CPSC 3200 Object-Oriented Development

Programming Assignment #1: Due Tuesday, April 5, 2022 before MIDNIGHT

For an acceptable P1 submission:

1. use **C#** and Visual Studio
2. upload all files (NOT a project) to Canvas
3. use Unit Testing to verify the functionality of class *gridFlea*
4. use Programming by Contract to specify contractual design, placing
    a. class and interface invariants at the top of *gridFlea.cs* file
    b. implementation invariant at the end of *gridFlea.cs* file
    c. pre & postconditions (if needed) before each method header
5. write readable code – see codingStd.docx in the **coding** folder under Files on Canvas
    a. use functional decomposition => NO monolithic drivers
    b. ***do NOT hard code: replace arbitrary literals, such as '42, with constants***
6. employ the OOP tenets of abstraction, encapsulation and information hiding
7. document your driver:
    a. ProgrammingByContract NOT used for drivers
    b. Assume that the reader does NOT have access to this assignment specification
    c. *provide an overview of your program*
    d. *explicitly state <u>ALL assumptions</u>*

P1:	**Type definition**	class *gridFlea*
	Driver	*P1.cs*

<u>=> TWO perspectives supported via P1 fulfillment:</u>

class designer	designs and implements *gridFlea* class
client	i.e. software that uses *gridFlea* objects; simulated by driver *P1.cs*

Part I: Class Design (gridFlea.cs)

Each *gridFlea* object encapsulates, at minimum, the following numbers: **x**, which represents its location along the x-axis; **y**, which represents its location along the y-axis; **size** and **reward**, which determine its value. Upon `g.move(p)`, an active *gridFlea* object `g` jumps p squares along one axis of the grid if in *energetic* mode; otherwise `g` moves only one square. 'reward' is reduced by the number of squares moved. 'size' does not change. Upon `g.value()`, an active *gridFlea* object `g` returns reward*size*change, where 'change' represents the how far `g` has moved, i.e. (initialX – x) + (initialY –y).

Every *gridFlea* object is initially *active* and *energetic* but becomes 'inactive' after some number of moves (a bound that should vary from object to object). Any `g.move(p)` that moves `g` outside grid boundaries permanently deactivates `g`. The client may reset as well as revive any *gridFlea* object that has not been permanently deactivated.

Many details are missing.

How is each *gridFlea* object initialized?
How is state defined and controlled?
How is movement defined and controlled?
What must be exposed to the client? What must NOT?
…

Hint: x and y are internal values, related to *gridFlea* object movement
⇨ client does NOT control *gridFlea* object placement directly
⇨ there should NOT be any setX(int) or setY(int) public methods

You MUST make and DOCUMENT your own design decisions

This assignment is an abstract realization of a data sink (store) that yields specific information upon query but can age and become invalid. With the interface described above, your design should encapsulate and control state as well as the release of information.

Do NOT tie your type definition to the Console.

Use Unit Testing to verify your class functionality.

Part II: Driver (P1.cs) -- External Perspective of Client – tests your class design

Design a **functionally decomposed** driver to demonstrate program requirements.

Use many distinct objects, *stored in an array* (do NOT use vectors, lists, …), and initialized so that there is a seemingly random distribution of gridFleas, etc.

Adequate testing requires sufficiently varied objects (… in different states).
Verification of state and state transitions streamlined with many objects.

Craft output, readable but not exceedingly lengthy, to demonstrate expected functionality

Given that P1 rests on a single type definition, there may not be much difference between the driver and the unit testing.

The rubric below is ONLY a general sample
NOTE: Regardless of rubric, points deducted for non-professional coding styles
Consult codingStd.docx in the coding folder under Files

Class Design (70 points)

Contractual Design	20 points
Interface and Implementation invariants	
Pre & Post conditions	
Proper Accessibility (public, private)	5 points
Appropriate state set in constructor (or default)	5 points
Error design	5 points
Definition and control of state	10 points
Appropriately defined and supported functionality	25 points

Unit Testing 10 points

Driver (20 points)

Appropriate functional decomposition & documentation	10 points
PROGRAMMER name, date, revision history, platform, etc.	
Description of process(es) performed by program	
Explanation of user interface (input, meaning of output)	
Comments on use and validity (error processing)	
Statement of assumptions	
Required functionality verified	5 points
Appropriate allocation and manipulation of objects	5 points