Com S 227  
Fall 2013  
Assignment 3  
300 points  
Due Date: Monday, December 9, 11:59 pm  
“Late” deadline (25% penalty): Tuesday, December 10, 11:59 pm

General information

This assignment is to be done on your own. If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions; see below.

Your work will not be accepted for credit unless we have on file a signed “Acknowledgment of Academic Dishonesty Policy.” These are normally completed as part of Lab 2; otherwise, you can download and print it the form and give it to your instructor or TA:

Please start the assignment as soon as possible and get your questions answered early. Reading a technical specification like this document is not like reading a story from beginning to happy ending. You will probably have to read it all multiple times before things start to click. You can start writing and testing code before you fully understand all the details, though. There is a “Getting Started” section towards the end of this document that might have some useful advice.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder assignment3. If you don’t find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag assignment3. But remember, do not post any source code for the classes that are to be turned in. It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled “pre” to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post “private” so that only the instructors and TAs can see it. Be sure you have stated a
specific question; vague requests of the form “read all my code and tell me what’s wrong with it” will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled “Official Clarification” are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

**Test code and test cases**

You are not required to submit any test code for this assignment, so any such code may be freely shared. You are encouraged to post any questions you have about the correct behavior of any of the methods, and to post ideas for test cases for others to review and comment on.

**Introduction**

For this assignment you will implement a number of classes for an implementation of a dice game based on the game *Yahtzee*. The purposes of this assignment are:

- To give you a chance to make some design decisions related to inheritance
- To give you some more practice using arrays and ArrayLists

**Summary of tasks**

You will implement the following classes in package `hw3`:

```
Game
DiceGroup
```

and, *at a minimum*, the following eight classes in package `hw3.categories`:

```
ChanceCategory
FullHouseCategory
LargeStraightCategory
```
SmallStraightCategory
CountOccurrencesCategory
YahtzeeCategory
YahtzeeLessOneCategory
YahtzeeLessTwoCategory

In addition, you will implement, in package `hw3.categories`, whatever additional classes you determine are necessary in order to exploit inheritance to facilitate code reuse. See the discussion of scoring categories below for more information.

The exact definition of each of the eight scoring category classes listed above can be found by reading the class javadoc in the skeleton code.

**Overview**

Yahtzee is normally played with five six-sided dice and a scorecard. One turn consists of rolling the dice up to three times and then recording the result in one of the categories on the scorecard. In our version, the number of dice, the range of numbers on the dice, the maximum number of rolls, and the categories on the scorecard will all be configurable.

If you are not at all familiar with the game, don’t worry, it is not too complicated. Take a look at the Wikipedia page for an overview, [http://en.wikipedia.org/wiki/Yahtzee](http://en.wikipedia.org/wiki/Yahtzee)

The two key abstractions in the design are *dice groups* and *scoring categories*.

**Dice groups**

A dice group, represented by the `DiceGroup` class that you will implement, is a basically a list of integers representing the states of N dice (in the standard game there would be 5 numbers 1 through 6). However, the dice are partitioned into two lists: *available* dice and *completed* dice. Initially, all dice are available. When the dice are rolled, random values in the appropriate range are generated for the available dice only; the completed values are fixed. If the maximum number of rolls has not yet been reached, the player can select some of the dice values to be moved to the completed list. Those values are now fixed, and will not be modified with the next roll. Note that in this design, there is no such thing as an individual “die” object; a die value is just an integer. All methods that return arrays containing dice values just return the values in ascending order.

After the maximum number of rolls is reached, all dice are automatically moved to the completed list and the dice group can no longer be modified. A new dice group must be created for the next turn.
See the online javadoc for details. Also note that the client normally obtains a new `DiceGroup` using the method `createNewDiceGroup` in the `Game` class, not by calling the constructor directly. (You can see how this works by looking at the `doOneTurn()` method of the sample UI.)

**Dice group example.** Suppose we represent a dice group as a string by listing first the available dice and then the completed dice in parentheses. For example, in a game with 5 dice, after the first roll we might see

\[
2 \ 3 \ 3 \ 4 \ 6 \ ()
\]

Depending on which categories you need to satisfy, you might select 2, 3, and 4 (perhaps in the hope of completing a straight). Now you have

\[
3 \ 6 \ (2 \ 3 \ 4)
\]

The 6 and the other 3 are then replaced by random values on the next roll, but the 2, 3, and 4 you selected remain fixed. If (for example) you rolled a 2 and a 5, you would have

\[
2 \ 5 \ (2 \ 3 \ 4)
\]

At this point you could select the 5 (to make a small straight, hoping that rerolling the 2 might give you a 1 or 6 to make a large straight). Now we have

\[
2 \ (2 \ 3 \ 4 \ 5)
\]

If, as luck would have it, you then rolled a 4, you’d have

\[
(2 \ 3 \ 4 \ 4 \ 5)
\]

Since (in the traditional game) we only get three rolls, the 4 is automatically put into the completed dice since we can’t reroll it.

**Scoring categories**

A scoring category represents one row of the score sheet. It stores the actual score for the category and the dice group that was used to satisfy the category, and it contains the algorithms needed to a) determine whether a given dice group satisfies the category, and b) determine what the potential score would be for a given dice group.

There are many different possible categories, each with its own particular algorithms. For example, the traditional game has a three-of-a-kind category: a dice group satisfies the category if it contains any three numbers that are the same, and it is scored by summing the dice. The traditional game also has a “large straight” category: a dice group satisfies the category if it has 5 consecutive values, and it always receives a fixed score of 40.
This is where polymorphism becomes useful. The client using this code (e.g., think of the client as the text-based UI provided in the sample code) does not care about the details of what the categories are or how to calculate the scores. It just needs to be able to invoke methods on a category to find out whether a given dice group satisfies it, what the score would be, and to inform the category whether it was selected for a given dice group.

Therefore, a scoring category is defined by an interface IScoreCategory. This interface is already written and you should not modify it. See the javadoc for detailed descriptions of the methods. See the text-based UI to see how it is used from the client’s point of view. In particular, you can see in the printCategories method where the UI just iterates over the categories and displays the potential score or actual score from each one.

There are eight concrete classes that you are required to implement in package hw3.categories, as listed above. Each of them must implement the IScoreCategory interface. However, it need not do so directly: it could instead extend some other class that implements IScoreCategory. If you just added the declaration implements IScoreCategory to each of these classes and then proceeded to fill in all the required methods, you would find yourself writing the same code over and over again. Even though there are several different algorithms involved (e.g. three of a kind vs. a large straight), there is also a lot in common between the classes. You should carefully think about how to design an inheritance hierarchy so that you can minimize duplicated code between the classes. You might think about starting with an abstract class containing features that are common to all category types, such as isDone() or getDice(). There are additional opportunities for sharing code to think about too. Are there code similarities between “large straight” and “small straight” that you can exploit? How about between three-of-a-kind and four-of-a-kind?

A portion of your grade on this assignment will be determined by the logical organization of classes in this hierarchy and by how effectively you have been able to use inheritance to reuse code.

Configurations

(Note that the class GameConfiguration discussed in this section is already implemented and you should not modify it.)

The design allows for many similar games to be implemented using the same basic infrastructure. You have seen above that a DiceGroup can be constructed to use, for example, a group of fourteen 9-sided dice allowing the player 5 re-rolls. The traditional game rules also provide a 35 point bonus if the total score for the “upper section” is greater than or equal to a cutoff value of 63. Therefore, we make these two values configurable as well. (The traditional game also has a bonus for additional “yahtzees” (all dice the same) beyond the one that satisfies the yahtzee scoring category, but we do not implement this feature.) All of these values are
stored in a `GameConfiguration` object, and you can see from the javadoc that these five constants are set in its constructor.

**Game**

A `Game` instance is simple: it encapsulates a `GameConfiguration` and two lists, one for the categories in the upper section, and one for the categories in the lower section. There is no other state information required, since each category object keeps track of the score for that category. The `Game` constructor requires a `GameConfiguration` object. The methods for returning scores just need to iterate over the scoring categories to total things up.

**ExampleGameFactory**

(Note that the class `ExampleGameFactory` is already implemented. However, you can modify it as needed for testing and experimentation.)

This class has some static methods for creating various games. You can take a look to see how the game configuration is constructed and how the categories are added to a `Game` object. This class is used in the sample UI.

Note that in the sample code, all the calls to the constructors of the `IScoreCategory` implementations are commented out. That’s because this code won’t compile until you actually have implementations of the classes that are named there. As you complete the implementations, you can start uncommenting the constructor calls to try them out (and uncomment the import statements too).

**The sample code**

The sample code is an Eclipse project that you can import. See the instructions given for homework 2 if you don’t remember how to import a project.

There are “complete” skeletons present for `DiceGroup` and `Game` classes in package `hw3`. In the package `hw3.categories` there are partial skeletons for the 8 required concrete subtypes of `IScoreCategory`. Each of these classes has a descriptive javadoc comment and a stub for the required constructor. *You will need to add an implements or extends clause to each of these class declarations to make it a subtype of `IScoreCategory`, and add whatever implementation code is necessary for your design, as discussed in the above section “Scoring Categories”.*

**About the UI**

There is a sample user interface `TextUIMain` provided in the default (top-level) package of the sample code. *This code will not run until you have implemented the `Game` and `DiceGroup`
classes, and you will not be able to play a game until you have started implementing the necessary category classes in the `hw3.categories` package. As you implement and test the categories, you can uncomment the corresponding constructors (and add relevant import statements) in the `ExampleGameFactory`.

This is a text-based UI using clunky console I/O. It is not as slick as a graphical UI but has the advantage that the code is entirely comprehensible, so you can read it to see how the other classes in the application are used. A typical screenshot of a game in progress is shown below:

Potential scores for this roll:

<table>
<thead>
<tr>
<th>Score</th>
<th>Category</th>
<th>Total</th>
<th>Dice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Aces</td>
<td>3 (1 1 1 2 3)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Twos</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Threes</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Fours</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Fives</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Sixes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3 of a kind</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4 of a kind</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Full House</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Small Straight</td>
<td>30</td>
<td>(3 3 4 5 6)</td>
</tr>
<tr>
<td>10</td>
<td>Large Straight</td>
<td>40</td>
<td>(2 3 4 5 6)</td>
</tr>
<tr>
<td>11</td>
<td>Yahtzee</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Chance</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

-----

Upper section total:  3 (3 plus bonus 0)
Lower section total:  70
SCORE:    73

You rolled 3 5 (3 3 3)

a) keep all remaining
b) reroll all remaining
c) select dice to keep
Your choice:

At this moment in the game above we have 3 points in the upper section for the “Aces” category and 70 points in the lower section for the “Small Straight” and “Large Straight” categories. The dashed lines in the second column indicate the categories that are done, and the rightmost columns show the score and the list of dice that were applied to the category. The “You rolled…” shows the current state of our dice group. Since the parentheses are nonempty, this is not the first roll in this turn. We must have previously chosen to “keep” three 3’s from the previous roll, and we just rolled another 3 and the 5. The second column in the list shows us
what potential score we would get in each category for this roll as it stands (e.g. 17 points for four-of-a-kind). One of our options might be to “keep” the additional 3 and reroll the 5 in the hope of getting a Yahtzee of all 3’s.

You can edit the main method to a) choose a seed for the random number generator, in case you want the dice results to be reproducible while you are developing the code, and b) to change the game type.

**Suggestions for getting started**

1. The *Game* class is very straightforward and doesn’t really depend on anything else.
2. *DiceGroup* can also be developed and tested independently. Note that the roll() method takes a Random as an argument, so for testing you can specify a Random with a known seed to make the results reproducible.
3. Any of the scoring category classes can be developed and tested with a simplified, partial implementation of *DiceGroup*. All they actually require is the `getAll()` method. For testing purposes you would also want to implement the *DiceGroup* constructor that allows you to specify starting values for the dice. Those two things are sufficient for testing the score category classes.
4. For the scoring categories, you might start with something like *CountOccurrencesCategory*. Add the clause `implements IScoringCategory` and stub in the required methods and start to implement them. (You may later move some of this code into an abstract superclass, but this is a good place to begin.)
5. Once you have *Game*, *DiceGroup*, and *CountOccurrencesCategory*, you can uncomment the constructor in the `createReallyTinyGame()` method of the *ExampleGameFactory*. Then, you can run the UI and play the Really Tiny Game!
6. Start thinking about the other scoring categories and start writing implementations for some of them. Soon, you will get tired of writing the same code over and over again, and you can think about how to organize it better using inheritance. Create an abstract superclass with code that is common for all the categories. Then look for other similarities. (e.g., Can YahtzeeLessOne share some implementation with YahtzeeLessTwo? Is LargeStraight similar to SmallStraight?)
7. As you complete and test each of the scoring categories, uncomment the relevant constructor from one of the games in *ExampleGameFactory* and try it out.

**Special documentation requirement**

You must add a comment to the top of the *Game* class with a couple of sentences explaining how you decided to organize the class hierarchy for the scoring categories.
Style and documentation

Roughly 15% of the points will be for documentation and code style, including the “Special documentation requirement” described above. Please refer to the “Style and documentation” section of homework 1 for general guidelines about documentation. Note the following for this assignment in particular:

- Add @author tags to each file that you create or modify
- You do not need to rewrite existing javadoc in the skeleton code. You do need to javadoc instance variables and helper methods that you add, as usual.
- You should javadoc any new classes you add to the hw3.categories package
- Methods that are documented in an interface or superclass normally do not need to be documented again when you implement the interface or extend the class (unless you have substantially modified the behavior to the point that the original documentation is incorrect).
- Some of the loops you write may be tricky. You are expected to provide internal, //-style comments to explain what you are doing. If you had to think for a few minutes to figure out how to write the code, it probably needs a comment to make it clear to the reader! Internal comments always come BEFORE the code they describe and are indented to the same level.

What to turn in

Remember to CHECK your submission after you upload it on Blackboard by looking at your submission history. You WILL lose points if your submission is incorrect.

Download the zip file you submitted and check it carefully. Extract the files into a clean, temporary directory and look at them before you submit. Are they .java files? Are all the required files present in the archive? Are they in the directory hw3? Are they the latest, working versions of your code? Did you accidentally submit just the skeleton code?

Please submit, on Blackboard, a zip file containing all of your source code (the .java files). Please do also include the api package (the UI and ExampleGameFactory do not matter.) When you check it you should see, at a minimum:

1. A directory hw3 containing the files Game.java and DiceGroup.java
2. A subdirectory hw3/api containing IScoreCategory.java and GameConfiguration.java
3. A subdirectory hw3/categories containing the eight required implementations of IScoreCategory AND any additional classes you have implemented to create your inheritance hierarchy.
Normally it is sufficient to right-click on the `src` directory of your project and select “Send To -> Compressed/zipped file”. *Do not zip up the entire project, just the src directory!* It is entirely acceptable to include additional test classes you have created, and to include the UI code. Use the default Windows or Mac zip utility, *not* Winrar or 7zip or whatever.

If you are not sure how to do any of these things, see the document “Assignment submission HOWTO” which can be found on the Blackboard Assignments page or linked on Piazza.