

## **CS6310 - Software Architecture & Design**

### **Assignment #4 [150 points]: OsMowSis - Design Updates & Modifications (v1)**

**Fall Term 2019 - Instructor: Mark Moss**

#### **Submission**

- This assignment must be completed as an individual, not as part of a group.
- You must submit:
  - (1) A UML Class Model Diagram named **class\_model.pdf**;
  - (2) A UML Sequence Diagram named **sequence\_model.pdf**;
  - (3) Answers to the provided design questions as **question\_answers.pdf**.
- You may divide your class and sequence model diagrams into multiple diagrams if needed. For example, if the initial diagram is so large that dividing it into multiples makes it significantly easier to read, then you may submit multiple files with reasonable name extensions such as **class\_model\_1.pdf, class\_model\_2.pdf**, etc. You must provide enough context for the multiple diagrams to allow us to connect them together to understand the single, unified diagram.
- Formats other than PDF must be pre-approved an Instructor or TA.
- You must notify the Instructors and TAs via a private post on Piazza BEFORE the Due Date if you are encountering difficulty submitting your project. Send the message to "Instructors" when posting your message in Piazza, as opposed to addressing a message to just one individual, to ensure that it is acknowledged as quickly as possible.
- 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 most 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.

#### **Deliverables**

UML Diagrams are normally divided into three fundamental categories: Functional, Structural and Behavioral. For this assignment, you must analyze the updated requirements for the OsMowSis problem description as presented above, and then revise your original designs based on what you've learned from earlier assignments. This time, we're asking you to provide an updated UML Class Model Diagram that accurately reflects the structure of the updated system, along with a UML Sequence Diagram that accurately affects the dynamic behavior of your system. Your Class Model and Sequence Diagrams must be consistent in terms of the classes, attribute names, operations, any other relevant factors. You must submit the following items:

- (1) **Class Model Diagram [45 points]** – You must provide a Class Model Diagram that includes the classes, attributes with basic types, operations, and relationships with proper cardinalities that accurately reflect the problem space. To make your class diagram more readable, you may omit very basic "setters and getters" [e.g. `set_()` and `get_()` based operations and methods], especially

those of the one-line variety that only access and/or modify an object's basic attribute. Your UML Class Diagram must match the same level of detail as shown in the Udacity videos.

- (2) **Sequence Diagram [45 points]** – You must provide a Sequence Diagram that represents the communication between your mowers and any other relevant objects from your design during the course of completing one turn of the simulation. You should use the appropriate symbols for selection (e.g. if/then) and iteration (e.g. for/while loops) in your diagram where appropriate. For your diagram, you must include at least enough mowers to demonstrate how you handle the situations in the questions below, but not so many that your diagram becomes unwieldy.

The emphasis of the Sequence Diagram is on the communication between objects, so you should ensure that the messages between the objects are reflected to reasonable names that are consistent with your Class Model operations. You don't have to provide incredibly detailed insight about functions, procedures and other operations and calculations that are completely internal to a specific object in your Sequence Diagram. On the other hand, most reasonably well-designed solutions will have the data needed to manage the simulation spread across various objects, and so the objects will need to interact in order to complete the simulation effectively. Sequence Diagrams that demonstrate little or no communication between objects are sometimes indications of ineffective (e.g. monolithic) design.

- (3) **Short-Answer Design Questions [60 points]** – The following is a set of questions designed to help ensure that your design meets the level of detail we are looking for in this assignment. For each question, provide a relatively short (3-7 sentences) answer that identifies the specific classes, attributes, operations and other aspects of your design that allow you to address the given issues.

Many of these questions are similar to the questions from earlier assignments, but we are specifically asking them again to give you the opportunity to address changes based on what you learned after implementing your original design, and the changes to the requirements presented above. *Please be careful of "simply parroting" your previous answers – if they aren't consistent with the design documents you submitted and/or the new requirements, you will lose a significant amount of points.*

- How do you track the current direction and location of each mower?
- How do you track the locations of gophers?
- How do you track the how much of the grass has been cut so far?
- How do you update the simulation state for each of the varied actions?
- How do you determine the appropriate output for a **cscan()** action?
- How do you determine the appropriate output for an **lscan()** action?
- How do you determine when the simulation should be halted?
- How do you keep track of the knowledge needed to display the final report?
- How do you keep track of the partial knowledge collected by each mower?
- How do you manage collaboration between the mowers?
- How do you determine the next action for a mower?

## Updated Scenario

The OsMowSis application that you originally designed and prototyped in earlier assignments is greatly appreciated by the clients. They would like to refine and expand the capabilities of the simulation system in a number of ways, which will be discussed in more detail below. The fundamental problem of developing and testing automated lawnmowers is still the main focus of your efforts – however, the most significant changes to the design requirements are as follows:

- The stationary craters as obstacles have been replaced by moving (and mischievous) gophers;
- The mowers have a new type of "long range" scanning capability;
- The mowers have a limited amount of energy, and will occasionally need to "recharge"; and,
- The mowers are allowed to communicate and share their information as needed.

We will provide more details below, to include brief explanations of the motivation for some of these changes. These changes are only additions to and/or modifications of the previous requirements from the earlier assignments. All of the previous requirements still apply; and, if you feel that any of the requirements are contradictory, let us know and we'll resolve the conflict.

## Gophers (Just when you thought it was safe to go out on the lawn...)

You approach the lawn, ready to place the mowers in their starting positions, and then you notice something strange... almost too good to be true...there are no craters. Well, unfortunately, your instincts are correct: even though the lawn has no craters, there's a bigger challenge waiting for you: gophers. For the remainder of the project, gophers will replace craters as the main obstacles.

- Gophers move around by tunneling under the surface of the lawn. We display them on the lawn images to better manage the simulation run.
- Since they are under the lawn's surface, gophers can be below squares that still have grass and squares that have already been cut. We will add tokens to the set of possible scan results to address this new fact.
- If a gopher is located in the same square as a mower, it will come up to the surface just long enough to chew up some of the mower's wires and other electronics, causing the mower to be removed from the simulation run. The gopher is unaffected and continues to hunt other mowers.
- Since gophers are tunneling below the surface of the lawn, they travel much more slowly than the mowers. Gophers only move every **T** turns, where **T** represents the period (i.e., inverse of the frequency) of gopher moves. If the period **T** is 5, then the gophers move at the end of each turn that is evenly divisible by 5 – turns 5, 10, 15, 20, etc. If the period **T** is 3, then the gophers move at the end of each turn that is evenly divisible by 3 – turns 3, 6, 9, 12, etc. The movement period **T** will be given in the scenario file.
- The gophers move one at a time like the mowers. They move in the order listed in the scenario file. Each gopher moves using a very specific process:
  1. Determine the closest mower on the surface. If two or more mowers are equally close, then select one at random [use a uniform distribution with equal chances for each mower to be selected].
  2. Check the square that would move the gopher one square closer to the selected mower.
    - If another gopher is occupying that square, then don't move – two gophers should not occupy the same square.
    - Otherwise, move the gopher to that square.

- The gophers don't have a movement orientation, nor do they have any memory of previous actions. Every move is based only on the current state of the lawn.

Gopher image at <https://www.vecteezy.com/vector-art/170903-free-flat-gopher-collection-vector>

## Linear Scanning Capability

The mowers now have a new type of scanning to complement the current capabilities. Let's rename the current scanning type **cscan()** to represent "circular scanning" since it provides information about the eight squares surrounding the mower, regardless of the mower's current direction.

The mowers have a new scanning capability called **lscan()** to represent linear scanning, which provides the opportunity to collect information about squares much farther away from mower's current location. Linear scanning is based on the mower's current direction, and provides information about the squares extending from the front of the mower until the first fence square. Consequently, the **lscan()** results can contain a variable number of squares, and linear scanning takes significantly more energy than circular scanning. Consider the next image as an example:





















3					
2					
1					
0					
	0	1	2	3	4

Figure 1 - cscan() and lscan() results

Consider the distinctions between the **cscan()** and **lscan()** results as shown in Figure 1:

```
...
m0,cscan
grass,gopher_empty,empty,grass,grass,fence,fence,fence
m1,cscan
grass,grass,grass,grass,grass,grass,gopher_grass,grass
m0,lscan
gopher_empty,grass,fence
m1,lscan
gopher_grass,empty,mower,fence
...
```

## Limited Energy & Recharging

In earlier phases of the project, the mowers had "unlimited" power and range, with the exception of the limited number of turns in the simulation run. In the next phases of the project, mowers will have a limited amount of energy to perform actions during the simulation. Here are the guidelines:

- Each mower will have the same maximum energy capacity, measured in generic "units." The energy capacity will be given as an input value in the scenario file.
- The starting square for each mower will also contain a recharging pad. Any mower that stops on this pad will be fully recharged at the end of its action.
- Recharging pads are static and don't change locations.
- If a mower runs out of energy while away from a recharging pad, it stalls and effectively becomes an obstacle to the other active mowers for the remainder of the simulation run.
- Mowers will expend different amounts of energy based on their actions:
  - **lscan()**: 3 units of energy
  - **move()**: 2 units of energy
  - **steer()**, **cscan()**: 1 unit of energy
  - **pass()**: 0 units of energy

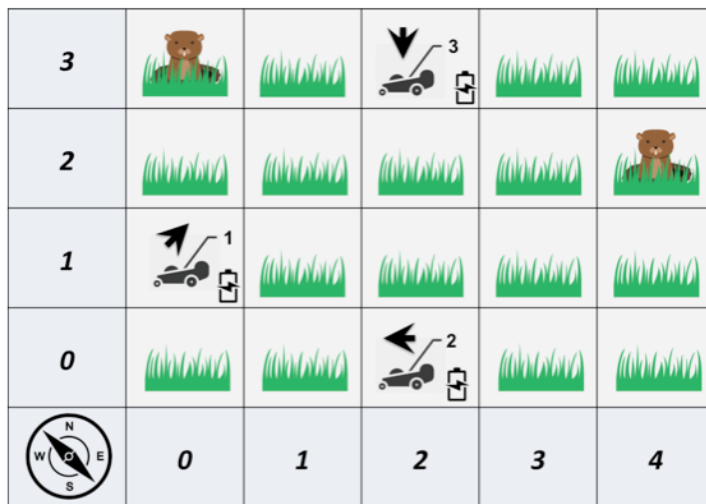
## Mower Communication

For the next iteration of the system, we will allow the mowers to collaborate in order to mow the lawn more efficiently. The clients liked the idea of the automated mowers, and are ready to engage "swarms" of mowers to tackle the much larger lawns.

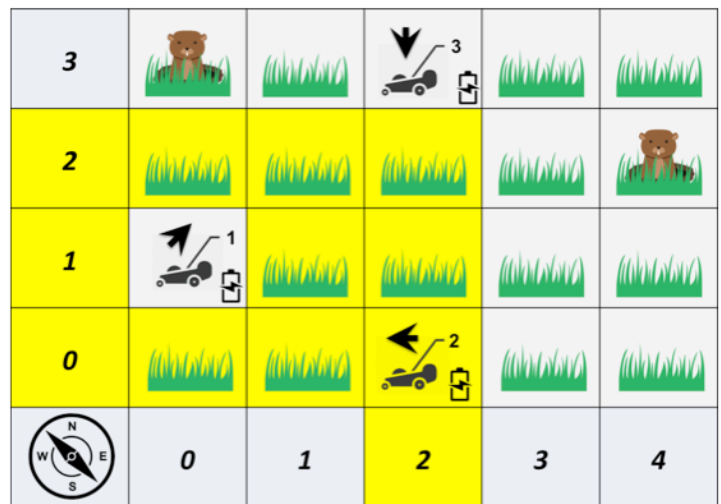
- The scenario file will specify the number of mowers that will be placed on the lawn, and the maximum number of mowers will be revised to eight (8) mowers per scenario. Your fundamental goal is still to cut all of the grass in as few turns as possible.
- At the start of the simulation, each mower knows:
  - Its unique identifier, referred to as "mowerID" in earlier assignments (e.g., **m0**, **m1**, **m2**)
  - Its original orientation (direction)
  - The total number of mowers on the lawn
- At the start of the simulation, each mower does NOT know:
  - Its "absolute location" with respect to the lawn (i.e. the X and Y coordinates)
  - The width or height of the lawn
  - The quantities or absolute locations of any of the obstacles
  - The locations (absolute or relative) of the other mowers
- The mowers are allowed to share data with each other using a persistent, shared communication channel. As each mower scans the lawn and learns more about its surroundings, it is allowed to combine its accumulated knowledge with the other mowers as you see fit.

## Test Scenario Example

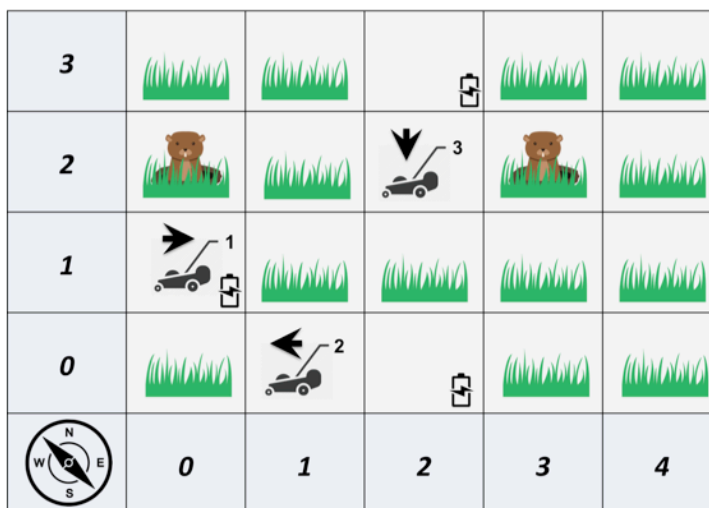
Consider the following sequence of actions. This scenario was designed to demonstrate how our earlier examples are modified in light of the recent requirement changes discussed above. It's not a comprehensive example that covers every possible sequence of events, but it does demonstrate the basic movements of multiple mowers and gophers (Figure 2).



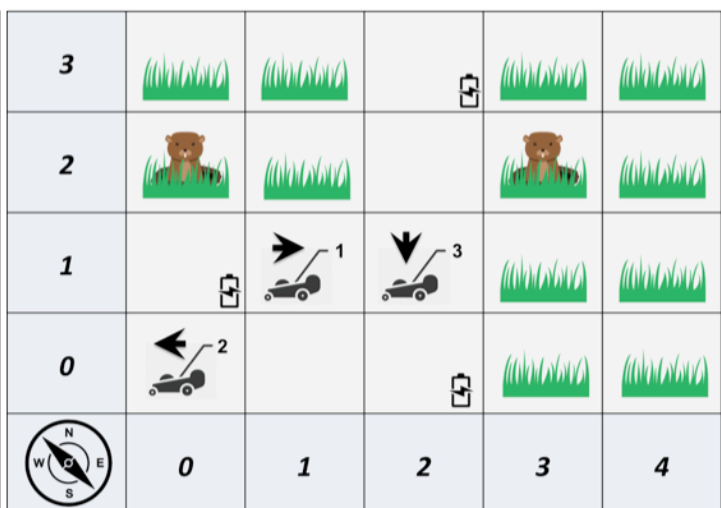
**starting state**



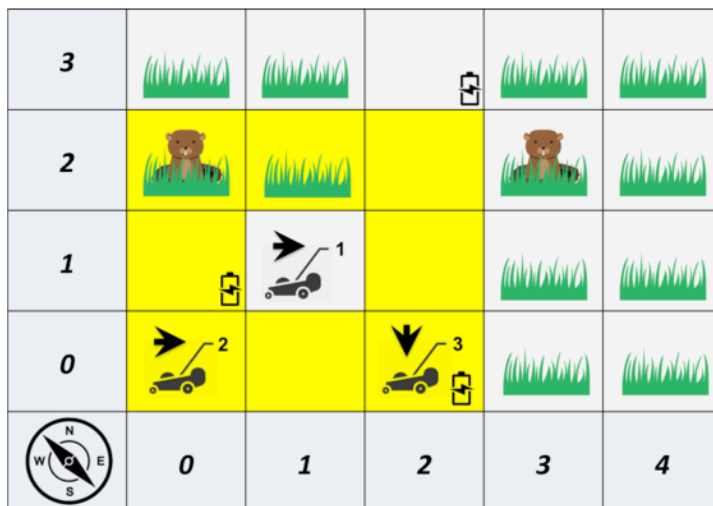
**turn #1**



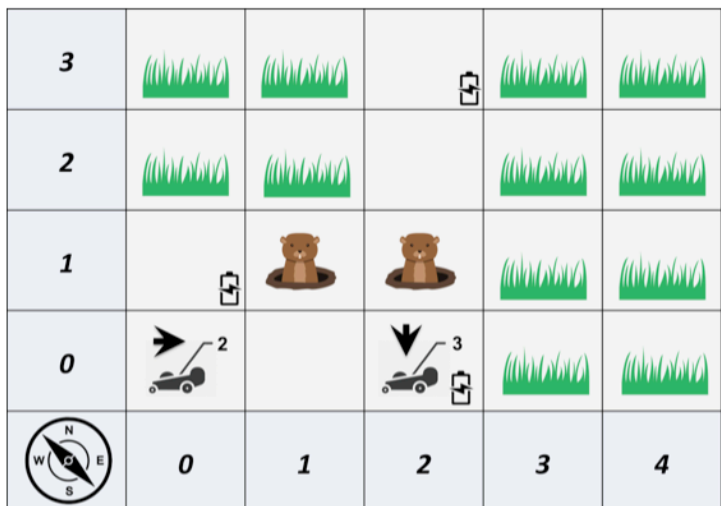
**turn #2**



**turn #3**



**turn #4 before gophers**



**turn #4 after gophers**

Figure 2 - Sequence of Actions

The scenario file format is also modified to support the updates to the requirements:

- (1) **<width (x-axis) of the lawn>**
- (2) **<height (y-axis) of the lawn>**
- (3) **<number of lawnmowers>**
- (4) **<initial location, direction and strategy of each lawnmower>**  
**[one line per lawnmower]**
- (5) **<maximum energy capacity for each mower>**
- (6) **<number of gophers>**
- (7) **<location of each gopher>**  
**[one line per gopher]**
- (8) **<gopher movement period (number of turns between gopher moves)>**
- (9) **<maximum number of turns>**

The starting state above is generated from the following file:

```
5
4
3
0,1,northeast,0
2,0,west,0
2,3,south,0
60
2
0,3
4,2
2
35
```

The lawn has a width of **5** squares and a height of **4** squares. There are three mowers starting at locations **(0,1)**, **(2,0)** and **(2,3)**, with initial directions as shown in the file. All three mowers are using a random strategy **(0)** for this scenario. And each mower has a maximum capacity of **60** units of energy. There are also **2** gophers initially located at **(0,3)** and **(4,2)**, and they move every **2** turns. Finally, the simulation run must end after **35** turns.

- During turn #1, the mowers perform scans to get a better idea of their surroundings. Mower #1 performs a circular scan, while mowers #2 and #3 perform linear scans.
- During turn #2, mower #1 changes its direction to the East, and mowers #2 and #3 both move forward. Also, since this turn is an even multiple of two, then the gophers get to move. Gopher #1 is two squares away from both mowers #1 and #3, so it selects mower #1 at random, and moves one square towards mower #1. Gopher #2 is closest to mower #2 (two squares away), so it moves one square towards mower #2.
- During turn #3, all of the mowers move one square forward.
- During turn #4 (before gophers), mower #1 executes a circular scan, mower #2 changes its direction to the East, and mower #3 moves forward. Once again, the gophers get an opportunity to move. Gopher #1 is closest to mower #1 (one square away), so it moves onto the same square as mower #1, removing it from the simulation run. Gopher #2 is two squares away from mower #3, so it moves one square towards mower #3.

The state of the simulation run after turn #4 has been completed is shown above. Note the energy levels of the remaining mowers at this point of the simulation. Mower #2 executed a linear scan, then moved forward twice, and changed direction. This means the mower #2 expended 8 units of energy with a remaining total of 52 units. Mower #3 executed a linear scan and moved forward three times thus expending 9 units of energy with a remaining total 51 units. However, since mower #3 concluded turn #4 on a recharging pad, its energy level is restored to 60 units.

### Scanning, Movement & Output File Updates

The updated requirements also require that we update the acceptable terms for scan results:

**{grass, empty, gopher\_grass, gopher\_empty, fence, mower}**

Suppose that the contents in our example scenario are stored in a file named **scenario0.csv**. The output log file for the actions as described above would be:

```
> java -jar osmowsis.jar scenario0.csv
m0,cscan
grass,grass,grass,grass,grass,fence,fence,fence
m1,lscan
grass,grass,fence
m2,lscan
grass,grass,mower,fence
m0,steer,east
ok
m1,move
ok
m2,move
ok
g0,m0,0,2
ok
g1,m2,3,2
ok
m0,move
ok
m1,move
ok
m2,move
ok
m0,cscan
grass,empty,empty,mower,empty,mower,empty,gopher_grass
m1,steer,east
ok
m2,move
ok
g0,m0,1,1
ok
g1,m2,2,1
ok
...
```

Note that the gopher moves are given in the format:

**<gopherID>,<selected\_mowerID>,<X\_destination>,<Y\_destination>**  
**ok**



## Simulation Monitoring Updates

Your initial success in developing this system has spurred new requests from your clients. Now, they are excited about being able to test a variety of lawnmower models and strategies with their system.

- Unfortunately, they are also concerned that some unscrupulous mower developers will attempt to give their mower models an unfair advantage by accessing the full map of the lawn (which includes information like the current numbers and absolute locations of the obstacles), instead of using the scanning functions to assemble that knowledge per the requirements.
- Another concern is the need to display factors like the size of the lawn and the actual number grass squares from the very beginning of the simulation run, before the mowers have been able to determine those values.
- Consequently, your design must ensure that the full map of knowledge will not be compromised by dishonest mower designs. One way is to implement a clear and distinct simulation monitor that provides the overall control for the different objects during the simulation run. There are other ways to implement this, but the burden of addressing these concerns rests on you.

## Evaluating Your UML Submissions

- You must generate your diagrams using an automated tool (e.g. Argo UML, LucidCharts, Microsoft Visio, etc.) so that they are as clear & legible as possible. Even PowerPoint is allowed, though this is an excellent opportunity to use a tool that is more appropriately designed for UML as opposed to a general drawing tool like PowerPoint. The choice of tool(s) is yours; however, you should do a “sanity check” to make sure that your final diagrams are readable when exported to PDF; or, if necessary, some reasonable graphical format (PNG, JPG or GIF).
- **Do not auto-generate your UML diagrams by reverse engineering source code.** For this assignment to be effective from a learning standpoint, you should spend time generating your own UML design diagrams from the requirements. Diagrams that are auto-generated may lose up to 50% of the total points for the assignment as a whole.
- We prefer that you submit your diagrams in dark, rich colors (particularly black). Diagrams submitted in certain colors, especially pastel or lighter shades, can be difficult to read.
- You must designate which version of UML you will be using – either 1.4 (the latest ISO-accepted version) or 2.0 (the latest OMG-accepted version). We highly recommend that you label all diagrams with a header that includes your name and the UML version being used. There are significant differences between the UML versions, so your diagram must be consistent with the standard you’ve designated. Either version is acceptable at this point in the course.

## 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. 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.