EGR 326, Spring 2016 Group Project: Yahtzee

On this assignment you will work in a group of 3 students to implement the dice game called Yahtzee. Your program will be implemented in Java and will display a graphical user interface using Swing/AWT.

There will be no sample solution will be posted. You can be creative and make your own unique implementation, so long as it satisfies the constraints listed in this document. The only class that I specify you should have is a class named `YahtzeeMain` in the default package that contains a `main` method to launch the program. You do not need to turn in any testing program (such as a JUnit test case) but you are welcome to do so if you like.

Yahtzee Game Description:

Yahtzee is a dice game played by one to four players. In this assignment, you need to support exactly **1 player** (computer AI player). But as part of the incremental development, you can first implement human player and then replace the code later by computer AI player.

A round of the game consists of each player taking a turn (but again in this assignment there is only 1 player). On each turn, a player rolls the five dice with the hope of getting them into a configuration that corresponds to one of 13 categories listed below. Each die rolled will randomly show a value from 1-6 inclusive with equal probability.

After an initial roll of all 5 dice, the player may choose to roll any or all of the dice again up to two more times (for a total of three rolls). By the end of the third roll, the player must assign the final dice configuration to one of the thirteen categories on the scorecard.

If the dice configuration meets the criteria for that category, the player receives the appropriate score for that category; otherwise the score for that category is 0. **Since there are thirteen categories and each category is used exactly once, a game consists of thirteen rounds.** After the thirteenth round, all players will have received scores for all categories. Once you have chosen the category for the dice configuration at each round, you cannot change afterwards. The player with the total highest score is declared the winner.

The thirteen categories of dice configurations and their scores are:

1. **Aces:** Any dice configuration is valid for this category. The score is equal to the sum of all of the 1s showing on the dice, or 0 if there are no 1s showing.
2. **Twos:** (Same as Aces but for the value 2). Any dice configuration is valid for this category. The score is equal to the sum of all of the 2s showing on the dice, or 0 if there are no 2s showing.

3. **Threes:** (Same as Aces but for the value 3). Any dice configuration is valid for this category. The score is equal to the sum of all of the 3s showing on the dice, or 0 if there are no 3s showing.
4. **Fours:** (Same as Aces but for the value 4). Any dice configuration is valid for this category. The score is equal to the sum of all of the 4s showing on the dice, or 0 if there are no 4s showing.
5. **Fives:** (Same as Aces but for the value 5). Any dice configuration is valid for this category. The score is equal to the sum of all of the 5s showing on the dice, or 0 if there are no 5s showing.
6. **Sixes:** (Same as Aces but for the value 6). Any dice configuration is valid for this category. The score is equal to the sum of all of the 6s showing on the dice, or 0 if there are no 6s showing.

Category	Description	Score	Example
Aces	Any combination	The sum of dice with the number 1	 scores 3
Twos	Any combination	The sum of dice with the number 2	 scores 6
Threes	Any combination	The sum of dice with the number 3	 scores 12
Fours	Any combination	The sum of dice with the number 4	 scores 8
Fives	Any combination	The sum of dice with the number 5	 scores 0
Sixes	Any combination	The sum of dice with the number 6	 scores 18

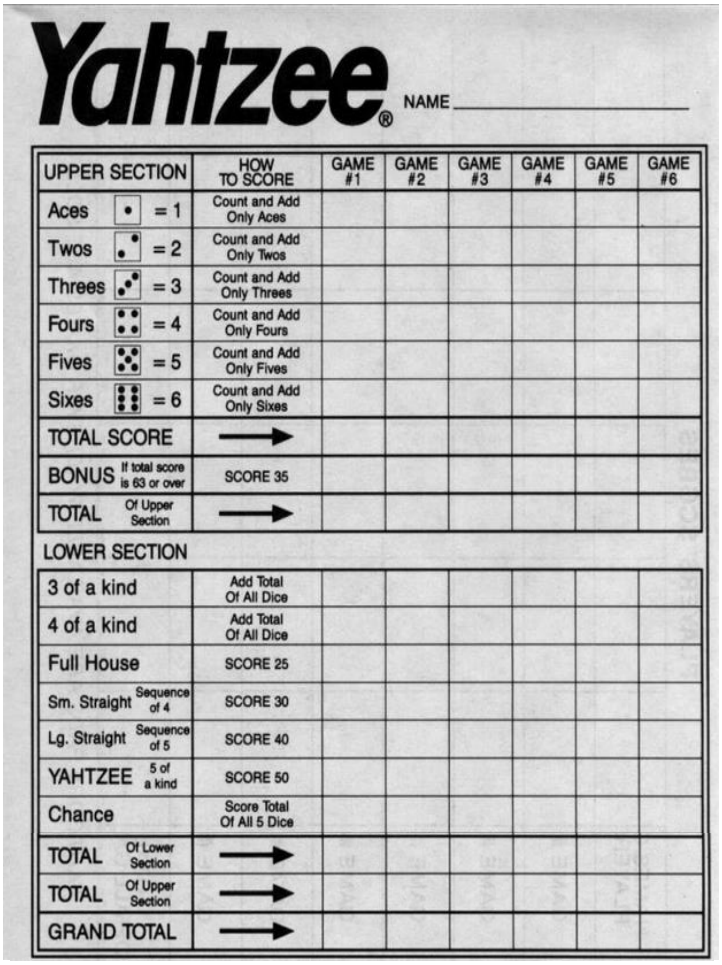
7. **Three of a Kind:** At least three of the dice must show the same value. The score is equal to the sum of all of the values showing on the dice.
8. **Four of a Kind:** At least four of the dice must show the same value. The score is equal to the sum of all of the values showing on the dice.
9. **Full House:** The dice must show three of one value and two of another value. The score is 25 points.
10. **Small Straight:** The dice must contain at least four consecutive values, such as 2-3-4-5. The score is 30 points.
11. **Large Straight:** The dice must contain five consecutive values, such as 1-2-3-4-5. The score is 40 points.
12. **Yahtzee!:** All of the dice must show the same value.. The score is 50 points.
13. **Chance:** Any dice configuration is valid. The score is equal to the sum of all of the values showing on the dice.

Scoring:

A given hand of dice might fit into several categories. For example, the hand [4, 5, 5, 4, 5] would receive a nonzero score if placed into the Fives, Sixes, Three of a Kind, Full

House, or Chance categories. In such a case, it is up to the player which category to place the hand. Some amount of strategy is required here: Should you use the hand for Three of a Kind, scoring 4+5+5+4+5 = 23 points? Should you use it for Full House, earning 25 points? It might seem wise to use it for the Full House, since it is worth more. But 23 is a lot to earn for the Three of a Kind category, relative to other threes of a kind. So perhaps it is good to allocate this hand there, if we think we're likely to see another full house on a later turn. A later hand of [1, 2, 1, 1, 2] might be a better one to use for Full House since it is still worth 25 there but only worth 7 under Three of a Kind or Chance.

The 13 categories that make up the game are divided into two sections. The upper section contains the categories Aces, Twos, Threes, Fours, Fives, and Sixes. At the end of the game, the values in these categories are added to generate the value in the entry labeled **Upper Score**. If a player's score for the upper section ends up totaling 63 or more, that player is awarded a 35-point bonus on the next line. The scores in the lower section of the scorecard are also added together to generate the entry labeled Lower Score. The total score for each player is then computed by adding together the upper score, the bonus (if any), and the lower score. (The real game of Yahtzee gives the player a bonus score for rolling multiple Yahtzees in a game, but do NOT support that for this project.) Below figure is the scorecard that depicts Upper Section and Lower Section to get the Grand Total.



Yahtzee®

NAME _____

UPPER SECTION	HOW TO SCORE	GAME #1	GAME #2	GAME #3	GAME #4	GAME #5	GAME #6
Aces = 1	Count and Add Only Aces						
Twos = 2	Count and Add Only Twos						
Threes = 3	Count and Add Only Threes						
Fours = 4	Count and Add Only Fours						
Fives = 5	Count and Add Only Fives						
Sixes = 6	Count and Add Only Sixes						
TOTAL SCORE	→						
BONUS If total score is 63 or over	SCORE 35						
TOTAL Of Upper Section	→						
LOWER SECTION							
3 of a kind	Add Total Of All Dice						
4 of a kind	Add Total Of All Dice						
Full House	SCORE 25						
Sm. Straight Sequence of 4	SCORE 30						
Lg. Straight Sequence of 5	SCORE 40						
YAHTZEE 5 of a kind	SCORE 50						
Chance	Score Total Of All 5 Dice						
TOTAL Of Lower Section	→						
TOTAL Of Upper Section	→						
GRAND TOTAL	→						

You can read more about the game of Yahtzee on Wikipedia: http://en.wikipedia.org/wiki/Yahtzee

Your Implementation:

You will implement a graphical Yahtzee game in Java that implements the basic game play with the following features. Various categories of features are described in more detail in later sections of this document.

- One computer player
- Four specific computer player strategies as described below
- The basic game play and ability to select and roll/re-roll dice, then assign a given hand of dice into one of the thirteen categories previously described;
- Keeping score properly for this computer player using the score rules previously described (Must show final dice configuration and then corresponding score)
- Allowing multiple games to be played and storing a cumulative score for the player, separate from that player's score for the current game.

One of the more challenging tasks you will be faced with is determining whether a dice configuration meets the requirements for a given category, and is therefore valid. For example, Three of a Kind requires a dice configuration in which at least three of the dice show the same value, Small Straight requires at least four of the dice values to be

consecutive, and so forth. If a player assigns an invalid dice configuration to a category, they receive 0 points for it.

Four Computer Player Strategies:

Your game should implement a computer AI player that uses various specific strategies described below. The computer player will choose which the strategy per round based on his judgement. Upon selecting the strategy, the computer player will automatically and immediately make its move. Your code should provide a means to show which strategy was selected for the round.

• **Random:** Randomly chooses which dice to re-roll each turn, then chooses a random category to assign the hand with equal probability.
• **Of-a-Kinder:** Keeps dice values that occur 2 or more times and re-rolls the rest. For example, if his initial hands [2, 4, 3, 1, 4], he keeps the 4s and re-rolls the rest. If he now has [1, 4, 1, 5, 4], he keeps the 1s and 4s and re-rolls the 5. At the end of all rolls, he places the hand into the category that gives the max score for that hand.
• **Upper-Sectioner:** Has the same behavior about which dice to keep/roll as an Of-a-Kinder. But once the hand is done, if there are any Upper Section categories left (such as Aces, Twos, etc.), he places it into the Upper Section category that provides the maximum score for that hand. If no Upper Section categories remain, he will choose the remaining category that provides the maximum score for that hand.
• **Four-and-Up:** Keeps any dice whose values are 4 or above, and re-rolls the others. At the end of all rolls, he places the hand into the category that provides the maximum score for that hand.

Your algorithms must implement the Strategy pattern as taught in class. In particular, strategies must fall under a common type hierarchy, and the rest of your model code should be largely ignorant of strategy algorithms. It should simply be supplied and use a strategy without worrying about which one it is.

Development Strategy and Hints:

Here are some possibly helpful notes regarding testing for hand categories and scoring:
• There's not much difference between determining validity for Three/Four of a Kind, Yahtzee, and Full House.
• There's not much difference between determining the validity for Small Straight and Large Straight.
• Any dice configuration is valid for Aces, Twos, Threes, Fours, Fives, Sixes, and Chance.

Check for errors when the player selects the category to assign a hand. The user cannot re-use any previous category.

You might find it worthwhile to create a "cheat" mode during development. If you are running in cheat mode, you can hard-code or prompt the user for dice values instead of choosing the dice randomly. Implementing this feature will make it easier for you to check the various situations that can come up during the game.

Output:

Your program must output the current state of the game for each round as follows:

- 1st Dice roll
- If any, 2nd and 3rd dice roll, and which set of dices was re-rolled
- Final dice configuration
- The strategy picked for the current round
- Current score of the player
- Total score of the player at the end of the final round
- Previous scores of previous N games

Graphical User Interface and Views (10% Extra-Credit):

GUI is completely optional and will count as 10% extra-credit. If you choose to implement GUI, your program's graphical user interface should meet the following appearance and behavioral constraints:

• The overall window size should be appropriate to show the state of the game.
• The window title should be an appropriate message indicating that this is a Yahtzee game for EGR 326.
• When the main window closes, the program should exit.
• The view must represent all major aspects of the current state of the application, including the current dice, the strategy selected for the round, which dice the player's score card and total points earned so far.
• The view and GUI should enable and disable various components such that the user is prevented from giving invalid input to the system and is prevented from interacting with components that are not presently relevant.
• The view need to be able to handle resizing gracefully for a reasonable range of window sizes. The views do not need to scale proportionally, but they should not crash or become damaged by a resize. (If the window is resized too small to fit the view's contents, it is expected that some contents would become obscured or unable to be seen on the screen)

Design Constraints:

The source code of your project should meet the following design constraints:

• Your computer play algorithms must implement the Strategy pattern as taught in class. In particular, strategies must fall under a common type hierarchy, and the rest of your model code should be largely ignorant of strategy algorithms. It should simply be supplied and use a strategy without worrying about which one it is.

• You must use at least three other patterns in your solution. For example, you could use the Flyweight pattern for one of your classes if appropriate. Or you could use the Prototype pattern and cloning if necessary. Or you could use the State pattern to keep track of your model's state. (Using iterators of collections, while it may seem to be a use of the Iterator pattern, also does not count for this requirement. Making one of your own objects have an iterator, though, would.)
• Make at least one class in your system **immutable**.
• Organize the classes in your program into packages. Use packages to separate your model from your views, and for any other major categories of functionality. You may add as many if you like. Your main class to run the program should remain in the default unnamed package.

Groups:

For this assignment you must work in a group of 2 to 3 students. Your group should turn in a single copy of your assignment (you don't all need to turn it in).
You should include the primary author(s) of each source file in the `@author` tag of any Javadoc comment headings atop your classes. If multiple group members work on a given file, list all of their names in descending order of contribution.
We expect that all group members should contribute substantially to the final program. It is not appropriate for one or two group members to do the vast majority of the work. This includes willingness to do all of the following:
• meet at least once weekly with your group at a scheduled time
• communicate regularly with your group partners as needed by email, in person, by phone, or otherwise
• hold your group partners accountable for their work, and talk to them and/or report to instructor if they fail to do it
Your group's project will receive a percentage grade as specified previously. By default, this grade will be shared among all group members. However, if particular group members contribute especially more or especially less than their partners, they may be subject to an individual multiplier between 0.5 and 2. Missing meetings, failing to communicate, not doing one's share of work, or other negligence may result in a multiplier less than 1.0. This will only be done in circumstances where the group members and instructor agree clearly that there was a significant lack of work performed.
If one of your partners is not doing his/her share of the work, it is your responsibility to attempt to convince them to contribute. If this is not successful, notify your instructor promptly. If we are not notified of a group problem until the last minute, there is not a good chance that a group problem can be resolved, nor different grades assigned

Deliverable Milestones:

You will submit your work incrementally over the next 3 weeks. The work is divided into the following "milestone" deliverables. Each milestone can be submitted up to 24 hours late for a -5% penalty, and will not be accepted more than 24 hours late. Each phase is graded on functionality ("external correctness") only, aside from Milestone 4. If any

group members have late days remaining, those group members will lose late days rather than points.

Milestone 1: Zero-Feature Check-in (Wed, March 30, 5:00 PM) (8% of grade)

By this date we will check to see that you have at least set up your github or bitbucket repository and that each group member has made a minimal check-in to it. You will receive full credit for this milestone if every member has checked in something, anything, to the repository. If some group members have not made a check-in, you will lose points.

Milestone 2: Beta Code (Tue, April 7, 2 PM) (25% of grade)

The first version of your code implementation will be called the "milestone" version. For this version we want to see that your group has completed a significant chunk of the work of the project, though we do not expect the game to be fully functional. We expect that by this date, we should be able to go to your repository, check out its contents, and compile and run the program successfully. When the program is run, we should be able to view the dice configuration and basic player information. It should be possible to do a single human game play. That is, you should be able to select dice and roll / re-roll them, and then assign a hand to a category.

But you do not need to implement any scoring, computer AI, multiple games, or many of the other features. If you choose to implement GUI, your GUI can also be unpolished and un-robust against errors such as invalid moves or empty strings. It is allowed for this milestone to contain some bugs as long as the general functionality can be run and tested.

Milestone 3: In-Class Project Demo/Competition (Tue, April 19, 2 PM, in class) (15% of grade)

On the last day of lecture in class, each team will perform short demos of all groups' projects. Your project should be finished by this date so that it can be demoed successfully. Also for fun, we will do #5 rounds of game play to determine the winner.

Milestone 4: Final Code Submission (Tue, April 19, 11:59 PM) (50% of grade)

You will submit the final version of your code and resources. The final version will be graded more strictly than the milestone, and will be expected to meet all requirements for the project as described in this document. The correctness of your code will be graded by running your code. Exceptions should not occur under normal usage.

Milestone 5: Group Peer Evaluations (Tue, April 19, 11:59 PM) (2% of grade)

After you've submitted your final code, you will also fill out a short survey about each of your group partners and what work you and the others did on the project. (Survey link will be emailed

out to you). This will help us make sure that every group member contributed significantly to the project. Details about this milestone will be announced later.

Submitting Your Files:

Since we don't know what classes you will choose, you should submit a .zip file named `yahtzee.zip` containing all .java source files necessary to build and execute your program. We should be able to compile your code and run your `YahtzeeMain` class (which should be located in the default package) using only the resources you turn in. Do not include .class files, but do include any supporting resources that you use in your program or view, such as images.