COMS W1007 Assignment 1
Due Sept. 25, 2018

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 interconnectedness 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 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 student would decide on which dormitory to choose, based on its design. 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, "has elevator" is like "efficient data access". Is there a "best" dormitory at Columbia? Explain.

1.2 Hello World again (6 points)

Extend the class NameMaker with a "natural" default constructor (which you defend in the Javadoc), and with another constructor that also allows multiple names.

1.3 arrays, ArrayLists, LinkedLists (6 points)

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

- The contacts information (name, phone, etc.) for your smartphone.
- A re-loadable stapler that holds a row of staples.
- The bouncer at a popular social establishment, who keeps patrons waiting in a line, but who may allow special handling for distinguished guests or people he doesn't like.
- Repotting plants as they outgrow their pots.

Then, give one example each in the real world in which aggregations of real objects are manipulated like a[], AL, LL. You must give examples that differ from the ones given in class and this question: no parking lots, spring-loaded bookcases, beads on a chain, contact info, staplers, bouncers, pots.

1.4 Bad user requirements (6 points)

Critique the following use case, by listing the questions you would go back to ask the user, in order to clarify the user's intentions. Look for classes, methods, fields, performance requirements, incompletenesses, redundancies, etc.

> I want to buy one of your robot used car salesmen. When customers come into my store, it should greet them all quickly, find out what they want and what they can afford, check if they are credit worthy, then show them a bunch of the junkers on the lot that are most suitable but that we are trying to get rid of. Then at the appropriate time, it draws up all the bills of sale, all the while carrying on feel-good conversations with everyone.

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

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

> The ATM, which has a screen, a keyboard, and an input slot and an output slot. It takes a credit card which is valided by keying in a PIN, determines the transaction type (if valid), and either takes bills of several denominations or gives bills of several denominations, then gives back the credit card (unless it has been reported stolen). If necessary, it produces error messages on the screen.

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 the following way. First, select three names you used in your solution. Show how each of them follows the suggestions in Chapter 2. Then, pick three methods, and show how each follows the suggestions in Chapter 3. Lastly, pick two comments, and show how each follows the suggestions in Chapter 4.

2 Programming Part (50 points)

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 (very wordy) requirement specification, together with some friendly advice (which you can ignore), and 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 should come later in the course. Nevertheless, the system will show behaviors that may be difficult to predict..

Further, this Programming Part shows in miniature what happens when a system is successful and the initial design has to be extended in several Steps. Yes, if you read ahead through all the Steps 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.

Your eventual goal is to implement the 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 a simple Rock Paper Scissors game against a real human friend. Your program plays randomly, but it should therefore win more often than your friend does. The second Step adds some AI, and it should win even more often against a friend. The third Step plays the full RPSLK, with the same AI, against a friend. The fourth Step replaces the friend with a simulated friend, so that you can get many games played quickly and gather statistics on both your program and your simulated friend. The fifth and final Step uses the simulated friend to help your refine a better AI-based strategy, 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.

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 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 its abstract of the 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 agile programming will always keep a working version of each Step before proceeding to the next. Eclipse can help with this; look up its "local history" feature.

2.1 Step 1: Basic interactive RPS system (20 points)

For this Step, you should start small, with a console application that uses text output and text input, and that just plays Rock, Paper, Scissors with a friend. 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 friend for a single character from 'r', 'p', or 's'. When a valid character is thrown by the friend–you must design a loop to do sanity checking!–some class prints out a message indicating the two throws, and who won, and why, roughly: "I win: My paper covers your rock".

We will assume that the computer is honest, and really does throw without looking at the friend's throw, and that your friend trusts both you 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 nested executable *code*. You should make this effort now in Step 1, because the design will soon be expanded, and you would appreciate an expansion that is simple and easy.

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

To pull this all together, have a short, simple, solution-oriented Runner class 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, particularly in order to 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.

Once your fried plays the game, it is likely that both of you will discover that the computer–even by playing randomly–tends to win more than the friend does, since real humans tend to "get

stuck" with foolish misconceptions about strategy. This 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.

It would be good to point these things out to your friend at the end of the game. And, most importantly, you must record these observations about the testing in the Javadoc (where?).

If you stop at this Step, turn in your *.java files (generally, this means what you wrote in the Eclipse project), and some trial runs captured from the console output to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Do not use a README file; anything that would go in the README should go into the Runner class instead, where it is less likely to get lost.

If you continue, you must still submit *output* for Step 1. But it is likely that your system will grow Step by Step. So, you don't have show the design, documentation, or code for Step 1 explicitly, *as long as* you eventually submit a final system that can do this Step. This means that, if necessary, the grader can take your single Eclipse project and test Step 1 from it.

2.2 Step 2: Adding simple AI (8 points)

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 your friend has thrown 'r', 'p', and 's' during the game so far, and exploits the human inability to be totally random. So, if the AI version of Thrower notes that the friend really likes Rock, it will tend to prefer, in some way, to suggest a throw of Paper.

This means at this Step, Thrower has to be augmented to have some methods for learning the games history, recording some useful information derived from the history, computing the "best" throw, and outputting its decision, all of which are again 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. Again, you test your system by its playing 100 rounds with your friend, and reporting what happens with this testing, in Javadoc somewhere, as in Step 1. Note that at this Step, Thrower has two Step-dependent strategies: random, plus simple AI.

If you stop at this Step, turn in your *.java files (generally, this means what you wrote in the Eclipse project), and some trial runs captured from the console output to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Do not use a README file–anything that would go in the README should go into the Runner class instead, where it is less likely to get lost.

If you continue, you must still submit *output* for Steps 1 and 2. But it is likely that your system will grow Step by Step. So, you don't have show the design, documentation, or code for Step 2 explicitly, *as long as* you eventually submit a final system that can do this Step. This means that, if necessary, the grader can take your single Eclipse project, and test Steps 1 and 2 from it.

2.3 Step 3: Extending your system to play RPSLK (8 points)

This is just like Step 2, 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.

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

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

or "RPS-101":

`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.

To test this Step, you will need to do about 200 rounds, and again record your observations about the testing somewhere in Javadoc. You will need a very patient friend. Or, maybe see:

`https://www.wikihow.com/Be-Your-Own-Best-Friend`

If you stop at this Step, turn in your *.java files (generally, this means what you wrote in the Eclipse project), and some trial runs captured from the console output to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Do not use a README file–anything that would go in the README should go into the Runner class instead, where it is less likely to get lost.

If you continue, you must still submit *output* for Steps 1, 2, and 3, since those are different and have to be tested.. But it is likely that your system will grow Step by Step. So, you don't have show the design, documentation, or code for Step 3 explicitly, *as long as* you eventually submit a final system that can do this Step. This means that, if necessary, the grader can take your single Eclipse project, and test Steps 1, 2, and 3 from it. You should probably at this point be thinking something about a constant named STEP.

2.4 Step 4: Simulating your friend (8 points)

Having amused your friend with the game, it is time to see if you can make your system do even better. To do this, your system first will have to remove the friend from the testing loop, so that it can run the experiments on *simulated* friends. This way it can more readily test out better strategies, without any apparent social cost.

So, this Step should augment the system somewhere (in the Talker class, maybe?) so that it doesn't have to wait for a friend to input a throw. Instead, it interacts with a Sim class, which observes the game in progress, then executes a "strategy", and then outputs a throw automatically.

It is up to you to determine what kind of strategy to use. You can even test out several. Note that this Step helps demonstrate the utility of CRC and UML, since there is some similarity, overlap, and dependencies within these simulated friends. These similarities are easier to see if they are dealt with upfront in the design phase, rather in the implementation phase of the process.

Here are some suggestions for a simulated friend, who then can throw:

- randomly
- the same character every time
- cyclically, in order:: 'r', 'p', 's', 'l', 'k', 'r', 'p', etc.

- reflectively, using whatever the computer threw the previous time
- the same character until it loses, then throws something else, determined in some way
- based on the code of Thrower, which was stolen through industrial espionage(!)
- based on a more complicated algorithm derived from "experience"

Your system should now play some games against the simulated friend, reporting the outcomes. 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 friend. (We will later see that this is an example of the Decorator Pattern.)

As part of the testing, you must decide *how many* rounds is "enough" in order to get reliable outputs. Your final decision on "enough" should become part of the Javadoc of the appropriate class or classes. And, after you have done the testing, also note in the Javadoc documentation what happened to the win percentage and tie percentage for the friend you simulated.

If you stop at this Step, turn in your *.java files (generally, this means what you wrote in the Eclipse project), and some trial runs captured from the console output to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Do not use a README file–anything that would go in the README should go into the Runner class instead, where it is less likely to get lost.

If you continue, you must still submit *output* for Steps 1, 2, 3, and 4, since those are different and have to be tested.. But it is likely that your system will grow Step by Step. So, you don't have show the design, documentation, or code for Step 4 explicitly, *as long as* you eventually submit a final system that can do this Step. This means that, if necessary, the grader can take your single Eclipse project, and test Steps 1, 2, 3, and 4 from it.

2.5 Step 5: Creativity: add a bit more AI (6 points)

Now you have all the tools and experimentation you need to improve your Thrower. It already has two strategies: random, and a simple AI based on a simple view of a friend's human preferences. Add a third strategy, called MAIGA ("Make AI Great Again"), which uses more information from the game's history. For example, it can look at the simulated friend's *last two* throws to see if there are pairwise patterns, such as, "a Spock very often follows a Lizard". Or, it can see if the simulated friend is reacting to the previous throw of the *system*, such as very often throwing the same thing that the system threw the round before. Or, it can use negative information, such as noting that the simulated friend rarely throws Scissors.

This quickly gets into issues of Machine Learning, and we aren't going to go there. Because, in the extreme, the Thrower can record the *entire game*, and then, at each throw, look for patterns going all the way back to the 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. So, please design your Thrower 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 MAIGA system to be the best ever against the simulated friend, 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* it. 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 friend, followed by the results of Step 5 against the *same* simulated friend.

If you do this Step, turn in your *.java files (generally, this means what you wrote in the Eclipse project), and some trial runs captured from the console output to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Do not use a README file–anything that would go in the README should go into the Runner class instead, where it is less likely to get lost.

You must still submit *output* for Steps 1, 2, 3, 4, and 5, since those are different and have to be tested. It should still be the case that, if necessary, the grader can take your single Eclipse project, and test Steps 1, 2, 3, 4, and 5 from it.

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, this foreknowledge doesn't by itself solve all of the design issues.

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 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.