CSE 331, Spring 2011
Group Project: Yahtzee

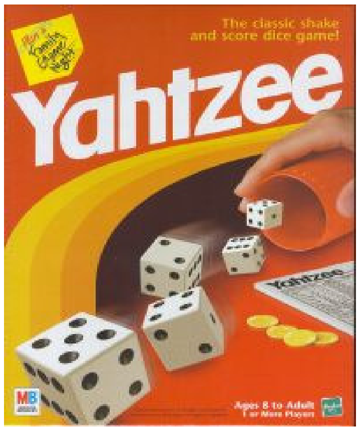
(see later in this document for due dates)

Portions of this spec are based on a similar assignment given at Stanford University by Julie Zelenski and Eric Roberts.

On this assignment you will work in a group of 3-4 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.

Unlike in some other assignments, **no sample solution** will be posted to the course web site. We want you to be creative and make your own unique implementation, so long as it satisfies the constraints listed in this document. There are also no support files provided for this assignment. We do not specify exactly what classes you should have, other than 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) on this assignment, but you are welcome to do so if you like.

Yahtzee Game Description:



Yahtzee is a dice game played by **one to four players** (though you only have to support exactly 2 players in this project). A round of the game consists of each player taking a turn. 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.

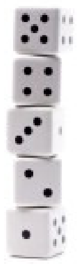
After an initial roll of all 5 dice, the player may choose to roll any or all of the dice again up to two more times (for a total of **three rolls**). By the end of the third roll, the player must assign the final dice configuration to one of the thirteen categories on the scorecard. If the dice configuration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

The thirteen **categories** of dice configurations and their scores are:

1. **Ones**. 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 Ones 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 Ones 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 Ones 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 Ones 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 Ones 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.
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 Ones, 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 you don't need to support that for this project.

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.

- A Swing/AWT graphical interface (GUI);
- Two-player mode with two human players or one human versus one computer player;
- At least four specific computer player strategies;
- The basic game play and ability to select and roll dice, then assign a given hand of dice into one of the thirteen categories previously described;
- Keeping score properly for each of the two players using the score rules previously described;
- Determining who won each game and indicating this to the user;
- Allowing multiple games to be played and storing a cumulative score for each 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.

Graphical User Interface and Views:

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 CSE 331.
- The window should appear in the **center** of the screen when it first loads.
- When the main window closes, the program should exit.

Your GUI should support at least **two views**: one "component view" and one "painted view". A view is implemented as a class that extends `JComponent` or `JPanel` and displays the state of the application's data model using graphical components or 2D graphics painting. It should be possible through the GUI to change the view at any reasonable time during the execution of the program.

The exact appearance of each view is not specified in this document, so you can be creative. But your views must have the following characteristics:

- At least one view should display the model primarily **using existing Java components** such as buttons, labels, text fields, combo boxes, and so on.
- The set of graphical components chosen for said view should be **non-trivial**, such as using several different components and having them respond to various events intelligently. **You must use at least two (2) graphical components that were not used in any of the previous homework assignments.**
- At least one control button on the screen should use an **icon** of your own choosing. You can find icons through Google Image Search on the web. Turn in your icon file(s) with your program.

- At least one view should draw the model primarily **using 2D graphics** and painting.
- Your 2D graphics should be **non-trivial**, such as using many colors/paints, gradients, shapes, stroke thicknesses, images, anti-aliasing, and/or other features shown in class.
- Your 2D graphics should draw any onscreen shapes by creating shape objects rather than using the procedural drawing methods such as `drawRect` and `fillOval`.
- Both views must represent all major aspects of the **current state** of the application, including the current dice, each player's score card and total points earned so far, and who is winning (or wins) the game.
- The views 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.
- Your 2D graphical view must respond to clicks on the various game board squares using **mouse events**.
- Both views do 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.

Strategies:

Your game should allow for both players to be human or for one of the players to be a computer AI opponent that plays using various specific strategies described below. Your GUI should provide a means to choose from all available strategies. When one of the computer strategies is selected, the computer player strategy will automatically and immediately make its move when it is its turn.

A computer player makes his moves using one of the following selectable movement strategies. Whenever a strategy algorithm describes moving randomly, the algorithm should choose from available choices with equal probability.

- **Random:** Randomly chooses which dice to re-roll each turn, then chooses a random category to assign the hand.

- **Of-a-Kinder:** Keeps dice values that occur 2 or more times and re-rolls the rest. For example, if his initial hand is [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 Ones, 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.

To make moves on the game, strategies will have to have some sort of communication with the overall game board in one direction or the other. Though you should not expose the game model's internal representation to strategies or otherwise, you may assume that there is not an evil or "rogue" strategy that would try to do malicious things such as making multiple moves or making poorly chosen moves on behalf of the opponent.

Every strategy may assume a precondition while making its move that the game is still in progress and that it is that player's turn. It is your responsibility to ensure that it is used in such a way by your game.

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.

Extra Features:

Your game should include at least **two extra features** from the following list. If you complete more than one extra feature, each additional one you implement properly will earn you **+1 extra point** on this assignment, up to a max of +5 extra features. Your overall assignment score cannot exceed 100% of the possible points. If you have an idea for another feature not shown on this list, you can ask the instructor and/or TAs to see if we will grant permission to use it for credit.

- **High score feature**: Save the top ten highest scores and names to a file and make it persistent between runs of the program. Read the file when you start and print out the hall of fame. If a player gets a score that preempts one of the early high scores, congratulate them and update the file before you quit so it is recorded for next time.
- **Bonus scores for multiple Yahtzees in a game**: As long as you have not entered a 0 in the Yahtzee box, the rules of the game give you a bonus chip worth 100 points for each additional Yahtzee you roll during the same game.
- **Two computer player mode**: Allow both players to be computer players using your strategies.
- **Up to 4 players mode**: The real game of Yahtzee allows up to 4 players. Make your game allow one, two, three, or four players to participate.
- **Third view**: Provide a third non-trivial view of the game in addition to the two already described.
- **Fifth strategy**: Provide a fifth non-trivial computer AI strategy in addition to the four already described. If you want to try to write a more advanced player that performs better, consider the following general observations:
    - You should generally aim for the 35 points bonus by filling the upper section with high scores (so as to get at least 63 points) near the beginning of the game.
    - If you roll a combination for which your available boxes give a low score, fill in a zero in the 1's box (or even the 2's or 3's), since high scores in the other boxes may compensate for this.
    - Early in the game you should use the Chance box only as a fall-back for when you are trying for a high point combination and fail. Keep the Chance box available for a situation where your options are very limited.
    - If you have a bad hand near the end of the game, enter a 0 in the Yahtzee box or another difficult category.
    - If you roll a pair of 1's or 2's early in the game, go for a full house, straight, or one of the other high numbers you rolled. Keeping the 1's or 2's to seek a 3 or 4 of a kind will not get you many points. Plus, it is a good idea

to keep the 1's and 2's boxes available for later in the game to have the possibility of putting a zero there. As an example, if on your first turn your initial roll is [1, 1, 2, 5, 6], it turns out that keeping only the 5 is the choice that maximizes your expected score for the game.

  - If you roll four of a kind with 4's, 5's, or 6's, take the points in the upper section if they are available, not in the 4 of a kind box. This will help you get the 35 points bonus at the end of the game.
  - You should always try for a Yahtzee early in the game. If you roll a 3 or 4 of a kind on your first or second roll and you have the corresponding box in the upper section open, you should definitely go for the Yahtzee. Plus, if you saved your chance, 1's, or 2's boxes, you can score a zero there if you fail to get the Yahtzee.

- **Animation**: Make a view use animation timers in a non-trivial way to animate various game state and events.
- **Sound**: Play sound effects in response to various in-game events.
- **Network game**: Support the ability for players to connect to the game over the network. (This is hard.)
- **JAR archive**: Make your game runnable from a JAR archive with no other external resources present.

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 Ones, 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.

Design Constraints:

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

- At least one class from your model must use **Observable** objects. Your views should be **Observers** watching these observable objects. As much as possible, changes that occur in the appearance of the view in response to events should come through observer updates. The game model should notify its observers on any externally visible mutation to its state, whether or not that notification is needed by your GUI for this particular program.
- 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 the Composite pattern for layout does not count. 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.

Style and Design Guidelines:

This document does not specify exactly what classes you should write, nor what behavior, fields, methods, etc. each class should have. Part of this assignment is for you to choose appropriate classes and contents for each class based on the specification of desired functionality and based on our class discussions of proper class design. A major part of your grade will come from how effectively you design the contents of each class to solve this problem elegantly. In general, if an entity is important in this system, you should represent that entity with a Java class.

Part of your grade comes from having good **model/view separation**. In particular, the core data classes of your program should not perform any direct user interaction or store information directly related to a specific kind of user interaction. Instead, your user interface classes should perform all graphical user input/output. Also do not place core system functionality into the GUI classes that would be better placed in the model.

Your model classes are responsible for the same **error handling and exception checking** that was described in previous assignments. Namely, you should check for `null` parameters, empty/whitespace strings, and negative numbers and throw an appropriate exception if a bad parameter is passed.

You should work diligently to reduce **redundancy** in your code. If you find yourself repeating code, make the code into a method or pull it out into its own class to reduce the redundancy.

The guidelines described on pages 7-8 of the **Homework 4 spec** apply here, such as using a good object-oriented design, using design patterns as appropriate, following the Expert pattern, avoiding representation exposure, and ensuring invariants on the valid state of your objects by throwing exceptions. For brevity's sake, we will not repeat all of those guidelines here, though you are still responsible for meeting them as appropriate.

Document all of your files by commenting them descriptively *in your own words* at the top of each class, each method/ constructor, and on complex sections of code. Use **Javadoc comments** for all external documentation (atop class headers or public method/constructor headers). Include descriptive initial summaries along with proper tags such as `@param`. See Homework 4's spec for a more detailed description of our expectations regarding comments.

Groups:

For this assignment you must work in a group of **3 to 4 students**. Your group should turn in a single copy of your assignment (you don't all need to turn it in). The instructor and TAs will provide you with a Unix group and an **SVN repository** on the department's shared servers to facilitate working as a team.

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 your TA or 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 TA and/or 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 several 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 (Mon, May 23, 5:00 PM) (8% of grade)

By this date we will check to see that you have at least set up your SVN 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 (Sat, May 28, 11:30 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 see the dice and basic player information for at least one view. It should be possible to at least play part of a single human vs. human game with some amount of success. That is, we 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, multiple views, or many of the other features. Your GUI can also be unpolished and un-robust against errors such as invalid moves or empty strings. You do not need to have any of the "extra" features implemented in this milestone. 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 (Friday, June 3, 10:30 AM, in class) (5% of grade)

On the last day of lecture in class, we will perform short demos of all groups' projects. During these demos, the instructor will run your game on his laptop and use it to play a short game for a few minutes while your group speaks briefly about your work and the features you chose to implement. Some members of your group must be present at the lecture on the last day of class and speak briefly about your project to receive credit for this milestone. Your project should be generally finished by this date so that it can be demoed successfully. But your source code does not have to be 100% complete for submission until the following evening as described in the next milestone.

Milestone 4: Final Code (Saturday, June 4, 11:30 PM) (60% of grade)

On the weekend after the last lecture demo, 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 product and manipulating its graphical user interface. Exceptions should not occur under normal usage. Your code should not produce any console output, such as `System.out.println` statements in your code that were not specified.

Milestone 5: Group Peer Evaluations (Sunday, June 5, 11:30 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. 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.