CS6310 – Software Architecture & Design
Assignment #4 [125 points]: Star Search Simulation System – Implementation (v4 updates)
Spring Term 2020 – Instructor: Mark Moss

Submission Deliverables

- This assignment must be completed as an individual, not as part of a group.
- You must submit:
  (1) A Java JAR file named **star_search.jar** that performs the core steps of the application as described below
  (2) The corresponding Java source code for your JAR file (including external dependencies and libraries) in a ZIP file named **star_search_source.zip** with a Main class as a common executable starting point in case we need to compile your source files.
- You must notify us via a private post on Canvas and/or Piazza BEFORE the Due Date if you are encountering difficulty submitting your project. You will not be penalized for situations where Canvas is encountering significant technical problems. However, you must alert us before the due date – not well after the fact. You are responsible for submitting your answers on time in all other cases.
- Please consider that uploading files to Canvas might occasionally take a long time, even in the case of seemingly "relatively small" submissions. Plan accordingly, as submissions outside of the Canvas Availability Date will likely not be accepted. You are permitted to do unlimited submissions, thus we recommend you save, upload and submit often. You should use the same file naming standards for the (optional) "interim submissions" that you do for the final submission.

Problem Scenario

Your requirement for this assignment involves implementing the core architecture and functionality for the Star Search Simulation System that you designed in the earlier assignments. Your system must implement the following functionality:

(0) The simulation system must read in the provided scenario file which designates the initial state of the simulation, to include the initial locations of drones, suns and other configuration values
(1) The simulation system alerts the next drone that it needs to provide a proposed action
(2) The selected drone uses its internal decision-making processes to determine its next action
(3) The selected drone sends its proposed action back to the simulation system
(4) If the selected action is a **`thrust()`**, then the simulation system must evaluate the state of the simulation to determine if the movement leads to a crash or vaporization
(5) If the selected action is a **`scan()`**, then the system returns the scan results to the selected drone
(6) The selected drone is allowed to digest any received information (from scans or collisions) and communicate with the other drones as needed
(7) The simulation system must update the state of the simulation
(8) The simulation system must determine if the simulation run should be halted
(9) The simulation system must halt the simulation run if required and display the final report; otherwise, it must continue the simulation run at step (1) with the next drone

Your prototype does not have to produce a graphical display. At this point, we will evaluate your prototype based on the text-based output and the structure of the source code. You'll have ample opportunity to develop a more advanced graphical user interface during the group project.

Sample Source Code

- We've provided a small amount of sample code to help get you started with the project. *Using this code is optional – you may use portions of it or all of it, but this is completely your choice.* However, this code might be helpful for those who haven't worked with Java recently (or at all).
- *The provided code is NOT fully correct!* It contains snippets of algorithms that might be useful for implementing the required functions, but it does NOT contain all of the functionality needed for the application. You are welcome to use any of this code but please note that you are ultimately responsible for ensuring the correctness of your application.
- *The code is in a very "monolithic" design intentionally.* One of the key things that you absolutely must do is transform the sections of the code that are helpful to you into a more appropriate class and file structure that is consistent with your proposed design.

General Intent

In the first phase of the Star Search Project, you were required to provide design artifacts to describe your approach to structuring the classes, attributes, operations, methods and relationships needed to simulate the problem. Now, you are required to provide a lightweight implementation of the system in Java that reflects your design. Your system must support the Star Search Simulation System tasks as defined in the earlier assignment instructions, along with any further updates as clarified below.

Also, even though we aren't asking you to provide updated architectural diagrams, you are required to demonstrate some fundamental separation of classes in your source code indicative of a reasonable design. This means that your source code should show some indication of being divided up into classes, files, etc. that match your design. You will lose points if you submit poorly structured code – for example, if you simply try to add a few lines to the source code that we provided to you and simply return it to us.

And don't panic if you realize that your original design had flaws while working on this assignment – this is common for "agile styles" of development. Make sure that your implementation works per these requirements and specifications and note your earlier design oversights with some brief comments in your code. You'll be asked to provide improved design artifacts based on your "revised and improved understanding" of the problem space in the later phases of the project.

We provide examples of the input and output file formats, along with an explanation of the required tasks. We've also provided some sample code that you are welcome to use in the development of your system. You must provide the actual Java source code, along with all references to any external packages that you used to develop your system, in a ZIP file named **star_search_source.zip**. You must also provide an executable JAR file named **star_search.jar** to support testing and evaluation of your system. We must also be able to recompile your application from your source code as part of the evaluation process.

Evaluation/Grading Your Submissions

- Your submission will be evaluated based on four main areas:

(1) 20 points for your system's correctness: working JAR file, the capability to recompile your code if necessary, etc.
(2) 55 points for reasonably structured source code
(3) 50 points for correct operation of the system on the selected scenario (test) files

Don't lose sight of the goal for the course: searching performance is important (and fun to work on), but you really need to think about how you will separate and distribute the complexity of the overall problem across separate classes; and, how the objects instantiated from those classes then communicate and collaborate to solve the problem. Helping you develop a solid structure for your design is the most important aspect of working on this assignment.

(1) *20 points for your submission correctness: working JAR file, the capability to recompile your code if necessary, etc.*
Common issues that cause you to lose points in this category:

- Not submitting a working JAR file
- The JAR file doesn't function properly on the VM during testing – for example, JNI errors, etc.
- Gross formatting errors – we've designed the testing harness to be as robust as possible when processing the submissions, but errors caused by including graphical displays of the space region, diagnostic output, and other random messages will also cause you to lose points.
- Formatting is important! Syntax is important! Use the matching characters for the output strings, and don't put extra spaces between elements of the output strings. Strings that do not match the correct output because of formatting (syntax) errors might receive significant penalties.

(2) *55 points for reasonably structured source code*
The main issue that will cause you to lose points in this category is submitting poorly structured, possibly monolithic source code that doesn't display any indication of separation of responsibilities among classes, objects, etc. There isn't a specific set of objects that you must have, but you do need to display some significant effort to apply object-oriented analysis & design principles.

(3) *50 points for correct operation of the system on the selected scenario (test) files*
Common issues that cause you to lose points in this category:

- Command Errors: when your system fails to acknowledge an invalid action, such as actions other than **`{steer, thrust, scan, pass}`**; or, if your **`steer()`** direction – or your **`thrust()`** distance – is invalid
- Response Errors: when your system responds "**`ok`**" when it's actually a "**`crash`**", or vice versa
- Scan Errors: when your system doesn't give the correct sequence of locations surrounding the drone that issued the **`scan()`** action
- Reporting Errors: when your system doesn't provide the correct four-value report at the end of the simulation run

Input & Output Specifications

- Here are the specifications for the scenario file that your system must read in step (0), along with the line-by-line and final report details that your system must produce as the simulation runs:
  - Here are some of the limits on values within the (input) scenario files:
    ```
    (1)  <width (x-axis) of the space region> [min: 1, max: 20]
    (2)  <height (y-axis) of the space region> [min: 1, max: 15]
    (3)  <number of drones> [min: 1, max: 10]
    (4)  <initial location, direction and strategy of each drone>
         [one line per drone]
    (5)  <number of suns> [min:0, max: 50% of the space region]
    (6)  <location of each sun>
         [one line per sun]
    (7)  <maximum number of turns for this run> [min: 1, max: 200]
    ```
  - The official terms for scan results are **`{stars, empty, sun, barrier, drone}`** as described earlier.
  - The official terms for directions are **`{north, northeast, east, southeast, south, southwest, west, northwest}`** as described earlier.
  - We've added a **`strategy`** value for each drone corresponding to that drone's control logic:
    - Zero (**`0`**) = select an action randomly
    - One (**`1`**) = leverage environment info and drone collaboration to select "best" action
    - Two (**`2`**) = allow the user to select an action via the input prompt (diagnostic only!)

- **The required file output for the assignment is to provide a pair of lines for each action during the simulation, followed by the final report at the end of the simulation run.** The first line of output must echo the drone's ID & the drone's selected action. The second output line must list the corresponding validation of the action and/or response from the simulation system:
  - If the drone selects **`steer(<direction>)`** then the simulation system must display **`ok`** if the direction is valid.
  - If the drone selects **`thrust(<distance>)`** then the simulation system must display **`crash`** if the distance is valid and the action causes the drone to collide with a sun or another drone and must display **`ok`** otherwise.
  - If the drone selects **`scan()`** then the simulation system must display the contents of the surrounding squares in sequence using the terms **`stars, empty, sun, barrier`** and **`drone`**. The simulation system must also provide this information to the drone itself.
  - If the drone selects **`pass()`** then the simulation system must display **`ok`**.
  - Finally, if the drone selects an invalid action (e.g., steer **`direction`** or thrust **`distance`** is invalid) then the simulation system must display an "**`action_not_recognized`**" message and ignore the invalid action as if a **`pass`** action had been requested.
  - Please note the underscores that have been added to the "**`action_not_recognized`**" message. It makes parsing the files easier if the message is handled as a single token (i.e., string) as opposed to three separate words.

- At the end of the simulation run, your system must display the following information as a single line of four numbers separated by commas (no other spaces, delimiters, etc.):
  - The total number of squares in the space region

   - The initial number of squares containing stars in the space region, including the squares initially occupied by drones
   - The number of squares successfully explored (i.e., visited) by the drones
   - The total number of fully completed turns for the simulation run
- The required file output does NOT include the graphical display of the space region. This functionality was provided to help you visualize the drone actions while developing your strategy. Be sure to disable, comment out, or delete this aspect of the system before your final submission.

Scenario Example

Consider the following scenario file, which is a simple extension of an earlier example:

```
5
4
3
1,2,north,0
0,1,northeast,0
3,1,west,0
3
1,1
1,3
3,0
3
```

The initial simulation state is shown here (Figure 1):

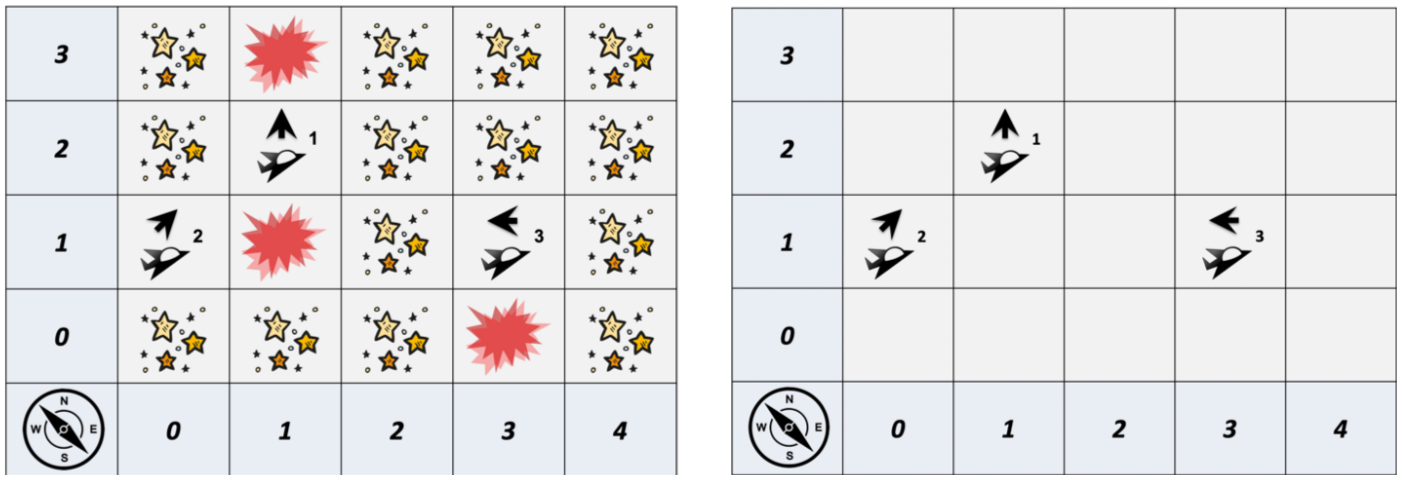

Figure 1 – The Initial Position: The Full Simulation State (*left*) and What the Drones "Know" (*right*)

For the first turn, after the drones execute **`thrust(1)`**, **`thrust(2)`**, and **`scan()`** (in ID order):

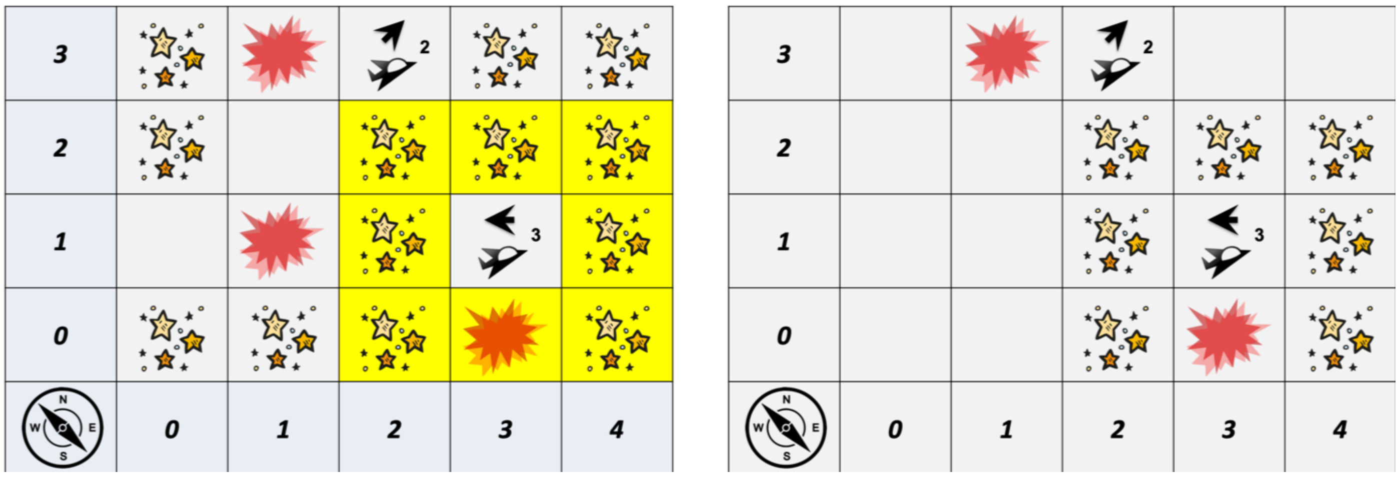
Figure 2 – Position After the First Turn: The Full Simulation State (*left*) and What the Drones "Know" (*right*)

For the second turn, after the remaining drones execute **`scan()`** and **`thrust(3)`** (in ID order):

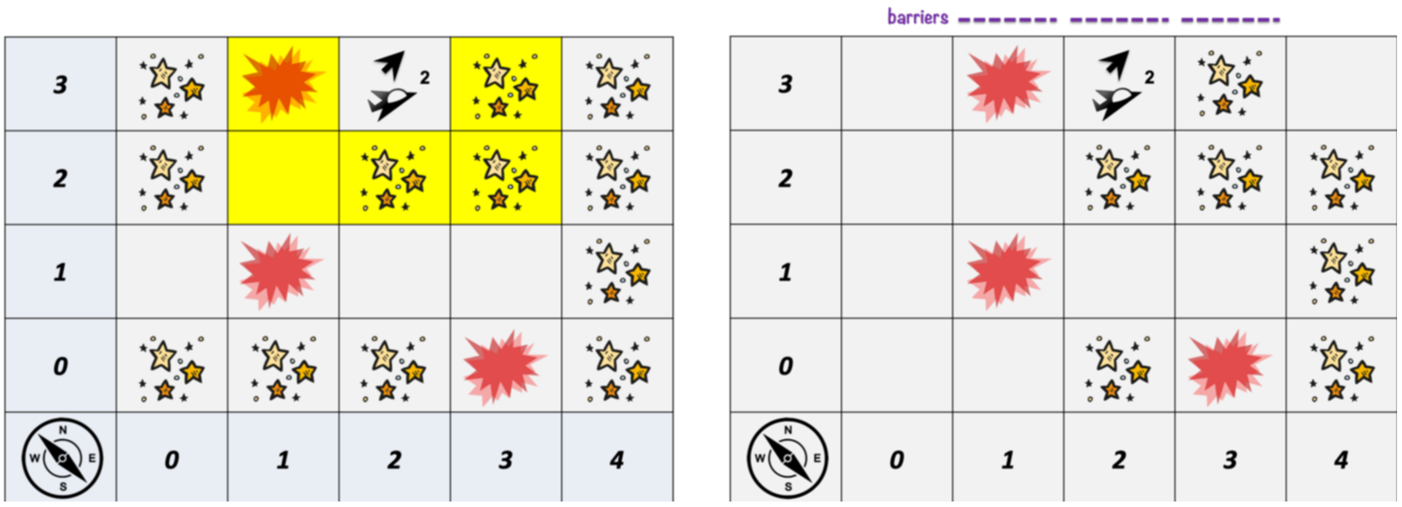

Figure 3 – Position After the Second Turn: The Full Simulation State (*left*) and What the Drones "Know" (*right*)

Finally, suppose that the remaining drone simply changes its direction with the **`steer(southeast)`** command for the third turn to complete the simulation run (no diagram shown, nor really needed). Let the contents of the scenario described above be stored in a file named **`scenario0.csv`**. We will place the scenario file in the same directory (i.e. the current working directory – **`cwd`**) along with your **`star_search.jar`** file, and then test your program using a **`java -jar`** command. Based on the drone actions shown above, the following output should be displayed:

```
> java -jar starsearch.jar scenario0.csv
d0,thrust,1
crash
d1,thrust,2
ok
d2,scan
stars,stars,stars,stars,sun,stars,stars,stars
d1,scan
barrier,barrier,stars,stars,stars,empty,sun,barrier
d2,thrust,3
crash
d1,steer,southeast
```

```
ok
20,17,5,3
>_
```

Note that the **`scan()`** results are returned beginning with the contents of the square that is North of the drone's current location, regardless of the drone's current direction. The remaining squares are given in a clockwise order: Northeast, East, Southeast, etc. Also, the output file syntax is a relatively straightforward, text-based Commas Separated Value (CSV) format organized as:
`<droneID>,<action>[,<direction>|,<distance>]`

- The **`<droneID>`** represents the order of the drone as listed in the scenario file: the first drone is **`d0`**, the second drone is **`d1`**, etc.
- The **`<droneID>`** and **`<action>`** values must always be listed. The only valid **`<action>`** values are: **`{steer, thrust, scan, pass}`**
- A direction must be listed if (and only if) the action is **`steer`**. The valid **`<direction>`** values are: **`{north, northeast, east, southeast, south, southwest, west, northwest}`**
- Finally, a distance must be listed if (and only if) the action is **`thrust`**. The valid **`<distance>`** values are 1, 2 or 3.
- All other **`<action>, <direction>`** and **`<distance>`** values are invalid and will cause the drone to enter "maintenance mode" for that turn, effectively "passing" until the next turn.

Submission Details

- You must submit your source code in a ZIP file named **star_search_source.zip** with a project structure as described below. You must include all of your source *.java code – you do not need to include the (compiled) *.class files. Also, you must submit (separately) a runnable JAR file of your program named **star_search.jar** to aid in evaluating correctness.

- We will test your program by copying it into the same directory (i.e. the current working directory – **`cwd`**) with an existing test case, and then executing it at a terminal prompt in the provided VM with the following command(s):
  ```
  > java -jar star_search.jar <name of scenario file>
  ```

- Your system must complete each scenario file in less than five (5) seconds in real time – otherwise, your result is considered to be a failure for that scenario. The scale of these scenarios shouldn't require more computing resources than this – in fact, many systems from the previous term completed all of the scenario files (e.g., 20+ files) in less than three (3) seconds. Let us know if you feel that you are unable to develop your system to meet this requirement.

- We might (in rare cases) also test your program by recompiling your source files submitted in your **star_search_source.zip** file. In this case, we will use the following steps:
  ```
  > javac *.java
  > jar cfe star_search.jar Main *.class
  ```
  and then we will run the newly generated **`star_search.jar`** file against the scenario files. If you would like to use a more complex recompilation sequence, then that is OK, but you must provide clear instructions and the supporting files to do this in a relatively straightforward manner.

- We've also provided an auto-checker called **star_search_monitor.jar** that you can use to help troubleshoot your programs. You can use the program by saving your **star_search.jar** output to an appropriately named results file, and executing the auto-checker as follows:

  ```
  > java -jar star_search.jar scenario<N>.csv > scenario<N>_results.csv
  > java -jar star_search_monitor.jar <first test> <last test>
  ```

  For example, you can test one file as follows:

  ```
  > java -jar star_search.jar scenario2.csv > scenario2_results.csv
  > java -jar star_search_monitor.jar 2 2
  ```

  You can test multiple (adjacent) files as well:

  ```
  > java -jar star_search.jar scenario3.csv > scenario3_results.csv
  > java -jar star_search.jar scenario4.csv > scenario4_results.csv
  > java -jar star_search.jar scenario5.csv > scenario5_results.csv
  > java -jar star_search_monitor.jar 3 5
  ```

  The **star_search_monitor.jar** program uses the output file produced by your submission, and basically traces the actions that your drones have executed in its own separate copy of the initial scenario file. If differences are detected, then it displays a simple text-based graphical display representing the current state of the system along with a **FAULT** message identifying the error in the file being tested.

- The **star_search.jar** and **star_search_monitor.jar** programs have a "verbose" (**-v** or **-verbose**) option that displays the state of the simulation run after each move:

  ```
  > java -jar star_search.jar scenario8.csv -verbose
  > java -jar star_search_monitor.jar 3 5 -v
  ```

  This can help you visualize what is happening in the system. By the same token, you are not required to implement this functionality in your system. And you should ensure that you DO NOT INTEGRATE THESE EXTRA GRAPHICS INTO THE FINAL OUTPUT of your **star_search.jar** program – the extra lines will disrupt the evaluation system and the test results.

- We've just added a newer version of the **star_search_monitor_v3.jar** program. Version 3 operates just like version 2 – for example, it's more "forgiving" on invalid moves by the drones, but still stops if the drones move "out of order". The only real difference is that this new version handles the "**action_not_recognized**" message checks more consistently.

  The **star_search_monitor_v3.jar** program is very similar to the program that we will use to evaluate your final submission, and we highly recommend that you use it to check your work before your final submission to avoid any surprises. With the exception of the file name, this second version of the program operates exactly like the original version.

  We've also added a new version of the **star_search_v2.jar** program. This version operates just like the original version with the exception that it handles the "**action_not_recognized**" message checks more consistently.

- To clarify: when a strategy value of '**`2`**' is used for a drone in any of the scenario test files, then that drone will be controlled manually by the user while the **`star_search.jar`** program is executed. This is purely for 'diagnostic testing' - in short, so that you can experiment with how drones move and interact within the space region. It can also be very helpful for discovering potential errors in the current version of the **`star_search.jar`** program that we've provided.

  When I test your drones, however, I will NOT use this option. I will only test your submission using scenario test files with combinations of:
  - strategy value '**`0`**', to ensure that your simulation system handles crashes with other drones, vaporizations with suns, and bumping into the space barrier correctly; and,
  - strategy value '**`1`**', which is your opportunity to implement your collaborative drone controls to search the space region as efficiently as possible – well, at least better than randomly for now.

- Your program should display the output directly to "standard output" on the Linux Terminal – not to a file, special console, etc. Also, note that the test file must be located in the same directory where your JAR is currently located. More importantly, we will copy your JAR file to different locations during our testing processes, so be sure that your program reads the test files from the current local directory, and that your program is NOT hard-coded to read from a fixed directory path or other structure.

- We've provided a virtual machine (VM) for you to use to develop your program, but you are not required to use it for development if you have other preferences. We do recommend, however, that you test your finished system on the VM before you make your final submission. The VM will be considered the "execution environment standard" for testing purposes, and the occasionally received excuses of "…it worked on my home or work computer…" will not be sufficient.

- On a related note, it's highly suggested that you test your finished system on a separate machine if possible, away from the original development environment. Unfortunately, we do receive otherwise correct solutions that fail during our tests because of certain common errors:
  - forgetting to include one or more external libraries, etc.
  - having your program read files from a specific, hardcoded (and non-existent) folder
  - having your program read test files embedded files in its own JAR package

- Many modern Integrated Development Environments (IDEs) such as Eclipse, IntelliJ, Android Studio, etc. offer very straightforward features that will allow you to create a runnable JAR file fairly easily. Also, please be aware that we might recompile your source code to verify the functionality & evaluation of your solution/JAR file compared to your submitted source code.

Closing Comments & Suggestions

This is the information that has been provided by the customer so far. We (the OMSCS 6310 Team) will likely conduct an Office Hours where you will be permitted to ask us questions in order to clarify the client's intent, etc. We will answer some of the questions, but we will not necessarily answer all of them. Also, though this current version of the system if "the core" of the system going forward, our clients will very likely add, update, and possibly remove some of the requirements over the span of the course. One of your main tasks will be to ensure that your architectural documents and related artifacts remain consistent with the problem requirements – and with your system implementations – over time.

Quick Reminder on Collaborating with Others

Please use Piazza for your questions and/or comments, and post publicly whenever it is appropriate. If your questions or comments contain information that specifically provides an answer for some part of the assignment, then please make your post private first, and we (the OMSCS 6310 Team) will review it and decide if it is suitable to be shared with the larger class. Best of luck on to you this assignment, and please contact us if you have questions or concerns.