256         * represent all die values other than 1 or 5) Requirement 6.1.0 and
2257         * 6.2.0
2258         */
2259
2260        // Generate all permutations of the roll
2261        automatedPermutations = integerPermutations("155AX");
2262
2263        // Convert the list of Strings to a list of lists of integers
2264        for (String numbers : automatedPermutations) {
2265            myPermutations = permutations(numbers);
2266
2267            // Check each list of integers for proper scoring
2268            for (List<Integer> currentNumbers : myPermutations) {
2269                // Test the total score
2270                score = game.calculateScore(currentNumbers, false);
2271                assertEquals(200, score);
2272
2273                // Test the held score
2274                score = game.calculateScore(currentNumbers, true);
2275                assertEquals(0, score);
2276
2277                // Test the calculateHighestScore method
2278                highestScore = game.calculateHighestScore(currentNumbers);
2279                assertEquals(200, highestScore[0]);
2280                highestRoll = (List<Integer>) highestScore[1];
```

```
2281                assertEquals(highestRoll.size(), 3);
2282                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2283                assertTrue(Collections.frequency(highestRoll, 5) == 2);
2284            }
2285        }
2286
2287        /*
2288         * Test 65: Check scoring for a die roll of 155AA (A represents all die
2289         * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
2290         */
2291
2292        // Generate all permutations of the roll
2293        automatedPermutations = integerPermutations("155AA");
2294
2295        // Convert the list of Strings to a list of lists of integers
2296        for (String numbers : automatedPermutations) {
2297            myPermutations = permutations(numbers);
2298
2299            // Check each list of integers for proper scoring
2300            for (List<Integer> currentNumbers : myPermutations) {
2301                // Test the total score
2302                score = game.calculateScore(currentNumbers, false);
2303                assertEquals(200, score);
2304
2305                // Test the held score
2306                score = game.calculateScore(currentNumbers, true);
2307                assertEquals(0, score);
2308
2309                // Test the calculateHighestScore method
2310                highestScore = game.calculateHighestScore(currentNumbers);
2311                assertEquals(200, highestScore[0]);
2312                highestRoll = (List<Integer>) highestScore[1];
2313                assertEquals(highestRoll.size(), 3);
2314                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2315                assertTrue(Collections.frequency(highestRoll, 5) == 2);
2316            }
2317        }
2318
2319        /*
2320         * Test 66: Check scoring for a die roll of 115AX (A and X each
2321         * represent all die values other than 1 or 5) Requirement 6.1.0 and
2322         * 6.2.0
2323         */
2324
2325        // Generate all permutations of the roll
2326        automatedPermutations = integerPermutations("115AX");
2327
2328        // Convert the list of Strings to a list of lists of integers
2329        for (String numbers : automatedPermutations) {
2330            myPermutations = permutations(numbers);
2331
2332            // Check each list of integers for proper scoring
2333            for (List<Integer> currentNumbers : myPermutations) {
2334                // Test the total score
2335                score = game.calculateScore(currentNumbers, false);
2336                assertEquals(250, score);
2337
```

```
2338                // Test the held score
2339                score = game.calculateScore(currentNumbers, true);
2340                assertEquals(0, score);
2341
2342                // Test the calculateHighestScore method
2343                highestScore = game.calculateHighestScore(currentNumbers);
2344                assertEquals(250, highestScore[0]);
2345                highestRoll = (List<Integer>) highestScore[1];
2346                assertEquals(highestRoll.size(), 3);
2347                assertTrue(Collections.frequency(highestRoll, 1) == 2);
2348                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2349            }
2350        }
2351
2352        /*
2353         * Test 67: Check scoring for a die roll of 115AA (A represents all die
2354         * values other than 1 or 5) Requirement 6.1.0 and 6.2.0
2355         */
2356
2357        // Generate all permutations of the roll
2358        automatedPermutations = integerPermutations("115AA");
2359
2360        // Convert the list of Strings to a list of lists of integers
2361        for (String numbers : automatedPermutations) {
2362            myPermutations = permutations(numbers);
2363
2364            // Check each list of integers for proper scoring
2365            for (List<Integer> currentNumbers : myPermutations) {
2366                // Test the total score
2367                score = game.calculateScore(currentNumbers, false);
2368                assertEquals(250, score);
2369
2370                // Test the held score
2371                score = game.calculateScore(currentNumbers, true);
2372                assertEquals(0, score);
2373
2374                // Test the calculateHighestScore method
2375                highestScore = game.calculateHighestScore(currentNumbers);
2376                assertEquals(250, highestScore[0]);
2377                highestRoll = (List<Integer>) highestScore[1];
2378                assertEquals(highestRoll.size(), 3);
2379                assertTrue(Collections.frequency(highestRoll, 1) == 2);
2380                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2381            }
2382        }
2383
2384        /*
2385         * Test 68: Check scoring for a die roll of AAAXY (A, X and Y each
2386         * represent all die values other than 1 or 5) Requirement 6.4.0
2387         */
2388
2389        // Generate all permutations of the roll
2390        automatedPermutations = integerPermutations("AAAXY");
2391
2392        // Convert the list of Strings to a list of lists of integers
2393        for (String numbers : automatedPermutations) {
2394            myPermutations = permutations(numbers);
```

```
2395
2396            // Get the die value for calculating the score
2397            dieOneValue = Character.getNumericValue(numbers.charAt(0));
2398
2399            // Check each list of integers for proper scoring
2400            for (List<Integer> currentNumbers : myPermutations) {
2401                // Test the total score
2402                score = game.calculateScore(currentNumbers, false);
2403                assertEquals(100 * dieOneValue, score);
2404
2405                // Test the held score
2406                score = game.calculateScore(currentNumbers, true);
2407                assertEquals(0, score);
2408
2409                // Test the calculateHighestScore method
2410                highestScore = game.calculateHighestScore(currentNumbers);
2411                assertEquals(100 * dieOneValue, highestScore[0]);
2412                highestRoll = (List<Integer>) highestScore[1];
2413                assertEquals(highestRoll.size(), 3);
2414                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2415            }
2416        }
2417
2418        /*
2419         * Test 69: Check scoring for a die roll of AAAXX (A and X each
2420         * represent all die values other than 1 or 5) Requirement 6.4.0
2421         */
2422
2423        // Generate all permutations of the roll
2424        automatedPermutations = integerPermutations("AAAXX");
2425
2426        // Convert the list of Strings to a list of lists of integers
2427        for (String numbers : automatedPermutations) {
2428            myPermutations = permutations(numbers);
2429
2430            // Get the die value for calculating the score
2431            dieOneValue = Character.getNumericValue(numbers.charAt(0));
2432
2433            // Check each list of integers for proper scoring
2434            for (List<Integer> currentNumbers : myPermutations) {
2435                // Test the total score
2436                score = game.calculateScore(currentNumbers, false);
2437                assertEquals(100 * dieOneValue, score);
2438
2439                // Test the held score
2440                score = game.calculateScore(currentNumbers, true);
2441                assertEquals(0, score);
2442
2443                // Test the calculateHighestScore method
2444                highestScore = game.calculateHighestScore(currentNumbers);
2445                assertEquals(100 * dieOneValue, highestScore[0]);
2446                highestRoll = (List<Integer>) highestScore[1];
2447                assertEquals(highestRoll.size(), 3);
2448                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2449            }
2450        }
2451
```

```
2452        /*
2453         * Test 70: Check scoring for a die roll of 5AAAX (A and X each
2454         * represent all die values other than 1 or 5) Requirement 6.2.0 and
2455         * 6.4.0
2456         */
2457
2458        // Generate all permutations of the roll
2459        automatedPermutations = integerPermutations("5AAAX");
2460
2461        // Convert the list of Strings to a list of lists of integers
2462        for (String numbers : automatedPermutations) {
2463            myPermutations = permutations(numbers);
2464
2465            // Get the die value for calculating the score
2466            dieOneValue = Character.getNumericValue(numbers.charAt(1));
2467
2468            // Check each list of integers for proper scoring
2469            for (List<Integer> currentNumbers : myPermutations) {
2470                // Test the total score
2471                score = game.calculateScore(currentNumbers, false);
2472                assertEquals(50 + 100 * dieOneValue, score);
2473
2474                // Test the held score
2475                score = game.calculateScore(currentNumbers, true);
2476                assertEquals(0, score);
2477
2478                // Test the calculateHighestScore method
2479                highestScore = game.calculateHighestScore(currentNumbers);
2480                assertEquals(100 * dieOneValue + 50, highestScore[0]);
2481                highestRoll = (List<Integer>) highestScore[1];
2482                assertEquals(highestRoll.size(), 4);
2483                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2484                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2485            }
2486        }
2487
2488        /*
2489         * Test 71: Check scoring for a die roll of 1AAAX (A and X each
2490         * represent all die values other than 1 or 5) Requirement 6.1.0 and
2491         * 6.4.0
2492         */
2493
2494        // Generate all permutations of the roll
2495        automatedPermutations = integerPermutations("1AAAX");
2496
2497        // Convert the list of Strings to a list of lists of integers
2498        for (String numbers : automatedPermutations) {
2499            myPermutations = permutations(numbers);
2500
2501            // Get the die value for calculating the score
2502            dieOneValue = Character.getNumericValue(numbers.charAt(1));
2503
2504            // Check each list of integers for proper scoring
2505            for (List<Integer> currentNumbers : myPermutations) {
2506                // Test the total score
2507                score = game.calculateScore(currentNumbers, false);
2508                assertEquals(100 + 100 * dieOneValue, score);
```

```
2509
2510                // Test the held score
2511                score = game.calculateScore(currentNumbers, true);
2512                assertEquals(0, score);
2513
2514                // Test the calculateHighestScore method
2515                highestScore = game.calculateHighestScore(currentNumbers);
2516                assertEquals(100 * dieOneValue + 100, highestScore[0]);
2517                highestRoll = (List<Integer>) highestScore[1];
2518                assertEquals(highestRoll.size(), 4);
2519                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2520                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2521            }
2522        }
2523
2524        /*
2525         * Test 72: Check scoring for a die roll of 55AAA (A represents all die
2526         * values other than 1 or 5) Requirement 6.2.0 and 6.4.0
2527         */
2528
2529        // Generate all permutations of the roll
2530        automatedPermutations = integerPermutations("55AAA");
2531
2532        // Convert the list of Strings to a list of lists of integers
2533        for (String numbers : automatedPermutations) {
2534            myPermutations = permutations(numbers);
2535
2536            // Get the die value for calculating the score
2537            dieOneValue = Character.getNumericValue(numbers.charAt(2));
2538
2539            // Check each list of integers for proper scoring
2540            for (List<Integer> currentNumbers : myPermutations) {
2541                // Test the total score
2542                score = game.calculateScore(currentNumbers, false);
2543                assertEquals(100 + 100 * dieOneValue, score);
2544
2545                // Test the held score
2546                score = game.calculateScore(currentNumbers, true);
2547                assertEquals(100 + 100 * dieOneValue, score);
2548
2549                // Test the calculateHighestScore method
2550                highestScore = game.calculateHighestScore(currentNumbers);
2551                assertEquals(100 * dieOneValue + 100, highestScore[0]);
2552                highestRoll = (List<Integer>) highestScore[1];
2553                assertEquals(highestRoll.size(), 5);
2554                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2555                assertTrue(Collections.frequency(highestRoll, 5) == 2);
2556            }
2557        }
2558
2559        /*
2560         * Test 73: Check scoring for a die roll of 15AAA (A represents all die
2561         * values other than 1 or 5) Requirement 6.1.0, 6.2.0, and 6.4.0
2562         */
2563
2564        // Generate all permutations of the roll
2565        automatedPermutations = integerPermutations("15AAA");
```

```
2566
2567        // Convert the list of Strings to a list of lists of integers
2568        for (String numbers : automatedPermutations) {
2569            myPermutations = permutations(numbers);
2570
2571            // Get the die value for calculating the score
2572            dieOneValue = Character.getNumericValue(numbers.charAt(2));
2573
2574            // Check each list of integers for proper scoring
2575            for (List<Integer> currentNumbers : myPermutations) {
2576                // Test the total score
2577                score = game.calculateScore(currentNumbers, false);
2578                assertEquals(150 + 100 * dieOneValue, score);
2579
2580                // Test the held score
2581                score = game.calculateScore(currentNumbers, true);
2582                assertEquals(150 + 100 * dieOneValue, score);
2583
2584                // Test the calculateHighestScore method
2585                highestScore = game.calculateHighestScore(currentNumbers);
2586                assertEquals(100 * dieOneValue + 150, highestScore[0]);
2587                highestRoll = (List<Integer>) highestScore[1];
2588                assertEquals(highestRoll.size(), 5);
2589                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2590                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2591                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2592            }
2593        }
2594
2595        /*
2596         * Test 74: Check scoring for a die roll of 11AAA (A represents all die
2597         * values other than 1 or 5) Requirement 6.1.0 and 6.4.0
2598         */
2599
2600        // Generate all permutations of the roll
2601        automatedPermutations = integerPermutations("11AAA");
2602
2603        // Convert the list of Strings to a list of lists of integers
2604        for (String numbers : automatedPermutations) {
2605            myPermutations = permutations(numbers);
2606
2607            // Get the die value for calculating the score
2608            dieOneValue = Character.getNumericValue(numbers.charAt(2));
2609
2610            // Check each list of integers for proper scoring
2611            for (List<Integer> currentNumbers : myPermutations) {
2612                // Test the total score
2613                score = game.calculateScore(currentNumbers, false);
2614                assertEquals(200 + 100 * dieOneValue, score);
2615
2616                // Test the held score
2617                score = game.calculateScore(currentNumbers, true);
2618                assertEquals(200 + 100 * dieOneValue, score);
2619
2620                // Test the calculateHighestScore method
2621                highestScore = game.calculateHighestScore(currentNumbers);
2622                assertEquals(100 * dieOneValue + 200, highestScore[0]);
```

```
2623                highestRoll = (List<Integer>) highestScore[1];
2624                assertEquals(highestRoll.size(), 5);
2625                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
2626                assertTrue(Collections.frequency(highestRoll, 1) == 2);
2627            }
2628        }
2629
2630        /*
2631         * Test 75: Check scoring for a die roll of 555AX (A and X each
2632         * represent all die values other than 1 or 5) Requirement 6.4.0
2633         */
2634
2635        // Generate all permutations of the roll
2636        automatedPermutations = integerPermutations("555AX");
2637
2638        // Convert the list of Strings to a list of lists of integers
2639        for (String numbers : automatedPermutations) {
2640            myPermutations = permutations(numbers);
2641
2642            // Check each list of integers for proper scoring
2643            for (List<Integer> currentNumbers : myPermutations) {
2644                // Test the total score
2645                score = game.calculateScore(currentNumbers, false);
2646                assertEquals(500, score);
2647
2648                // Test the held score
2649                score = game.calculateScore(currentNumbers, true);
2650                assertEquals(0, score);
2651
2652                // Test the calculateHighestScore method
2653                highestScore = game.calculateHighestScore(currentNumbers);
2654                assertEquals(500, highestScore[0]);
2655                highestRoll = (List<Integer>) highestScore[1];
2656                assertEquals(highestRoll.size(), 3);
2657                assertTrue(Collections.frequency(highestRoll, 5) == 3);
2658            }
2659        }
2660
2661        /*
2662         * Test 76: Check scoring for a die roll of 555AA (A represents all die
2663         * values other than 1 or 5) Requirement 6.4.0
2664         */
2665
2666        // Generate all permutations of the roll
2667        automatedPermutations = integerPermutations("555AA");
2668
2669        // Convert the list of Strings to a list of lists of integers
2670        for (String numbers : automatedPermutations) {
2671            myPermutations = permutations(numbers);
2672
2673            // Check each list of integers for proper scoring
2674            for (List<Integer> currentNumbers : myPermutations) {
2675                // Test the total score
2676                score = game.calculateScore(currentNumbers, false);
2677                assertEquals(500, score);
2678
2679                // Test the held score
```

```
2680                score = game.calculateScore(currentNumbers, true);
2681                assertEquals(0, score);
2682
2683                // Test the calculateHighestScore method
2684                highestScore = game.calculateHighestScore(currentNumbers);
2685                assertEquals(500, highestScore[0]);
2686                highestRoll = (List<Integer>) highestScore[1];
2687                assertEquals(highestRoll.size(), 3);
2688                assertTrue(Collections.frequency(highestRoll, 5) == 3);
2689            }
2690        }
2691
2692        /*
2693         * Test 77: Check scoring for a die roll of 1555A (A represents all die
2694         * values other than 1 or 5) Requirement 6.1.0 and 6.4.0
2695         */
2696
2697        // Generate all permutations of the roll
2698        automatedPermutations = integerPermutations("1555A");
2699
2700        // Convert the list of Strings to a list of lists of integers
2701        for (String numbers : automatedPermutations) {
2702            myPermutations = permutations(numbers);
2703
2704            // Check each list of integers for proper scoring
2705            for (List<Integer> currentNumbers : myPermutations) {
2706                // Test the total score
2707                score = game.calculateScore(currentNumbers, false);
2708                assertEquals(600, score);
2709
2710                // Test the held score
2711                score = game.calculateScore(currentNumbers, true);
2712                assertEquals(0, score);
2713
2714                // Test the calculateHighestScore method
2715                highestScore = game.calculateHighestScore(currentNumbers);
2716                assertEquals(600, highestScore[0]);
2717                highestRoll = (List<Integer>) highestScore[1];
2718                assertEquals(highestRoll.size(), 4);
2719                assertTrue(Collections.frequency(highestRoll, 5) == 3);
2720                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2721            }
2722        }
2723
2724        /*
2725         * Test 78: Check scoring for a die roll of 11555 Requirement 6.1.0 and
2726         * 6.4.0
2727         */
2728
2729        // Generate all permutations of the roll and store in a list of lists of
2730        // integers
2731        myPermutations = permutations("11555");
2732
2733        // Check each list of integers for correct scoring
2734        for (List<Integer> numbers : myPermutations) {
2735            // Test the total score
2736            score = game.calculateScore(numbers, false);
```

```
2737            assertEquals(700, score);
2738
2739            // Test the held score
2740            score = game.calculateScore(numbers, true);
2741            assertEquals(700, score);
2742
2743            // Test the calculateHighestScore method
2744            highestScore = game.calculateHighestScore(numbers);
2745            assertEquals(700, highestScore[0]);
2746            highestRoll = (List<Integer>) highestScore[1];
2747            assertEquals(highestRoll.size(), 5);
2748            assertTrue(Collections.frequency(highestRoll, 5) == 3);
2749            assertTrue(Collections.frequency(highestRoll, 1) == 2);
2750
2751        }
2752
2753        /*
2754         * Test 79: Check scoring for a die roll of 111AX (A and X each
2755         * represent all die values other than 1 or 5) Requirement 6.3.0
2756         */
2757
2758        // Generate all permutations of the roll
2759        automatedPermutations = integerPermutations("111AX");
2760
2761        // Convert the list of Strings to a list of lists of integers
2762        for (String numbers : automatedPermutations) {
2763            myPermutations = permutations(numbers);
2764
2765            // Check each list of integers for proper scoring
2766            for (List<Integer> currentNumbers : myPermutations) {
2767                // Test the total score
2768                score = game.calculateScore(currentNumbers, false);
2769                assertEquals(1000, score);
2770
2771                // Test the held score
2772                score = game.calculateScore(currentNumbers, true);
2773                assertEquals(0, score);
2774
2775                // Test the calculateHighestScore method
2776                highestScore = game.calculateHighestScore(currentNumbers);
2777                assertEquals(1000, highestScore[0]);
2778                highestRoll = (List<Integer>) highestScore[1];
2779                assertEquals(highestRoll.size(), 3);
2780                assertTrue(Collections.frequency(highestRoll, 1) == 3);
2781            }
2782        }
2783
2784        /*
2785         * Test 80: Check scoring for a die roll of 111AA (A represents all die
2786         * values other than 1 or 5) Requirement 6.3.0
2787         */
2788
2789        // Generate all permutations of the roll
2790        automatedPermutations = integerPermutations("111AA");
2791
2792        // Convert the list of Strings to a list of lists of integers
2793        for (String numbers : automatedPermutations) {
```

```
2794            myPermutations = permutations(numbers);
2795
2796            // Check each list of integers for proper scoring
2797            for (List<Integer> currentNumbers : myPermutations) {
2798                // Test the total score
2799                score = game.calculateScore(currentNumbers, false);
2800                assertEquals(1000, score);
2801
2802                // Test the held score
2803                score = game.calculateScore(currentNumbers, true);
2804                assertEquals(0, score);
2805
2806                // Test the calculateHighestScore method
2807                highestScore = game.calculateHighestScore(currentNumbers);
2808                assertEquals(1000, highestScore[0]);
2809                highestRoll = (List<Integer>) highestScore[1];
2810                assertEquals(highestRoll.size(), 3);
2811                assertTrue(Collections.frequency(highestRoll, 1) == 3);
2812            }
2813        }
2814
2815        /*
2816         * Test 81: Check scoring for a die roll of 1115A (A represents all die
2817         * values other than 1 or 5) Requirement 6.2.0 and 6.3.0
2818         */
2819
2820        // Generate all permutations of the roll
2821        automatedPermutations = integerPermutations("1115A");
2822
2823        // Convert the list of Strings to a list of lists of integers
2824        for (String numbers : automatedPermutations) {
2825            myPermutations = permutations(numbers);
2826
2827            // Check each list of integers for proper scoring
2828            for (List<Integer> currentNumbers : myPermutations) {
2829                // Test the total score
2830                score = game.calculateScore(currentNumbers, false);
2831                assertEquals(1050, score);
2832
2833                // Test the held score
2834                score = game.calculateScore(currentNumbers, true);
2835                assertEquals(0, score);
2836
2837                // Test the calculateHighestScore method
2838                highestScore = game.calculateHighestScore(currentNumbers);
2839                assertEquals(1050, highestScore[0]);
2840                highestRoll = (List<Integer>) highestScore[1];
2841                assertEquals(highestRoll.size(), 4);
2842                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2843                assertTrue(Collections.frequency(highestRoll, 1) == 3);
2844            }
2845        }
2846
2847        /*
2848         * Test 82: Check scoring for a die roll of 11155 Requirement 6.2.0 and
2849         * 6.3.0
2850         */
```

```
2851
2852        // Generate all permutations of the roll and store in a list of lists of
2853        // integers
2854        myPermutations = permutations("11155");
2855
2856        // Check each list of integers for correct scoring
2857        for (List<Integer> numbers : myPermutations) {
2858            // Test the total score
2859            score = game.calculateScore(numbers, false);
2860            assertEquals(1100, score);
2861
2862            // Test the held score
2863            score = game.calculateScore(numbers, true);
2864            assertEquals(1100, score);
2865
2866            // Test the calculateHighestScore method
2867            highestScore = game.calculateHighestScore(numbers);
2868            assertEquals(1100, highestScore[0]);
2869            highestRoll = (List<Integer>) highestScore[1];
2870            assertEquals(highestRoll.size(), 5);
2871            assertTrue(Collections.frequency(highestRoll, 5) == 2);
2872            assertTrue(Collections.frequency(highestRoll, 1) == 3);
2873        }
2874
2875        /*
2876         * Test 83: Check scoring for a die roll of AAAAX (A and X each
2877         * represent all die values other than 1 or 5) Requirement 6.5.0
2878         */
2879
2880        // Generate all permutations of the roll
2881        automatedPermutations = integerPermutations("AAAAX");
2882
2883        // Convert the list of Strings to a list of lists of integers
2884        for (String numbers : automatedPermutations) {
2885            myPermutations = permutations(numbers);
2886
2887            // Get the die value for calculating the score
2888            dieOneValue = Character.getNumericValue(numbers.charAt(0));
2889
2890            // Check each list of integers for proper scoring
2891            for (List<Integer> currentNumbers : myPermutations) {
2892                // Test the total score
2893                score = game.calculateScore(currentNumbers, false);
2894                assertEquals(200 * dieOneValue, score);
2895
2896                // Test the held score
2897                score = game.calculateScore(currentNumbers, true);
2898                assertEquals(0, score);
2899
2900                // Test the calculateHighestScore method
2901                highestScore = game.calculateHighestScore(currentNumbers);
2902                assertEquals(200 * dieOneValue, highestScore[0]);
2903                highestRoll = (List<Integer>) highestScore[1];
2904                assertEquals(highestRoll.size(), 4);
2905                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
2906            }
2907        }
```

```
2908
2909        /*
2910         * Test 84: Check scoring for a die roll of 5AAAA (A represents all die
2911         * values other than 1 or 5) Requirement 6.2.0 and 6.5.0
2912         */
2913
2914        // Generate all permutations of the roll
2915        automatedPermutations = integerPermutations("5AAAA");
2916
2917        // Convert the list of Strings to a list of lists of integers
2918        for (String numbers : automatedPermutations) {
2919            myPermutations = permutations(numbers);
2920
2921            // Get the die value for calculating the score
2922            dieOneValue = Character.getNumericValue(numbers.charAt(1));
2923
2924            // Check each list of integers for proper scoring
2925            for (List<Integer> currentNumbers : myPermutations) {
2926                // Test the total score
2927                score = game.calculateScore(currentNumbers, false);
2928                assertEquals(200 * dieOneValue + 50, score);
2929
2930                // Test the held score
2931                score = game.calculateScore(currentNumbers, true);
2932                assertEquals(200 * dieOneValue + 50, score);
2933
2934                // Test the calculateHighestScore method
2935                highestScore = game.calculateHighestScore(currentNumbers);
2936                assertEquals(200 * dieOneValue + 50, highestScore[0]);
2937                highestRoll = (List<Integer>) highestScore[1];
2938                assertEquals(highestRoll.size(), 5);
2939                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
2940                assertTrue(Collections.frequency(highestRoll, 5) == 1);
2941            }
2942        }
2943
2944        /*
2945         * Test 85: Check scoring for a die roll of 1AAAA (A represents all die
2946         * values other than 1 or 5) Requirement 6.1.0 and 6.5.0
2947         */
2948
2949        // Generate all permutations of the roll
2950        automatedPermutations = integerPermutations("1AAAA");
2951
2952        // Convert the list of Strings to a list of lists of integers
2953        for (String numbers : automatedPermutations) {
2954            myPermutations = permutations(numbers);
2955
2956            // Get the die value for calculating the score
2957            dieOneValue = Character.getNumericValue(numbers.charAt(1));
2958
2959            // Check each list of integers for proper scoring
2960            for (List<Integer> currentNumbers : myPermutations) {
2961                // Test the total score
2962                score = game.calculateScore(currentNumbers, false);
2963                assertEquals(200 * dieOneValue + 100, score);
2964
```

```
2965                // Test the held score
2966                score = game.calculateScore(currentNumbers, true);
2967                assertEquals(200 * dieOneValue + 100, score);
2968
2969                // Test the calculateHighestScore method
2970                highestScore = game.calculateHighestScore(currentNumbers);
2971                assertEquals(200 * dieOneValue + 100, highestScore[0]);
2972                highestRoll = (List<Integer>) highestScore[1];
2973                assertEquals(highestRoll.size(), 5);
2974                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
2975                assertTrue(Collections.frequency(highestRoll, 1) == 1);
2976            }
2977        }
2978
2979        /*
2980         * Test 86: Check scoring for a die roll of 5555A (A represents all die
2981         * values other than 1 or 5) Requirement 6.5.0
2982         */
2983
2984        // Generate all permutations of the roll
2985        automatedPermutations = integerPermutations("5555A");
2986
2987        // Convert the list of Strings to a list of lists of integers
2988        for (String numbers : automatedPermutations) {
2989            myPermutations = permutations(numbers);
2990
2991            // Check each list of integers for proper scoring
2992            for (List<Integer> currentNumbers : myPermutations) {
2993                // Test the total score
2994                score = game.calculateScore(currentNumbers, false);
2995                assertEquals(1000, score);
2996
2997                // Test the held score
2998                score = game.calculateScore(currentNumbers, true);
2999                assertEquals(0, score);
3000
3001                // Test the calculateHighestScore method
3002                highestScore = game.calculateHighestScore(currentNumbers);
3003                assertEquals(1000, highestScore[0]);
3004                highestRoll = (List<Integer>) highestScore[1];
3005                assertEquals(highestRoll.size(), 4);
3006                assertTrue(Collections.frequency(highestRoll, 5) == 4);
3007            }
3008        }
3009
3010        /*
3011         * Test 87: Check scoring for a die roll of 15555 Requirement 6.1.0 and
3012         * 6.5.0
3013         */
3014
3015        // Generate all permutations of the roll and store in a list of lists of
3016        // integers
3017        myPermutations = permutations("15555");
3018
3019        // Check each list of integers for correct scoring
3020        for (List<Integer> numbers : myPermutations) {
3021            // Test the total score
```

```
3022            score = game.calculateScore(numbers, false);
3023            assertEquals(1100, score);
3024
3025            // Test the held score
3026            score = game.calculateScore(numbers, true);
3027            assertEquals(1100, score);
3028
3029            // Test the calculateHighestScore method
3030            highestScore = game.calculateHighestScore(numbers);
3031            assertEquals(1100, highestScore[0]);
3032            highestRoll = (List<Integer>) highestScore[1];
3033            assertEquals(highestRoll.size(), 5);
3034            assertTrue(Collections.frequency(highestRoll, 5) == 4);
3035            assertTrue(Collections.frequency(highestRoll, 1) == 1);
3036        }
3037
3038        /*
3039         * Test 88: Check scoring for a die roll of 1111A (A represents all die
3040         * values other than 1 or 5) Requirement 6.5.0
3041         */
3042
3043        // Generate all permutations of the roll
3044        automatedPermutations = integerPermutations("1111A");
3045
3046        // Convert the list of Strings to a list of lists of integers
3047        for (String numbers : automatedPermutations) {
3048            myPermutations = permutations(numbers);
3049
3050            // Check each list of integers for proper scoring
3051            for (List<Integer> currentNumbers : myPermutations) {
3052                // Test the total score
3053                score = game.calculateScore(currentNumbers, false);
3054                assertEquals(2000, score);
3055
3056                // Test the held score
3057                score = game.calculateScore(currentNumbers, true);
3058                assertEquals(0, score);
3059
3060                // Test the calculateHighestScore method
3061                highestScore = game.calculateHighestScore(currentNumbers);
3062                assertEquals(2000, highestScore[0]);
3063                highestRoll = (List<Integer>) highestScore[1];
3064                assertEquals(highestRoll.size(), 4);
3065                assertTrue(Collections.frequency(highestRoll, 1) == 4);
3066            }
3067        }
3068
3069        /*
3070         * Test 89: Check scoring for a die roll of 11115 Requirement 6.2.0 and
3071         * 6.5.0
3072         */
3073
3074        // Generate all permutations of the roll and store in a list of lists of
3075        // integers
3076        myPermutations = permutations("11115");
3077
3078        // Check each list of integers for correct scoring
```

```
3079        for (List<Integer> numbers : myPermutations) {
3080            // Test the total score
3081            score = game.calculateScore(numbers, false);
3082            assertEquals(2050, score);
3083
3084            // Test the held score
3085            score = game.calculateScore(numbers, true);
3086            assertEquals(2050, score);
3087
3088            // Test the calculateHighestScore method
3089            highestScore = game.calculateHighestScore(numbers);
3090            assertEquals(2050, highestScore[0]);
3091            highestRoll = (List<Integer>) highestScore[1];
3092            assertEquals(highestRoll.size(), 5);
3093            assertTrue(Collections.frequency(highestRoll, 5) == 1);
3094            assertTrue(Collections.frequency(highestRoll, 1) == 4);
3095        }
3096
3097        /*
3098         * Test 90: Check scoring for a die roll of AAAAA (A represents all die
3099         * values other than 1 or 5) Requirement 6.5.0
3100         */
3101
3102        // Generate all permutations of the roll
3103        automatedPermutations = integerPermutations("AAAAA");
3104
3105        // Convert the list of Strings to a list of lists of integers
3106        for (String numbers : automatedPermutations) {
3107            myPermutations = permutations(numbers);
3108
3109            // Get the die value for calculating the score
3110            dieOneValue = Character.getNumericValue(numbers.charAt(0));
3111
3112            // Check each list of integers for proper scoring
3113            for (List<Integer> currentNumbers : myPermutations) {
3114                // Test the total score
3115                score = game.calculateScore(currentNumbers, false);
3116                assertEquals(400 * dieOneValue, score);
3117
3118                // Test the held score
3119                score = game.calculateScore(currentNumbers, true);
3120                assertEquals(400 * dieOneValue, score);
3121
3122                // Test the calculateHighestScore method
3123                highestScore = game.calculateHighestScore(currentNumbers);
3124                assertEquals(400 * dieOneValue, highestScore[0]);
3125                highestRoll = (List<Integer>) highestScore[1];
3126                assertEquals(highestRoll.size(), 5);
3127                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 5);
3128            }
3129        }
3130
3131        /*
3132         * Test 91: Check scoring for a die roll of 55555 Requirement 6.5.0
3133         */
3134
3135        // Generate all permutations of the roll and store in a list of lists of
```

```
3136        // integers
3137        myPermutations = permutations("55555");
3138
3139        // Check each list of integers for correct scoring
3140        for (List<Integer> numbers : myPermutations) {
3141            // Test the total score
3142            score = game.calculateScore(numbers, false);
3143            assertEquals(2000, score);
3144
3145            // Test the held score
3146            score = game.calculateScore(numbers, true);
3147            assertEquals(2000, score);
3148
3149            // Test the calculateHighestScore method
3150            highestScore = game.calculateHighestScore(numbers);
3151            assertEquals(2000, highestScore[0]);
3152            highestRoll = (List<Integer>) highestScore[1];
3153            assertEquals(highestRoll.size(), 5);
3154            assertTrue(Collections.frequency(highestRoll, 5) == 5);
3155        }
3156
3157        /*
3158         * Test 92: Check scoring for a die roll of 11111 Requirement 6.5.0
3159         */
3160
3161        // Generate all permutations of the roll and store in a list of lists of
3162        // integers
3163        myPermutations = permutations("11111");
3164
3165        // Check each list of integers for correct scoring
3166        for (List<Integer> numbers : myPermutations) {
3167            // Test the total score
3168            score = game.calculateScore(numbers, false);
3169            assertEquals(4000, score);
3170
3171            // Test the held score
3172            score = game.calculateScore(numbers, true);
3173            assertEquals(4000, score);
3174
3175            // Test the calculateHighestScore method
3176            highestScore = game.calculateHighestScore(numbers);
3177            assertEquals(4000, highestScore[0]);
3178            highestRoll = (List<Integer>) highestScore[1];
3179            assertEquals(highestRoll.size(), 5);
3180            assertTrue(Collections.frequency(highestRoll, 1) == 5);
3181        }
3182    }
3183
3184    /**
3185     * This method tests the calculateScore method of the Game class for all
3186     * permutations of 6 dice, requirements 6.1.0, 6.2.0, 6.3.0, 6.4.0, 6.5.0,
3187     * 6.6.0, and 6.7.0:<br />
3188     * <br />
3189     * 6.1.0 Each 1 rolled is worth 100 points<br />
3190     * 6.2.0 Each 5 rolled is worth 50 points<br />
3191     * 6.3.0 Three 1â€™s are worth 1000 points<br />
3192     * 6.4.0 Three of a kind of any value other than 1 is worth 100 times the
```

```
3193     * value of the die (e.g. three 4â€™s is worth 400 points).<br />
3194     * 6.5.0 Four, five, or six of a kind is scored by doubling the three of a
3195     * kind value for every additional matching die (e.g. five 3â€™s would be
3196     * scored as 300 X 2 X 2 = 1200.<br />
3197     * 6.6.0 Three doubles (e.g. 1-1-2-2-3-3) is worth 750 points.<br />
3198     * 6.7.0 A straight (e.g. 1-2-3-4-5-6), which can only be achieved when all
3199     * 6 dice are rolled, is worth 1500 points.
3200     */
3201    @SuppressWarnings("unchecked")
3202    @Test
3203    public void testCalculateScore6() {
3204
3205        /*
3206         * Test 93: Check scoring for a die roll of AAXXYZ (A, X, Y and Z each
3207         * represent all die values other than 1 or 5)
3208         */
3209
3210        // Generate all permutations of the roll
3211        automatedPermutations = integerPermutations("AAXXYZ");
3212
3213        // Convert the list of Strings to a list of lists of integers
3214        for (String numbers : automatedPermutations) {
3215            myPermutations = permutations(numbers);
3216
3217            // Check each list of integers for proper scoring
3218            for (List<Integer> currentNumbers : myPermutations) {
3219                // Test the total score
3220                score = game.calculateScore(currentNumbers, false);
3221                assertEquals(0, score);
3222
3223                // Test the held score
3224                score = game.calculateScore(currentNumbers, true);
3225                assertEquals(0, score);
3226
3227                // Test the calculateHighestScore method
3228                highestScore = game.calculateHighestScore(currentNumbers);
3229                assertEquals(0, highestScore[0]);
3230                highestRoll = (List<Integer>) highestScore[1];
3231                assertTrue(highestRoll.isEmpty());
3232            }
3233        }
3234
3235        /*
3236         * Test 94: Check scoring for a die roll of AAXXYY (A, X and Y each
3237         * represent all die values other than 1 or 5) Requirement 6.6.0
3238         */
3239
3240        // Generate all permutations of the roll
3241        automatedPermutations = integerPermutations("AAXXYY");
3242
3243        // Convert the list of Strings to a list of lists of integers
3244        for (String numbers : automatedPermutations) {
3245            myPermutations = permutations(numbers);
3246
3247            // Get the die value for calculating the score
3248            dieOneValue = Character.getNumericValue(numbers.charAt(0));
3249            dieTwoValue = Character.getNumericValue(numbers.charAt(2));
```

```
3250            int dieThreeValue = Character.getNumericValue(numbers.charAt(4));
3251
3252            // Check each list of integers for proper scoring
3253            for (List<Integer> currentNumbers : myPermutations) {
3254                // Test the total score
3255                score = game.calculateScore(currentNumbers, false);
3256                assertEquals(750, score);
3257
3258                // Test the held score
3259                score = game.calculateScore(currentNumbers, true);
3260                assertEquals(750, score);
3261
3262                // Test the calculateHighestScore method
3263                highestScore = game.calculateHighestScore(currentNumbers);
3264                assertEquals(750, highestScore[0]);
3265                highestRoll = (List<Integer>) highestScore[1];
3266                assertEquals(highestRoll.size(), 6);
3267                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 2);
3268                assertTrue(Collections.frequency(highestRoll, dieTwoValue) == 2);
3269                assertTrue(Collections.frequency(highestRoll, dieThreeValue) == 2);
3270            }
3271        }
3272
3273        /*
3274         * Test 95: Check scoring for a die roll of 11AAXX (A and X each
3275         * represent all die values other than 1 or 5) Requirement 6.6.0
3276         */
3277
3278        // Generate all permutations of the roll
3279        automatedPermutations = integerPermutations("11AAXX");
3280
3281        // Convert the list of Strings to a list of lists of integers
3282        for (String numbers : automatedPermutations) {
3283            myPermutations = permutations(numbers);
3284
3285            dieOneValue = Character.getNumericValue(numbers.charAt(2));
3286            dieTwoValue = Character.getNumericValue(numbers.charAt(4));
3287
3288            // Check each list of integers for proper scoring
3289            for (List<Integer> currentNumbers : myPermutations) {
3290                // Test the total score
3291                score = game.calculateScore(currentNumbers, false);
3292                assertEquals(750, score);
3293
3294                // Test the held score
3295                score = game.calculateScore(currentNumbers, true);
3296                assertEquals(750, score);
3297
3298                // Test the calculateHighestScore method
3299                highestScore = game.calculateHighestScore(currentNumbers);
3300                assertEquals(750, highestScore[0]);
3301                highestRoll = (List<Integer>) highestScore[1];
3302                assertEquals(highestRoll.size(), 6);
3303                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 2);
3304                assertTrue(Collections.frequency(highestRoll, dieTwoValue) == 2);
3305                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3306            }
```

```
3307        }
3308
3309        /*
3310         * Test 96: Check scoring for a die roll of 55AAXX (A and X each
3311         * represent all die values other than 1 or 5) Requirement 6.6.0
3312         */
3313
3314        // Generate all permutations of the roll
3315        automatedPermutations = integerPermutations("55AAXX");
3316
3317        // Convert the list of Strings to a list of lists of integers
3318        for (String numbers : automatedPermutations) {
3319            myPermutations = permutations(numbers);
3320
3321            dieOneValue = Character.getNumericValue(numbers.charAt(2));
3322            dieTwoValue = Character.getNumericValue(numbers.charAt(4));
3323
3324            // Check each list of integers for proper scoring
3325            for (List<Integer> currentNumbers : myPermutations) {
3326                // Test the total score
3327                score = game.calculateScore(currentNumbers, false);
3328                assertEquals(750, score);
3329
3330                // Test the held score
3331                score = game.calculateScore(currentNumbers, true);
3332                assertEquals(750, score);
3333
3334                // Test the calculateHighestScore method
3335                highestScore = game.calculateHighestScore(currentNumbers);
3336                assertEquals(750, highestScore[0]);
3337                highestRoll = (List<Integer>) highestScore[1];
3338                assertEquals(highestRoll.size(), 6);
3339                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 2);
3340                assertTrue(Collections.frequency(highestRoll, dieTwoValue) == 2);
3341                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3342            }
3343        }
3344
3345        /*
3346         * Test 97: Check scoring for a die roll of 1155AA (A represents all die
3347         * values other than 1 or 5) Requirement 6.6.0
3348         */
3349
3350        // Generate all permutations of the roll
3351        automatedPermutations = integerPermutations("1155AA");
3352
3353        // Convert the list of Strings to a list of lists of integers
3354        for (String numbers : automatedPermutations) {
3355            myPermutations = permutations(numbers);
3356
3357            dieOneValue = Character.getNumericValue(numbers.charAt(4));
3358
3359            // Check each list of integers for proper scoring
3360            for (List<Integer> currentNumbers : myPermutations) {
3361                // Test the total score
3362                score = game.calculateScore(currentNumbers, false);
3363                assertEquals(750, score);
```

```
3364
3365                // Test the held score
3366                score = game.calculateScore(currentNumbers, true);
3367                assertEquals(750, score);
3368
3369                // Test the calculateHighestScore method
3370                highestScore = game.calculateHighestScore(currentNumbers);
3371                assertEquals(750, highestScore[0]);
3372                highestRoll = (List<Integer>) highestScore[1];
3373                assertEquals(highestRoll.size(), 6);
3374                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 2);
3375                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3376                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3377            }
3378        }
3379
3380        /*
3381         * Test 98: Check scoring for a die roll of 1AXY5Z (A, X, Y and Z each
3382         * represent all die values other than 1 or 5) Requirement 6.7.0
3383         */
3384
3385        // Generate all permutations of the roll
3386        automatedPermutations = integerPermutations("1AXY5Z");
3387
3388        // Convert the list of Strings to a list of lists of integers
3389        for (String numbers : automatedPermutations) {
3390            myPermutations = permutations(numbers);
3391
3392            // Check each list of integers for proper scoring
3393            for (List<Integer> currentNumbers : myPermutations) {
3394                // Test the total score
3395                score = game.calculateScore(currentNumbers, false);
3396                assertEquals(1500, score);
3397
3398                // Test the held score
3399                score = game.calculateScore(currentNumbers, true);
3400                assertEquals(1500, score);
3401
3402                // Test the calculateHighestScore method
3403                highestScore = game.calculateHighestScore(currentNumbers);
3404                assertEquals(1500, highestScore[0]);
3405                highestRoll = (List<Integer>) highestScore[1];
3406                assertEquals(highestRoll.size(), 6);
3407                assertTrue(highestRoll.contains(1));
3408                assertTrue(highestRoll.contains(2));
3409                assertTrue(highestRoll.contains(3));
3410                assertTrue(highestRoll.contains(4));
3411                assertTrue(highestRoll.contains(5));
3412                assertTrue(highestRoll.contains(6));
3413            }
3414        }
3415
3416        /*
3417         * Test 99: Check scoring for a die roll of 5AAXXY (A, X and Y each
3418         * represent all die values other than 1 or 5) Requirement 6.2.0
3419         */
3420
```

```
3421        // Generate all permutations of the roll
3422        automatedPermutations = integerPermutations("5AAXXY");
3423
3424        // Convert the list of Strings to a list of lists of integers
3425        for (String numbers : automatedPermutations) {
3426            myPermutations = permutations(numbers);
3427
3428            // Check each list of integers for proper scoring
3429            for (List<Integer> currentNumbers : myPermutations) {
3430                // Test the total score
3431                score = game.calculateScore(currentNumbers, false);
3432                assertEquals(50, score);
3433
3434                // Test the held score
3435                score = game.calculateScore(currentNumbers, true);
3436                assertEquals(0, score);
3437
3438                // Test the calculateHighestScore method
3439                highestScore = game.calculateHighestScore(currentNumbers);
3440                assertEquals(50, highestScore[0]);
3441                highestRoll = (List<Integer>) highestScore[1];
3442                assertEquals(highestRoll.size(), 1);
3443                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3444            }
3445        }
3446
3447        /*
3448         * Test 100: Check scoring for a die roll of 5AAXYZ (A, X, Y and Z each
3449         * represent all die values other than 1 or 5) Requirement 6.2.0
3450         */
3451
3452        // Generate all permutations of the roll
3453        automatedPermutations = integerPermutations("5AAXYZ");
3454
3455        // Convert the list of Strings to a list of lists of integers
3456        for (String numbers : automatedPermutations) {
3457            myPermutations = permutations(numbers);
3458
3459            // Check each list of integers for proper scoring
3460            for (List<Integer> currentNumbers : myPermutations) {
3461                // Test the total score
3462                score = game.calculateScore(currentNumbers, false);
3463                assertEquals(50, score);
3464
3465                // Test the held score
3466                score = game.calculateScore(currentNumbers, true);
3467                assertEquals(0, score);
3468
3469                // Test the calculateHighestScore method
3470                highestScore = game.calculateHighestScore(currentNumbers);
3471                assertEquals(50, highestScore[0]);
3472                highestRoll = (List<Integer>) highestScore[1];
3473                assertEquals(highestRoll.size(), 1);
3474                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3475            }
3476        }
3477
```

```
3478        /*
3479         * Test 101: Check scoring for a die roll of 1AAXXY (A, X and Y each
3480         * represent all die values other than 1 or 5) Requirement 6.1.0
3481         */
3482
3483        // Generate all permutations of the roll
3484        automatedPermutations = integerPermutations("1AAXXY");
3485
3486        // Convert the list of Strings to a list of lists of integers
3487        for (String numbers : automatedPermutations) {
3488            myPermutations = permutations(numbers);
3489
3490            // Check each list of integers for proper scoring
3491            for (List<Integer> currentNumbers : myPermutations) {
3492                // Test the total score
3493                score = game.calculateScore(currentNumbers, false);
3494                assertEquals(100, score);
3495
3496                // Test the held score
3497                score = game.calculateScore(currentNumbers, true);
3498                assertEquals(0, score);
3499
3500                // Test the calculateHighestScore method
3501                highestScore = game.calculateHighestScore(currentNumbers);
3502                assertEquals(100, highestScore[0]);
3503                highestRoll = (List<Integer>) highestScore[1];
3504                assertEquals(highestRoll.size(), 1);
3505                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3506            }
3507        }
3508
3509        /*
3510         * Test 102: Check scoring for a die roll of 1AAXYZ (A, X, Y and Z each
3511         * represent all die values other than 1 or 5) Requirement 6.1.0
3512         */
3513
3514        // Generate all permutations of the roll
3515        automatedPermutations = integerPermutations("1AAXYZ");
3516
3517        // Convert the list of Strings to a list of lists of integers
3518        for (String numbers : automatedPermutations) {
3519            myPermutations = permutations(numbers);
3520
3521            // Check each list of integers for proper scoring
3522            for (List<Integer> currentNumbers : myPermutations) {
3523                // Test the total score
3524                score = game.calculateScore(currentNumbers, false);
3525                assertEquals(100, score);
3526
3527                // Test the held score
3528                score = game.calculateScore(currentNumbers, true);
3529                assertEquals(0, score);
3530
3531                // Test the calculateHighestScore method
3532                highestScore = game.calculateHighestScore(currentNumbers);
3533                assertEquals(100, highestScore[0]);
3534                highestRoll = (List<Integer>) highestScore[1];
```

```
3535                assertEquals(highestRoll.size(), 1);
3536                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3537            }
3538        }
3539
3540        /*
3541         * Test 103: Check scoring for a die roll of 55AAXY (A, X and Y each
3542         * represent all die values other than 1 or 5) Requirement 6.2.0
3543         */
3544
3545        // Generate all permutations of the roll
3546        automatedPermutations = integerPermutations("55AAXY");
3547
3548        // Convert the list of Strings to a list of lists of integers
3549        for (String numbers : automatedPermutations) {
3550            myPermutations = permutations(numbers);
3551
3552            // Check each list of integers for proper scoring
3553            for (List<Integer> currentNumbers : myPermutations) {
3554                // Test the total score
3555                score = game.calculateScore(currentNumbers, false);
3556                assertEquals(100, score);
3557
3558                // Test the held score
3559                score = game.calculateScore(currentNumbers, true);
3560                assertEquals(0, score);
3561
3562                // Test the calculateHighestScore method
3563                highestScore = game.calculateHighestScore(currentNumbers);
3564                assertEquals(100, highestScore[0]);
3565                highestRoll = (List<Integer>) highestScore[1];
3566                assertEquals(highestRoll.size(), 2);
3567                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3568            }
3569        }
3570
3571        /*
3572         * Test 104: Check scoring for a die roll of 55AXYZ (A, X, Y and Z each
3573         * represent all die values other than 1 or 5) Requirement 6.2.0
3574         */
3575
3576        // Generate all permutations of the roll
3577        automatedPermutations = integerPermutations("55AXYZ");
3578
3579        // Convert the list of Strings to a list of lists of integers
3580        for (String numbers : automatedPermutations) {
3581            myPermutations = permutations(numbers);
3582
3583            // Check each list of integers for proper scoring
3584            for (List<Integer> currentNumbers : myPermutations) {
3585                // Test the total score
3586                score = game.calculateScore(currentNumbers, false);
3587                assertEquals(100, score);
3588
3589                // Test the held score
3590                score = game.calculateScore(currentNumbers, true);
3591                assertEquals(0, score);
```

```
3592
3593                // Test the calculateHighestScore method
3594                highestScore = game.calculateHighestScore(currentNumbers);
3595                assertEquals(100, highestScore[0]);
3596                highestRoll = (List<Integer>) highestScore[1];
3597                assertEquals(highestRoll.size(), 2);
3598                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3599            }
3600        }
3601
3602        /*
3603         * Test 105: Check scoring for a die roll of 15AAXY (A, X and Y each
3604         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3605         * 6.2.0
3606         */
3607
3608        // Generate all permutations of the roll
3609        automatedPermutations = integerPermutations("15AAXY");
3610
3611        // Convert the list of Strings to a list of lists of integers
3612        for (String numbers : automatedPermutations) {
3613            myPermutations = permutations(numbers);
3614
3615            // Check each list of integers for proper scoring
3616            for (List<Integer> currentNumbers : myPermutations) {
3617                // Test the total score
3618                score = game.calculateScore(currentNumbers, false);
3619                assertEquals(150, score);
3620
3621                // Test the held score
3622                score = game.calculateScore(currentNumbers, true);
3623                assertEquals(0, score);
3624
3625                // Test the calculateHighestScore method
3626                highestScore = game.calculateHighestScore(currentNumbers);
3627                assertEquals(150, highestScore[0]);
3628                highestRoll = (List<Integer>) highestScore[1];
3629                assertEquals(highestRoll.size(), 2);
3630                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3631                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3632            }
3633        }
3634
3635        /*
3636         * Test 106: Check scoring for a die roll of 15AAXx (A and X each
3637         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3638         * 6.2.0
3639         */
3640
3641        // Generate all permutations of the roll
3642        automatedPermutations = integerPermutations("15AAXX");
3643
3644        // Convert the list of Strings to a list of lists of integers
3645        for (String numbers : automatedPermutations) {
3646            myPermutations = permutations(numbers);
3647
3648            // Check each list of integers for proper scoring
```

```
3649            for (List<Integer> currentNumbers : myPermutations) {
3650                // Test the total score
3651                score = game.calculateScore(currentNumbers, false);
3652                assertEquals(150, score);
3653
3654                // Test the held score
3655                score = game.calculateScore(currentNumbers, true);
3656                assertEquals(0, score);
3657
3658                // Test the calculateHighestScore method
3659                highestScore = game.calculateHighestScore(currentNumbers);
3660                assertEquals(150, highestScore[0]);
3661                highestRoll = (List<Integer>) highestScore[1];
3662                assertEquals(highestRoll.size(), 2);
3663                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3664                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3665            }
3666        }
3667
3668        /*
3669         * Test 107: Check scoring for a die roll of 11AAXY (A, X and Y each
3670         * represent all die values other than 1 or 5) Requirement 6.1.0
3671         */
3672
3673        // Generate all permutations of the roll
3674        automatedPermutations = integerPermutations("11AAXY");
3675
3676        // Convert the list of Strings to a list of lists of integers
3677        for (String numbers : automatedPermutations) {
3678            myPermutations = permutations(numbers);
3679
3680            // Check each list of integers for proper scoring
3681            for (List<Integer> currentNumbers : myPermutations) {
3682                // Test the total score
3683                score = game.calculateScore(currentNumbers, false);
3684                assertEquals(200, score);
3685
3686                // Test the held score
3687                score = game.calculateScore(currentNumbers, true);
3688                assertEquals(0, score);
3689
3690                // Test the calculateHighestScore method
3691                highestScore = game.calculateHighestScore(currentNumbers);
3692                assertEquals(200, highestScore[0]);
3693                highestRoll = (List<Integer>) highestScore[1];
3694                assertEquals(highestRoll.size(), 2);
3695                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3696            }
3697        }
3698
3699        /*
3700         * Test 108: Check scoring for a die roll of 11AXYZ (A, X, Y and Z each
3701         * represent all die values other than 1 or 5) Requirement 6.1.0
3702         */
3703
3704        // Generate all permutations of the roll
3705        automatedPermutations = integerPermutations("11AXYZ");
```

```
3706
3707        // Convert the list of Strings to a list of lists of integers
3708        for (String numbers : automatedPermutations) {
3709            myPermutations = permutations(numbers);
3710
3711            // Check each list of integers for proper scoring
3712            for (List<Integer> currentNumbers : myPermutations) {
3713                // Test the total score
3714                score = game.calculateScore(currentNumbers, false);
3715                assertEquals(200, score);
3716
3717                // Test the held score
3718                score = game.calculateScore(currentNumbers, true);
3719                assertEquals(0, score);
3720
3721                // Test the calculateHighestScore method
3722                highestScore = game.calculateHighestScore(currentNumbers);
3723                assertEquals(200, highestScore[0]);
3724                highestRoll = (List<Integer>) highestScore[1];
3725                assertEquals(highestRoll.size(), 2);
3726                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3727            }
3728        }
3729
3730        /*
3731         * Test 109: Check scoring for a die roll of 155AAX (A and X each
3732         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3733         * 6.2.0
3734         */
3735
3736        // Generate all permutations of the roll
3737        automatedPermutations = integerPermutations("155AAX");
3738
3739        // Convert the list of Strings to a list of lists of integers
3740        for (String numbers : automatedPermutations) {
3741            myPermutations = permutations(numbers);
3742
3743            // Check each list of integers for proper scoring
3744            for (List<Integer> currentNumbers : myPermutations) {
3745                // Test the total score
3746                score = game.calculateScore(currentNumbers, false);
3747                assertEquals(200, score);
3748
3749                // Test the held score
3750                score = game.calculateScore(currentNumbers, true);
3751                assertEquals(0, score);
3752
3753                // Test the calculateHighestScore method
3754                highestScore = game.calculateHighestScore(currentNumbers);
3755                assertEquals(200, highestScore[0]);
3756                highestRoll = (List<Integer>) highestScore[1];
3757                assertEquals(highestRoll.size(), 3);
3758                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3759                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3760            }
3761        }
3762
```

```
3763        /*
3764         * Test 110: Check scoring for a die roll of 155AXY (A, X and Y each
3765         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3766         * 6.2.0
3767         */
3768
3769        // Generate all permutations of the roll
3770        automatedPermutations = integerPermutations("155AXY");
3771
3772        // Convert the list of Strings to a list of lists of integers
3773        for (String numbers : automatedPermutations) {
3774            myPermutations = permutations(numbers);
3775
3776            // Check each list of integers for proper scoring
3777            for (List<Integer> currentNumbers : myPermutations) {
3778                // Test the total score
3779                score = game.calculateScore(currentNumbers, false);
3780                assertEquals(200, score);
3781
3782                // Test the held score
3783                score = game.calculateScore(currentNumbers, true);
3784                assertEquals(0, score);
3785
3786                // Test the calculateHighestScore method
3787                highestScore = game.calculateHighestScore(currentNumbers);
3788                assertEquals(200, highestScore[0]);
3789                highestRoll = (List<Integer>) highestScore[1];
3790                assertEquals(highestRoll.size(), 3);
3791                assertTrue(Collections.frequency(highestRoll, 1) == 1);
3792                assertTrue(Collections.frequency(highestRoll, 5) == 2);
3793            }
3794        }
3795
3796        /*
3797         * Test 111: Check scoring for a die roll of 115AAX (A and X each
3798         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3799         * 6.2.0
3800         */
3801
3802        // Generate all permutations of the roll
3803        automatedPermutations = integerPermutations("115AAX");
3804
3805        // Convert the list of Strings to a list of lists of integers
3806        for (String numbers : automatedPermutations) {
3807            myPermutations = permutations(numbers);
3808
3809            // Check each list of integers for proper scoring
3810            for (List<Integer> currentNumbers : myPermutations) {
3811                // Test the total score
3812                score = game.calculateScore(currentNumbers, false);
3813                assertEquals(250, score);
3814
3815                // Test the held score
3816                score = game.calculateScore(currentNumbers, true);
3817                assertEquals(0, score);
3818
3819                // Test the calculateHighestScore method
```

```
3820                highestScore = game.calculateHighestScore(currentNumbers);
3821                assertEquals(250, highestScore[0]);
3822                highestRoll = (List<Integer>) highestScore[1];
3823                assertEquals(highestRoll.size(), 3);
3824                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3825                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3826            }
3827        }
3828
3829        /*
3830         * Test 112: Check scoring for a die roll of 115AXY (A, X and Y each
3831         * represent all die values other than 1 or 5) Requirement 6.1.0 and
3832         * 6.2.0
3833         */
3834
3835        // Generate all permutations of the roll
3836        automatedPermutations = integerPermutations("115AXY");
3837
3838        // Convert the list of Strings to a list of lists of integers
3839        for (String numbers : automatedPermutations) {
3840            myPermutations = permutations(numbers);
3841
3842            // Check each list of integers for proper scoring
3843            for (List<Integer> currentNumbers : myPermutations) {
3844                // Test the total score
3845                score = game.calculateScore(currentNumbers, false);
3846                assertEquals(250, score);
3847
3848                // Test the held score
3849                score = game.calculateScore(currentNumbers, true);
3850                assertEquals(0, score);
3851
3852                // Test the calculateHighestScore method
3853                highestScore = game.calculateHighestScore(currentNumbers);
3854                assertEquals(250, highestScore[0]);
3855                highestRoll = (List<Integer>) highestScore[1];
3856                assertEquals(highestRoll.size(), 3);
3857                assertTrue(Collections.frequency(highestRoll, 1) == 2);
3858                assertTrue(Collections.frequency(highestRoll, 5) == 1);
3859            }
3860        }
3861
3862        /*
3863         * Test 113: Check scoring for a die roll of AAAXXY (A, X and Y
3864         * represent all die values other than 1 or 5) Requirement 6.4.0
3865         */
3866
3867        // Generate all permutations of the roll
3868        automatedPermutations = integerPermutations("AAAXXY");
3869
3870        // Convert the list of Strings to a list of lists of integers
3871        for (String numbers : automatedPermutations) {
3872            myPermutations = permutations(numbers);
3873
3874            // Get the die value for calculating the score
3875            dieOneValue = Character.getNumericValue(numbers.charAt(0));
3876
```

```
3877            // Check each list of integers for proper scoring
3878            for (List<Integer> currentNumbers : myPermutations) {
3879                // Test the total score
3880                score = game.calculateScore(currentNumbers, false);
3881                assertEquals(100 * dieOneValue, score);
3882
3883                // Test the held score
3884                score = game.calculateScore(currentNumbers, true);
3885                assertEquals(0, score);
3886
3887                // Test the calculateHighestScore method
3888                highestScore = game.calculateHighestScore(currentNumbers);
3889                assertEquals(100 * dieOneValue, highestScore[0]);
3890                highestRoll = (List<Integer>) highestScore[1];
3891                assertEquals(highestRoll.size(), 3);
3892                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
3893            }
3894        }
3895
3896        /*
3897         * Test 114: Check scoring for a die roll of AAAXYZ (A, X, Y and Z
3898         * represent all die values other than 1 or 5) Requirement 6.4.0
3899         */
3900
3901        // Generate all permutations of the roll
3902        automatedPermutations = integerPermutations("AAAXYZ");
3903
3904        // Convert the list of Strings to a list of lists of integers
3905        for (String numbers : automatedPermutations) {
3906            myPermutations = permutations(numbers);
3907
3908            // Get the die value for calculating the score
3909            dieOneValue = Character.getNumericValue(numbers.charAt(0));
3910
3911            // Check each list of integers for proper scoring
3912            for (List<Integer> currentNumbers : myPermutations) {
3913                // Test the total score
3914                score = game.calculateScore(currentNumbers, false);
3915                assertEquals(100 * dieOneValue, score);
3916
3917                // Test the held score
3918                score = game.calculateScore(currentNumbers, true);
3919                assertEquals(0, score);
3920
3921                // Test the calculateHighestScore method
3922                highestScore = game.calculateHighestScore(currentNumbers);
3923                assertEquals(100 * dieOneValue, highestScore[0]);
3924                highestRoll = (List<Integer>) highestScore[1];
3925                assertEquals(highestRoll.size(), 3);
3926                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
3927            }
3928        }
3929
3930        /*
3931         * Test 115: Check scoring for a die roll of 555AAX (A and X represent
3932         * all die values other than 1 or 5) Requirement 6.4.0
3933         */
```

```
3934
3935        // Generate all permutations of the roll
3936        automatedPermutations = integerPermutations("555AAX");
3937
3938        // Convert the list of Strings to a list of lists of integers
3939        for (String numbers : automatedPermutations) {
3940            myPermutations = permutations(numbers);
3941
3942            // Check each list of integers for proper scoring
3943            for (List<Integer> currentNumbers : myPermutations) {
3944                // Test the total score
3945                score = game.calculateScore(currentNumbers, false);
3946                assertEquals(500, score);
3947
3948                // Test the held score
3949                score = game.calculateScore(currentNumbers, true);
3950                assertEquals(0, score);
3951
3952                // Test the calculateHighestScore method
3953                highestScore = game.calculateHighestScore(currentNumbers);
3954                assertEquals(500, highestScore[0]);
3955                highestRoll = (List<Integer>) highestScore[1];
3956                assertEquals(highestRoll.size(), 3);
3957                assertTrue(Collections.frequency(highestRoll, 5) == 3);
3958            }
3959        }
3960
3961        /*
3962         * Test 116: Check scoring for a die roll of 555AXY (A, X and Y
3963         * represent all die values other than 1 or 5) Requirement 6.4.0
3964         */
3965
3966        // Generate all permutations of the roll
3967        automatedPermutations = integerPermutations("555AXY");
3968
3969        // Convert the list of Strings to a list of lists of integers
3970        for (String numbers : automatedPermutations) {
3971            myPermutations = permutations(numbers);
3972
3973            // Check each list of integers for proper scoring
3974            for (List<Integer> currentNumbers : myPermutations) {
3975                // Test the total score
3976                score = game.calculateScore(currentNumbers, false);
3977                assertEquals(500, score);
3978
3979                // Test the held score
3980                score = game.calculateScore(currentNumbers, true);
3981                assertEquals(0, score);
3982
3983                // Test the calculateHighestScore method
3984                highestScore = game.calculateHighestScore(currentNumbers);
3985                assertEquals(500, highestScore[0]);
3986                highestRoll = (List<Integer>) highestScore[1];
3987                assertEquals(highestRoll.size(), 3);
3988                assertTrue(Collections.frequency(highestRoll, 5) == 3);
3989            }
3990        }
```

```
3991
3992        /*
3993         * Test 117: Check scoring for a die roll of 111AAX (A and X represent
3994         * all die values other than 1 or 5) Requirement 6.3.0
3995         */
3996
3997        // Generate all permutations of the roll
3998        automatedPermutations = integerPermutations("111AAX");
3999
4000        // Convert the list of Strings to a list of lists of integers
4001        for (String numbers : automatedPermutations) {
4002            myPermutations = permutations(numbers);
4003
4004            // Check each list of integers for proper scoring
4005            for (List<Integer> currentNumbers : myPermutations) {
4006                // Test the total score
4007                score = game.calculateScore(currentNumbers, false);
4008                assertEquals(1000, score);
4009
4010                // Test the held score
4011                score = game.calculateScore(currentNumbers, true);
4012                assertEquals(0, score);
4013
4014                // Test the calculateHighestScore method
4015                highestScore = game.calculateHighestScore(currentNumbers);
4016                assertEquals(1000, highestScore[0]);
4017                highestRoll = (List<Integer>) highestScore[1];
4018                assertEquals(highestRoll.size(), 3);
4019                assertTrue(Collections.frequency(highestRoll, 1) == 3);
4020            }
4021        }
4022
4023        /*
4024         * Test 118: Check scoring for a die roll of 111AXY (A, X and Y
4025         * represent all die values other than 1 or 5) Requirement 6.3.0
4026         */
4027
4028        // Generate all permutations of the roll
4029        automatedPermutations = integerPermutations("111AXY");
4030
4031        // Convert the list of Strings to a list of lists of integers
4032        for (String numbers : automatedPermutations) {
4033            myPermutations = permutations(numbers);
4034
4035            // Check each list of integers for proper scoring
4036            for (List<Integer> currentNumbers : myPermutations) {
4037                // Test the total score
4038                score = game.calculateScore(currentNumbers, false);
4039                assertEquals(1000, score);
4040
4041                // Test the held score
4042                score = game.calculateScore(currentNumbers, true);
4043                assertEquals(0, score);
4044
4045                // Test the calculateHighestScore method
4046                highestScore = game.calculateHighestScore(currentNumbers);
4047                assertEquals(1000, highestScore[0]);
```

```
4048                highestRoll = (List<Integer>) highestScore[1];
4049                assertEquals(highestRoll.size(), 3);
4050                assertTrue(Collections.frequency(highestRoll, 1) == 3);
4051            }
4052        }
4053
4054        /*
4055         * Test 119: Check scoring for a die roll of 1AAAXX (A and X each
4056         * represent all die values other than 1 or 5) Requirement 6.1.0 and
4057         * 6.4.0
4058         */
4059
4060        // Generate all permutations of the roll
4061        automatedPermutations = integerPermutations("1AAAXX");
4062
4063        // Convert the list of Strings to a list of lists of integers
4064        for (String numbers : automatedPermutations) {
4065            myPermutations = permutations(numbers);
4066
4067            // Get the die value for calculating the score
4068            dieOneValue = Character.getNumericValue(numbers.charAt(1));
4069
4070            // Check each list of integers for proper scoring
4071            for (List<Integer> currentNumbers : myPermutations) {
4072                // Test the total score
4073                score = game.calculateScore(currentNumbers, false);
4074                assertEquals(100 * dieOneValue + 100, score);
4075
4076                // Test the held score
4077                score = game.calculateScore(currentNumbers, true);
4078                assertEquals(0, score);
4079
4080                // Test the calculateHighestScore method
4081                highestScore = game.calculateHighestScore(currentNumbers);
4082                assertEquals(100 * dieOneValue + 100, highestScore[0]);
4083                highestRoll = (List<Integer>) highestScore[1];
4084                assertEquals(highestRoll.size(), 4);
4085                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4086                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4087            }
4088        }
4089
4090        /*
4091         * Test 120: Check scoring for a die roll of 1AAAXY (A, X and Y each
4092         * represent all die values other than 1 or 5) Requirement 6.1.0 and
4093         * 6.4.0
4094         */
4095
4096        // Generate all permutations of the roll
4097        automatedPermutations = integerPermutations("1AAAXY");
4098
4099        // Convert the list of Strings to a list of lists of integers
4100        for (String numbers : automatedPermutations) {
4101            myPermutations = permutations(numbers);
4102
4103            // Get the die value for calculating the score
4104            dieOneValue = Character.getNumericValue(numbers.charAt(1));
```

```
4105
4106            // Check each list of integers for proper scoring
4107            for (List<Integer> currentNumbers : myPermutations) {
4108                // Test the total score
4109                score = game.calculateScore(currentNumbers, false);
4110                assertEquals(100 * dieOneValue + 100, score);
4111
4112                // Test the held score
4113                score = game.calculateScore(currentNumbers, true);
4114                assertEquals(0, score);
4115
4116                // Test the calculateHighestScore method
4117                highestScore = game.calculateHighestScore(currentNumbers);
4118                assertEquals(100 * dieOneValue + 100, highestScore[0]);
4119                highestRoll = (List<Integer>) highestScore[1];
4120                assertEquals(highestRoll.size(), 4);
4121                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4122                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4123            }
4124        }
4125
4126        /*
4127         * Test 121: Check scoring for a die roll of 5AAAXX (A and X each
4128         * represent all die values other than 1 or 5) Requirement 6.2.0 and
4129         * 6.4.0
4130         */
4131
4132        // Generate all permutations of the roll
4133        automatedPermutations = integerPermutations("5AAAXX");
4134
4135        // Convert the list of Strings to a list of lists of integers
4136        for (String numbers : automatedPermutations) {
4137            myPermutations = permutations(numbers);
4138
4139            // Get the die value for calculating the score
4140            dieOneValue = Character.getNumericValue(numbers.charAt(1));
4141
4142            // Check each list of integers for proper scoring
4143            for (List<Integer> currentNumbers : myPermutations) {
4144                // Test the total score
4145                score = game.calculateScore(currentNumbers, false);
4146                assertEquals(100 * dieOneValue + 50, score);
4147
4148                // Test the held score
4149                score = game.calculateScore(currentNumbers, true);
4150                assertEquals(0, score);
4151
4152                // Test the calculateHighestScore method
4153                highestScore = game.calculateHighestScore(currentNumbers);
4154                assertEquals(100 * dieOneValue + 50, highestScore[0]);
4155                highestRoll = (List<Integer>) highestScore[1];
4156                assertEquals(highestRoll.size(), 4);
4157                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4158                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4159            }
4160        }
4161
```

```
4162        /*
4163         * Test 122: Check scoring for a die roll of 5AAAXY (A, X and Y each
4164         * represent all die values other than 1 or 5) Requirement 6.2.0 and
4165         * 6.4.0
4166         */
4167
4168        // Generate all permutations of the roll
4169        automatedPermutations = integerPermutations("5AAAXY");
4170
4171        // Convert the list of Strings to a list of lists of integers
4172        for (String numbers : automatedPermutations) {
4173            myPermutations = permutations(numbers);
4174
4175            // Get the die value for calculating the score
4176            dieOneValue = Character.getNumericValue(numbers.charAt(1));
4177
4178            // Check each list of integers for proper scoring
4179            for (List<Integer> currentNumbers : myPermutations) {
4180                // Test the total score
4181                score = game.calculateScore(currentNumbers, false);
4182                assertEquals(100 * dieOneValue + 50, score);
4183
4184                // Test the held score
4185                score = game.calculateScore(currentNumbers, true);
4186                assertEquals(0, score);
4187
4188                // Test the calculateHighestScore method
4189                highestScore = game.calculateHighestScore(currentNumbers);
4190                assertEquals(100 * dieOneValue + 50, highestScore[0]);
4191                highestRoll = (List<Integer>) highestScore[1];
4192                assertEquals(highestRoll.size(), 4);
4193                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4194                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4195            }
4196        }
4197
4198        /*
4199         * Test 123: Check scoring for a die roll of 55AAAX (A and X each
4200         * represent all die values other than 1 or 5) Requirement 6.2.0 and
4201         * 6.4.0
4202         */
4203
4204        // Generate all permutations of the roll
4205        automatedPermutations = integerPermutations("55AAAX");
4206
4207        // Convert the list of Strings to a list of lists of integers
4208        for (String numbers : automatedPermutations) {
4209            myPermutations = permutations(numbers);
4210
4211            // Get the die value for calculating the score
4212            dieOneValue = Character.getNumericValue(numbers.charAt(2));
4213
4214            // Check each list of integers for proper scoring
4215            for (List<Integer> currentNumbers : myPermutations) {
4216                // Test the total score
4217                score = game.calculateScore(currentNumbers, false);
4218                assertEquals(100 * dieOneValue + 100, score);
```

```
4219
4220                // Test the held score
4221                score = game.calculateScore(currentNumbers, true);
4222                assertEquals(0, score);
4223
4224                // Test the calculateHighestScore method
4225                highestScore = game.calculateHighestScore(currentNumbers);
4226                assertEquals(100 * dieOneValue + 100, highestScore[0]);
4227                highestRoll = (List<Integer>) highestScore[1];
4228                assertEquals(highestRoll.size(), 5);
4229                assertTrue(Collections.frequency(highestRoll, 5) == 2);
4230                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4231            }
4232        }
4233
4234        /*
4235         * Test 124: Check scoring for a die roll of 15AAAX (A and X each
4236         * represent all die values other than 1 or 5) Requirement 6.1.0, 6.2.0,
4237         * and 6.4.0
4238         */
4239
4240        // Generate all permutations of the roll
4241        automatedPermutations = integerPermutations("15AAAX");
4242
4243        // Convert the list of Strings to a list of lists of integers
4244        for (String numbers : automatedPermutations) {
4245            myPermutations = permutations(numbers);
4246
4247            // Get the die value for calculating the score
4248            dieOneValue = Character.getNumericValue(numbers.charAt(2));
4249
4250            // Check each list of integers for proper scoring
4251            for (List<Integer> currentNumbers : myPermutations) {
4252                // Test the total score
4253                score = game.calculateScore(currentNumbers, false);
4254                assertEquals(100 * dieOneValue + 150, score);
4255
4256                // Test the held score
4257                score = game.calculateScore(currentNumbers, true);
4258                assertEquals(0, score);
4259
4260                // Test the calculateHighestScore method
4261                highestScore = game.calculateHighestScore(currentNumbers);
4262                assertEquals(100 * dieOneValue + 150, highestScore[0]);
4263                highestRoll = (List<Integer>) highestScore[1];
4264                assertEquals(highestRoll.size(), 5);
4265                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4266                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4267                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4268            }
4269        }
4270
4271        /*
4272         * Test 125: Check scoring for a die roll of 11AAAX (A and X each
4273         * represent all die values other than 1 or 5) Requirement 6.1.0 and
4274         * 6.4.0
4275         */
```

```
4276
4277        // Generate all permutations of the roll
4278        automatedPermutations = integerPermutations("11AAAX");
4279
4280        // Convert the list of Strings to a list of lists of integers
4281        for (String numbers : automatedPermutations) {
4282            myPermutations = permutations(numbers);
4283
4284            // Get the die value for calculating the score
4285            dieOneValue = Character.getNumericValue(numbers.charAt(2));
4286
4287            // Check each list of integers for proper scoring
4288            for (List<Integer> currentNumbers : myPermutations) {
4289                // Test the total score
4290                score = game.calculateScore(currentNumbers, false);
4291                assertEquals(100 * dieOneValue + 200, score);
4292
4293                // Test the held score
4294                score = game.calculateScore(currentNumbers, true);
4295                assertEquals(0, score);
4296
4297                // Test the calculateHighestScore method
4298                highestScore = game.calculateHighestScore(currentNumbers);
4299                assertEquals(100 * dieOneValue + 200, highestScore[0]);
4300                highestRoll = (List<Integer>) highestScore[1];
4301                assertEquals(highestRoll.size(), 5);
4302                assertTrue(Collections.frequency(highestRoll, 1) == 2);
4303                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4304            }
4305        }
4306
4307        /*
4308         * Test 126: Check scoring for a die roll of 155AAA (A represents all
4309         * die values other than 1 or 5) Requirement 6.1.0, 6.2.0, and 6.4.0
4310         */
4311
4312        // Generate all permutations of the roll
4313        automatedPermutations = integerPermutations("155AAA");
4314
4315        // Convert the list of Strings to a list of lists of integers
4316        for (String numbers : automatedPermutations) {
4317            myPermutations = permutations(numbers);
4318
4319            // Get the die value for calculating the score
4320            dieOneValue = Character.getNumericValue(numbers.charAt(3));
4321
4322            // Check each list of integers for proper scoring
4323            for (List<Integer> currentNumbers : myPermutations) {
4324                // Test the total score
4325                score = game.calculateScore(currentNumbers, false);
4326                assertEquals(100 * dieOneValue + 200, score);
4327
4328                // Test the held score
4329                score = game.calculateScore(currentNumbers, true);
4330                assertEquals(100 * dieOneValue + 200, score);
4331
4332                // Test the calculateHighestScore method
```

```
4333                highestScore = game.calculateHighestScore(currentNumbers);
4334                assertEquals(100 * dieOneValue + 200, highestScore[0]);
4335                highestRoll = (List<Integer>) highestScore[1];
4336                assertEquals(highestRoll.size(), 6);
4337                assertTrue(Collections.frequency(highestRoll, 5) == 2);
4338                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4339                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4340            }
4341        }
4342
4343        /*
4344         * Test 127: Check scoring for a die roll of 115AAA (A represents all
4345         * die values other than 1 or 5) Requirement 6.1.0, 6.2.0, and 6.4.0
4346         */
4347
4348        // Generate all permutations of the roll
4349        automatedPermutations = integerPermutations("115AAA");
4350
4351        // Convert the list of Strings to a list of lists of integers
4352        for (String numbers : automatedPermutations) {
4353            myPermutations = permutations(numbers);
4354
4355            // Get the die value for calculating the score
4356            dieOneValue = Character.getNumericValue(numbers.charAt(3));
4357
4358            // Check each list of integers for proper scoring
4359            for (List<Integer> currentNumbers : myPermutations) {
4360                // Test the total score
4361                score = game.calculateScore(currentNumbers, false);
4362                assertEquals(100 * dieOneValue + 250, score);
4363
4364                // Test the held score
4365                score = game.calculateScore(currentNumbers, true);
4366                assertEquals(100 * dieOneValue + 250, score);
4367
4368                // Test the calculateHighestScore method
4369                highestScore = game.calculateHighestScore(currentNumbers);
4370                assertEquals(100 * dieOneValue + 250, highestScore[0]);
4371                highestRoll = (List<Integer>) highestScore[1];
4372                assertEquals(highestRoll.size(), 6);
4373                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4374                assertTrue(Collections.frequency(highestRoll, 1) == 2);
4375                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4376            }
4377        }
4378
4379        /*
4380         * Test 128: Check scoring for a die roll of AAAXXX (A and X each
4381         * represent all die values other than 1 or 5) Requirement 6.4.0
4382         */
4383
4384        // Generate all permutations of the roll
4385        automatedPermutations = integerPermutations("AAAXXX");
4386
4387        // Convert the list of Strings to a list of lists of integers
4388        for (String numbers : automatedPermutations) {
4389            myPermutations = permutations(numbers);
```

```
4390
4391            // Get the die value for calculating the score
4392            dieOneValue = Character.getNumericValue(numbers.charAt(0));
4393
4394            // Get the die value for calculating the score
4395            dieTwoValue = Character.getNumericValue(numbers.charAt(3));
4396
4397            // Check each list of integers for proper scoring
4398            for (List<Integer> currentNumbers : myPermutations) {
4399                // Test the total score
4400                score = game.calculateScore(currentNumbers, false);
4401                assertEquals(100 * dieOneValue + 100 * dieTwoValue, score);
4402
4403                // Test the held score
4404                score = game.calculateScore(currentNumbers, true);
4405                assertEquals(100 * dieOneValue + 100 * dieTwoValue, score);
4406
4407                // Test the calculateHighestScore method
4408                highestScore = game.calculateHighestScore(currentNumbers);
4409                assertEquals(100 * dieOneValue + 100 * dieTwoValue,
4410                        highestScore[0]);
4411                highestRoll = (List<Integer>) highestScore[1];
4412                assertEquals(highestRoll.size(), 6);
4413                assertTrue(Collections.frequency(highestRoll, dieTwoValue) == 3);
4414                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4415            }
4416        }
4417
4418        /*
4419         * Test 129: Check scoring for a die roll of 555AAA (A represents all
4420         * die values other than 1 or 5) Requirement 6.4.0
4421         */
4422
4423        // Generate all permutations of the roll
4424        automatedPermutations = integerPermutations("555AAA");
4425
4426        // Convert the list of Strings to a list of lists of integers
4427        for (String numbers : automatedPermutations) {
4428            myPermutations = permutations(numbers);
4429
4430            // Get the die value for calculating the score
4431            dieOneValue = Character.getNumericValue(numbers.charAt(3));
4432
4433            // Check each list of integers for proper scoring
4434            for (List<Integer> currentNumbers : myPermutations) {
4435                // Test the total score
4436                score = game.calculateScore(currentNumbers, false);
4437                assertEquals(100 * dieOneValue + 500, score);
4438
4439                // Test the held score
4440                score = game.calculateScore(currentNumbers, true);
4441                assertEquals(100 * dieOneValue + 500, score);
4442
4443                // Test the calculateHighestScore method
4444                highestScore = game.calculateHighestScore(currentNumbers);
4445                assertEquals(100 * dieOneValue + 500, highestScore[0]);
4446                highestRoll = (List<Integer>) highestScore[1];
```

```
4447                assertEquals(highestRoll.size(), 6);
4448                assertTrue(Collections.frequency(highestRoll, 5) == 3);
4449                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4450            }
4451        }
4452
4453        /*
4454         * Test 130: Check scoring for a die roll of 111AAA (A represents all
4455         * die values other than 1 or 5) Requirement 6.3.0 and 6.4.0
4456         */
4457
4458        // Generate all permutations of the roll
4459        automatedPermutations = integerPermutations("111AAA");
4460
4461        // Convert the list of Strings to a list of lists of integers
4462        for (String numbers : automatedPermutations) {
4463            myPermutations = permutations(numbers);
4464
4465            // Get the die value for calculating the score
4466            dieOneValue = Character.getNumericValue(numbers.charAt(3));
4467
4468            // Check each list of integers for proper scoring
4469            for (List<Integer> currentNumbers : myPermutations) {
4470                // Test the total score
4471                score = game.calculateScore(currentNumbers, false);
4472                assertEquals(100 * dieOneValue + 1000, score);
4473
4474                // Test the held score
4475                score = game.calculateScore(currentNumbers, true);
4476                assertEquals(100 * dieOneValue + 1000, score);
4477
4478                // Test the calculateHighestScore method
4479                highestScore = game.calculateHighestScore(currentNumbers);
4480                assertEquals(100 * dieOneValue + 1000, highestScore[0]);
4481                highestRoll = (List<Integer>) highestScore[1];
4482                assertEquals(highestRoll.size(), 6);
4483                assertTrue(Collections.frequency(highestRoll, 1) == 3);
4484                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 3);
4485            }
4486        }
4487
4488        /*
4489         * Test 131: Check scoring for a die roll of 111555 Requirement 6.3.0
4490         * and 6.4.0
4491         */
4492
4493        // Generate all permutations of the roll and store in a list of lists of
4494        // integers
4495        myPermutations = permutations("111555");
4496
4497        // Check each list of integers for correct scoring
4498        for (List<Integer> numbers : myPermutations) {
4499            // Test the total score
4500            score = game.calculateScore(numbers, false);
4501            assertEquals(1500, score);
4502
4503            // Test the held score
```

```
4504            score = game.calculateScore(numbers, true);
4505            assertEquals(1500, score);
4506
4507            // Test the calculateHighestScore method
4508            highestScore = game.calculateHighestScore(numbers);
4509            assertEquals(1500, highestScore[0]);
4510            highestRoll = (List<Integer>) highestScore[1];
4511            assertEquals(highestRoll.size(), 6);
4512            assertTrue(Collections.frequency(highestRoll, 5) == 3);
4513            assertTrue(Collections.frequency(highestRoll, 1) == 3);
4514        }
4515
4516        /*
4517         * Test 132: Check scoring for a die roll of 5551AA (A represents all
4518         * die values other than 1 or 5) Requirement 6.1.0 and 6.4.0
4519         */
4520
4521        // Generate all permutations of the roll
4522        automatedPermutations = integerPermutations("5551AA");
4523
4524        // Convert the list of Strings to a list of lists of integers
4525        for (String numbers : automatedPermutations) {
4526            myPermutations = permutations(numbers);
4527
4528            // Check each list of integers for proper scoring
4529            for (List<Integer> currentNumbers : myPermutations) {
4530                // Test the total score
4531                score = game.calculateScore(currentNumbers, false);
4532                assertEquals(600, score);
4533
4534                // Test the held score
4535                score = game.calculateScore(currentNumbers, true);
4536                assertEquals(0, score);
4537
4538                // Test the calculateHighestScore method
4539                highestScore = game.calculateHighestScore(currentNumbers);
4540                assertEquals(600, highestScore[0]);
4541                highestRoll = (List<Integer>) highestScore[1];
4542                assertEquals(highestRoll.size(), 4);
4543                assertTrue(Collections.frequency(highestRoll, 5) == 3);
4544                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4545            }
4546        }
4547
4548        /*
4549         * Test 133: Check scoring for a die roll of 5551AX (A and X each
4550         * represent all die values other than 1 or 5) Requirement 6.1.0 and
4551         * 6.4.0
4552         */
4553
4554        // Generate all permutations of the roll
4555        automatedPermutations = integerPermutations("5551AX");
4556
4557        // Convert the list of Strings to a list of lists of integers
4558        for (String numbers : automatedPermutations) {
4559            myPermutations = permutations(numbers);
4560
```

```
4561            // Check each list of integers for proper scoring
4562            for (List<Integer> currentNumbers : myPermutations) {
4563                // Test the total score
4564                score = game.calculateScore(currentNumbers, false);
4565                assertEquals(600, score);
4566
4567                // Test the held score
4568                score = game.calculateScore(currentNumbers, true);
4569                assertEquals(0, score);
4570
4571                // Test the calculateHighestScore method
4572                highestScore = game.calculateHighestScore(currentNumbers);
4573                assertEquals(600, highestScore[0]);
4574                highestRoll = (List<Integer>) highestScore[1];
4575                assertEquals(highestRoll.size(), 4);
4576                assertTrue(Collections.frequency(highestRoll, 5) == 3);
4577                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4578            }
4579        }
4580
4581        /*
4582         * Test 134: Check scoring for a die roll of 55511A (A represents all
4583         * die values other than 1 or 5) Requirement 6.1.0 and 6.4.0
4584         */
4585
4586        // Generate all permutations of the roll
4587        automatedPermutations = integerPermutations("55511A");
4588
4589        // Convert the list of Strings to a list of lists of integers
4590        for (String numbers : automatedPermutations) {
4591            myPermutations = permutations(numbers);
4592
4593            // Check each list of integers for proper scoring
4594            for (List<Integer> currentNumbers : myPermutations) {
4595                // Test the total score
4596                score = game.calculateScore(currentNumbers, false);
4597                assertEquals(700, score);
4598
4599                // Test the held score
4600                score = game.calculateScore(currentNumbers, true);
4601                assertEquals(0, score);
4602
4603                // Test the calculateHighestScore method
4604                highestScore = game.calculateHighestScore(currentNumbers);
4605                assertEquals(700, highestScore[0]);
4606                highestRoll = (List<Integer>) highestScore[1];
4607                assertEquals(highestRoll.size(), 5);
4608                assertTrue(Collections.frequency(highestRoll, 5) == 3);
4609                assertTrue(Collections.frequency(highestRoll, 1) == 2);
4610            }
4611        }
4612
4613        /*
4614         * Test 135: Check scoring for a die roll of 1115AA (A represents all
4615         * die values other than 1 or 5) Requirement 6.2.0 and 6.3.0
4616         */
4617
```

```
4618        // Generate all permutations of the roll
4619        automatedPermutations = integerPermutations("1115AA");
4620
4621        // Convert the list of Strings to a list of lists of integers
4622        for (String numbers : automatedPermutations) {
4623            myPermutations = permutations(numbers);
4624
4625            // Check each list of integers for proper scoring
4626            for (List<Integer> currentNumbers : myPermutations) {
4627                // Test the total score
4628                score = game.calculateScore(currentNumbers, false);
4629                assertEquals(1050, score);
4630
4631                // Test the held score
4632                score = game.calculateScore(currentNumbers, true);
4633                assertEquals(0, score);
4634
4635                // Test the calculateHighestScore method
4636                highestScore = game.calculateHighestScore(currentNumbers);
4637                assertEquals(1050, highestScore[0]);
4638                highestRoll = (List<Integer>) highestScore[1];
4639                assertEquals(highestRoll.size(), 4);
4640                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4641                assertTrue(Collections.frequency(highestRoll, 1) == 3);
4642            }
4643        }
4644
4645        /*
4646         * Test 136: Check scoring for a die roll of 1115AX (A and X each
4647         * represent all die values other than 1 or 5) Requirement 6.2.0 and
4648         * 6.4.0
4649         */
4650
4651        // Generate all permutations of the roll
4652        automatedPermutations = integerPermutations("1115AX");
4653
4654        // Convert the list of Strings to a list of lists of integers
4655        for (String numbers : automatedPermutations) {
4656            myPermutations = permutations(numbers);
4657
4658            // Check each list of integers for proper scoring
4659            for (List<Integer> currentNumbers : myPermutations) {
4660                // Test the total score
4661                score = game.calculateScore(currentNumbers, false);
4662                assertEquals(1050, score);
4663
4664                // Test the held score
4665                score = game.calculateScore(currentNumbers, true);
4666                assertEquals(0, score);
4667
4668                // Test the calculateHighestScore method
4669                highestScore = game.calculateHighestScore(currentNumbers);
4670                assertEquals(1050, highestScore[0]);
4671                highestRoll = (List<Integer>) highestScore[1];
4672                assertEquals(highestRoll.size(), 4);
4673                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4674                assertTrue(Collections.frequency(highestRoll, 1) == 3);
```

```
4675            }
4676        }
4677
4678        /*
4679         * Test 137: Check scoring for a die roll of 11155A (A represents all
4680         * die values other than 1 or 5) Requirement 6.2.0 and 6.3.0
4681         */
4682
4683        // Generate all permutations of the roll
4684        automatedPermutations = integerPermutations("11155A");
4685
4686        // Convert the list of Strings to a list of lists of integers
4687        for (String numbers : automatedPermutations) {
4688            myPermutations = permutations(numbers);
4689
4690            // Check each list of integers for proper scoring
4691            for (List<Integer> currentNumbers : myPermutations) {
4692                // Test the total score
4693                score = game.calculateScore(currentNumbers, false);
4694                assertEquals(1100, score);
4695
4696                // Test the held score
4697                score = game.calculateScore(currentNumbers, true);
4698                assertEquals(0, score);
4699
4700                // Test the calculateHighestScore method
4701                highestScore = game.calculateHighestScore(currentNumbers);
4702                assertEquals(1100, highestScore[0]);
4703                highestRoll = (List<Integer>) highestScore[1];
4704                assertEquals(highestRoll.size(), 5);
4705                assertTrue(Collections.frequency(highestRoll, 5) == 2);
4706                assertTrue(Collections.frequency(highestRoll, 1) == 3);
4707            }
4708        }
4709
4710        /*
4711         * Test 138: Check scoring for a die roll of AAAAXX (A and X each
4712         * represent all die values other than 1 or 5) Requirement 6.5.0
4713         */
4714
4715        // Generate all permutations of the roll
4716        automatedPermutations = integerPermutations("AAAAXX");
4717
4718        // Convert the list of Strings to a list of lists of integers
4719        for (String numbers : automatedPermutations) {
4720            myPermutations = permutations(numbers);
4721
4722            // Get the die value for calculating the score
4723            dieOneValue = Character.getNumericValue(numbers.charAt(0));
4724
4725            // Check each list of integers for proper scoring
4726            for (List<Integer> currentNumbers : myPermutations) {
4727                // Test the total score
4728                score = game.calculateScore(currentNumbers, false);
4729                assertEquals(100 * dieOneValue * 2, score);
4730
4731                // Test the held score
```

```
4732                score = game.calculateScore(currentNumbers, true);
4733                assertEquals(0, score);
4734
4735                // Test the calculateHighestScore method
4736                highestScore = game.calculateHighestScore(currentNumbers);
4737                assertEquals(100 * dieOneValue * 2, highestScore[0]);
4738                highestRoll = (List<Integer>) highestScore[1];
4739                assertEquals(highestRoll.size(), 4);
4740                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
4741            }
4742        }
4743
4744        /*
4745         * Test 139: Check scoring for a die roll of AAAAXY (A, X and Y each
4746         * represent all die values other than 1 or 5) Requirement 6.5.0
4747         */
4748
4749        // Generate all permutations of the roll
4750        automatedPermutations = integerPermutations("AAAAXY");
4751
4752        // Convert the list of Strings to a list of lists of integers
4753        for (String numbers : automatedPermutations) {
4754            myPermutations = permutations(numbers);
4755
4756            // Get the die value for calculating the score
4757            dieOneValue = Character.getNumericValue(numbers.charAt(0));
4758
4759            // Check each list of integers for proper scoring
4760            for (List<Integer> currentNumbers : myPermutations) {
4761                // Test the total score
4762                score = game.calculateScore(currentNumbers, false);
4763                assertEquals(100 * dieOneValue * 2, score);
4764
4765                // Test the held score
4766                score = game.calculateScore(currentNumbers, true);
4767                assertEquals(0, score);
4768
4769                // Test the calculateHighestScore method
4770                highestScore = game.calculateHighestScore(currentNumbers);
4771                assertEquals(100 * dieOneValue * 2, highestScore[0]);
4772                highestRoll = (List<Integer>) highestScore[1];
4773                assertEquals(highestRoll.size(), 4);
4774                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
4775            }
4776        }
4777
4778        /*
4779         * Test 140: Check scoring for a die roll of 5555AA (A represents all
4780         * die values other than 1 or 5) Requirement 6.5.0
4781         */
4782
4783        // Generate all permutations of the roll
4784        automatedPermutations = integerPermutations("5555AA");
4785
4786        // Convert the list of Strings to a list of lists of integers
4787        for (String numbers : automatedPermutations) {
4788            myPermutations = permutations(numbers);
```

```
4789
4790            // Check each list of integers for proper scoring
4791            for (List<Integer> currentNumbers : myPermutations) {
4792                // Test the total score
4793                score = game.calculateScore(currentNumbers, false);
4794                assertEquals(1000, score);
4795
4796                // Test the held score
4797                score = game.calculateScore(currentNumbers, true);
4798                assertEquals(0, score);
4799
4800                // Test the calculateHighestScore method
4801                highestScore = game.calculateHighestScore(currentNumbers);
4802                assertEquals(1000, highestScore[0]);
4803                highestRoll = (List<Integer>) highestScore[1];
4804                assertEquals(highestRoll.size(), 4);
4805                assertTrue(Collections.frequency(highestRoll, 5) == 4);
4806            }
4807        }
4808
4809        /*
4810         * Test 141: Check scoring for a die roll of 5555AX (A represents all
4811         * die values other than 1 or 5) Requirement 6.5.0
4812         */
4813
4814        // Generate all permutations of the roll
4815        automatedPermutations = integerPermutations("5555AX");
4816
4817        // Convert the list of Strings to a list of lists of integers
4818        for (String numbers : automatedPermutations) {
4819            myPermutations = permutations(numbers);
4820
4821            // Check each list of integers for proper scoring
4822            for (List<Integer> currentNumbers : myPermutations) {
4823                // Test the total score
4824                score = game.calculateScore(currentNumbers, false);
4825                assertEquals(1000, score);
4826
4827                // Test the held score
4828                score = game.calculateScore(currentNumbers, true);
4829                assertEquals(0, score);
4830
4831                // Test the calculateHighestScore method
4832                highestScore = game.calculateHighestScore(currentNumbers);
4833                assertEquals(1000, highestScore[0]);
4834                highestRoll = (List<Integer>) highestScore[1];
4835                assertEquals(highestRoll.size(), 4);
4836                assertTrue(Collections.frequency(highestRoll, 5) == 4);
4837            }
4838        }
4839
4840        /*
4841         * Test 142: Check scoring for a die roll of 1111AA (A represents all
4842         * die values other than 1 or 5) Requirement 6.5.0
4843         */
4844
4845        // Generate all permutations of the roll
```

```
4846        automatedPermutations = integerPermutations("1111AA");
4847
4848        // Convert the list of Strings to a list of lists of integers
4849        for (String numbers : automatedPermutations) {
4850            myPermutations = permutations(numbers);
4851
4852            // Check each list of integers for proper scoring
4853            for (List<Integer> currentNumbers : myPermutations) {
4854                // Test the total score
4855                score = game.calculateScore(currentNumbers, false);
4856                assertEquals(2000, score);
4857
4858                // Test the held score
4859                score = game.calculateScore(currentNumbers, true);
4860                assertEquals(0, score);
4861
4862                // Test the calculateHighestScore method
4863                highestScore = game.calculateHighestScore(currentNumbers);
4864                assertEquals(2000, highestScore[0]);
4865                highestRoll = (List<Integer>) highestScore[1];
4866                assertEquals(highestRoll.size(), 4);
4867                assertTrue(Collections.frequency(highestRoll, 1) == 4);
4868            }
4869        }
4870
4871        /*
4872         * Test 143: Check scoring for a die roll of 1111AX (A represents all
4873         * die values other than 1 or 5) Requirement 6.5.0
4874         */
4875
4876        // Generate all permutations of the roll
4877        automatedPermutations = integerPermutations("1111AX");
4878
4879        // Convert the list of Strings to a list of lists of integers
4880        for (String numbers : automatedPermutations) {
4881            myPermutations = permutations(numbers);
4882
4883            // Check each list of integers for proper scoring
4884            for (List<Integer> currentNumbers : myPermutations) {
4885                // Test the total score
4886                score = game.calculateScore(currentNumbers, false);
4887                assertEquals(2000, score);
4888
4889                // Test the held score
4890                score = game.calculateScore(currentNumbers, true);
4891                assertEquals(0, score);
4892
4893                // Test the calculateHighestScore method
4894                highestScore = game.calculateHighestScore(currentNumbers);
4895                assertEquals(2000, highestScore[0]);
4896                highestRoll = (List<Integer>) highestScore[1];
4897                assertEquals(highestRoll.size(), 4);
4898                assertTrue(Collections.frequency(highestRoll, 1) == 4);
4899            }
4900        }
4901
4902        /*
```

```
4903         * Test 144: Check scoring for a die roll of 1AAAAX (A amd X each
4904         * represent all die values other than 1 or 5) Requirement 6.1.0 and
4905         * 6.5.0
4906         */
4907
4908        // Generate all permutations of the roll
4909        automatedPermutations = integerPermutations("1AAAAX");
4910
4911        // Convert the list of Strings to a list of lists of integers
4912        for (String numbers : automatedPermutations) {
4913            myPermutations = permutations(numbers);
4914
4915            // Get the die value for calculating the score
4916            dieOneValue = Character.getNumericValue(numbers.charAt(1));
4917
4918            // Check each list of integers for proper scoring
4919            for (List<Integer> currentNumbers : myPermutations) {
4920                // Test the total score
4921                score = game.calculateScore(currentNumbers, false);
4922                assertEquals(100 * dieOneValue * 2 + 100, score);
4923
4924                // Test the held score
4925                score = game.calculateScore(currentNumbers, true);
4926                assertEquals(0, score);
4927
4928                // Test the calculateHighestScore method
4929                highestScore = game.calculateHighestScore(currentNumbers);
4930                assertEquals(100 * dieOneValue * 2 + 100, highestScore[0]);
4931                highestRoll = (List<Integer>) highestScore[1];
4932                assertEquals(highestRoll.size(), 5);
4933                assertTrue(Collections.frequency(highestRoll, 1) == 1);
4934                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
4935            }
4936        }
4937
4938        /*
4939         * Test 145: Check scoring for a die roll of 5AAAAX (A amd X each
4940         * represent all die values other than 1 or 5) Requirement 6.2.0 and
4941         * 6.5.0
4942         */
4943
4944        // Generate all permutations of the roll
4945        automatedPermutations = integerPermutations("5AAAAX");
4946
4947        // Convert the list of Strings to a list of lists of integers
4948        for (String numbers : automatedPermutations) {
4949            myPermutations = permutations(numbers);
4950
4951            // Get the die value for calculating the score
4952            dieOneValue = Character.getNumericValue(numbers.charAt(1));
4953
4954            // Check each list of integers for proper scoring
4955            for (List<Integer> currentNumbers : myPermutations) {
4956                // Test the total score
4957                score = game.calculateScore(currentNumbers, false);
4958                assertEquals(100 * dieOneValue * 2 + 50, score);
4959
```

```
4960                // Test the held score
4961                score = game.calculateScore(currentNumbers, true);
4962                assertEquals(0, score);
4963
4964                // Test the calculateHighestScore method
4965                highestScore = game.calculateHighestScore(currentNumbers);
4966                assertEquals(100 * dieOneValue * 2 + 50, highestScore[0]);
4967                highestRoll = (List<Integer>) highestScore[1];
4968                assertEquals(highestRoll.size(), 5);
4969                assertTrue(Collections.frequency(highestRoll, 5) == 1);
4970                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
4971            }
4972        }
4973
4974        /*
4975         * Test 146: Check scoring for a die roll of 55AAAA (A represents all
4976         * die values other than 1 or 5) Requirement 6.2.0 and 6.5.0
4977         */
4978
4979        // Generate all permutations of the roll
4980        automatedPermutations = integerPermutations("55AAAA");
4981
4982        // Convert the list of Strings to a list of lists of integers
4983        for (String numbers : automatedPermutations) {
4984            myPermutations = permutations(numbers);
4985
4986            // Get the die value for calculating the score
4987            dieOneValue = Character.getNumericValue(numbers.charAt(2));
4988
4989            // Check each list of integers for proper scoring
4990            for (List<Integer> currentNumbers : myPermutations) {
4991                // Test the total score
4992                score = game.calculateScore(currentNumbers, false);
4993                assertEquals(100 * dieOneValue * 2 + 100, score);
4994
4995                // Test the held score
4996                score = game.calculateScore(currentNumbers, true);
4997                assertEquals(100 * dieOneValue * 2 + 100, score);
4998
4999                // Test the calculateHighestScore method
5000                highestScore = game.calculateHighestScore(currentNumbers);
5001                assertEquals(100 * dieOneValue * 2 + 100, highestScore[0]);
5002                highestRoll = (List<Integer>) highestScore[1];
5003                assertEquals(highestRoll.size(), 6);
5004                assertTrue(Collections.frequency(highestRoll, 5) == 2);
5005                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
5006            }
5007        }
5008
5009        /*
5010         * Test 147: Check scoring for a die roll of 15AAAA (A represents all
5011         * die values other than 1 or 5) Requirement 6.1.0, 6.2.0, and 6.5.0
5012         */
5013
5014        // Generate all permutations of the roll
5015        automatedPermutations = integerPermutations("15AAAA");
5016
```

```
5017        // Convert the list of Strings to a list of lists of integers
5018        for (String numbers : automatedPermutations) {
5019            myPermutations = permutations(numbers);
5020
5021            // Get the die value for calculating the score
5022            dieOneValue = Character.getNumericValue(numbers.charAt(2));
5023
5024            // Check each list of integers for proper scoring
5025            for (List<Integer> currentNumbers : myPermutations) {
5026                // Test the total score
5027                score = game.calculateScore(currentNumbers, false);
5028                assertEquals(100 * dieOneValue * 2 + 150, score);
5029
5030                // Test the held score
5031                score = game.calculateScore(currentNumbers, true);
5032                assertEquals(100 * dieOneValue * 2 + 150, score);
5033
5034                // Test the calculateHighestScore method
5035                highestScore = game.calculateHighestScore(currentNumbers);
5036                assertEquals(100 * dieOneValue * 2 + 150, highestScore[0]);
5037                highestRoll = (List<Integer>) highestScore[1];
5038                assertEquals(highestRoll.size(), 6);
5039                assertTrue(Collections.frequency(highestRoll, 1) == 1);
5040                assertTrue(Collections.frequency(highestRoll, 5) == 1);
5041                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
5042            }
5043        }
5044
5045        /*
5046         * Test 148: Check scoring for a die roll of 11AAAA (A represents all
5047         * die values other than 1 or 5) Requirement 6.1.0 and 6.5.0
5048         */
5049
5050        // Generate all permutations of the roll
5051        automatedPermutations = integerPermutations("11AAAA");
5052
5053        // Convert the list of Strings to a list of lists of integers
5054        for (String numbers : automatedPermutations) {
5055            myPermutations = permutations(numbers);
5056
5057            // Get the die value for calculating the score
5058            dieOneValue = Character.getNumericValue(numbers.charAt(2));
5059
5060            // Check each list of integers for proper scoring
5061            for (List<Integer> currentNumbers : myPermutations) {
5062                // Test the total score
5063                score = game.calculateScore(currentNumbers, false);
5064                assertEquals(100 * dieOneValue * 2 + 200, score);
5065
5066                // Test the held score
5067                score = game.calculateScore(currentNumbers, true);
5068                assertEquals(100 * dieOneValue * 2 + 200, score);
5069
5070                // Test the calculateHighestScore method
5071                highestScore = game.calculateHighestScore(currentNumbers);
5072                assertEquals(100 * dieOneValue * 2 + 200, highestScore[0]);
5073                highestRoll = (List<Integer>) highestScore[1];
```

```
5074                assertEquals(highestRoll.size(), 6);
5075                assertTrue(Collections.frequency(highestRoll, 1) == 2);
5076                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 4);
5077            }
5078        }
5079
5080        /*
5081         * Test 149: Check scoring for a die roll of 15555A (A represents all
5082         * die values other than 1 or 5) Requirement 6.1.0 and 6.5.0
5083         */
5084
5085        // Generate all permutations of the roll
5086        automatedPermutations = integerPermutations("15555A");
5087
5088        // Convert the list of Strings to a list of lists of integers
5089        for (String numbers : automatedPermutations) {
5090            myPermutations = permutations(numbers);
5091
5092            // Check each list of integers for proper scoring
5093            for (List<Integer> currentNumbers : myPermutations) {
5094                // Test the total score
5095                score = game.calculateScore(currentNumbers, false);
5096                assertEquals(1100, score);
5097
5098                // Test the held score
5099                score = game.calculateScore(currentNumbers, true);
5100                assertEquals(0, score);
5101
5102                // Test the calculateHighestScore method
5103                highestScore = game.calculateHighestScore(currentNumbers);
5104                assertEquals(1100, highestScore[0]);
5105                highestRoll = (List<Integer>) highestScore[1];
5106                assertEquals(highestRoll.size(), 5);
5107                assertTrue(Collections.frequency(highestRoll, 1) == 1);
5108                assertTrue(Collections.frequency(highestRoll, 5) == 4);
5109            }
5110        }
5111
5112        /*
5113         * Test 150: Check scoring for a die roll of 115555 Requirement 6.1.0
5114         * and 6.5.0
5115         */
5116
5117        // Generate all permutations of the roll and store in a list of lists of
5118        // integers
5119        myPermutations = permutations("115555");
5120
5121        // Check each list of integers for correct scoring
5122        for (List<Integer> numbers : myPermutations) {
5123            // Test the total score
5124            score = game.calculateScore(numbers, false);
5125            assertEquals(1200, score);
5126
5127            // Test the held score
5128            score = game.calculateScore(numbers, true);
5129            assertEquals(1200, score);
5130
```

```
5131            // Test the calculateHighestScore method
5132            highestScore = game.calculateHighestScore(numbers);
5133            assertEquals(1200, highestScore[0]);
5134            highestRoll = (List<Integer>) highestScore[1];
5135            assertEquals(highestRoll.size(), 6);
5136            assertTrue(Collections.frequency(highestRoll, 1) == 2);
5137            assertTrue(Collections.frequency(highestRoll, 5) == 4);
5138        }
5139
5140        /*
5141         * Test 151: Check scoring for a die roll of 11115A (A represents all
5142         * die values other than 1 or 5) Requirement 6.2.0 and 6.5.0
5143         */
5144
5145        // Generate all permutations of the roll
5146        automatedPermutations = integerPermutations("11115A");
5147
5148        // Convert the list of Strings to a list of lists of integers
5149        for (String numbers : automatedPermutations) {
5150            myPermutations = permutations(numbers);
5151
5152            // Check each list of integers for proper scoring
5153            for (List<Integer> currentNumbers : myPermutations) {
5154                // Test the total score
5155                score = game.calculateScore(currentNumbers, false);
5156                assertEquals(2050, score);
5157
5158                // Test the held score
5159                score = game.calculateScore(currentNumbers, true);
5160                assertEquals(0, score);
5161
5162                // Test the calculateHighestScore method
5163                highestScore = game.calculateHighestScore(currentNumbers);
5164                assertEquals(2050, highestScore[0]);
5165                highestRoll = (List<Integer>) highestScore[1];
5166                assertEquals(highestRoll.size(), 5);
5167                assertTrue(Collections.frequency(highestRoll, 1) == 4);
5168                assertTrue(Collections.frequency(highestRoll, 5) == 1);
5169            }
5170        }
5171
5172        /*
5173         * Test 152: Check scoring for a die roll of 111155 Requirement 6.2.0
5174         * and 6.5.0
5175         */
5176
5177        // Generate all permutations of the roll and store in a list of lists of
5178        // integers
5179        myPermutations = permutations("111155");
5180
5181        // Check each list of integers for correct scoring
5182        for (List<Integer> numbers : myPermutations) {
5183            // Test the total score
5184            score = game.calculateScore(numbers, false);
5185            assertEquals(2100, score);
5186
5187            // Test the held score
```

```
5188            score = game.calculateScore(numbers, true);
5189            assertEquals(2100, score);
5190
5191            // Test the calculateHighestScore method
5192            highestScore = game.calculateHighestScore(numbers);
5193            assertEquals(2100, highestScore[0]);
5194            highestRoll = (List<Integer>) highestScore[1];
5195            assertEquals(highestRoll.size(), 6);
5196            assertTrue(Collections.frequency(highestRoll, 1) == 4);
5197            assertTrue(Collections.frequency(highestRoll, 5) == 2);
5198        }
5199
5200        /*
5201         * Test 153: Check scoring for a die roll of AAAAAX (A and X each
5202         * represent all die values other than 1 or 5) Requirement 6.5.0
5203         */
5204
5205        // Generate all permutations of the roll
5206        automatedPermutations = integerPermutations("AAAAAX");
5207
5208        // Convert the list of Strings to a list of lists of integers
5209        for (String numbers : automatedPermutations) {
5210            myPermutations = permutations(numbers);
5211
5212            // Get the die value for calculating the score
5213            dieOneValue = Character.getNumericValue(numbers.charAt(0));
5214
5215            // Check each list of integers for proper scoring
5216            for (List<Integer> currentNumbers : myPermutations) {
5217                // Test the total score
5218                score = game.calculateScore(currentNumbers, false);
5219                assertEquals(100 * dieOneValue * 2 * 2, score);
5220
5221                // Test the held score
5222                score = game.calculateScore(currentNumbers, true);
5223                assertEquals(0, score);
5224
5225                // Test the calculateHighestScore method
5226                highestScore = game.calculateHighestScore(currentNumbers);
5227                assertEquals(100 * dieOneValue * 2 * 2, highestScore[0]);
5228                highestRoll = (List<Integer>) highestScore[1];
5229                assertEquals(highestRoll.size(), 5);
5230                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 5);
5231            }
5232        }
5233
5234        /*
5235         * Test 154: Check scoring for a die roll of 5AAAAA (A represents all
5236         * die values other than 1 or 5) Requirement 6.2.0 and 6.5.0
5237         */
5238
5239        // Generate all permutations of the roll
5240        automatedPermutations = integerPermutations("5AAAAA");
5241
5242        // Convert the list of Strings to a list of lists of integers
5243        for (String numbers : automatedPermutations) {
5244            myPermutations = permutations(numbers);
```

```
5245
5246            // Get the die value for calculating the score
5247            dieOneValue = Character.getNumericValue(numbers.charAt(1));
5248
5249            // Check each list of integers for proper scoring
5250            for (List<Integer> currentNumbers : myPermutations) {
5251                // Test the total score
5252                score = game.calculateScore(currentNumbers, false);
5253                assertEquals(100 * dieOneValue * 2 * 2 + 50, score);
5254
5255                // Test the held score
5256                score = game.calculateScore(currentNumbers, true);
5257                assertEquals(100 * dieOneValue * 2 * 2 + 50, score);
5258
5259                // Test the calculateHighestScore method
5260                highestScore = game.calculateHighestScore(currentNumbers);
5261                assertEquals(100 * dieOneValue * 2 * 2 + 50, highestScore[0]);
5262                highestRoll = (List<Integer>) highestScore[1];
5263                assertEquals(highestRoll.size(), 6);
5264                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 5);
5265                assertTrue(Collections.frequency(highestRoll, 5) == 1);
5266            }
5267        }
5268
5269        /*
5270         * Test 155: Check scoring for a die roll of 1AAAAA (A represents all
5271         * die values other than 1 or 5) Requirement 6.1.0 and 6.5.0
5272         */
5273
5274        // Generate all permutations of the roll
5275        automatedPermutations = integerPermutations("1AAAAA");
5276
5277        // Convert the list of Strings to a list of lists of integers
5278        for (String numbers : automatedPermutations) {
5279            myPermutations = permutations(numbers);
5280
5281            // Get the die value for calculating the score
5282            dieOneValue = Character.getNumericValue(numbers.charAt(1));
5283
5284            // Check each list of integers for proper scoring
5285            for (List<Integer> currentNumbers : myPermutations) {
5286                // Test the total score
5287                score = game.calculateScore(currentNumbers, false);
5288                assertEquals(100 * dieOneValue * 2 * 2 + 100, score);
5289
5290                // Test the held score
5291                score = game.calculateScore(currentNumbers, true);
5292                assertEquals(100 * dieOneValue * 2 * 2 + 100, score);
5293
5294                // Test the calculateHighestScore method
5295                highestScore = game.calculateHighestScore(currentNumbers);
5296                assertEquals(100 * dieOneValue * 2 * 2 + 100, highestScore[0]);
5297                highestRoll = (List<Integer>) highestScore[1];
5298                assertEquals(highestRoll.size(), 6);
5299                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 5);
5300                assertTrue(Collections.frequency(highestRoll, 1) == 1);
5301            }
```

```
5302        }
5303
5304        /*
5305         * Test 156: Check scoring for a die roll of 55555A (A represents all
5306         * die values other than 1 or 5) Requirement 6.5.0
5307         */
5308
5309        // Generate all permutations of the roll
5310        automatedPermutations = integerPermutations("55555A");
5311
5312        // Convert the list of Strings to a list of lists of integers
5313        for (String numbers : automatedPermutations) {
5314            myPermutations = permutations(numbers);
5315
5316            // Check each list of integers for proper scoring
5317            for (List<Integer> currentNumbers : myPermutations) {
5318                // Test the total score
5319                score = game.calculateScore(currentNumbers, false);
5320                assertEquals(2000, score);
5321
5322                // Test the held score
5323                score = game.calculateScore(currentNumbers, true);
5324                assertEquals(0, score);
5325
5326                // Test the calculateHighestScore method
5327                highestScore = game.calculateHighestScore(currentNumbers);
5328                assertEquals(2000, highestScore[0]);
5329                highestRoll = (List<Integer>) highestScore[1];
5330                assertEquals(highestRoll.size(), 5);
5331                assertTrue(Collections.frequency(highestRoll, 5) == 5);
5332            }
5333        }
5334
5335        /*
5336         * Test 157: Check scoring for a die roll of 155555 Requirement 6.1.0
5337         * and 6.5.0
5338         */
5339
5340        // Generate all permutations of the roll and store in a list of lists of
5341        // integers
5342        myPermutations = permutations("155555");
5343
5344        // Check each list of integers for correct scoring
5345        for (List<Integer> numbers : myPermutations) {
5346            // Test the total score
5347            score = game.calculateScore(numbers, false);
5348            assertEquals(2100, score);
5349
5350            // Test the held score
5351            score = game.calculateScore(numbers, true);
5352            assertEquals(2100, score);
5353
5354            // Test the calculateHighestScore method
5355            highestScore = game.calculateHighestScore(numbers);
5356            assertEquals(2100, highestScore[0]);
5357            highestRoll = (List<Integer>) highestScore[1];
5358            assertEquals(highestRoll.size(), 6);
```

```
5359            assertTrue(Collections.frequency(highestRoll, 5) == 5);
5360            assertTrue(Collections.frequency(highestRoll, 1) == 1);
5361        }
5362
5363        /*
5364         * Test 158: Check scoring for a die roll of 11111A (A represents all
5365         * die values other than 1 or 5) Requirement 6.5.0
5366         */
5367
5368        // Generate all permutations of the roll
5369        automatedPermutations = integerPermutations("11111A");
5370
5371        // Convert the list of Strings to a list of lists of integers
5372        for (String numbers : automatedPermutations) {
5373            myPermutations = permutations(numbers);
5374
5375            // Check each list of integers for proper scoring
5376            for (List<Integer> currentNumbers : myPermutations) {
5377                // Test the total score
5378                score = game.calculateScore(currentNumbers, false);
5379                assertEquals(4000, score);
5380
5381                // Test the held score
5382                score = game.calculateScore(currentNumbers, true);
5383                assertEquals(0, score);
5384
5385                // Test the calculateHighestScore method
5386                highestScore = game.calculateHighestScore(currentNumbers);
5387                assertEquals(4000, highestScore[0]);
5388                highestRoll = (List<Integer>) highestScore[1];
5389                assertEquals(highestRoll.size(), 5);
5390                assertTrue(Collections.frequency(highestRoll, 1) == 5);
5391            }
5392        }
5393
5394        /*
5395         * Test 159: Check scoring for a die roll of 111115 Requirement 6.2.0
5396         * and 6.5.0
5397         */
5398
5399        // Generate all permutations of the roll and store in a list of lists of
5400        // integers
5401        myPermutations = permutations("111115");
5402
5403        // Check each list of integers for correct scoring
5404        for (List<Integer> numbers : myPermutations) {
5405            // Test the total score
5406            score = game.calculateScore(numbers, false);
5407            assertEquals(4050, score);
5408
5409            // Test the held score
5410            score = game.calculateScore(numbers, true);
5411            assertEquals(4050, score);
5412
5413            // Test the calculateHighestScore method
5414            highestScore = game.calculateHighestScore(numbers);
5415            assertEquals(4050, highestScore[0]);
```

```
5416            highestRoll = (List<Integer>) highestScore[1];
5417            assertEquals(highestRoll.size(), 6);
5418            assertTrue(Collections.frequency(highestRoll, 5) == 1);
5419            assertTrue(Collections.frequency(highestRoll, 1) == 5);
5420        }
5421
5422        /*
5423         * Test 160: Check scoring for a die roll of AAAAAA (A represents all
5424         * die values other than 1 or 5) Requirement 6.5.0
5425         */
5426
5427        // Generate all permutations of the roll
5428        automatedPermutations = integerPermutations("AAAAAA");
5429
5430        // Convert the list of Strings to a list of lists of integers
5431        for (String numbers : automatedPermutations) {
5432            myPermutations = permutations(numbers);
5433
5434            // Get the die value for calculating the score
5435            dieOneValue = Character.getNumericValue(numbers.charAt(1));
5436
5437            // Check each list of integers for proper scoring
5438            for (List<Integer> currentNumbers : myPermutations) {
5439                // Test the total score
5440                score = game.calculateScore(currentNumbers, false);
5441                assertEquals(100 * dieOneValue * 2 * 2 * 2, score);
5442
5443                // Test the held score
5444                score = game.calculateScore(currentNumbers, true);
5445                assertEquals(100 * dieOneValue * 2 * 2 * 2, score);
5446
5447                // Test the calculateHighestScore method
5448                highestScore = game.calculateHighestScore(currentNumbers);
5449                assertEquals(100 * dieOneValue * 2 * 2 * 2, highestScore[0]);
5450                highestRoll = (List<Integer>) highestScore[1];
5451                assertEquals(highestRoll.size(), 6);
5452                assertTrue(Collections.frequency(highestRoll, dieOneValue) == 6);
5453            }
5454        }
5455
5456        /*
5457         * Test 161: Check scoring for a die roll of 555555 Requirement 6.5.0
5458         */
5459
5460        // Generate all permutations of the roll and store in a list of lists of
5461        // integers
5462        myPermutations = permutations("555555");
5463
5464        // Check each list of integers for correct scoring
5465        for (List<Integer> numbers : myPermutations) {
5466            // Test the total score
5467            score = game.calculateScore(numbers, false);
5468            assertEquals(4000, score);
5469
5470            // Test the held score
5471            score = game.calculateScore(numbers, true);
5472            assertEquals(4000, score);
```

```
5473
5474            // Test the calculateHighestScore method
5475            highestScore = game.calculateHighestScore(numbers);
5476            assertEquals(4000, highestScore[0]);
5477            highestRoll = (List<Integer>) highestScore[1];
5478            assertEquals(highestRoll.size(), 6);
5479            assertTrue(Collections.frequency(highestRoll, 5) == 6);
5480        }
5481
5482        /*
5483         * Test 162: Check scoring for a die roll of 111111 Requirement 6.5.0
5484         */
5485
5486        // Generate all permutations of the roll and store in a list of lists of
5487        // integers
5488        myPermutations = permutations("111111");
5489
5490        // Check each list of integers for correct scoring
5491        for (List<Integer> numbers : myPermutations) {
5492            // Test the total score
5493            score = game.calculateScore(numbers, false);
5494            assertEquals(8000, score);
5495
5496            // Test the held score
5497            score = game.calculateScore(numbers, true);
5498            assertEquals(8000, score);
5499
5500            // Test the calculateHighestScore method
5501            highestScore = game.calculateHighestScore(numbers);
5502            assertEquals(8000, highestScore[0]);
5503            highestRoll = (List<Integer>) highestScore[1];
5504            assertEquals(highestRoll.size(), 6);
5505            assertTrue(Collections.frequency(highestRoll, 1) == 6);
5506        }
5507
5508    }
5509
5510    /* The following are helper methods used for retrieving roll permutations */
5511
5512    /**
5513     * The permutations method is a wrapper class to the stringPermutations
5514     * method. This method passes a string of integers for which the
5515     * permutations are sought to the stringPermutations method that then
5516     * returns a list of all the string permutations of the input string. The
5517     * permutations method than converts the list of strings to a list of lists
5518     * of integers.
5519     *
5520     * @param myString
5521     *            string of integers for which all permutations are sought
5522     * @return List of Lists of integers representing distinct permutations of
5523     *         the input string
5524     */
5525    public static List<List<Integer>> permutations(String myString) {
5526
5527        // If myString is null, return null
5528        if (myString == null) {
5529            return null;
```

```
5530        }
5531
5532        // If any character in myString is not a digit, return null
5533        int length = myString.length();
5534
5535        // For each character in myString
5536        for (int i = 0; i < length; i++) {
5537            if (!Character.isDigit(myString.charAt(i))) {
5538                return null;
5539            }
5540        }
5541
5542        // Retrieve a list of all distinct permutations of the string
5543        List<String> myList = stringPermutations(myString);
5544
5545        // Convert the list of strings to a list of lists of integers
5546        List<List<Integer>> myIntegers = new LinkedList<List<Integer>>();
5547
5548        // For each string in myList, convert that string to a list of integers
5549        // and add
5550        // the resulting list to the myIntegers list
5551        for (String number : myList) {
5552            List<Integer> currentPermutation = new LinkedList<Integer>();
5553            for (int i = 0; i < number.length(); i++) {
5554                currentPermutation.add(Character.getNumericValue(number
5555                        .charAt(i)));
5556            }
5557            myIntegers.add(currentPermutation);
5558        }
5559
5560        return myIntegers;
5561    }
5562
5563    /**
5564     * The stringPermutation method uses recursion to find all permutations of a
5565     * string of characters, returning a list of those strings
5566     *
5567     * @param myString
5568     *            string of characters for which all permutations are to be
5569     *            found
5570     * @return List of strings representing all distinct permutations of the
5571     *         input string
5572     */
5573    public static List<String> stringPermutations(String myString) {
5574
5575        // If myString is null, return null
5576        if (myString == null) {
5577            return null;
5578        }
5579
5580        // Instantiate the permutationsList as a Linked List of strings
5581        List<String> permutationsList = new LinkedList<String>();
5582
5583        int length = myString.length();
5584
5585        // If the length of myString is less than 2, there are no more
5586        // permutations to perform
```

```
5587        // so add myString to the permutationsList
5588        if (length < 2) {
5589            permutationsList.add(myString);
5590            return permutationsList;
5591        }
5592
5593        char currentChar;
5594
5595        // For each character in myString
5596        for (int i = 0; i < length; i++) {
5597            currentChar = myString.charAt(i);
5598
5599            // Retrieve the substring excluding the current character
5600            String subString = myString.substring(0, i)
5601                    + myString.substring(i + 1);
5602
5603            // Retrieve all of the permutations for a substring excluding the
5604            // current character
5605            List<String> subPermutations = stringPermutations(subString);
5606
5607            // For each permutation of the substring, add the string that
5608            // includes the current
5609            // character plus the substring to the permutations list.
5610            for (String item : subPermutations) {
5611                if (!permutationsList.contains(currentChar + item))
5612                    permutationsList.add(currentChar + item);
5613            }
5614        }
5615
5616        return permutationsList;
5617    }
5618
5619    /**
5620     * The integerPermutations method converts a string, representing a Farkle
5621     * dice roll, into a list of strings including all roll permutations for the
5622     * input string.
5623     *
5624     * @param myString
5625     *            input string to be converted, with variables as placeholders
5626     *            for dice that could be 2,3,4 or 6. E.g. the string "1AABC"
5627     *            represents a roll including a die with the value 1, and 4 dice
5628     *            with the value 2 through 6. For each permuation, 2 of the 4
5629     *            dice, represented by A, are of the same value. On a given
5630     *            roll, the value for die B never equals that for A or C, and
5631     *            the value of die C never equals that for A or B.
5632     * @return List of strings with all permutations of 2, 3, 4, or 6 valued die
5633     *         in place of variables in the input string. E.g. an input string
5634     *         of "1A" would return the list {"12", "13", "14", "16"}
5635     */
5636    public static List<String> integerPermutations(String myString) {
5637
5638        if (myString == null) {
5639            return null;
5640        }
5641
5642        // The number of distinct variables in the input string
5643        int numberOfVariables;
```

```
5644
5645        // List of present variables in the input string
5646        List<Character> presentVariables = new LinkedList<Character>();
5647
5648        // List of strings representing the roll permutations calculated from
5649        // the input string
5650        List<String> rollPermutations = new LinkedList<String>();
5651
5652        // The current character of the input string
5653        char currentCharacter;
5654
5655        // The length of the input string
5656        int length = myString.length();
5657
5658        // The current permutation that will be added to the list
5659        String currentPermutation = "";
5660
5661        // Retrieve the different variables in myString
5662        for (int i = 0; i < length; i++) {
5663            currentCharacter = myString.charAt(i);
5664
5665            // If the current character is a letter, and the variables list is
5666            // empty or the variables list does not contain the letter
5667            // Add it to the variables list
5668            if (Character.isLetter(currentCharacter)
5669                    && (presentVariables.isEmpty() || !presentVariables
5670                            .contains(currentCharacter))) {
5671                presentVariables.add(currentCharacter);
5672            }
5673        }
5674
5675        // Determine the number of distinct variables present in myString
5676        numberOfVariables = presentVariables.size();
5677
5678        // There can be a maximum of 4 variables (representing 2, 3, 4, 6),
5679        // Depending on the number of variables present in the input string, add
5680        // the string representing each possible permutation to the
5681        // rollPermutations
5682        // list
5683        if (numberOfVariables > 0 && numberOfVariables <= 4) {
5684
5685            // Outer loop for 1 die
5686            for (int i = 2; i <= 6; i++) {
5687                if (i == 5)
5688                    continue;
5689                // currentPermutation += i;
5690
5691                // Embedded loop for a variable representing a second die
5692                if (numberOfVariables > 1) {
5693                    for (int j = 2; j <= 6; j++) {
5694                        if (j == i || j == 5)
5695                            continue;
5696                        // currentPermutation += j;
5697
5698                        // Embedded loop for a variable representing a third die
5699                        if (numberOfVariables > 2) {
5700                            for (int k = 2; k <= 6; k++) {
```

```
5701                    if (k == i || k == j || k == 5)
5702                        continue;
5703                    // currentPermutation += k;
5704
5705                    // Embedded loop for a variable representing a
5706                    // fourth die
5707                    if (numberOfVariables > 3) {
5708                        for (int l = 2; l <= 6; l++) {
5709                            if (l == k || l == j || l == i
5710                                    || l == 5)
5711                                continue;
5712                            // Clear the currentPermutation String
5713                            currentPermutation = "";
5714
5715                            // Loop through the input string adding
5716                            // this permutation of the roll.
5717                            for (int x = 0; x < length; x++) {
5718                                // Clear the currentPermutation
5719                                // String
5720                                currentCharacter = myString
5721                                        .charAt(x);
5722
5723                                // If the current character is a
5724                                // letter, and the variables list is
5725                                // empty
5726                                // or the variables list does not
5727                                // contain the letter, add it to the
5728                                // variables
5729                                // list
5730                                if (Character
5731                                        .isLetter(currentCharacter)) {
5732                                    switch (presentVariables
5733                                            .indexOf(currentCharacter)) {
5734                                    case 0:
5735                                        currentPermutation += i;
5736                                        break;
5737                                    case 1:
5738                                        currentPermutation += j;
5739                                        break;
5740                                    case 2:
5741                                        currentPermutation += k;
5742                                        break;
5743                                    case 3:
5744                                        currentPermutation += l;
5745                                        break;
5746                                    }
5747                                } else {
5748                                    currentPermutation += currentCharacter;
5749                                }
5750                            }
5751                            rollPermutations
5752                                    .add(currentPermutation);
5753                        }
5754                    } else {
5755                        // Clear the currentPermutation String
5756                        currentPermutation = "";
5757
```

```
5758                            // Loop through the input string adding this
5759                            // permutation of the roll.
5760                            for (int x = 0; x < length; x++) {
5761                                currentCharacter = myString.charAt(x);
5762
5763                                // If the current character is a letter,
5764                                // and the variables list is empty
5765                                // or the variables list does not
5766                                // contain the letter, add it to the
5767                                // variables
5768                                // list
5769                                if (Character
5770                                        .isLetter(currentCharacter)) {
5771                                    switch (presentVariables
5772                                            .indexOf(currentCharacter)) {
5773                                    case 0:
5774                                        currentPermutation += i;
5775                                        break;
5776                                    case 1:
5777                                        currentPermutation += j;
5778                                        break;
5779                                    case 2:
5780                                        currentPermutation += k;
5781                                        break;
5782                                    }
5783                                } else {
5784                                    currentPermutation += currentCharacter;
5785                                }
5786                            }
5787                            rollPermutations.add(currentPermutation);
5788                        }
5789                    }
5790                } else {
5791                    // Clear the currentPermutation String
5792                    currentPermutation = "";
5793
5794                    // Loop through the input string adding this
5795                    // permutation of the roll.
5796                    for (int x = 0; x < length; x++) {
5797
5798                        currentCharacter = myString.charAt(x);
5799
5800                        // If the current character is a letter, and the
5801                        // variables list is empty
5802                        // or the variables list does not contain the
5803                        // letter, add it to the variables
5804                        // list
5805                        if (Character.isLetter(currentCharacter)) {
5806                            switch (presentVariables
5807                                    .indexOf(currentCharacter)) {
5808                            case 0:
5809                                currentPermutation += i;
5810                                break;
5811                            case 1:
5812                                currentPermutation += j;
5813                                break;
5814                            }
```

```
5815                            } else {
5816                                currentPermutation += currentCharacter;
5817                            }
5818                        }
5819                        rollPermutations.add(currentPermutation);
5820                    }
5821                }
5822            } else {
5823                // Clear the currentPermutation String
5824                currentPermutation = "";
5825
5826                // Loop through the input string adding this permutation of
5827                // the roll.
5828                for (int x = 0; x < length; x++) {
5829                    currentCharacter = myString.charAt(x);
5830
5831                    // If the current character is a letter, and the
5832                    // variables list is empty
5833                    // or the variables list does not contain the letter,
5834                    // add it to the variables
5835                    // list
5836                    if (Character.isLetter(currentCharacter)) {
5837                        currentPermutation += i;
5838
5839                    } else {
5840                        currentPermutation += currentCharacter;
5841                    }
5842                }
5843                rollPermutations.add(currentPermutation);
5844            }
5845        }
5846    }
5847
5848    // If more than 4 variables were in the string it is incorrect input,
5849    // return null
5850    else if (numberOfVariables > 4) {
5851        return null;
5852    } else {
5853        // There are no variables present to perform the permutation on,
5854        // simply add myString
5855        // to the rollPermutations
5856        rollPermutations.add(myString);
5857    }
5858
5859    return rollPermutations;
5860  }
5861}
5862
```

```java
package com.lotsofun.farkle;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

/**
 * Tests the Game Class (except the calculateHighestScore(List<Integer>) and the
 * calculateScore(List<Integer>, boolean) methods
 *
 * @author Jacob Davidson
 * @version 3.0.0
 */
public class GameTest {

    /** The FarkleController object used for testing */
    FarkleController farkleController;

    /** The Single Player Game object used for testing */
    Game singlePlayerGame;

    /** The Multi Player Game object used for testing */
    Game multiPlayerGame;

    /**
     * Instantiate all objects before testing begins
     *
     * @throws Exception
     */
    @Before
    public void setUp() throws Exception {

        // Instantiate the farklecontroller
        farkleController = new FarkleController(true);

        // Instantiate the single player game object
        singlePlayerGame = new Game(GameMode.SINGLEPLAYER, farkleController);

        // Instantiate the single player game object
        multiPlayerGame = new Game(GameMode.MULTIPLAYER, farkleController);

    }

    /**
     * Test the Game constructor
     */
    @Test
    public void testGameConstructor() {

        // Assert that the farkleController is not null
        assertNotNull(farkleController);

        // Test the single player mode game constructor
        assertNotNull(singlePlayerGame);
        assertTrue(singlePlayerGame.getPlayers()[0].equals(singlePlayerGame
                .getCurrentPlayer()));
        assertEquals(1, singlePlayerGame.getTurnNumberForCurrentPlayer());
```

```
58          assertEquals(0, singlePlayerGame.getGameScoreForPlayer(1));
59
60          // Test the multi player mode game constructor
61          assertNotNull(multiPlayerGame);
62          assertTrue(multiPlayerGame.getPlayers()[0].equals(multiPlayerGame
63                  .getCurrentPlayer()));
64          assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
65          assertEquals(0, multiPlayerGame.getGameScoreForPlayer(1));
66          assertEquals(1, multiPlayerGame.getPlayers()[1].getTurnNumber());
67          assertEquals(0, multiPlayerGame.getGameScoreForPlayer(2));
68
69      }
70
71      /**************************************************************************
72       * 4.8.0 - If the player selects the bank button, the current turn point
73       * total is added to the player's game point total, and the turn is over.
74       **************************************************************************/
75      /**
76       * Test the bank() method
77       */
78      @Test
79      public void testBank() {
80
81          // Test bank for singlePlayerGame by setting the game score and ensuring
82          // the bank method
83          // returns that score.
84          singlePlayerGame.getCurrentPlayer().setGameScore(500);
85          assertEquals(500, singlePlayerGame.bank());
86
87          // Test bank for multiPlayerGame, set the game score for player 1 and
88          // ensure the bank method
89          // returns that score.
90          multiPlayerGame.getCurrentPlayer().setGameScore(100);
91          assertEquals(100, multiPlayerGame.bank());
92
93          // Test to check the current player is player two after the first call
94          // to bank()
95          assertTrue(multiPlayerGame.getPlayers()[1].equals(multiPlayerGame
96                  .getCurrentPlayer()));
97
98          // Set the game score for the current player to 1000
99          multiPlayerGame.getCurrentPlayer().setGameScore(1000);
100
101         // Test to ensure the bank method returns 1000 for the current player
102         // (player 2)
103         assertEquals(1000, multiPlayerGame.bank());
104
105         // Test to make sure the next bank method call returns the game score
106         // for player 1
107         assertEquals(100, multiPlayerGame.bank());
108
109     }
110
111     /**************************************************************************
112      * 4.2.0 - The resulting roll is analyzed according to requirement 6.0.0 to
113      * determine if the player farkled. A farkle occurs if the roll results in 0
114      * points.
```

```
115      ************************************************************************/
116     /**
117      * Test the farkle() method
118      */
119     @Test
120     public void testFarkle() {
121
122         // Test farkle for singlePlayerGame by adding a roll score, and testing
123         // to make sure
124         // that the roll scare is not added to the game score after a farkle
125         singlePlayerGame.getCurrentPlayer().setGameScore(1000);
126         singlePlayerGame.getCurrentPlayer().scoreRoll(500);
127         singlePlayerGame.farkle();
128         assertEquals(1000, singlePlayerGame.getGameScoreForCurrentPlayer());
129
130         // Test farkle for multiPlayerGame. set the game score to 100 and the
131         // roll score to 500
132         // for the current player
133         multiPlayerGame.getCurrentPlayer().setGameScore(100);
134         multiPlayerGame.getCurrentPlayer().scoreRoll(500);
135
136         // Upon a farkle, the current player should be player 2 and the roll
137         // score should not
138         // have been added to the game score for player 1
139         multiPlayerGame.farkle();
140
141         // Test to check the current player is player two after the first call
142         // to farkle()
143         assertTrue(multiPlayerGame.getPlayers()[1].equals(multiPlayerGame
144                 .getCurrentPlayer()));
145
146         // Test to make sure the roll score wasn't added to player 1's game
147         // score
148         assertEquals(100, multiPlayerGame.getPlayers()[0].getGameScore());
149
150         // Set the game score for the current player to 1000, and the roll score
151         // to 500
152         multiPlayerGame.getCurrentPlayer().setGameScore(1000);
153         multiPlayerGame.getCurrentPlayer().scoreRoll(500);
154
155         // Upon a farkle, the current player should be player 1 and the roll
156         // score should not
157         // have been added to the game score for player 2.
158         multiPlayerGame.farkle();
159
160         // Test to check the current player is player one after the second call
161         // to farkle()
162         assertTrue(multiPlayerGame.getPlayers()[0].equals(multiPlayerGame
163                 .getCurrentPlayer()));
164
165         // Test to make sure the roll score wasn't added to player 1's game
166         // score
167         assertEquals(1000, multiPlayerGame.getPlayers()[1].getGameScore());
168
169     }
170
171     /**
```

```
172      * Test the setCurrentPlayer(int) and getCurrentPlayer() method
173      */
174     @Test
175     public void testSetAndGetCurrentPlayer() {
176
177         // Check if player 1 is the current player for singlePlayerGame
178         assertTrue(singlePlayerGame.getCurrentPlayer().equals(
179                 singlePlayerGame.getPlayers()[0]));
180
181         // Check if player 1 is the current player for multiplayergame
182         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
183                 multiPlayerGame.getPlayers()[0]));
184
185         // Change the multiplayergame current player to player 2
186         multiPlayerGame.setCurrentPlayer(2);
187
188         // Check if player 2 is the current player for multiplayergame
189         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
190                 multiPlayerGame.getPlayers()[1]));
191
192     }
193
194     /**
195      * Test the getGameMode() method
196      */
197     @Test
198     public void testGetAndSetGameMode() {
199
200         // Test with the singlePlayerGame
201         assertTrue(singlePlayerGame.getGameMode() == GameMode.SINGLEPLAYER);
202
203         // Test with multiPlayerGame
204         assertTrue(multiPlayerGame.getGameMode() == GameMode.MULTIPLAYER);
205
206     }
207
208     /**
209      * Test the getGameScoreForCurrentPlayer() method
210      */
211     @Test
212     public void testGetGameScoreForCurrentPlayer() {
213
214         // Test that the score is 0 at the beginning of the single player game
215         assertTrue(singlePlayerGame.getGameScoreForCurrentPlayer() == 0);
216
217         // Change the player score and test again
218         singlePlayerGame.getCurrentPlayer().setGameScore(1000);
219         assertTrue(singlePlayerGame.getGameScoreForCurrentPlayer() == 1000);
220
221         // Test that the game score is 0 for both players at the beginning of a
222         // multiPlayer game
223         assertTrue(multiPlayerGame.getGameScoreForCurrentPlayer() == 0
224                 && multiPlayerGame.getCurrentPlayer().equals(
225                         multiPlayerGame.getPlayers()[0]));
226         multiPlayerGame.setCurrentPlayer(2);
227         assertTrue(multiPlayerGame.getGameScoreForCurrentPlayer() == 0
228                 && multiPlayerGame.getCurrentPlayer().equals(
```

```
229                     multiPlayerGame.getPlayers()[1]));
230         multiPlayerGame.setCurrentPlayer(1);
231 
232         // Change each player score and test again
233         multiPlayerGame.getCurrentPlayer().setGameScore(500);
234         assertTrue(multiPlayerGame.getGameScoreForCurrentPlayer() == 500
235                 && multiPlayerGame.getCurrentPlayer().equals(
236                         multiPlayerGame.getPlayers()[0]));
237         multiPlayerGame.setCurrentPlayer(2);
238         multiPlayerGame.getCurrentPlayer().setGameScore(100);
239         assertTrue(multiPlayerGame.getGameScoreForCurrentPlayer() == 100
240                 && multiPlayerGame.getCurrentPlayer().equals(
241                         multiPlayerGame.getPlayers()[1]));
242 
243     }
244 
245     /**
246      * Test the getGameScoreForPlayer(int) method
247      */
248     @Test
249     public void testGetGameScoreForPlayer() {
250 
251         // Test for single player game, make sure the game score for player 1 is
252         // 0 at the start
253         assertEquals(0, singlePlayerGame.getGameScoreForPlayer(1));
254 
255         // Set the game score to an integer, and ensure the
256         // getGameScoreForPlayer returns that integer
257         singlePlayerGame.getCurrentPlayer().setGameScore(500);
258         assertEquals(500, singlePlayerGame.getGameScoreForPlayer(1));
259 
260         // Make sure player 2 is null for single player game
261         assertNull(singlePlayerGame.getPlayers()[1]);
262 
263         // Test for multi player game, make sure the game score is initially 0
264         // for both players
265         assertEquals(0, multiPlayerGame.getGameScoreForPlayer(1));
266         assertEquals(0, multiPlayerGame.getGameScoreForPlayer(2));
267 
268         // Set the game score to an integer for each player, and ensure the
269         // getGameScoreForPlayer
270         // returns that integer for both
271         multiPlayerGame.getPlayers()[0].setGameScore(1000);
272         multiPlayerGame.getPlayers()[1].setGameScore(700);
273         assertEquals(1000, multiPlayerGame.getGameScoreForPlayer(1));
274         assertEquals(700, multiPlayerGame.getGameScoreForPlayer(2));
275 
276     }
277 
278     /**
279      * Test the getHighScore() and setHighScore(int) methods
280      */
281     @Test
282     public void testGetAndSetHighScore() {
283 
284         // Set the high score, and then get it to make sure it matches. This is
285         // a Preference, so it will be the same regardless of setting and
```

```
286         // getting
287         // it through the single player game or multi player game
288         singlePlayerGame.setHighScore(5000);
289         assertEquals(5000, singlePlayerGame.getHighScore());
290         assertEquals(5000, multiPlayerGame.getHighScore());
291 
292         // Because we can't rely on the JUnits to necessarily fire in
293         // the order in which they're written, this preference needs
294         // to be reset to prevent it from persisting in the build environment.
295         singlePlayerGame.setHighScore(0);
296         assertEquals(0, singlePlayerGame.getHighScore());
297         assertEquals(0, multiPlayerGame.getHighScore());
298 
299     }
300 
301     /**
302      * Test the getNextPlayer() method
303      */
304     @Test
305     public void testGetNextPlayer() {
306 
307         // In single player mode, this should always return 1
308         assertEquals(1, singlePlayerGame.getNextPlayer());
309 
310         // Farkle will end the current players turn, and set the next player
311         singlePlayerGame.farkle();
312 
313         // In single player mode, getNextPlayer() should still result in 1
314         assertEquals(1, singlePlayerGame.getNextPlayer());
315 
316         // In multi player mode, this will be one or two. At the start of the
317         // game
318         // the current player is player 1 so getNextPlayer should return 2
319         assertEquals(2, multiPlayerGame.getNextPlayer());
320 
321         // Farkle will end the current players turn, and set the next player
322         multiPlayerGame.farkle();
323 
324         // In multi player mode, getNextPlayer() should now result in 1
325         assertEquals(1, multiPlayerGame.getNextPlayer());
326 
327         // Bank will also end the current players turn, and set the next player
328         multiPlayerGame.bank();
329 
330         // In multi player mode, getNextPlayer() should now result in 2 again
331         assertEquals(2, multiPlayerGame.getNextPlayer());
332 
333     }
334 
335     /**
336      * Test the getPlayerName(int) and setPlayerName(int, String) methods
337      */
338     @Test
339     public void testGetAndSetPlayerName() {
340 
341         // Test the single player game. The player's name should initially be
342         // null
```

```
343         assertNull(singlePlayerGame.getPlayerName(1));
344 
345         // Assigning null to the player's name should result in a name of null
346         singlePlayerGame.setPlayerName(1, null);
347         assertNull(singlePlayerGame.getPlayerName(1));
348 
349         // Assigning any string to the player's name should result in that
350         // string
351         singlePlayerGame.setPlayerName(1, "Jake");
352         assertTrue(singlePlayerGame.getPlayerName(1).equals("Jake"));
353 
354         // Test the multi player game. The player's names should initially be
355         // null
356         assertNull(multiPlayerGame.getPlayerName(1));
357         assertNull(multiPlayerGame.getPlayerName(2));
358 
359         // Assigning null to the player's name should result in a name of null
360         multiPlayerGame.setPlayerName(1, null);
361         multiPlayerGame.setPlayerName(2, null);
362         assertNull(multiPlayerGame.getPlayerName(1));
363         assertNull(multiPlayerGame.getPlayerName(2));
364 
365         // Assigning any string to the player's name should result in that
366         // string
367         multiPlayerGame.setPlayerName(1, "bill");
368         multiPlayerGame.setPlayerName(2, "roger");
369         assertTrue(multiPlayerGame.getPlayerName(1).equals("bill"));
370         assertTrue(multiPlayerGame.getPlayerName(2).equals("roger"));
371 
372     }
373 
374     /**
375      * Test getPlayerNumberForCurrentPlayer() method
376      */
377     @Test
378     public void testGetPlayerNumberForCurrentPlayer() {
379 
380         // The current player number for a single player game should always be 1
381         assertEquals(1, singlePlayerGame.getPlayerNumberForCurrentPlayer());
382 
383         // Same should be true after a farkle or bank
384         singlePlayerGame.farkle();
385         assertEquals(1, singlePlayerGame.getPlayerNumberForCurrentPlayer());
386         singlePlayerGame.bank();
387         assertEquals(1, singlePlayerGame.getPlayerNumberForCurrentPlayer());
388 
389         // The current player number for a multi player game should start as 1
390         assertEquals(1, multiPlayerGame.getPlayerNumberForCurrentPlayer());
391 
392         // After the first farkle it should be 2
393         multiPlayerGame.farkle();
394         assertEquals(2, multiPlayerGame.getPlayerNumberForCurrentPlayer());
395 
396         // After the second farkle it should be 1 again
397         multiPlayerGame.farkle();
398         assertEquals(1, multiPlayerGame.getPlayerNumberForCurrentPlayer());
399 
```

```
400         // After the first bank it should be 2
401         multiPlayerGame.bank();
402         assertEquals(2, multiPlayerGame.getPlayerNumberForCurrentPlayer());
403 
404         // After the second bank it should be 1 again
405         multiPlayerGame.bank();
406         assertEquals(1, multiPlayerGame.getPlayerNumberForCurrentPlayer());
407 
408     }
409 
410     /**
411      * Test the getPlayers() method
412      */
413     @Test
414     public void testGetPlayers() {
415 
416         // Make sure the number of players returned for the single player game
417         // is 1
418         Player[] singlePlayerGamePlayers = singlePlayerGame.getPlayers();
419         int playerCount = 0;
420         for (int i = 0; i < singlePlayerGamePlayers.length; i++) {
421             if (null != singlePlayerGamePlayers[i]) {
422                 playerCount++;
423             }
424         }
425         assertEquals(1, playerCount);
426 
427         // Make sure that player is the current player
428         assertTrue(singlePlayerGamePlayers[0].equals(singlePlayerGame
429                 .getCurrentPlayer()));
430 
431         // Make sure the number of players returned for the multi player game is
432         // 2
433         Player[] multiPlayerGamePlayers = multiPlayerGame.getPlayers();
434         playerCount = 0;
435         for (int i = 0; i < multiPlayerGamePlayers.length; i++) {
436             if (null != multiPlayerGamePlayers[i]) {
437                 playerCount++;
438             }
439         }
440         assertEquals(2, playerCount);
441 
442         // Make sure the current player is the first player in the
443         // multiPlayerGamePlayers array
444         assertTrue(multiPlayerGamePlayers[0].equals(multiPlayerGame
445                 .getCurrentPlayer()));
446 
447         // bank or farkle to change the current player, and check that player
448         // against the array
449         multiPlayerGame.bank();
450         assertTrue(multiPlayerGamePlayers[1].equals(multiPlayerGame
451                 .getCurrentPlayer()));
452 
453     }
454 
455     /**
456      * Test the getPlayerTypeForCurrentPlayer() and setPlayerType(int,
```

```
457      * PlayerType) methods
458      */
459     @Test
460     public void testGetAndSetPlayerType() {
461
462         // Test for single player mode. It should start as PlayerType.USER
463         assertEquals(PlayerType.USER,
464                 singlePlayerGame.getPlayerTypeForCurrentPlayer());
465
466         // Set the player type to computer and check it
467         singlePlayerGame.setPlayerType(1, PlayerType.COMPUTER);
468         assertEquals(PlayerType.COMPUTER,
469                 singlePlayerGame.getPlayerTypeForCurrentPlayer());
470
471         // Test for multi player mode. Both players should start out as
472         // PlayerType.USER
473         assertEquals(PlayerType.USER,
474                 multiPlayerGame.getPlayerTypeForCurrentPlayer());
475         multiPlayerGame.setPlayerType(1, PlayerType.COMPUTER);
476         assertEquals(PlayerType.COMPUTER,
477                 multiPlayerGame.getPlayerTypeForCurrentPlayer());
478
479         // Change the player via a bank() call, and test player 2
480         multiPlayerGame.bank();
481         assertEquals(PlayerType.USER,
482                 multiPlayerGame.getPlayerTypeForCurrentPlayer());
483         multiPlayerGame.setPlayerType(2, PlayerType.COMPUTER);
484         assertEquals(PlayerType.COMPUTER,
485                 multiPlayerGame.getPlayerTypeForCurrentPlayer());
486
487     }
488
489     /**
490      * Test the getRollScores() method
491      */
492     @Test
493     public void testGetRollScores() {
494
495         // Test the single player game. The getRollScores method should return 0
496         // at the start
497         assertEquals(0, singlePlayerGame.getRollScores());
498
499         // Add a few rolls and test again
500         singlePlayerGame.getCurrentPlayer().scoreRoll(100);
501         singlePlayerGame.processRoll();
502         assertEquals(100, singlePlayerGame.getRollScores());
503         singlePlayerGame.getCurrentPlayer().scoreRoll(100);
504         singlePlayerGame.processRoll();
505         assertEquals(200, singlePlayerGame.getRollScores());
506
507         // End the turn by banking and check to make sure it's back to 0
508         singlePlayerGame.bank();
509         assertEquals(0, singlePlayerGame.getRollScores());
510
511         // Test the multi player game. The getRollScores method should return 0
512         // at the start
513         assertEquals(0, multiPlayerGame.getRollScores());
```

```
514
515         // add a few rolls and test again
516         multiPlayerGame.getCurrentPlayer().scoreRoll(100);
517         multiPlayerGame.processRoll();
518         assertEquals(100, multiPlayerGame.getRollScores());
519         multiPlayerGame.getCurrentPlayer().scoreRoll(100);
520         multiPlayerGame.processRoll();
521         assertEquals(200, multiPlayerGame.getRollScores());
522
523         // End the turn by farkle and test the second player
524         multiPlayerGame.farkle();
525
526         // Test the multi player game. The getRollScores method should return 0
527         // at the start
528         assertEquals(0, multiPlayerGame.getRollScores());
529
530         // add a few rolls and test again
531         multiPlayerGame.getCurrentPlayer().scoreRoll(100);
532         multiPlayerGame.processRoll();
533         assertEquals(100, multiPlayerGame.getRollScores());
534         multiPlayerGame.getCurrentPlayer().scoreRoll(100);
535         multiPlayerGame.processRoll();
536         assertEquals(200, multiPlayerGame.getRollScores());
537
538     }
539
540     /**
541      * Test the getTurnNumberForCurrentPlayer() method
542      */
543     @Test
544     public void testGetTurnNumberForCurrentPlayer() {
545
546         // Test for single player game. This should initially be 1
547         assertEquals(1, singlePlayerGame.getTurnNumberForCurrentPlayer());
548
549         // End the turn via bank() or farkle() and test again
550         singlePlayerGame.bank();
551         assertEquals(2, singlePlayerGame.getTurnNumberForCurrentPlayer());
552         singlePlayerGame.farkle();
553         assertEquals(3, singlePlayerGame.getTurnNumberForCurrentPlayer());
554
555         // Test for multiPlayerGame
556         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
557         multiPlayerGame.bank();
558
559         // The bank should change the current player to player 2, current turn
560         // should still be 1
561         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
562
563         // End turn via bank() or farkle() and test again
564         multiPlayerGame.bank();
565         assertEquals(2, multiPlayerGame.getTurnNumberForCurrentPlayer());
566         multiPlayerGame.farkle();
567         assertEquals(2, multiPlayerGame.getTurnNumberForCurrentPlayer());
568         multiPlayerGame.farkle();
569         assertEquals(3, multiPlayerGame.getTurnNumberForCurrentPlayer());
570
```

```
571     }
572
573     /**
574      * Test the getWinningPlayerInfo() method
575      */
576     @Test
577     public void testGetWinningPlayerInfo() {
578
579         // Test for single player game. The name is initially null
580         String[] winner = singlePlayerGame.getWinningPlayerInfo();
581         assertNull(winner[0]);
582         assertTrue(winner[1].equals("0"));
583
584         // Change the name and score for the single player game and test again
585         singlePlayerGame.setPlayerName(1, "Brant");
586         singlePlayerGame.getCurrentPlayer().setGameScore(4100);
587         winner = singlePlayerGame.getWinningPlayerInfo();
588         assertTrue(winner[0].equals("Brant"));
589         assertTrue(winner[1].equals("4100"));
590
591         // Test for each player of the multiPlayerGame. The name's should
592         // initially be null and they should be tied
593         winner = multiPlayerGame.getWinningPlayerInfo();
594         assertNull(winner[0]);
595         assertNull(winner[1]);
596         assertTrue(winner[2].equals("0"));
597
598         // Set names and various scores for testing multiplayer
599         multiPlayerGame.setPlayerName(1, "Curtis");
600         multiPlayerGame.setPlayerName(2, "Jacob");
601
602         // Set scores so player 1 wins
603         multiPlayerGame.getPlayers()[0].setGameScore(1000);
604         multiPlayerGame.getPlayers()[1].setGameScore(500);
605         winner = multiPlayerGame.getWinningPlayerInfo();
606         assertTrue(winner[0].equals("Curtis"));
607         assertTrue(winner[1].equals("1000"));
608
609         // Set scores so player 2 wins
610         multiPlayerGame.getPlayers()[0].setGameScore(500);
611         multiPlayerGame.getPlayers()[1].setGameScore(700);
612         winner = multiPlayerGame.getWinningPlayerInfo();
613         assertTrue(winner[0].equals("Jacob"));
614         assertTrue(winner[1].equals("700"));
615
616         // Set scores so players tie
617         multiPlayerGame.getPlayers()[0].setGameScore(500);
618         multiPlayerGame.getPlayers()[1].setGameScore(500);
619         winner = multiPlayerGame.getWinningPlayerInfo();
620         assertTrue(winner[0].equals("Curtis"));
621         assertTrue(winner[1].equals("Jacob"));
622         assertTrue(winner[2].equals("500"));
623
624     }
625
626     /**
627      * Test the isBonusTurn() and setBonusTurn(boolean) methods
```

```
628      */
629     @Test
630     public void testIsAndSetBonusTurn() {
631
632         // test single player game
633         assertFalse(singlePlayerGame.isBonusTurn());
634         singlePlayerGame.setBonusTurn(true);
635         assertTrue(singlePlayerGame.isBonusTurn());
636         singlePlayerGame.setBonusTurn(false);
637         assertFalse(singlePlayerGame.isBonusTurn());
638
639         // Test multi player game
640         assertFalse(multiPlayerGame.isBonusTurn());
641         multiPlayerGame.setBonusTurn(true);
642         assertTrue(multiPlayerGame.isBonusTurn());
643         multiPlayerGame.setBonusTurn(false);
644         assertFalse(multiPlayerGame.isBonusTurn());
645
646     }
647
648     /**
649      * Test the processHold(int) method
650      */
651     @Test
652     public void testProcessHold() {
653
654         // Test for single player game
655         assertEquals(0, singlePlayerGame.getRollScores());
656         singlePlayerGame.processHold(100);
657         assertEquals(100, singlePlayerGame.getRollScores());
658         singlePlayerGame.processHold(-100);
659         assertEquals(-100, singlePlayerGame.getRollScores());
660
661         // test for multi player game - player 1
662         assertEquals(0, multiPlayerGame.getRollScores());
663         multiPlayerGame.processHold(100);
664         assertEquals(100, multiPlayerGame.getRollScores());
665         multiPlayerGame.processHold(-100);
666         assertEquals(-100, multiPlayerGame.getRollScores());
667
668         // test for multi player game - player 2
669         multiPlayerGame.farkle();
670         assertEquals(0, multiPlayerGame.getRollScores());
671         multiPlayerGame.processHold(100);
672         assertEquals(100, multiPlayerGame.getRollScores());
673         multiPlayerGame.processHold(-100);
674         assertEquals(-100, multiPlayerGame.getRollScores());
675
676     }
677
678     /**
679      * Test the processRoll() method
680      */
681     @Test
682     public void testProcessRoll() {
683
684         // Test for single player game
```

```
685
686         // The roll number and roll score should be initially 0
687         assertEquals(0, singlePlayerGame.getCurrentPlayer().getRollNumber());
688         assertEquals(0, singlePlayerGame.getRollScores());
689
690         // Add 100 to the roll and process it
691         singlePlayerGame.processHold(100);
692         singlePlayerGame.processRoll();
693
694         // The roll number should be 1 and the roll score should be 100
695         assertEquals(1, singlePlayerGame.getCurrentPlayer().getRollNumber());
696         assertEquals(100, singlePlayerGame.getRollScores());
697
698         // Add -50 to the roll and process it
699         singlePlayerGame.processHold(-50);
700         singlePlayerGame.processRoll();
701
702         // The roll number should be 2 and the roll score should be 50
703         assertEquals(2, singlePlayerGame.getCurrentPlayer().getRollNumber());
704         assertEquals(50, singlePlayerGame.getRollScores());
705
706         // Test for multi player game
707
708         // The roll number and roll score should be initially 0
709         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
710         assertEquals(0, multiPlayerGame.getRollScores());
711
712         // Add 100 to the roll and process it
713         multiPlayerGame.processHold(100);
714         multiPlayerGame.processRoll();
715
716         // The roll number should be 1 and the roll score should be 100
717         assertEquals(1, multiPlayerGame.getCurrentPlayer().getRollNumber());
718         assertEquals(100, multiPlayerGame.getRollScores());
719
720         // Add -50 to the roll and process it
721         multiPlayerGame.processHold(-50);
722         multiPlayerGame.processRoll();
723
724         // The roll number should be 2 and the roll score should be 50
725         assertEquals(2, multiPlayerGame.getCurrentPlayer().getRollNumber());
726         assertEquals(50, multiPlayerGame.getRollScores());
727
728         // call the fakle method to change to player 2 and test again
729         multiPlayerGame.farkle();
730
731         // The roll number and roll score should be initially 0
732         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
733         assertEquals(0, multiPlayerGame.getRollScores());
734
735         // Add 100 to the roll and process it
736         multiPlayerGame.processHold(100);
737         multiPlayerGame.processRoll();
738
739         // The roll number should be 1 and the roll score should be 100
740         assertEquals(1, multiPlayerGame.getCurrentPlayer().getRollNumber());
741         assertEquals(100, multiPlayerGame.getRollScores());
```

```
742
743         // Add -50 to the roll and process it
744         multiPlayerGame.processHold(-50);
745         multiPlayerGame.processRoll();
746
747         // The roll number should be 2 and the roll score should be 50
748         assertEquals(2, multiPlayerGame.getCurrentPlayer().getRollNumber());
749         assertEquals(50, multiPlayerGame.getRollScores());
750
751     }
752
753     /**************************************************************************
754      * 1.2.10.b - If the user selects the "Restart Game" option, the current
755      * game with all current configurations (player mode, player names, and
756      * player types) is restarted.
757      *************************************************************************/
758     /**************************************************************************
759      * 1.5.1.a - If the user selects the "Play Again?" button, the game will be
760      * restarted with all of the same configuration options of the previous game
761      * (player mode, player's names, and player types).
762      *************************************************************************/
763     /**
764      * Test the resetGame() method
765      */
766     @Test
767     public void testResetGame() {
768
769         // Test reset on a single player game
770
771         /*
772          * Test for turn number = 1, game score = 0, roll number = 0, roll score
773          * = 0, turnScores is empty, and rollScore is empty
774          */
775         assertEquals(1, singlePlayerGame.getTurnNumberForCurrentPlayer());
776         assertEquals(0, singlePlayerGame.getGameScoreForCurrentPlayer());
777         assertEquals(0, singlePlayerGame.getCurrentPlayer().getRollNumber());
778         assertEquals(0, singlePlayerGame.getRollScores());
779         assertTrue(singlePlayerGame.getCurrentPlayer().getTurnScores()
780                 .isEmpty());
781         assertTrue(singlePlayerGame.getCurrentPlayer().getRollScore().isEmpty());
782
783         // simulate a few rolls and turns
784         singlePlayerGame.processHold(100);
785         singlePlayerGame.processRoll();
786         singlePlayerGame.processHold(150);
787         singlePlayerGame.processRoll();
788         singlePlayerGame.bank();
789         singlePlayerGame.farkle();
790         singlePlayerGame.processHold(100);
791         singlePlayerGame.processRoll();
792         assertEquals(3, singlePlayerGame.getTurnNumberForCurrentPlayer());
793         assertEquals(250, singlePlayerGame.getGameScoreForCurrentPlayer());
794         assertEquals(1, singlePlayerGame.getCurrentPlayer().getRollNumber());
795         assertEquals(100, singlePlayerGame.getRollScores());
796
797         // Call the reset method, and test again
798         singlePlayerGame.resetGame();
```

```
799         assertEquals(1, singlePlayerGame.getTurnNumberForCurrentPlayer());
800         assertEquals(0, singlePlayerGame.getGameScoreForCurrentPlayer());
801         assertEquals(0, singlePlayerGame.getCurrentPlayer().getRollNumber());
802         assertEquals(0, singlePlayerGame.getRollScores());
803         assertTrue(singlePlayerGame.getCurrentPlayer().getTurnScores()
804                 .isEmpty());
805         assertTrue(singlePlayerGame.getCurrentPlayer().getRollScore().isEmpty());
806 
807         // Test reset on a multi player game
808 
809         /*
810          * Test for turn number = 1, game score = 0, roll number = 0, roll score
811          * = 0, turnScores is empty, and rollScore is empty for player 1
812          */
813         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
814         assertEquals(0, multiPlayerGame.getGameScoreForCurrentPlayer());
815         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
816         assertEquals(0, multiPlayerGame.getRollScores());
817         assertTrue(multiPlayerGame.getCurrentPlayer().getTurnScores().isEmpty());
818         assertTrue(multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
819         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
820                 multiPlayerGame.getPlayers()[0]));
821 
822         // Set player 2 and check the same tests
823         multiPlayerGame.setCurrentPlayer(2);
824         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
825         assertEquals(0, multiPlayerGame.getGameScoreForCurrentPlayer());
826         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
827         assertEquals(0, multiPlayerGame.getRollScores());
828         assertTrue(multiPlayerGame.getCurrentPlayer().getTurnScores().isEmpty());
829         assertTrue(multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
830         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
831                 multiPlayerGame.getPlayers()[1]));
832 
833         // Change current player back to player 1 and process a few rolls
834         multiPlayerGame.setCurrentPlayer(1);
835         multiPlayerGame.processHold(100);
836         multiPlayerGame.processRoll();
837         multiPlayerGame.processHold(150);
838         multiPlayerGame.processRoll();
839         multiPlayerGame.bank();
840 
841         // Player 1 now has a game score of 250, and is on turn 2
842         multiPlayerGame.farkle();
843 
844         // Player 2 now has a game score of 0, and is on turn 2
845         multiPlayerGame.processHold(100);
846         multiPlayerGame.processRoll();
847         multiPlayerGame.bank();
848 
849         // Player 1 now has a game score of 350, and is on turn 3
850         multiPlayerGame.processHold(550);
851         multiPlayerGame.processRoll();
852         multiPlayerGame.bank();
853 
854         // Player 2 now has a score of 550, and is on turn 3
855         multiPlayerGame.processHold(100);
```

```
856         multiPlayerGame.processRoll();
857         multiPlayerGame.processHold(150);
858         multiPlayerGame.processRoll();
859 
860         // PLayer 1 now has a roll score of 250, and is on roll 2 (starts from
861         // 0)
862 
863         // Test player 1 game state
864         assertEquals(3, multiPlayerGame.getTurnNumberForCurrentPlayer());
865         assertEquals(350, multiPlayerGame.getGameScoreForCurrentPlayer());
866         assertEquals(2, multiPlayerGame.getCurrentPlayer().getRollNumber());
867         assertEquals(250, multiPlayerGame.getRollScores());
868         assertTrue(!multiPlayerGame.getCurrentPlayer().getTurnScores()
869                 .isEmpty());
870         assertTrue(!multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
871         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
872                 multiPlayerGame.getPlayers()[0]));
873 
874         // Test player 2 game state
875         multiPlayerGame.farkle();
876         assertEquals(3, multiPlayerGame.getTurnNumberForCurrentPlayer());
877         assertEquals(550, multiPlayerGame.getGameScoreForCurrentPlayer());
878         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
879         assertEquals(0, multiPlayerGame.getRollScores());
880         assertTrue(!multiPlayerGame.getCurrentPlayer().getTurnScores()
881                 .isEmpty());
882         assertTrue(!multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
883         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
884                 multiPlayerGame.getPlayers()[1]));
885 
886         // Reset the game and test each player
887         multiPlayerGame.resetGame();
888         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
889         assertEquals(0, multiPlayerGame.getGameScoreForCurrentPlayer());
890         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
891         assertEquals(0, multiPlayerGame.getRollScores());
892         assertTrue(multiPlayerGame.getCurrentPlayer().getTurnScores().isEmpty());
893         assertTrue(multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
894         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
895                 multiPlayerGame.getPlayers()[0]));
896 
897         // Change the current player and check again
898         multiPlayerGame.setCurrentPlayer(2);
899         assertEquals(1, multiPlayerGame.getTurnNumberForCurrentPlayer());
900         assertEquals(0, multiPlayerGame.getGameScoreForCurrentPlayer());
901         assertEquals(0, multiPlayerGame.getCurrentPlayer().getRollNumber());
902         assertEquals(0, multiPlayerGame.getRollScores());
903         assertTrue(multiPlayerGame.getCurrentPlayer().getTurnScores().isEmpty());
904         assertTrue(multiPlayerGame.getCurrentPlayer().getRollScore().isEmpty());
905         assertTrue(multiPlayerGame.getCurrentPlayer().equals(
906                 multiPlayerGame.getPlayers()[1]));
907 
908     }
909 
910     /**************************************************************************
911      * 1.2.10.f - If the user selects the "Reset High Score" option, the high
912      * score is reset to 0.
```

```
913      ********************************************************************/
914     /**
915      * Test the resetHighScore() method
916      */
917     @Test
918     public void testResetHighScore() {
919
920         // Set the high score, and then reset it to make sure it is reset to 0.
921         // This is
922         // a Preference, so it will be the same regardless of setting and
923         // getting
924         // it through the single player game or multi player game
925         singlePlayerGame.setHighScore(5000);
926         assertEquals(5000, singlePlayerGame.getHighScore());
927         assertEquals(5000, multiPlayerGame.getHighScore());
928         singlePlayerGame.resetHighScore();
929         assertEquals(0, singlePlayerGame.getHighScore());
930         assertEquals(0, multiPlayerGame.getHighScore());
931
932     }
933 }
934
```

```java
package com.lotsofun.farkle;

import static org.junit.Assert.*;
import java.util.ArrayList;
import java.util.HashMap;
import org.junit.Before;
import org.junit.Test;

/**
 * Tests the Player Class and the calculateScore(List<Integer>, boolean) methods
 *
 * @author Jacob Davidson
 * @version 3.0.0
 */
public class PlayerTest {
    /** The Player object used for testing */
    Player player;

    /** rollScore Hash Map used for testing */
    HashMap<Integer, Integer> rollScore;

    /** An array list of turn scores used for testing */
    ArrayList<Integer> turnScores;

    /**
     * Instantiate all objects before testing begins
     *
     * @throws Exception
     */
    @Before
    public void setUp() throws Exception {
        // Instantiate the player object used for testing
        player = new Player(1);

        // Instantiate a Hash Map for the roll scores
        rollScore = new HashMap<Integer, Integer>();

        // Instantiate the turnScore array list
        turnScores = new ArrayList<Integer>();
    }

    /**
     * Test the Player constructor
     */
    @Test
    public void test() {
        assertNotNull(player);
        assertNotNull(player.getTurnScores());
        assertTrue(player.getTurnScores().isEmpty());
        assertEquals(1, player.getPlayerNumber());
        assertEquals(PlayerType.USER, player.getType());
    }

    /**
     * Test the endTurn(boolean) method
     */
    @Test
```

```
58      public void testEndTurn() {
59
60          // Add a few rolls to the rollScore hash map
61          rollScore.put(0, 50);
62          rollScore.put(1, 100);
63          rollScore.put(2, 150);
64
65          // Set the roll score for the player
66          player.setRollScore(rollScore);
67
68          // Set the roll number for the player
69          player.setRollNumber(3);
70
71          // Add a few turns to the turnScores array list
72          turnScores.add(400);
73          turnScores.add(300);
74
75          // Set the turnScores for the player
76          player.setTurnScores(turnScores);
77
78          // Set the turn number for the player
79          player.setTurnNumber(3);
80
81          // Set the game score for the player
82          player.setGameScore(700);
83
84          // End turn via a farkle, and test
85          player.endTurn(true);
86
87          assertTrue(0 == player.getTurnScores().get(2));
88          assertEquals(700, player.getGameScore());
89          assertEquals(4, player.getTurnNumber());
90          assertEquals(0, player.getRollNumber());
91          assertTrue(player.getRollScore().isEmpty());
92          assertEquals(player.getRollScores(), 0);
93
94          // Reset the player information, and test the endTurn(false)
95
96          // Clear and Add a few rolls to the rollScore hash map
97          rollScore.clear();
98          rollScore.put(0, 50);
99          rollScore.put(1, 100);
100         rollScore.put(2, 150);
101
102         // reset the roll score for the player
103         player.setRollScore(rollScore);
104
105         // reet the roll number for the player
106         player.setRollNumber(3);
107
108         // Clear and add a few turns to the turnScores array list
109         turnScores.clear();
110         turnScores.add(400);
111         turnScores.add(300);
112
113         // reet the turnScores for the player
114         player.setTurnScores(turnScores);
```

```
115
116         // reet the turn number for the player
117         player.setTurnNumber(3);
118
119         // reet the game score for the player
120         player.setGameScore(700);
121
122         // End turn via a bank, and test
123         player.endTurn(false);
124
125         assertTrue(300 == player.getTurnScores().get(2));
126         assertEquals(1000, player.getGameScore());
127         assertEquals(4, player.getTurnNumber());
128         assertEquals(0, player.getRollNumber());
129         assertTrue(player.getRollScore().isEmpty());
130         assertEquals(player.getRollScores(), 0);
131     }
132
133     /**
134      * Test the getGameScore() and setGameScore(int) methods
135      */
136     @Test
137     public void testGetAndSetGameScore() {
138
139         // The player game score should initially be 0
140         assertEquals(0, player.getGameScore());
141
142         // Set the game score and test again
143         player.setGameScore(1000);
144         assertEquals(1000, player.getGameScore());
145     }
146
147     /**
148      * Test the getPlayerName() and setPlayerName(String) methods
149      */
150     @Test
151     public void testGetAndSetPlayerName() {
152         // The player name should initially be null
153         assertNull(player.getPlayerName());
154
155         // Set the player name to something else and test it again
156         player.setPlayerName("Jake");
157         assertTrue(player.getPlayerName().equals("Jake"));
158     }
159
160     /**
161      * Test the getPlayerNumber() method
162      */
163     @Test
164     public void testGetPlayerNumber() {
165         // The player number is set with the constructor
166         assertEquals(1, player.getPlayerNumber());
167
168         // Initialize another player with a different number and test it
169         Player playerTwo = new Player(2);
170         assertEquals(2, playerTwo.getPlayerNumber());
171     }
```

```
172
173     /**
174      * Test the getRollNumber() and setRollNumber(int) methods
175      */
176     @Test
177     public void testGetAndSetRollNumber() {
178         // The roll number should be initially set to 0
179         assertEquals(0, player.getRollNumber());
180
181         // Set the roll number and test again
182         player.setRollNumber(5);
183         assertEquals(5, player.getRollNumber());
184
185         // Test with a negative number
186         player.setRollNumber(-5);
187         assertEquals(-5, player.getRollNumber());
188     }
189
190     /**
191      * Test the getRollScore() and setRollScore(HashMap<integer, integer>)
192      * methods
193      */
194     @Test
195     public void testGetAndSetRollScore() {
196         // rollScore should initially be empty
197         assertTrue(player.getRollScore().isEmpty());
198
199         // Create a rollScore and add it to the player
200         rollScore.put(0, 100);
201         rollScore.put(1, 200);
202         rollScore.put(2, 50);
203
204         // Assign this rollScore to the player
205         player.setRollScore(rollScore);
206
207         // Test the getRollScore() method
208         assertTrue(player.getRollScore().equals(rollScore));
209
210         // Test with null
211         player.setRollScore(null);
212         assertNull(player.getRollScore());
213     }
214
215     /**
216      * Test getRollScores()
217      */
218     @Test
219     public void testGetRollScores() {
220         // The rollScore should initially be null
221         assertEquals(0, player.getRollScores());
222
223         // Create a rollScore and add it to the player
224         rollScore.put(0, 100);
225         rollScore.put(1, 200);
226         rollScore.put(2, 50);
227
228         // Assign this rollScore to the player
```

```
229         player.setRollScore(rollScore);
230 
231         // Test the rollScore
232         assertEquals(350, player.getRollScores());
233 
234         // Check for negative numbers
235         rollScore.clear();
236         rollScore.put(0, -100);
237         rollScore.put(1, -200);
238 
239         // Assign this rollScore to the player
240         player.setRollScore(rollScore);
241 
242         // Test the rollScore
243         assertEquals(-300, player.getRollScores());
244     }
245 
246     /**
247      * Test the getTurnNumber() and setTurnNumnber(int) methods
248      */
249     @Test
250     public void testGetAndSetTurnNumber() {
251         // turnNumber should initially be 1
252         assertEquals(1, player.getTurnNumber());
253 
254         // Set the turn number to something else and retest
255         player.setTurnNumber(5);
256         assertEquals(5, player.getTurnNumber());
257     }
258 
259     /**
260      * Test the getTurnScores() and setTurnScores(ArrayList<Integers>)
261      */
262     @Test
263     public void testGetAndSetTurnScorees() {
264         // turnScores should initially be empty
265         assertTrue(player.getTurnScores().isEmpty());
266 
267         // Set the turnScores and test it
268         turnScores.add(400);
269         turnScores.add(300);
270         player.setTurnScores(turnScores);
271         assertTrue(player.getTurnScores().equals(turnScores));
272     }
273 
274     /**
275      * Test the getType() and setType(PlayerType) methods
276      */
277     @Test
278     public void testGetAndSetType() {
279         // Initially, player should be set to PlayerType.USER
280         assertEquals(PlayerType.USER, player.getType());
281 
282         // Change the type and test again
283         player.setType(PlayerType.COMPUTER);
284         assertEquals(PlayerType.COMPUTER, player.getType());
285     }
```

```
286
287     /**
288      * Test the resetRollScores() method
289      */
290     @Test
291     public void testResetRollScores() {
292
293         // Assign several rolls to roll score
294         rollScore.put(0, 100);
295         rollScore.put(1, 150);
296         rollScore.put(2, 50);
297
298         // Assign rollScore to player
299         player.setRollScore(rollScore);
300
301         // Test to make sure the roll score for player is not empty
302         assertTrue(!player.getRollScore().isEmpty());
303
304         // Reset the roll score and test to make sure it's empty
305         player.resetRollScores();
306         assertTrue(player.getRollScore().isEmpty());
307     }
308
309     /**
310      * Test the resetTurnScores() method
311      */
312     @Test
313     public void testResetTurnScores() {
314
315         // Assign several turns to turnScores
316         turnScores.add(400);
317         turnScores.add(300);
318         turnScores.add(500);
319
320         // Assign turnScores to player
321         player.setTurnScores(turnScores);
322
323         // Test to make sure the turn scores for player are not empty
324         assertTrue(!player.getTurnScores().isEmpty());
325
326         // Reset the turn scores and test to make sure it's empty
327         player.resetTurnScores();
328         assertTrue(player.getTurnScores().isEmpty());
329     }
330
331     /**
332      * Test the scoreRoll(int) method
333      */
334     @Test
335     public void testScoreRoll() {
336         // The rollScore should initially be empty and rollNumber should be 0
337         assertTrue(player.getRollScore().isEmpty());
338         assertEquals(0, player.getRollNumber());
339
340         // score a roll and test it
341         player.scoreRoll(550);
342         assertTrue(player.getRollScore().get(0).equals(550));
```

```
343
344         // Change the roll number, add a rollScore and test again
345         player.setRollNumber(1);
346         player.scoreRoll(300);
347         assertTrue(player.getRollScore().get(1).equals(300));
348         assertEquals(850, player.getRollScores());
349     }
350
351 }
352
```

Known Bugs and Issues

TEAM 1: CURTIS BROWNN
JACOB DAVIDSON
BRANT MULLINIX

CSC 478 – B
NOVEMBER 29, 2014

Known Bugs and Issues

1. File Menu

There is a known bug that causes the opened file menu to close while the computer player is taking its turn in two player mode against a computer opponent. If the file menu is opened by the user, it is automatically closed whenever the computer player rolls the dice, selects a die, gets a farkle, or selects the bank button. This minor bug does not detract from overall game play.

2. "Hint" File Menu Option

There is a known bug with the "Hint" file menu option that occurs in all game modes. The algorithm for the "Hint" file menu option only considers unheld dice. When the user selects any available dice after a roll, the hint menu option will only return the maximum scoring combination of dice for the remaining unheld dice. This does not help the user if they are selecting three or more of a kind, three pairs, or a straight. For this reason, we have chosen to disable the hint menu option if the user has selected any dice after a roll.

3. "New Game" File Menu Option

There is a known bug with the "New Game" menu option in a two player game versus a computer opponent. When the computer player's turn is animated in a separate thread, the "New Game" menu option does not properly reset the application to the farkle game mode options box. For this reason, we have chosen to disable the "New Game" file menu option when the computer player is taking its turn.

4. "Reset Game" File Menu Option

There is a known bug with the "Reset Game" menu option in a two player game versus a computer opponent. When the computer player's turn is animated in a separate thread, the "Reset Game" menu option does not properly reset the current game. For this reason, we have chosen to disable the "Reset Game" file menu option when the computer player is taking its turn.

5. Two Player Version Turn Scores Highlighting

There is a known bug with the turn score highlighting in two player mode. When player one begins their 11^{th} turn, a scroll bar is enabled allowing that user to scroll through all of the previous turn scores, and the 11^{th} turn is highlighted. Periodically, the turn score for the first turn of that player is also partially highlighted until the turn is over. This bug occurs sporadically and is difficult to replicate. This does not detract from overall game play, and is extremely minor in nature.