COMS W1007 Assignment 1
Due Sept. 26, 2019

1 Theory Part (50 points)

The following problems are worth the points indicated. They are generally based on the lectures about software development, data structure selection, and the early parts of Clean Code.

The instructor is mindful of the necessity for learning the new infrastructures used in the course (Courseworks, Piazza, Eclipse), and has taken this cultural transition into account. This assignment therefore gives more credit than will be usual for the programming section, and it has more interconnections between the Theory Part and the Programming Part.

"Paper" programs are those which are composed (handwritten or typed) on the same sheet that the rest of the theory problems are composed on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work for paper programs beyond the design and coding necessary to manually produce the objects or object fragments that are asked for, so text editor or computer output will have no effect on your grade. The same goes for the Theory Part in general: scanned clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

An important note about academic honesty: If you use *any* resource, including `stackoverflow.com`, you must properly attribute the source of your answer with the proper citation. Cutting and pasting without attribution is plagiarism. The CAs will electronically and physically check for cheating, and will report to the instructor any suspected violations. This even goes for `wikipedia.org`–which for this course often turns out to be inaccurate or overly complicated, anyway–since plagiarizing from Wikipedia is still plagiarism. Much better to read, think, and create your own sentences.

1.1 Conflicting design requirements (6 points)

Consider how a hungry person would decide on which eating establishment to choose for a meal, based on elements of its design and services. List six design considerations that are conflicting, like we did in class with the examples of a car. Show for each their analogy with a feature or consideration of Java classes. For example, "buffet" is like "API". Is there a "best" restaurant? Explain.

1.2 Hello World again (4 points)

Extend the class NameMaker with two more constructors:

a. A "natural" default constructor

b. A constructor that also allows an optional username, like your Columbia UNI

1.3 arrays, ArrayLists, LinkedLists (6 points)

Using the cheatsheet, explain which of the three data structures of a[], AL, LL–or which nested combinations of them–are best for use in the following scenarios, giving a reason or an example of its use:

a. A suitcase

b. A tie, scarf, or belt rack

c. A roll of paper towels

d. That part of a wallet that holds bills, which the user keeps sorted by denomination.

e. A waiting line for an event, but where people sneak their friends into the line wherever they happen to be in the line.

f. A silverware drawer, which has separate bins for forks, knives, etc.

1.4 Bad user requirements (6 points)

Critique the following use case, by listing questions you would go back to ask the user, in order to clarify the user's intentions. Look for classes, methods, fields, performance requirements, parallelisms, sequencing, inheritances, aggregations, incompletenesses, redundancies, etc. You only need to give six questions; no extra credit for more than six.

> I want to buy one of your robot pizza things for my store. Here is what I want it to do. When customers come, it should take their orders for pizzas, some cheap toppings, some premium toppings, and the drinks. Everybody has to buy something. Then it makes the orders up, and cooks them, maybe three or four at once in my ovens, and at the same time it folds up the boxes and does the drinks or the water. Then it takes the money and makes change if needed, then shoos them out so that new ones.can come in, unless it is closing time.

1.5 Good user requirements and CRC, UML (6 points)

Show the CRC, and the UML (with proper fields, methods, and connecting arrows) for the following use case:

> The vending machine holds packages of snacks on a collection of spiral coils. It accepts credit cards and mobile payments. It has a display that shows important messages, and a panel of buttons that allow a user to select the desired snack. The machine knows the price of each item. It either dispenses the snack or displays an error message, and adjusts the bank account.

1.6 CRC for RPSLK (6 points)

Give your CRC for the Programming Part, depending on how far you got. These can be free-hand, on standard paper, and you don't have to use actual 3x5 cards. There is *no* extra credit for using any design aids. Please do these *before* you start the coding of each Step! You only have to turn in one set, in the final form that documents whatever you developed and submitted in the Programming Part.

1.7 UML for RPSLK (6 points)

Give your UML for the Programming Part, depending on how far you got. These can be free-hand. There is *no* extra credit for using any design aids. Please do these *before* you start coding! You only have to turn one set, in the final form that documents whatever you developed and submitted in the Programming Part.

1.8 Clean code for RPSLK (8 points)

Justify the cleanliness of your code in each of the following ways.

a. Pick out three names you used in your solution. Show how each of them follows the suggestions in Chapter 2.

b. Pick out three methods. Show how each of them follows the suggestions in Chapter 3.

c. Pick out two comments. Show how each of them follows the suggestions in Chapter 4.

2 Programming Part (50 points)

2.0 Overview

2.0.1 In general

This assignment is intended to walk you through the design process, using a console application built on straightforward Java. It will also serve as a baseline to record your design proficiency at the beginning of the course, since that you will use part of your solution to this assignment in Assignment 5. So try to make it as clean, well-documented, and flexible as you can, as you will have to adapt your design later in the course in ways that are as yet unknown to you.

Basically, this assignment provides a requirement specification, which is very wordy, together with some friendly advice, which you are free to ignore. It asks you to do the CRC, UML, documentation, algorithm and data structure selection, coding, and testing of a relatively simple system. None of this assignment requires any graphical user interfaces, but that will come later in the course. Nevertheless, the system will show behaviors that may be difficult to predict.

Further, this Programming Part shows in miniature, step by step, what happens when a system is successful and the initial design has to be extended. Yes, if you read ahead through all the Steps of this assignment, you can have an unfair advantage over the real world, in that you will be able to see the future of your product perfectly. But you probably should want to design, document, code, and test by following the Steps in the exact order anyway.

Remember: it is always much better to have a saved version of a working system that does only some things well, rather than to have a non-working system that has tried to do everything all at once.

2.0.2 In specific

Your eventual goal is to implement the five-gesture game of Rock, Paper, Scissors, Lizard, Spock ("RPSLK"), and to give it at least a few bits of artificial intelligence. But you will do this in five Steps:

In brief: The first Step has your program play the traditional Rock Paper Scissors ("RPS") game against a real human, probably you, or maybe a friend. Although your program plays randomly, it should win more often than a human does. The second Step adds some AI, and it should win even more often against a human. The third Step levels up the game to play the full RPSLK, with the same AI, against a human. The fourth Step replaces the human with a *simulated* human, so that you can get many games played quickly and gather statistics on both your program and your simulated human, and this Sim gets tuned to be really good. The fifth and final Creativity Step uses the Sim to help your refine a better AI-based strategy according to your personal design, and demonstrates that your newer AI it is better (or at least not worse) than the original AI you came up with in the second Step.

In summary:

- Step 1: RPS, human player, random computer
- Step 2: RPS, human player, AI computer
- Step 3: RPSLK, human player, AI computer
- Step 4: RPSLK, Sim player, AI computer
- Step 5: RPSLK, Sim player, creative AI computer

2.0.3 Resources

For a quick summary of the rules of RPSLK, see:

`https://www.youtube.com/watch?v=x5Q6-wMx-K8`

For a very nice diagram, which illustrates the rules given above in exact order, and which also shows that there are five smaller three-gesture games embedded within RPSLK, including the traditional RPS, see:

`https://en.wiktionary.org/wiki/rock-paper-scissors-lizard-Spock`

A more formal presentation is by its creator, at:

`http://www.samkass.com/theories/RPSSL.html`

Just to bring this all into the real world, see the following article, which shows how this "game" has life and death consequences, at least for real lizards. We won't be using this resource, but it is fascinating nonetheless–and the text of the article's abstract even starts with Java's favorite word, "polymorphism"! Use your browser to search for the word "orange" to get to their algorithm, at:

`http://www.pnas.org/content/107/9/4254.full`

Please again recall that all programs must compile! So, the good practices of agile programming say you should always keep a working version of each Step before proceeding to the next, even if it is not functionally complete. Eclipse can help with this; look up its “local history” feature.

2.1 Step 1: Basic interactive RPS system (16 points)

2.1.1 Specification

For this Step, you should start small, with a console application that uses text output and text input, and that just plays the original game of Rock, Paper, Scissors with a human. The application should do something like the following.

A Talker class first displays some welcome text that give the rules. Then, for 100 consecutive times, a Thrower class selects the computer’s own move, or “throw”, and a class (which one?) asks the human for a single character from ’r’, ’p’, or ’s’. When a valid character is thrown by the human–you must design a loop to do sanity checking!–some class (which one?) prints out a message indicating the two throws, and who won, and why, roughly: “I win! My paper covers your rock!”. Note that there will be nine such messages (and in later Steps, 25), so use some form of aggregation to simplify them.

We will assume that the computer is honest, and that it really does its throw without looking at the human’s throw, and that the human trusts both the designer and the computer. Note that since you, as designer, know *exactly* the rules of the game before hand, you can hard-code these rules about who wins, as compact *data* (how?) rather than as long and nested executable *code*: this is again an example of aggregation. You should make this effort now in Step 1, because the design will soon be expanded, and you will appreciate any expansion that is simple and easy.

Just after the game ends, some class (which one?) computes how many rounds of the game the computer won, how many the human won, how many were tied, both as counts and as percentages. This is then displayed by some class (which one?)

To pull this all these classes together, have a short, simple, solution-oriented Runner class, with main(), which shows that your other classes work as required and as documented. Among other design issues, you probably should consider that the strings “rock”, “ paper”, and “scissors” have been determined by the real world, and, like the rules, are not negotiable. These strings probably should be encapsulated in a class (which one?). The various input and output specifications are likely to change, too, so make sure the Talker class design is extensible, even if there are no graphics involved yet. And make sure that it can handle sloppy or ignorant or malevolent inputs. And, even though the Thrower’s throws must be random in this Step, that strategy is certainly going to augmented, so make the Thrower class design extensible, too.

2.1.2 Deliverables

Once the human plays the game, it is likely that you, the designer, will observe that the computer–even by playing randomly–tends to win more than the human does, since real humans tend to “get stuck” with foolish misconceptions about strategy. This is very likely to happen even if the human player is you, the designer. This phenomenon is known as the “Gambler’s Fallacy”: the erroneous belief that somehow a coin, or a die, or deck of cards, or a random number generator “knows” that it has used "too much" of certain values, so it is more likely that the next move it will use one of the "neglected" values. Similarly, the ties in the game, which should be approximately 1/3 of the total number of rounds, probably will turn out to be less frequent than 1/3. Working also in this direction is that humans are really bad at generating random behavior, possibly because there was never any evolutionary advantage at being good at it. After the game, i t would be good to record your observations in the Javadoc comment(s) (where?).

If you stop at this Step, turn in your *.java files (generally, this means whatever classes you wrote in the Eclipse project), a trial run captured from the console output, and a summary of the game: number of throws, wins, ties, losses.

Submit them to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods, since a good part of your grade will depend on them. *Do not* use a README file, since anything that would go in the README should go into the Runner class instead, where it is less likely to get lost or get out of date.

Note that each Step builds on the previous Steps. So, if you continue to further Steps, you don't have show the design, documentation, or code for Step 1 *explicitly*, just as long as you eventually submit a final system that shows the documentation, code, and testing for that *higher* Step, including a summary of the game.

2.2 Step 2: Extending your system with simple AI (10 points)

2.2.1 Specification

Now, design, document, and write an enhanced and artificially intelligent Thrower class. This Thrower won't be particularly smart. It simply looks to see how often the human has thrown 'r', 'p', and 's' during the game so far, and exploits the human inability to be totally random. So, for example, if the AI version of Thrower notes that the human really likes Rock, it will tend to suggest a throw of Paper. Let's call this the "Recorder" strategy, and it should be better than the simpler "Random" strategy.

This means at this Step, Thrower has to be augmented, possibly by another class, to have some methods for learning the game's history: by recording some useful information derived from the history, by computing the "best" throw, and by outputting its decision. All of this should again be explained and justified in the Javadoc. It is also your job as designer to select the appropriate data structures (a[], AL, or LL) and algorithms for dealing with this history. Again, you must test your system by its playing 100 rounds with a human, and reporting what happens during this testing, in Javadoc somewhere, as in Step 1. Note that at this Step, Thrower still has access to two Step-dependent strategies: Step 1 Random, plus Sept 2 Recorder. Even if you won't be using the Random strategy, don't delete it; it may come in handy later.

2.2.2 Deliverables

If you stop at this Step, you must turn in the same kinds of things described at the end of Step 1, above: *.java files, trial runs, no README. If you submit these things for a system that does both Step 1 and Step 2, you don't have to submit anything explicitly for Step 1 alone. Similarly, if you go beyond Step 2, you don't have to explicitly show that the systems you built for Step 1 and for

Step 2 work as standalone systems–you are designing just one system that gets extended step by step.

2.3 Step 3: Extending your system to play RPSLK (6 points)

2.3.1 Specification

The system now should be extended, except it now plays a smart version of the full five character game. This may require some resizing of your data structures. But if you designed Steps 1 and 2 properly–particularly since you had the unfair advantage of exactly knowing the future–then those changes should be small and localized. Make sure you clearly comment any extensions you make!

One way to evaluate how good your design is, is to ask if it could easily be extended further, in order to handle something like "RPS-25", which is described at:

`http://www.umop.com/rps25.htm`

or even "RPS-101", shown at:

`http://www.umop.com/rps101.htm`

Note that the game in general must always have an odd number (why?) of characters, c, and that there are $\mathcal{O}(c^2)$ possible pair-wise outcomes. That grows too big to encode the rules as code, even for small c. For RPS-25, there are 300 rules, and for RPS-101, there are 5,050 rules. For these use cases, data wins over code. If you look closely at the diagram for the 101-character game, you should get a hint as to how to organize your data, even for the 5-character RPSLK case.

To test this Step, you will again need to do about 100 rounds, and again to record your observations about the testing somewhere in Javadoc. You will need to be very patient. Alternatively, "the human" could be a friend of yours, perhaps even someone also taking the course. Generally speaking, testing someone else's code is not considered to be dishonesty.

2.3.2 Deliverables

If you stop at this Step, turn in what was described above in the first two Steps, except that everything now shows your Step 3 system. Note that Step 3 is really just augmented Steps 1 and 2, so the grade you get for Steps 1 and 2 will be determined in part by the design, documentation, code, and testing you provide for Step 3. Note that if Steps 1 and 2 were properly designed, Step 3 only introduce very small changes, which is why this Step is the easiest Step and gets the fewest points. But any modifications to accommodate the additional characters must be documented.

And, as before, if you continue on to Step 4, you only have to submit a single system–which must of course compile and produce test runs.

2.4 Step 4: Extending your system by simulating the human (9 points)

2.4.1 Specification

Having amused yourself and possibly your friend with the game, it is time to see if you can make your system do even better. But to do this, your system first will have to remove the slow and probably grumpy human from the testing loop, so that your system can run many more experiments on a *simulated* human. This way you and the system can more readily test out better strategies, which you will do in Step 5, but without any apparent human social costs.

So, this Step should augment the system somewhere (in the Talker class, maybe?) so that it doesn't have to wait for a human to input a throw. Instead, it interacts with a Sim class, which observes the game in progress, and executes a Sim "strategy", which outputs a Sim throw automatically. Note that the Talker class still is useful, and that it should not lose its ability to play a genuine human. Make sure you document this increased functionality in the Javadoc.

Now, even though simulations are often done in the AI world, it is not easy to simulate a real human. Sometimes what is done instead is to have the system play itself: look up "Generative Adversarial Network". But that approach is often very slow, and often not successful. (It took a *lot* of work to get AlphaGo to run well!). So what you need to do here is to use the following quick and dirty approximation to some deep learning.

We know that humans are not good at playing the Random strategy. So, make the Sim can have two basic *non-random* strategies when it plays against the system.

a. "Rotator" strategy: the Sim plays its throws in the order R, P, S, L, K, R, P, etc. This very simple strategy has the virtue that the Sim doesn't "get stuck", and keeps the system from discovering any favorite throw of the Sim (why?).

b. "Reflector" strategy: the Sim plays whatever throw the system played last time. This very simple strategy also has the virtue that the Sim doesn't "get stuck", and it may confuse a system that is keeping only simple statistics on the Sim (why?).

But we also know that humans are not completely stupid, and will try to fool the system. Therefore, build your Sim so that it takes one more parameter, N, which says how many times in a row the Sim sticks with either strategy, starting with the Rotator. For example, if N==20, then the Sim plays the Rotator strategy for 20 throws, then switches to Reflector for the next 20, then back to the Rotator, etc. If N==1, then the Sim switches strategies every throw. And if N is a very large integer, say, Integer.MAX_VALUE, the Sim starts with the Rotator strategy and sticks with it.

Since Sims are fast and don't complain, you can now do many experiments with you Sim playing against your system, looking for the "best" N, and documenting the results of your investigation. Note that you are no longer limited to just doing 100 throws per game. On the other hand, a billion throws per game is overkill. It is up to you, and part of your AI efficiency, to experiment to find what a "good enough" number of throws is, in order to tune your Sim's value of N. You need to document how you did your testing, and why you think your value of N is a good one, within your Javadoc (where?).

Note that now there are four strategies (Random, Recorder, Rotator, Reflector) that you as designer have to build. This brings up two important design considerations:

a. This Step helps demonstrate the utility of CRC and UML, since there will be some similarity, overlap, and dependencies within these strategies. These similarities are easier to see if they are dealt with up front and with documentation in the design phase, rather then being discovered and hacked over in the implementation phase. We will later see that this is an example of what is called "refactoring".

b. A necessary part of the design of this Step is to determine how best to do this extension *without losing* the original functionality of playing against a real human. We will later see that this is an example of what is called the "Decorator Pattern".

2.4.2 Deliverables

As before: if you stop at this Step, turn in what was described above in the previous Steps, except that everything now shows your Step 4 system. Note that Step 4 should only augment a very few classes, so the grade you get for Step 4 will be determined in part by the design, documentation, code, and testing you have already provided for Step 3. Still, any modifications to accommodate the additional functionality of Step 4 must be documented in the Javadoc.

And, as before, even if you continue on to Step 5, you only have to submit a single system–which must of course compile and produce test runs. Once again, do not remove prior code: you or your friend will probably want to play the improved system as humans. Once again, document the increased functionality. You may find it helpful during testing to have the system indicate additional internal information, like what is the (experimental) value of N, and what (at any given throw) strategy is being pursued. This extended debugging information does not have to show up as part of your submission, but if you do use it, *do not* delete it from your code: future maintainers will thank you.

2.5 Step 5: Creativity: extending your system with your own AI (9 points)

2.5.1 Specification

Now you have all the tools and experimentation you need to improve your system's Thrower. It already has two strategies: Random, and Recorder. But its opponent now is more formidable. So, add a third strategy, called Revenge, which uses more information from the game's history rather than just simple statistics. This strategy is completely up to you. But you need to make sure you explain it thoroughly in its Javadoc, and test it "enough".

For example, Revenge can look to see if there are any patterns in the Sim's *last two* throws, such as, "a Spock very often follows a Lizard". Or, it can see if the simulated human is reacting to the previous throw of the *system*, such as, "the Sim is Reflecting". Or, it can even track the last three throws of the Sim, or the last three throws of the game (Sim then system then Sim), etc. Or, since the system knows something about the Sim's two basic strategies (maybe even because of industrial espionage!), the system can guess which strategy is being used at the moment. It might even be able to try to guess at the value of N.

This quickly gets into issues of Machine Learning (and maybe even issues of Cryptography), and we aren't going to go there in any depth. Because, in the extreme, the system can record the *entire game*, and then, at each throw, look for patterns going all the way back to the very start. But this is one of the well-known problems with Big Data: more data is not necessary more useful, since exact histories are unlikely to repeat, and since there are so many possible patterns one can detect, even in noise. For example, see:

`https://www.tylervigen.com/spurious-correlations`

So, please design your system carefully. It is better to have something dumber that actually runs, rather than something smarter that doesn't.

Although the purpose of this Step is for your revengeful system to be the best ever against the simulated human, the creativity in this Step is not just the creativity *in* your design. It is also in the creativity of your documenting the strengths and weaknesses *of* your design. You need to defend your design choices–that is, your style–and to record their performance, in the Javadoc somewhere.

One easy way to do this is to show the results of Step 4 against the simulated human, followed by the results of Step 5 against the *same* simulated human.

Note that by the end of this Programming Part, you should have experienced the ways in which good object-oriented design–including the CRC and UML tools–for the prior Steps can make modifying system functionality easier. And, it should be apparent that even if you had the luxury of perfectly knowing exactly what all the future enhancements of a system would be (the five Strategies, for example, and the two kinds of players), this foreknowledge doesn't by itself solve all of the design issues. You still have to carefully understand who does what, what the data structures and algorithms are, and how much to test.

2.5.2 Deliverables

As before, plus a Step 5 versus Step 4 comparison. Basically, this Step augments the same classes that you augmented in Step 2.

3 General Notes

For each Step, design and document the system, text edit its components into files, compile them, debug them, and execute them on your own test data. When you are ready, submit the text of its code, a copy of your testing, and whatever additional documentation is required. Make sure that the source code is clear and its user interface is comprehensive and informative.

Write your classes clearly, with a large block comment at the beginning describing what the class does, how it does it (i.e., in some form, talk about its CRC and/or UML), and justify how it will be tested. Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. Use the suggestions from *Clean Code*. For a related series of suggested practices and stylistic guidelines, see the website of a popular textbook sometimes used in COMS W1004, at:

`http://horstmann.com/bigj/style.html`

Remember that human designers, documenters, coders, and testers are limited in their abilities to understand something new, and that it is important to stay short, simple, searchable, solution-oriented, and standard. Be clean, be consistent.

4 Checklist

Here is an approximate checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below.

4.1 For the Theory Part

Answers, either handwritten then scanned, or in machine-readable form. No extra points for using a text editor or other formatting tools; hand-drawn diagrams are fine. Submitted electronically in Courseworks by the time specified. The theory file should be separate from the files for the programming. Name and UNI must be prominently included.

4.2 For the Programming Part

Answers submitted electronically in Courseworks by the time specified. All code (*.java files) documented according to Javadoc conventions, and all trial runs. These files should be separate from the files for the theory. Name and UNI must be prominently included in each class and on each output. Rather than providing a separate README file, one of your classes should have an obvious name that will attract the attention of the TAs (like "Runner", for example), and *inside* it there should be an overview of the components of your system. The classes, in whatever state of completion they are, *must compile*.

4.3 For both Parts

Please put all your submission files (theory, *.java, example runs) in one directory (named, for example,"HW1"), and then compress the contents of that directory into one file. Submit that compressed file to Coursework2. The name of that compressed file should be "myUNI_HW1", and it should have the appropriate extension like ".zip", except that "myUNI" should of course be replaced with your real Columbia UNI. *Please* use this convention as it makes the TAs' job much easier:

You may submit electronically as often as you wish, but it will be your *last* electronic submission before the deadline which is the official one that will be graded, and–if need be–executed.