The Game of Games – Implementation Time
(In Java)
Updated due Date: Sunday, December 4, 11:59 p.m.
Progress documentation must be available constantly
Implementation Phase: 300 pts

IMPLEMENTATION TIME!

Each group will be receiving the Use Case and Activity Diagrams designed by another group. This phase will reveal the joy and pain which can result from implementing another group's design and will be a test of the completeness and clarity of those designs. **Note the test tables are NOT being distributed nor should any design team send their test tables to the implementation team.**

Remember also that the theme of this entire class is managing and communicating expectations, ideas, responsibilities, and results throughout the entire lifecycle of a project. Documentation is absolutely vital to managing and creating an up-to-date as well as a lasting record of that process. While the implementation of this project is not complex, all of the required documentation surrounding it illustrates the infrastructure needed to support and record the implementation process. So if, down the line, anything needs to be updated, expanded or corrected, there is a paper trail that will be an enduring record to assist in maintaining , altering, and verifying the current and updated states of the project.

COMMUNICATION AND ORGANIZATION

Each group must have all communication (other than what happens in your own group meetings, which is captured in your meeting notes) through **Slack**, and I must be included in your groups. The group whose design you are implementing must also be included in your slack group, because, if there are any ambiguities in their design, you should be contacting them to get an answer via Slack with a "paper trail" of its resolution. So if you have questions, **you need to contact them through Slack**. If I see an implementation that does not match a design with no evidence of how it got to that point, it will be a problem.

Remember, however, the goal is to implement the design you are given, not to have the designers change their design to match the vision of the implementers. So for example, if their design involves the user typing in "heads" and "tails" for the Coin Flip game, "Can we change the input to a menu where the user picks 1 for heads and 2 for tails?" in NOT a legitimate clarification, whereas "Does upper or lower case matter?" is a legitimate

clarification. And when that clarification is made, the **designers should update the design documents** to reflect that clarification is needed. Remember, paper trail!

I play two roles: (1) the mile-high manager, who will keep an eye on things from far away but is not involved in the details and (2) the customer, who can be consulted if need be to clarify ambiguities regarding expectations. If there are any questions for me the customer or me the professor, send them to me via email (clearly, your customer should not be on slack seeing all the gory details of what is going on with their implementation).

DOCUMENTATION

Your implementation must be accompanied by the following forms of documentation, all of which are described in detail below:

- ***Planning and Progress*** documentation
- ***A Code Design*** document consisting of:
    - Per class**: UML Diagrams**, with **Method Glossary** and **Data Configuration Table** and per **Method Unit Tests** associated with each class.
    - A **unified Data Configuration Table** for the entire project organized by class
    - **Unit test** table
- Use **GitHub** for managing the versions of your project and also providing the final version of your code
- A ***Deployment Document*** indicating how the code for your program can be accessed, downloaded, configured and run. Your GitHub should also have a README defining all the files in the directory.
- A **Responsibilities document** indicating, when the implementation is complete, who did what

PLANNING AND PROGRESS DOCUMENTATION

- The management of your implementation **must be via Trello (Kanban!) or Monday (trello.com or Monday.com).** They both have template examples for agile, as well. As the mile-high manager, I should be in on those groups as well. **This documentation is to be visible constantly and changing as things are being**

assigned and completed. It should not just be constructed after the project is complete.

At a minimum I should see:

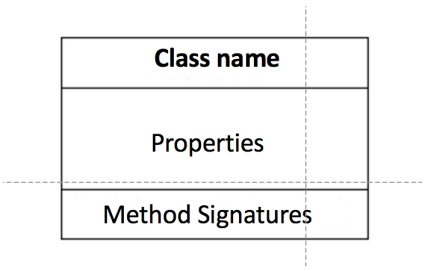
- A Gantt chart of the time line and assignments and projected completion times (in advance, not after the fact), which of course may change.
- Task assignments per person that show the up-to-the-minute status toward completion, with notes etc. if appropriate.

CODE DESIGN DOCUMENTS

****All projects must be written in JAVA****

UML diagrams

- For each class, having the format (consult class text or see https://www.edrawsoft.com/article/class-diagram-relationships.html?gclid=Cj0KCQiAveebBhD_ARIsAFaAvrGKBrpO7zxCTaHI9_r7uhegqD2glSLzn05Am5jcRC4dUZ1JpueqPc0aAsQTEALw_wcB



Method Glossary (organized by class and accompanying the class diagram above)

Must include:

- a generic form of the method signature, where parameter **types** are specified not names
- the name of the method must be in bold
- the purpose of each parameter

- the purpose of the return type if any
- the general purpose of the method

For example:

public int **purchaseChicken**(String)

param1: String, accepting the breed of a chicken
return: integer, representing the number of chickens purchased
purpose: accepts a breed of chicken, which is used to prompt the user to enter the number of chickens they want purchased, which is returned to the calling module.

Data Configuration Table

The data configuration information should accompany the class diagrams above AND ALSO be collected in a single table for the entire project (so, for example, it's easy to see if the coder for one class is showing the user "H" for heads and "T" to Tails, while another class is showing the entire words "HEADS" and "TAILS"). Be sure, in the unified table, that the classes where the data reside are clearly identified.

The data configuration information should be divided by class and method and show, for any important value, how it is represented. "Important" means it is key to the problem being solved. So, for example, how a spool of thread or the result of a coin flip is being represented would be important, a loop control variable, which would obviously be an integer, would not be. Such variables should be clearly understandable via the in-code documentation.

The information for each "important" variable would include:
- The generic name, with a corresponding variable name(s) in bold if appropriate
- The Input Type/Format, if appropriate
- The Output Type/Format, if appropriate
- The Internal Representation

For example: Consider this partial table for a card game, in which the user card values are input from a file and card values are also displayed to the user, and two internal variables for managing the various structures needed to simulate the game.

Value & Variable	Input Format	Output Type/Format	Internal Representation

Playing card	Two-character sequence, the first character representing suit with ‘H’, ‘C’, ‘D’, ‘S’, the second character the card value using 1-9, 0 for 10, and ‘J’. “Q’, ‘K’ or ‘A’.	String in the form “*value* of s*uit*”, for example, “4 of Clubs”; face cards have the name of the card fully spelled, for example “King of Hearts”	Two-character sequence, the first character representing suit with ‘H’, ‘C’, ‘D’, ‘S’, the second character the card value using 1-9, 0 for 10, and ‘J’. “Q’, ‘K’ or ‘A’.
Deck of cards for dealing, **deck**	N/A	N/A	Linked list of cards
Discard pile, **discards**	N/A	N/A	Stack

By having the above table to consult, if someone decided their code needed to pull a card out of the middle of the **discards** pile, they could see that plan would not be possible in the current design as **discards** is a stack. Or if it was necessary to remove a card from the middle of **discards**, the internal representation of the discard pile would need to be changed. If, on the other hand, someone’s code was displaying cards to the user in the 2-character format, a colleague could tactfully point to the design document that clearly states the output should be in form of “*value* of *suit*”, thus tactfully indicating the error of their colleague’s ways.

Unit Tests

In a table format similar to that used in developing your black-box (use case-based) testing document, for every method, there is to be a listing of unit tests done to determine that the method is working correctly. Remember, unit tests should be performed on a module **before** it is integrated and, although it need not be documented in the table, often includes writing extra code for purposes such as calling a method under specific circumstances, writing a stub method that will only return a value for testing rather than implementing the full method to generate the value to be returned, or assigning a value to a variable that would normally get its value randomly. This last purpose is needed to make sure that you games are correctly implemented without needing to play them over and over until they just happen to randomly generate the behavior you want to test. Such code will of course be removed after unit testing is complete. For each test (row) in the table indicate:

- Class/Method being tested
- General purpose of test
- How it will be tested (if input based, what is the input, if addressing parameter passing, what parameters will be passed, if condition based)
- Outcome
- Passed
- Failed (they should hopefully all be marked passed!)

Note that, unlike with the black box testing document, where correct/incorrect were not filled in as the tests couldn't be conducted until the code was written, here the unit tests should be performed and eventually (hopefully) all marked as passed.

Class/Method	Purpose	Test Steps	Expected Result	Actual Result	Passed	Failed
Utilities/isOdd	Returns true if passed an odd int	Call passing 5	True returned	True returned	X	
Utilities/isOdd	Returns false if passed an even int	Call passing 4	False returned	False returned	X	
Utilities/isOdd	Returns true if passed zero	Call passing 0	True returned	True returned	X	

Deployment Document

This document details step-by-step directions indicating how the code for your program (variations) can be accessed, downloaded, configured and run.

CODE AND CODING

YOU ARE TO WRITE THE CODE AS DESCRIBED IN THE REQUIREMENTS YOU RECEIVE. If there are omissions or ambiguities, you are to contact the team who designed your program via **slack**. They own the design and the documents, so if the design needs to be changed, it is their responsibility to do so **and provide you with the updated**

documents. Think of it as if INFOSYS here in Hartford designed an interface for an insurance company website, sent the design to India to be implemented and then it came back with a different design that the implementation team in India thought looked better. Surprise! And not a good one. In our scenario, the team whose design you are implementing is INFOSYS in Hartford. If design changes need to be made, the design team needs to make them. And again, the implementation team is not to re-work the design team's design, only ask questions to address omissions or ambiguities.

EVERY STUDENT IS EXPCECTED TO CODE A MINIMUM OF ONE CLASS, with the exception that one student in every group may be in charge of organizing and creating external documents and group management. However, all members of the group must agree if the 1-person-does-not-code option is to be used. The coding is to be, as much as possible, evenly divided between group members. One or two people should NOT be doing all the coding. Remember, the external documentation must accurately reflect coding responsibilities and accomplishments. **If it does not, it is considered academic dishonesty.**

Code Configuration

Your code MUST conform to the following architecture/requirements:

- The code must be written in Java.
- Every game must be in a separate class.
- The driver loop that displays the menu and controls the overall progress of the Game of Games must be in its own class, named **PlayGames**.
- A **GetInput** class will contain all/only the methods to prompt (if appropriate) and get input from the user, insuring that the input is valid (correct type and in the value range specified). The methods should be written as generically as possible, so that the minimum number of methods will be required for all the games. These methods will return the correct anticipated value to the calling code. Also, your program should be resilient (gracefully recognize, handle and not crash) in the face of user input errors.
- Your code may, of course, have additional classes if you see fit.

Code Variations

The implementation must have 3 variations:

Player mode – provides the player experience

Test mode- in order to test the program, test mode displays information allowing the user to cheat by seeing information in advance of or during game play. For example, it should show the contents of the "box" of thread while the red thread game is being played, or by show what hand the thimble is in before the player needs to guess etc.

This mode allows the program to be tested without playing it over and over until the program randomly displays the behavior to be tested.

Bug mode- this version of the code will, like test mode, display the information that will allow the user to cheat for testing purposes. However, it will also be corrupted by your team so that it has 3 bugs seeded into the code that the black box testing already created by the designers should be able to reveal. Make them interesting but still able to be found via test runs. They should be the type of bugs that might naturally occur in coding. This mode is also the reason why the design documents you received are missing the test cases. Since you don't know the test cases (and DO NOT ask for them), the bug mode is essentially your test of the design team's test cases.

In-code Style and Documentation

Your programs must be written using good programming style. Your code must be self-commenting with meaningful variable names. Every class and method must have a heading comment briefly describing what it does, as does any significant section of code within a method.

Also, be sure that literal values (like the number 12 or the letter 'T') do not randomly appear in your code but are stored in final variables with meaningful names so they are easy to find and easy to change. Appropriate data structures and data representations should be chosen and the same code should not appear in different places. Proper and clear indentation, and blank lines to improve readability, are required. **The quality, clarity and formatting of your input and output will also be judged.**

Drop in Moodle Folder

- all of the code (each variation (Player, Test, Bug) zipped separately)
- a single, unified PDF file containing **all** of the documents including the deployment and responsibilities documents, clearly organized and with a uniform look and feel **and a table of contents**
- the Deployment document separately in its own pdf file, which also includes where you code can be accessed.
- the Responsibilities document separately in its own pdf file
- all meeting notes