Overview

Task description:

Our client is an information-gathering agency that we cannot name for legal and contractual reasons. This client employs field agents that often find themselves gathering information in highly secure facilities.

Your task is to write a computer program in C# and .NET 6.0 that presents a menu allowing the agent to specify certain obstacles to the agent's mission, and provides the agent with information upon request, including the ability to request a safe path to a particular objective.

You are required to use an object-oriented approach to implementing the solution, and you are required to produce high quality, thoroughly documented source code so that the software can be maintained later. Your code must also feature proper use of exception handling.

Unit Learning Outcomes assessed:

ULO 3: Document software designs and computer code to ensure it is easy to maintain and complies with industry standards.

ULO 4: Apply object-oriented design and programming techniques to create software solutions.

Overview of assessment item

Estimated time for completion	Weighting	Group or Individual	How I will be assessed
25-50 hours	**50**% of final grade	Individual	See marking criteria below

What you need to do

Your software will present a menu-based user interface to the agent, presenting a series of options. Each option will have its own menu entry, consisting of a one-letter code and a description. The user will enter in a code to direct the software to perform one of the available actions. Here is one such example of a session with a user (user input is shown underlined):

```
Select one of the following options
g) Add 'Guard' obstacle
```

```
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
g
Enter the guard's location (X,Y):
2,1
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
d
Enter your current location (X,Y):
2,2
You can safely take any of the following directions: SEW
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
x
```

In this example, the user specifies that they know there is a guard located at (2,1) - which means 2 klicks east and 1 klick south of the map origin. The user then requests a list of safe directions, giving their own location as (2,2) - 2 klicks east and 2 klicks south of the origin. The program then reports that safe directions to travel in are **south**, **east** and **west** (as going north would result in the agent being caught by the guard. The user then enters `x` to close the program.

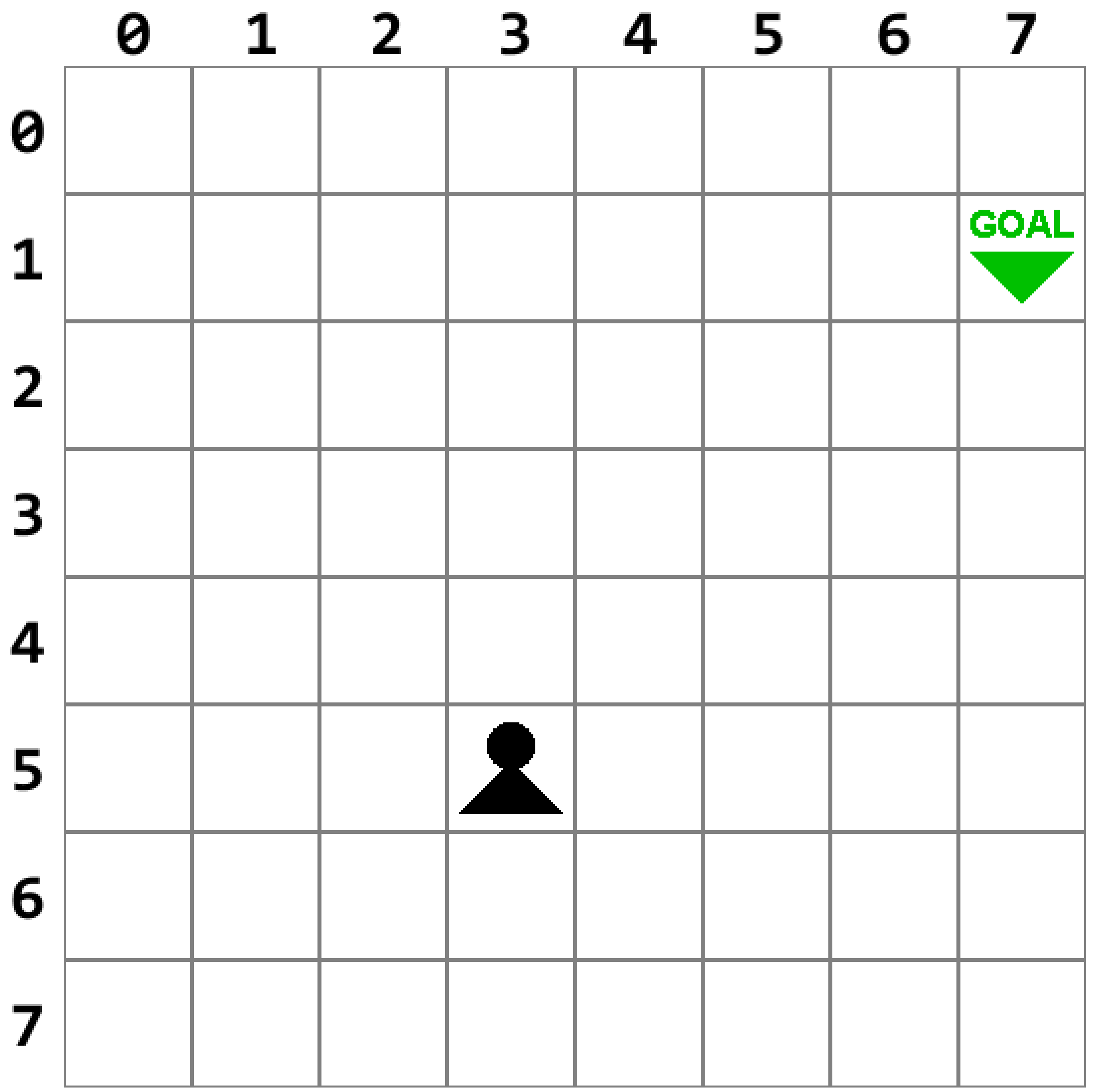
Another use of the program is navigation:

```
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
```

```
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
p
Enter your current location (X,Y):
3,5
Enter the location of your objective (X,Y):
7,1
The following path will take you to the objective:
NNNNEEEE
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
x
```

In this example, the agent is located 3 klicks east and 5 klicks south of the origin point, and the mission objective is located 7 klicks east and 1 klick south of the origin point. The agent has not observed any obstacles to the mission. The system then recommends that the agent go north for 4 klicks and east for 4 klicks.

Visualised, this problem would look like this:

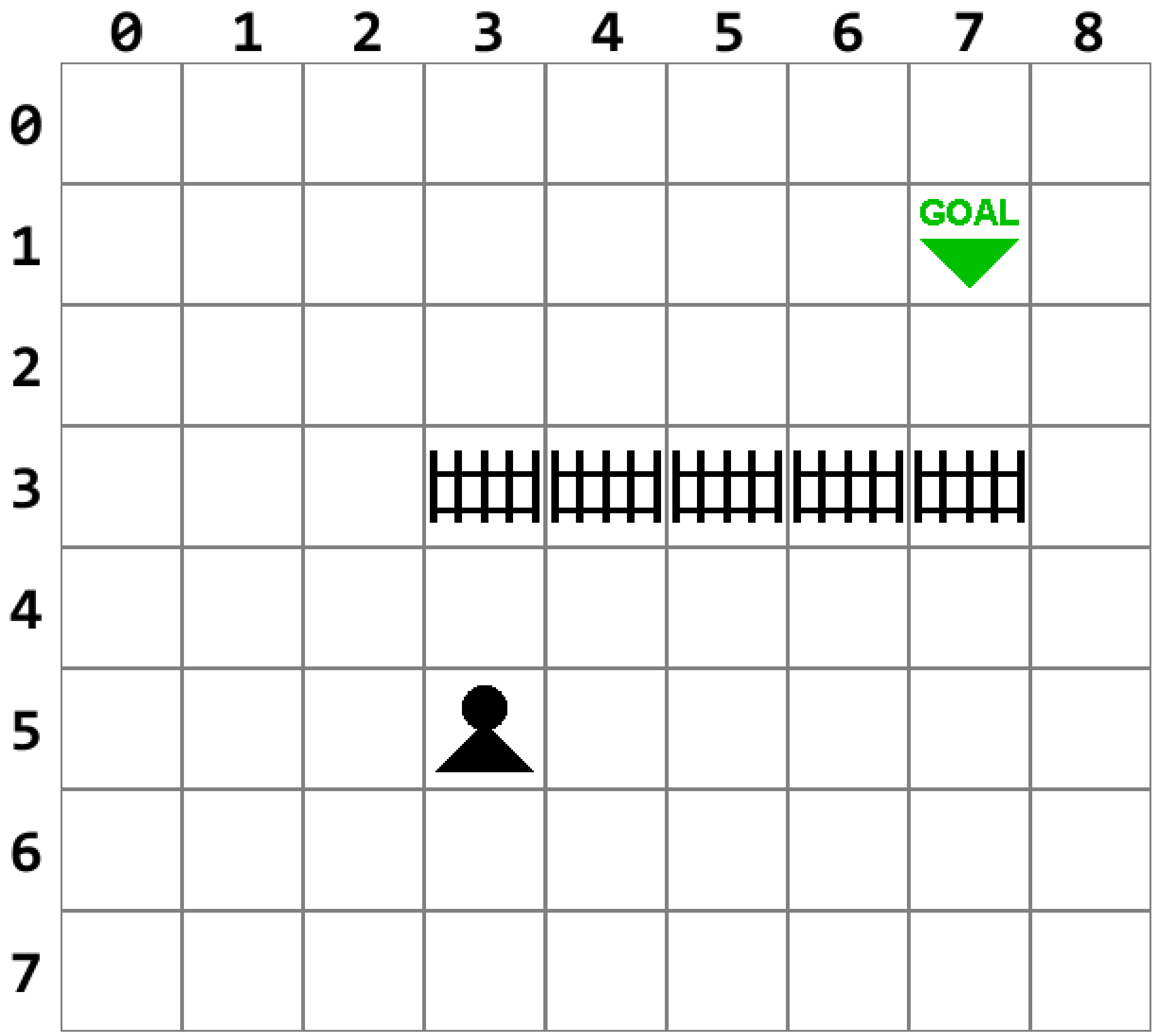


(Note that the dimensions are not restricted to 8,8 – this is purely for the purposes of this visualisation)

As there is nothing blocking the agent's path, any path that results in the agent moving 4 klicks north and 4 klicks east will be an acceptable solution. A path that results in the agent getting some of the way (but not all of the way) towards the goal will result in partial credit.

Most of the time, getting to the objective will be less straightforward, as there will be obstacles obstructing the agent's path. Each obstacle will ask for different information when it is added to the scenario. The previous example (a guard) is an obstacle that occupies a single location. Another example of an obstacle is a fence:

```
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
f
Enter the location where the fence starts (X,Y):
3,3
Enter the location where the fence ends (X,Y):
7,3
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
p
Enter your current location (X,Y):
3,5
Enter the location of your objective (X,Y):
7,1
The following path will take you to the objective:
NEEEEENNNW
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
x
```



In this case, the agent has observed a fence stretching from 3 klicks east and 3 klicks south to 7 klicks east and 3 klicks south. The fence is considered to obstruct all of the squares that it occupies, so the agent will have to move around it (the agent cannot enter the squares at 3,3 or 4,3 or 5,3 or 6,3 or 7,3 at all).

The software can also be used to provide a simple text-based visualisation of the obstacles that have been defined so far:

```
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
f
Enter the location where the fence starts (X,Y):
3,3
Enter the location where the fence ends (X,Y):
7,3
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
g
Enter the guard's location (X,Y):
4,1
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
g
Enter the guard's location (X,Y):
6,1
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
```

```
x) Exit
Enter code:
m
Enter the location of the top-left cell of the map (X,Y):
0,0
Enter the location of the bottom-right cell of the map (X,Y):
8,7
.........
....g.g..
.........
...fffff.
.........
.........
.........
.........
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
x
```

In this case, a map is displayed of the specified area, showing a `g`, `f`, `s` or `c` depending on whether that location is protected by a guard, fence, sensor or camera, and a `.` for all safe locations.

User interface

Your software will be tested via Gradescope and will therefore need to implement the following user interface requirements **precisely**. Remember that you can test your software by submitting via Gradescope at any time, and it is highly encouraged that you do so **often**. - minor issues can be caught very easily, but if you wait until the last week to submit you may find yourself losing many marks due to minor deviations from the spec.

Your program will begin by displaying a menu of choices:

Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle

c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:

Your menu may differ from this slightly if you have implemented **a custom obstacle**, but each menu item will be of the same format - a single lowercase letter for that menu item's code followed by `)` and a brief description.

After displaying Enter code: the program will wait for input from the user. If the user enters in something other than one of the single-letter codes listed in the menu it will display the message `Invalid option.`, then display `Enter code:` again and wait for input:

```
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
z
Invalid option.
Enter code:
Hello
Invalid option.
Enter code:
123
Invalid option.
Enter code:
G
Invalid option.
Enter code:

Invalid option.
Enter code:
```

Note that **every** situation where the program requests input from the user will be handled in a similar way - by displaying an error, then displaying the prompt again and waiting for another line of text to be entered.

After this, the program's next action will depend on the option chosen.

g, f, s, c) Add 'Guard/Fence/Sensor/Camera' obstacle

All of these options are used to specify obstacles. For details of what information they request and how they work, see the Obstacles section below. Each obstacle will request different information when added, and will obstruct the agent in different ways. For instance, if the obstacle is a guard, only its location will be requested. For a fence, its starting and ending locations. For a security camera, its location and direction. In all of these cases, the normal input rules will apply - if invalid input is entered, the program will display `Invalid input.` followed by the same prompt again. Any number of obstacles can be specified in this way, and the same type of obstacle can be specified multiple times - e.g. there might be multiple fences.

d) Show safe directions

If this option is chosen, the program will prompt the user for their current location:

```
Enter your current location (X,Y):
0,0
```

If the user enters anything other than an integer coordinate pair, the program will display `Invalid input.`, then repeat the prompt.

If the location specified is blocked by an obstacle (e.g. a guard was specified at 0,0, or a fence was specified at -3,0 to 3,0), the program will display:

```
Agent, your location is compromised. Abort mission.
```

Otherwise, if the location is itself safe but obstructed on all four sides by obstacles, the program will instead display:

```
You cannot safely move in any direction. Abort mission.
```

Otherwise, the program will display a list of directions (in the order NSEW) that the agent can safely move in from their current location. If all four directions are unobstructed:

```
You can safely take any of the following directions: NSEW
```

If, on the other hand, there are obstacles blocking the squares north and east of the agent's location:

```
You can safely take any of the following directions: SW
```

Regardless of the result, the program will then return to the menu.

m) Display obstacle map

If this option is specified, the program will request two pairs of coordinates - one of the top-left cell to display, one of the bottom-right cell. These coordinates are used to define the map range.

```
Enter the location of the top-left cell of the map (X,Y):
0,0
Enter the location of the bottom-right cell of the map (X,Y):
7,7
........
........
........
........
........
........
........
........
```

If the user enters something other than an integer coordinate pair, the program will display `Invalid input.`, then repeat that request.

For example:

```
Enter the location of the top-left cell of the map (X,Y):
4,5
Enter the location of the bottom-right cell of the map (X,Y):
7
Invalid input.
Enter the location of the bottom-right cell of the map (X,Y):
```

After both sets of coordinates have been input, if the bottom-right cell is **west** or **north** of the top-left cell, the program will display `Invalid map specification.`, then ask for both pairs of coordinates again:

```
Enter the location of the top-left cell of the map (X,Y):
4,5
Enter the location of the bottom-right cell of the map (X,Y):
8,4
Invalid map specification.
Enter the location of the top-left cell of the map (X,Y):
```

Otherwise, if the coordinates are valid, the program will display a text-based map (one line per row) of the specified area, such that the top-left character corresponds to the first pair of coordinates and the bottom-right character corresponds to the second pair of coordinates. Valid characters that can appear in the map are: `.` `g` `f` `s` `c`, for safe squares, guards, fences, locations protected by sensor and locations protected by camera respectively.

The program will then return to the menu.

p) Find safe path

When this option is selected, the program will prompt for the agent's current location:

```
Enter your current location (X,Y):
```

If something other than an integer coordinate pair is provided, the program will display Invalid input. and prompt the user again.

The program will then request the location of the agent's mission objective:

```
Enter the location of the mission objective (X,Y):
```

If the location is equal to the agent's current location, the program will print `Agent, you are already at the objective.` and then return to the menu.

If the location of the mission objective is obstructed by an obstacle (e.g. the objective is at 3,3 and there is a guard at 3,3) the program will print `The objective is blocked by an obstacle and cannot be reached.` and then return to the menu.

If there is a safe path between the agent's location and the objective, the program will print out:

```
The following path will take you to the objective:
```

...on a line by itself, followed by a series of characters consisting of N, S, E and W, indicating that the agent must move 1 klick north, south, east or west respectively for each step:

```
Enter your current location (X,Y):
0,0
Enter the location of your objective (X,Y):
4,4
The following path will take you to the objective:
SSSSEEEE
```

The program will then return to the menu.

If there is no safe path to the objective (due to obstacles), the program will print `There is no safe path to the objective.`, then return to the menu.

x) Exit

If Exit is chosen, the program will immediately terminate.

Coordinate system

This program utilises integer Cartesian coordinates, where the X coordinate indicates that many klicks **east** of an arbitrary origin point and the Y coordinate indicates that many klicks **south** of that point. **Negative coordinates are acceptable**. During our testing, we will restrict input values to within the range of values supported by a System.Int32 (-2147483648 to 2147483647). Your program is not expected to have to calculate routes that will go outside this range.

Note that 90% of the test cases will only use coordinates between (0,0) and (19,19), so if your program is able to handle values in that range, you will still be able to pass the majority of test cases.

Obstacles

There are a variety of obstacles that may be present. Obstacles are individually configured - for example, the Fence obstacle has start and end locations - and they obstruct the agent in different ways. Your program is expected to take an object-oriented approach to handling these obstacles, such that new obstacles can be added to your program with minimal effort and disruption to the rest of the program.

This section presents a catalogue of different obstacles that may be present. Note that all places where obstacles ask for input will handle the prompt in the usual way (displaying `Invalid input.` and then prompting the user again).

Each obstacle is specified in the format `x) Y`, where x is the letter code used to select the obstacle in the menu and Y is the descriptive name of the obstacle, which will also be shown in the menu (in the format `Add 'Y' obstacle`).

g) Guard

When a guard is added, it will prompt with the following:

```
Enter the guard's location (X,Y):
```

The guard will obstruct the agent from entering the specified square. For instance, if the guard is located at (7,3), the agent will not be able to enter that square.

f) Fence

When a fence is added, it will prompt with the following:

```
Enter the location where the fence starts (X,Y):
```

and

```
Enter the location where the fence ends (X,Y):
```

The fence will then extend between those two coordinates (inclusive of the endpoints). Fences can run either horizontally or vertically - in other words, the two coordinates specified must share an X coordinate or a Y coordinate. In addition, the two coordinates must be different (a fence cannot occupy just 1 square). If the coordinates entered do not follow these two restrictions, the program will display `Fences must be horizontal or vertical.`, then ask for coordinates again, starting from `Enter the location where the fence starts (X,Y):`

The fence will obstruct the agent from entering any square within its bounds. For example, a fence going from (4,3) to (4,6) will stop the agent from entering (4,3), (4,4), (4,5) or (4,6).

Note that the two fence coordinates can come in any order - the same fence could be specified as going from (1,1) to (1,10) or from (1,10) to (1,1).

s) Sensor

An sensor has a location and a range. If the agent moves within the range of the acoustic sensor, the agent will be caught.

The sensor will prompt with the following:

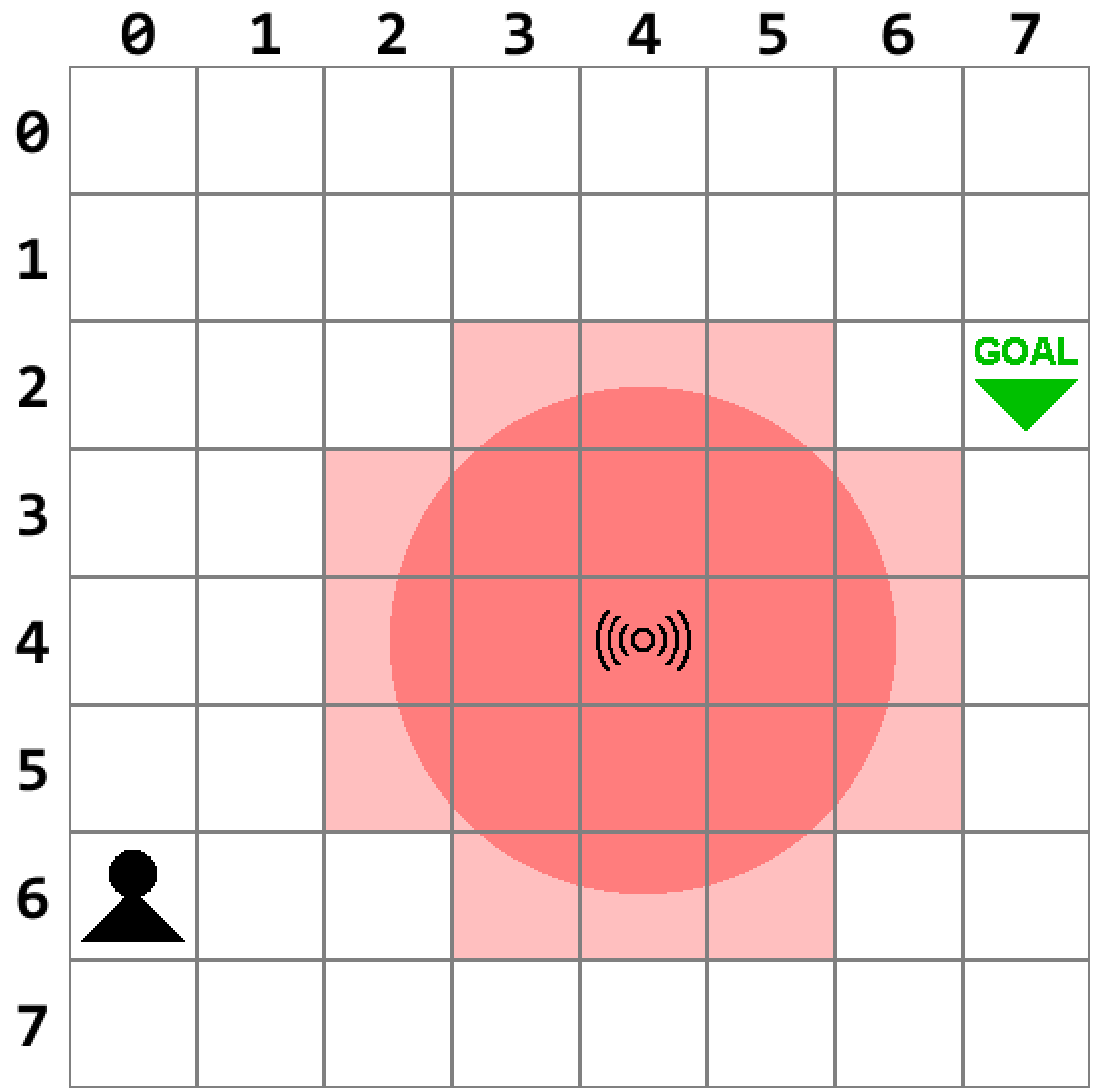
```
Enter the sensor's location (X,Y):
```

and

```
Enter the sensor's range (in klicks):
```

The sensor's range is a **floating point value** and a square is considered to be obstructed by the sensor if it is within the sensor's range, measured in Euclidean distance. The range cannot be less than or equal to 0 - this is considered invalid input.

For example, if the sensor is at (4,4) and has a range of 2.5:



Using the Pythagorean formula, we can compute the distance between each square and the sensor

So, for example, given that the sensor is at (4,4), we can check if (2,6) is within the range of the sensor:

2.828... is greater than 2.5, so (2,6) is not within the sensor's range.

Now consider (3,6):

2.236... is less than 2.5, so (3,6) is within the sensor's range (and therefore the agent cannot enter that square).

With this, we can see that squares (3,2), (4,2), (5,2), (2,3), (3,3), (4,3), (5,3), (6,3), (2,4), (3,4), (4,4), (5,4), (6,4), (2,7), (3,7), (4,7), (5,7), (6,7), (3,8), (4,8) and (5,8) are all within 2.5 klicks of the sensor. This means the agent cannot enter any of those squares.

c) Camera

Cameras have a location and a direction, and can spot anything within a 90 degree cone of vision centred on that direction. Cameras have infinite range.

The camera will prompt with the following:

```
Enter the camera's location (X,Y):
```

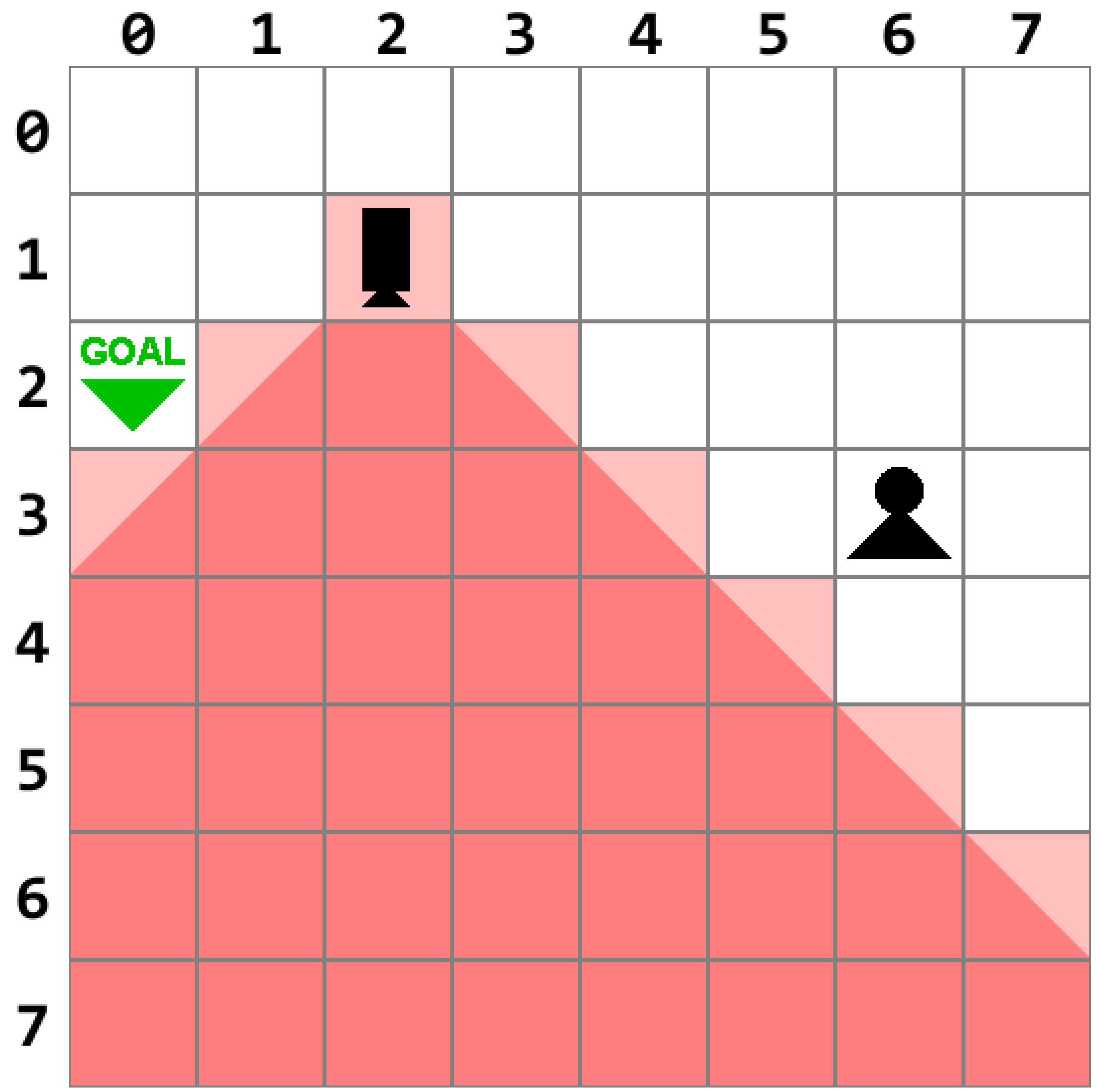
and

```
Enter the direction the camera is facing (n, s, e or w):
```

The location works as previously, while the direction prompt requires that user input a single lowercase character that is one of `n`, `s`, `e` or `w` for north, south, east or west respectively. This gives the direction the camera is facing. If anything other than `n`, `s`, `e` or `w` is entered, the program will respond with `Invalid direction.` and prompt again.

Cameras have infinite range and will obstruct the agent entering its 90 degree cone of vision, which extends from 45 degrees clockwise from the camera's direction to 45 degrees anticlockwise from the same direction.

Visualised, the camera's cone of influence looks like this (for a camera at (2,1) facing south):



The agent is unable to enter the camera's square, nor any of the squares that the camera can see. If the camera is located at (2,1) and facing south it will obstruct (2,1) and (1,2), (2,2), (3,2) and (0,3), (1,3), (2,3), (3,3), (4,3) and so on, extending infinitely.

?) Custom obstacle type

As part of the functionality for your program, and to demonstrate your program's extensibility, we want you to create an additional obstacle type of your choice. The requirements are as such:

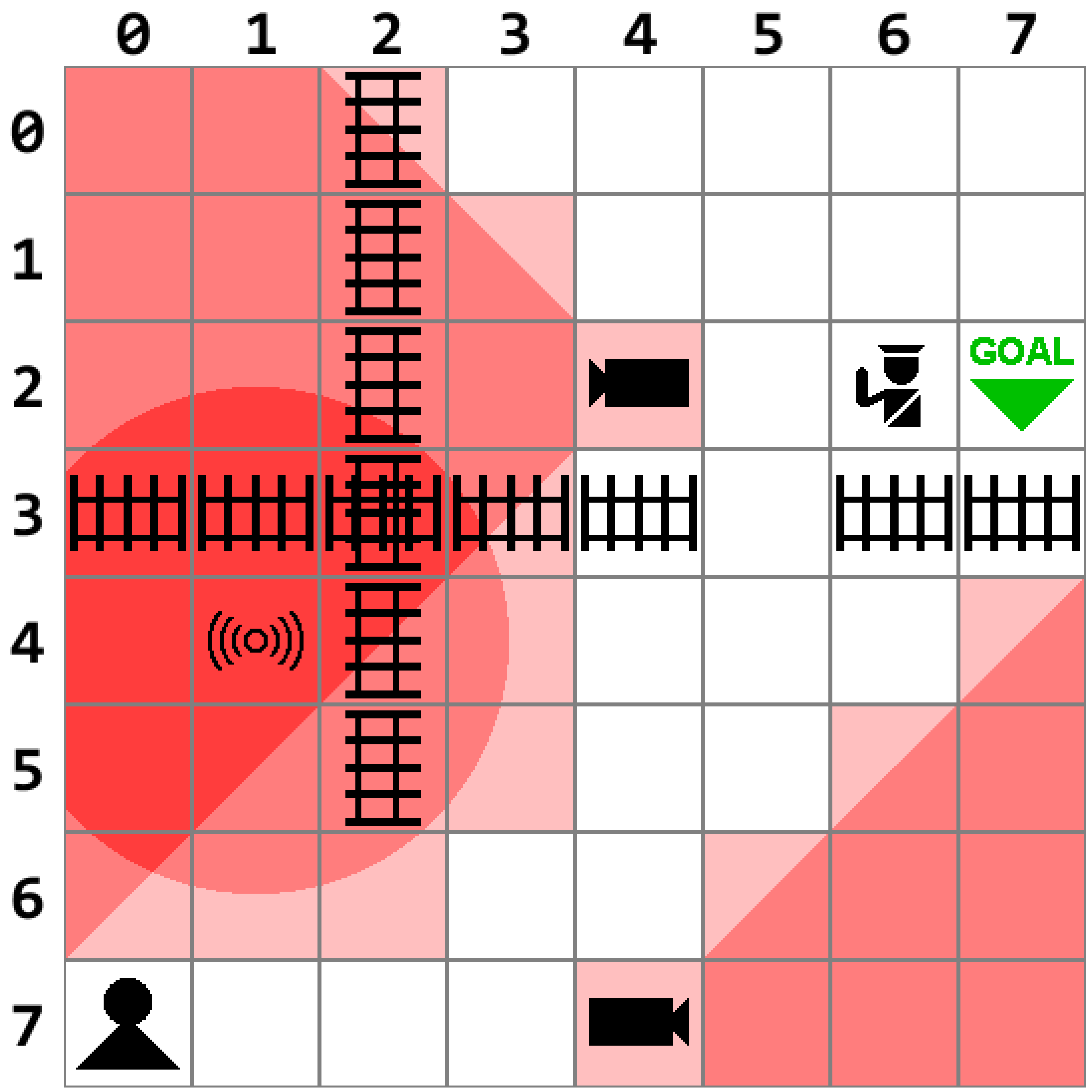
- It must use one of the unused lowercase letters for its 1-letter code: a b e h i j k l n o q r t u v w y z
- It must be able to obstruct the agent
- It must prompt the user for some information during creation (the same way all the other obstacles ask for location / sensor radius / direction etc.)
- It must have some unique functionality that is not already covered by existing obstacle types

You are not limited to only creating one custom obstacle, however one is sufficient to get full marks. The hard limit is 18 (as you will then run out of lowercase letters for codes.) Your obstacle must be included in the menu along with the other obstacles, and displayed in the same format - for example, if your custom obstacle is `Laser barrier` and the code for it is `l`, your menu will look like this:

```
Select one of the following options
g) Add 'Guard' obstacle
f) Add 'Fence' obstacle
s) Add 'Sensor' obstacle
c) Add 'Camera' obstacle
l) Add 'Laser barrier' obstacle
d) Show safe directions
m) Display obstacle map
p) Find safe path
x) Exit
Enter code:
```

Multiple and overlapping obstacles

Many scenarios will involve multiple obstacles being present. When multiple obstacles are present, they all contribute to preventing the agent from entering certain squares. Multiple obstacles do not interfere with each other - for instance, sensors and cameras can overlap with fences, guards and each other, and sensors and cameras can detect the agent through fences. Multiple fences can overlap, as well.



When multiple obstacles are overlapping, the map display can choose to display any of the obstacles obstructing a particular tile. For example, the above scenario might look like this in the map:

```
Enter the location of the top-left cell of the map (X,Y):
0,0
Enter the location of the bottom-right cell of the map (X,Y):
7,7
```

```
ccf.....
ccfc....
ccfcc.g.
fffff.ff
ccfs...c
ccfs..cc
css..ccc
....cccc
```

Marking criteria

There are a total of 50 marks available for this assessment item. Your submission will be marked with a mix of automated and manual processes.

Functionality (core task): 20 marks

When you submit your code to Gradescope, it will be automatically run against a battery of eighty (80) test cases, checking many different combinations of obstacles, agent location and objective location. These have been designed to vary in complexity, ensuring that you can pass some test cases even with a very early implementation. For this reason it is recommended that you submit to Gradescope **early and often**, to ensure that your code is successfully compiling and running on our grading platform.

Each test case have the same structure:

- It will use your menu to add some number of obstacles to the scenario (0, 1 or multiple obstacles)
- It will do one of `d) Show safe directions`, `m) Display obstacle path` or `p) Find safe path`
- It will enter x to exit the program

The marking for the test case will depend on the type of action performed

- For 'Show safe directions' your program will receive full marks if it displays the correct result and no marks otherwise
- For 'Display obstacle map' your program will receive full marks if the entire map is displayed correctly and no marks otherwise
- For 'Find safe path':
  - If your program successfully outputs a route and the agent can follow it to get to the objective without being blocked by an obstacle, you will receive **full marks** for that test case
  - If your program successfully outputs a route and the agent can follow it and get close to the objective without being blocked by an obstacle (in other words, walking into a fence or guard, or being spotted by a sensor or camera), you will receive marks for that test case based on how close the agent got: (d - r) ÷ d marks, where r is the Manhattan distance away from the objective that the agent got to and d is the

Manhattan distance between the agent's starting position and the objective. If the result is negative (that is, the agent wound up farther from the objective than the starting position was), no marks will be awarded

- If your program successfully outputs a route but it leads to the agent being obstructed/spotted by an obstacle (walking into a fence or being spotted by a camera, for example), you will receive no marks for that test case
- If your program does not successfully output a route, you will receive no marks for that test case
- If there is no valid route and your program displays `There is no safe path to the objective.`, you will receive full marks for that test case
- **Due to the scoring methodology, it is most important to handle the obstacles correctly and ensure that the agent does not stray into them. Even if you are unable to always find a path to the objective, if you direct the agent to get as close as possible to the objective without running into an obstacle and then stop, you should still get some marks.**

There are also a handful of test cases that purposefully provide invalid input. Your program will receive full marks for that test case if it behaves correctly in that situation, and no marks otherwise.

It is important that your program produce output **exactly** as specified in the problem description, user interface and obstacles list above. Variations in the output formatting (i.e. the messages your program prints out) may cause the autograder to misinterpret your input and cause you to fail test cases that you would have otherwise passed. If you have 'Find safe path' implemented, the most important line for those tests is `The following path will take you to the objective:`, because the line directly after that one appears is assumed to be the path. In general, however, just use the text given in this specification exactly and your program should have no issues with the autograder. One last thing is to make sure you use **Console.ReadLine()** for reading all text - other Console functions for reading keys may not work with the autograder, even though they might seem to be fine when run on your machine. Every input is on a line by itself and terminated by pressing the enter key, and that goes for selecting options from the menu, entering coordinates, entering a range for the sensor and entering a direction for the camera.

One last note: While we do not vary the test cases from run to run, if we discover during manual inspection of your program that you have **hard-coded** in our test cases and results (that is, instead of implementing the obstacle logic, you have built your program to specifically recognise our test cases and produce the output we expect) you will receive a score of 0 for functionality, as this is not be considered useful work.

Marking criterion

7: High Distinction (17-20 marks)	Submitted program implements the specification without hard-coding results and successfully passes 68/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 17 or greater.)
6: Distinction (15-16.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 60/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 15 or greater.)
5: Credit (13-15.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 52/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 13 or greater.)
4: Pass (10-12.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 40/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 10 or greater.)
3: Marginal fail (8-9.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 32/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 8 or greater.)

2: Fail (5-7.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 20/80 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 5 or greater.)
1: Low Fail (0-4.99 marks)	Submitted program passes fewer than 20 tests on Gradescope, or does not compile/run, or uses hard-coded results to attempt to subvert the assessment process

Functionality (custom obstacle): 3 marks

Your custom obstacle implementation will be checked by running your software and checking the source code.

Marking criterion

7-6: (High) Distinction (3 marks)	Custom obstacle serves a unique and novel purpose AND the submission meets the requirements for achievement level 4-5 as stated below
4-5: Pass/Credit (2 marks)	Custom obstacle is implemented and listed in the obstacle menu and behaves as expected
2-3: Fail (1 mark)	Custom obstacle is implemented and listed in the obstacle menu, but it does not obstruct the agent OR it does not possess unique functionality possessed by existing obstacle types OR it does not collect input from the user that is used to customise this instance of the obstacle
1: Low Fail (0 marks)	No custom obstacle implemented OR custom obstacle not listed in the obstacle menu OR programming errors / bugs / glitches are encountered when attempting to test the custom obstacle

Object-oriented design and implementation: 6 marks

Your object-oriented design and implementation will be checked by looking at your class inheritance hierarchy and examining source code to check your use of polymorphism and encapsulation, as well as the private/protected/internal/public state of your methods, fields and properties.

Marking criterion

7: High Distinction (6 marks)	Leverages advanced object-oriented techniques like polymorphism and inheritance to elegantly solve the task AND meets the requirements for achievement levels 4-6 as stated below
6: Distinction (5 marks)	Non-const fields are always private and all external access to them is controlled through public methods and/or properties which enforce appropriate domain restrictions AND the submission meets the requirements for achievement levels 4-6 as stated below
5: Credit (4 marks)	Features a high quality, thoughtful object-oriented design with appropriate classes, with minimal to no use of non-const public fields AND meets the requirements for achievement level 4 as stated below

4: Pass (3 marks)	The program is implemented using classes with appropriate subdivision of responsibilities and some kind of object-oriented design (classes have appropriate fields and methods)
2-3: Fail (2 marks)	The program is implemented using classes but with little use of object-oriented design or programming (e.g. mostly static methods), OR too many methods/fields are unnecessarily public/internal
1: Low Fail (0 marks)	Not an object-oriented design (e.g. all static methods or only one class) OR too little code submitted to evaluate the appropriateness of the design

Coupling and cohesion: 6 marks

Your coupling and cohesion will be checked by examining class interdependencies and looking at the methods/fields/properties present in each class.

As a refresher, **coupling** is something you are trying to minimise (it means unhealthy dependencies between classes that make your code less reusable), while **cohesion** is something you are trying to maximise (it means each class is responsible for a single unified task and all methods/fields/properties support that task).

Marking criterion

7: High Distinction (6 marks)	Coupling is minimal AND the submission meets the requirements for achievement levels 4-6 as stated below
6: Distinction (5 marks)	The worst types of coupling (content, common, temporal) are avoided AND the submission meets the requirements for achievement levels 4-5 as stated below
5: Credit (4 marks)	Classes are highly cohesive (responsible for a particular task and all methods/properties/fields contribute to this task) AND the submission meets the requirements for achievement level 4 as stated below
4: Pass (3 marks)	Classes are mostly cohesive, potentially with some responsibilities creeping into other classes
2-3: Fail (2 marks)	Classes have poor cohesion and responsibilities are mixed
1: Low Fail (0 marks)	Not divided into multiple classes OR too little code submitted to evaluate the quality of the design

Abstraction: 6 marks

Your use of abstraction is marked by analysing your source code. Under this criterion, the following will be penalised:

- Overly long or complex methods that should be broken up into simpler methods
- Repetitive coding constructs that should be a loop instead

- Sections of code or methods that are overly similar to other sections of code/methods and should be abstracted out into a common method
- Use of magic numbers (value literals other than -1, 0 or 1) in your code (these should be replaced by declaring a `const` field with a descriptive name containing the value)

In general, we are looking for anything other than code clarity and your OOP design (as those are marked in their own criteria) that makes your code more difficult to maintain.

Marking criterion

7: High Distinction (6 marks)	The submission contains no code that could be improved by abstracting out to an additional method AND the submission meets the requirements for achievement levels 4-6 as stated below
6: Distinction (5 marks)	No use of magic numbers and no repetitive coding constructs that should be a loop instead AND the submission meets the requirements for achievement levels 4-5 as stated below
5: Credit (4 marks)	The program is mostly broken up into appropriately-sized methods, with nearly all methods being under 50 lines long AND the submission meets the requirements for achievement level 4 as stated below
3-4: Pass (3 marks)	The program features no or only minor instances of repetition
2-3: Fail (2 marks)	The program features long methods with large amounts of repetition
1: Low Fail (0 marks)	Everything is in main() OR too little code submitted to evaluate your use of abstraction

Code clarity and comments: 5 marks

Code clarity is marked by looking at how your choice of identifiers (names of classes, methods, properties, fields, local variables) and your program's flow (use of loops / branching) affects the comprehensibility and maintainability of your code. For comments, we are looking for two different kinds: **top-level XML tag comments** (///) and body-level comments (usually single-line // comments). If your code has a 'goto' in it, unless you have an extremely good reason the maximum mark you can get for this section is 1.5/5.

Marking criterion

7: High Distinction (5 marks)	All public classes/methods/properties feature C# XML documentation comments at the top level explicitly defining the code's external interface AND the submission meets the requirements for achievement levels 4-6 as stated below
6: Distinction (4 marks)	All or nearly all public classes, methods and properties feature useful and informative comments at the top level AND the submission meets the requirements for achievement levels 4-5 as stated below
5: Credit (3 marks)	Many public classes feature useful and informative comments at the top level (outside the code body) AND the submission meets the requirements for achievement level 4 as stated below
4: Pass (2.5 marks)	Complex or confusing sections of code feature inline comments describing the programmer's intention. Comments are not unnecessarily frequent. Identifiers are well chosen and the program flow is logical

2-3: Fail (1.5 marks)	Useful comments are sparse OR comments are far too unnecessarily frequent to the point that they interfere with the reading of the code OR identifiers are poorly chosen OR the program flow is confusing
1: Low Fail (0 marks)	No useful comments OR too little code submitted to evaluate the quality of the code

Exception handling: 4 marks

Your use of exception handling (try / catch blocks and throw statements) will be marked by looking at your code (examining areas). If unhandled exceptions terminate the program at all during our testing (either automated or otherwise) the maximum mark you can get for this section is 1.5/5. However, this **only** applies to scenarios that are described in this specification - for instance, while your program must not terminate with an unhandled exception if provided with valid input, or if provided invalid input in the form of entering something other than a coordinate pair when prompted for one. However, if the user inputs something that makes the scenario invalid (for example, if there is no way to get to the objective, or if the agent starts out being blocked by an obstacle) you may throw an exception that terminates the program. We will not be testing for these kinds of problems.

Note that we aren't just looking for try / catch - swallowing exceptions that affect the running of the program and not taking the appropriate steps is is **not** considered exception handling (so you cannot pass this criterion simply by wrapping your entire program in a try/catch.)

Also note that 'NotImplementedException' does not count as exception handling either.

Marking criterion

6-7: (High) Distinction (4 marks)	Exception handling is used to enforce the external interfaces of classes AND the submission meets the requirements for achievement levels 4-5 as stated below
5: Credit (3 marks)	Exception handling is used to prevent undesired/unexpected behaviour and/or results AND the submission meets the requirements for achievement level 4 as stated below
4: Pass (2 marks)	Exception handling is used to handle exceptional events at the appropriate level as per the requirements around invalid data in the assignment specification
2-3: Fail (1 mark)	Only marginal/incidental exception handling is present
1: Low Fail (0 marks)	No use of exception handling (no throw and/or no try/catch)

What to submit

For this assignment, you are required to submit the source files that make up your project to **Gradescope**. As with Assignment 1, there is no need to submit a solution (.sln) file or C# project (.csproj) file - we will provide our own, and you are more likely to run into problems if you attempt to submit one. See **Submitting to Gradescope**for general information about submitting Gradescope assignments - the output format is quite different here, but the process of uploading your source files is the same.

You get unlimited submissions and submissions will give you useful feedback - in addition, if you have a submission in Gradescope by the end of Week 9 the teaching team will have a look at it and give you some early feedback on your progress. You are encouraged to take advantage of this opportunity.

Feedback

You will receive some automated feedback on the functionality portion of your grade soon after submitting. You can then use this to improve your program and resubmit. After the due date, once marking has been completed you will receive complete feedback and the rest of your marks.

If you have submitted to Gradescope by the end of Week 9, you will receive some early feedback on your progress by the end of Week 10. We will not mark your entire assignment at this time, but we will have a look and tell you whether you're on the right track or not.

Tips / Suggested order of implementation

When approaching a large project like this, it can be difficult to tell where to begin. Here are our suggestions as to how to best utilise your time working on this assignment:

1. Start by implementing the main menu and, at a minimum, make your program exit when `x` is entered. The way the test cases are designed, every test will send an `x` to exit the program as the final line of input. This means that, even if you haven't implemented anything else, your submission will at least get through every test case without timing out. You can then slowly add additional functionality to your program as you go, and submit to Gradescope to keep track of your progress.
2. Implement the first obstacle (the guard) and the first non-obstacle menu item (the direction checker). If you've done this successfully you should now have a score of 1.25/20 test cases.
3. Continue implementing obstacles in order of difficulty: fence, then sensor, then camera. As you complete each one you will be passing more and more test cases, and when you finish, you should have a score of 8/20 (or close). If you get stuck on this, take a break and move to step 4 for now, which should help you make a bit more progress and pass a few more test cases. You can then come back to step 3 later.

4. Implement the map. If you have a good OOP solution to handling obstacles, drawing the map should be easy, and completing this will get you an additional 4 marks, for a total of 12/20 marks if all the previous test cases passed too.
5. Work towards implementing pathfinding. As you improve your pathfinding approach, your test cases passed should steadily increase. Once you have it working perfectly, you should be getting close to 20/20 for functionality.
6. Implement the custom obstacle.