CSC461 Python, Iterator Pattern, Strategy Pattern

DUE: As stated on D2L

Checklist this time!

The purpose of this assignment is to give you practice with python and the iterator and strategy patterns. This is also to give you practice in completely handing off ownership of object class associations. This assignment hits D in SOLID,* really hard. *<u>If you do not make your classes reusable, the later tiers will be a nightmare!</u>

We will create a class diagram during lecture. You **must** use this diagram to code your project. This will be tested only on **Python 3.11.**

Overview

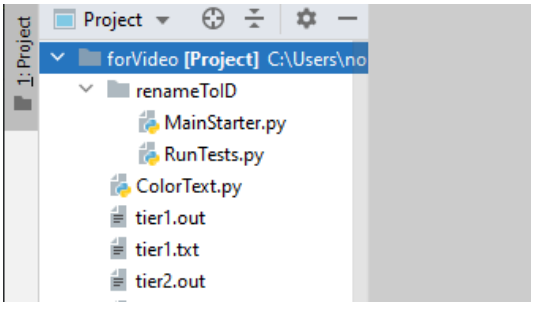
You will be coding a river lock simulator (https://en.wikipedia.org/wiki/Lock_(water_navigation)). To keep the project size reasonable, there will only be one river, with boats only going up river. You will also only add boats at the mouth of the river. A boat will travel the river based on its engine power and lock's behavior, which must be set using the strategy pattern.

Reminder: private/public variable rules still apply in python! You can search for public variables with regex (self\.[a-zA-Z]+[^)(] ?=)

I am providing you with **RunTests.py, ColorText.py** (used only to better mark discrepancies)**,** and **MainStarter.py** files that will add difference tests. **RunTests.py** and **MainStarter.py** must be inside your **lastName_firstName** package. Your grade will be based on these tests. **ColorText.py** is at the same level as your package folder.

MainStarter.py has a **main()** function that the tests use to start up your project. Therefore, you must begin your code in the provided **main()** function. **RunTests.py** works like it did in Java. Run this file and the tests will run. Run from **MainStarter.py**, and it will run as normal. In order to run redirected IO in python with extra newlines for readability, there is a custom input function that should be used rather than `input(…)`. It is `cleanInput(…)`. Use is the same as `input(…)`

The test files and **ColorText.py** are set up to be in the same folder (not inside) as your **lastName_firstName** package, such as in the following:



This is so I can use the same files for everyone when grading.

Menu Functionality

The menu must be in the following format as the initial state (there is a space after Choice:>):

```
~~~~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~.......~~~~~~~~~~

1) Add Default Boat
2) Update One Tick
3) Update X Ticks
4) Show Section Details
5) Add Boat
6) Make Tester
7) Make New Simulator
0) Quit

Choice:>
```

The tilde is a Unicode character, not ~. Copy the above character into your code.

The river is composed of a sequence of sections (shown as 〜〜〜 per unit of length), and locks are shown as _X(#)_. The # is the water level of the lock. There is a space before the # if needed. The default will be a section 6 units in length with 0 flow, followed by a lock with a depth of 0, followed by a section of 3 units in length with 1 flow, as shown above.

The menu must be able to handle out of range input by printing “Input an option in the range 0-7” and then reprint the menu. Warning, later on there will be a “hidden” debugging option for -1 and the out-of-range output, so the “real” range is -1 to 7. If the user inputs bad input (e.g., 2.5), output, “`Please, input a positive integer.`”

Hint: A try catch block around your menu, plus a message for specific types of exceptions, can make this input error and later input error requirements simple to handle. **Second hint**: `import traceback` can output the call stack in the event of an unhandled error. Since the try can block your error call stack from displaying, I suggest having the following at the end of the try-catch:

```
except:
    import traceback
    print( traceback.format_exc( ) )
```

Add Default Boat (Menu Option 1)

Add a default boat at the beginning (left end) of the river, and then reprint the system and the menu. A boat has an integer engine power and an optional engine behavior. Start the serial numbering at 1 and increment automatically for each boat added. **Note:** This serial number start is never reset during the entire run of the program. Therefore, later tiers will have numbers starting above 1.

A default boat will move at a speed of 1 unit per update regardless of the river flow.

To display a boat in a section, replace the first ~ with the boat character (🚤) in its associated spot. To display a boat in a lock, replace the X with the boat character. For example, if there was a boat in the section unit of the first section and one in the lock, using the default river, the program would show

~~~🚤~~~~~~~~~~~~~~~~_🚤(0)_~~~~~~~~~~

Following this line will be the boat's ID aligned to the boat position. Similar to above, replace the place holder characters with the ID. For example, if the above were the first two boats, the full output string would be

~~~🚤~~~~~~~~~~~~~~~~_🚤(0)_~~~~~~~~~~
~~~2~~~~~~~~~~~~~~~~1.....~~~~~~~~~~

Windows will damage the spacing since number and the Unicode character do not have the same width in any built-in monospace font I tried. But, the total number of characters will hold.

As another example, I added a boat and updated repeatedly until I saw ID 11

🚤~~🚤~~🚤~~🚤~~🚤~~🚤~~_🚤(0)_🚤~~🚤~~~~~~
11~10~9~~8~~7~~6~~5.....4~~3~~~~~~

If there is a boat already at the beginning of the river, make a new boat (to increment the serial number), then do not use it. This means the boat that was on the location stays. There will be no changes, other that the fact the next boat I make will have ID one higher.
~~~

Meeting D in SOLID

You are required to override the __str__ function for the river, and you will be required to meet D in SOLID for this. In other words, printing the system requires one line similar to: `print( mySim ).` In addition, to meet D in SOLID, printing a boat, section, or lock can be called independently, and still print properly. To simplify the problem, only the full river needs the second line. You may need to use python's string manipulation functions to complete this.

For grading purposes, there will be a hidden menu command of -1 to test the D in SOLID. This command will print a set of items. Make three new default boats. Add the first boat to a new section, and the second boat to a new lock. Add the last boat to the *current* system. Lastly, print all of these objects using something similar to:

```
print( boat1 )
print( section )
print( lock )
print( system )
```

If done properly, only `print(obj);` will be needed for each item as shown above. This task is tierless, although there is a test built for it (and should be working around tier 3). Since this run is last, let the grader know to run it manually if an earlier test fails *in the checklist area*. If run first, this is what would be output with comments for clarity:

```
🚢 #first boat
🚢~~ #first section
_🚢( 0)_ #lock
🚢~~~~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~ #river
3~~~~~~~~~~~~~~~~~~~~.......~~~~~~~~~~
```

Tag ONCE, the at the start of printing this sequence with GRADING: TO_STR.

Moving the Section (Menu Option 2)

Move the boats along the river based on their behavior, and then reprint the system and the menu. The default movement is one unit per update. If the boat reaches the end of a section\lock, move the boat onto the next section\lock if open. If the boat reaches the end of the river, simply remove the boat. Later, locks will need time to fill before releasing a boat, and time to empty before accepting another boat.

The starting lock doesn't need to fill or empty, so it will act like a regular unit area in a river section. As an example, this would be the first update after adding a boat:

```
Choice:> 2
~~~⛴~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~
~~~1~~~~~~~~~~~~~~~~.......~~~~~~~~~~
```

Updating from the contents of the simulator must be done with the iterator pattern. You must make these yourself **using python's built-in support**. This includes using the resulting supported for-each loop to confirm it is working. Under no circumstances should there be anything similar to

```
for i, v in enumerate(self.mySim.getList()):
    …
```

or

```
for v in iter(self.mySim.getList()):
    …
```

or

```
for v in self.__myLockList:
    …
```

The last one is less obvious, but you are using the array's built-in iterator rather than the one you made. If I put a break point inside your __iter__, it would not be hit in that last case. `for v in self` should be used instead. Basically, in some place outside your collection class, you should have something similar to the following:

```
for v in mySim:
  print(v)
```

You may make this iterator work backwards or forwards.
Tag ONCE, the __iter__(self) for this task with GRADING: ITER_ALL.

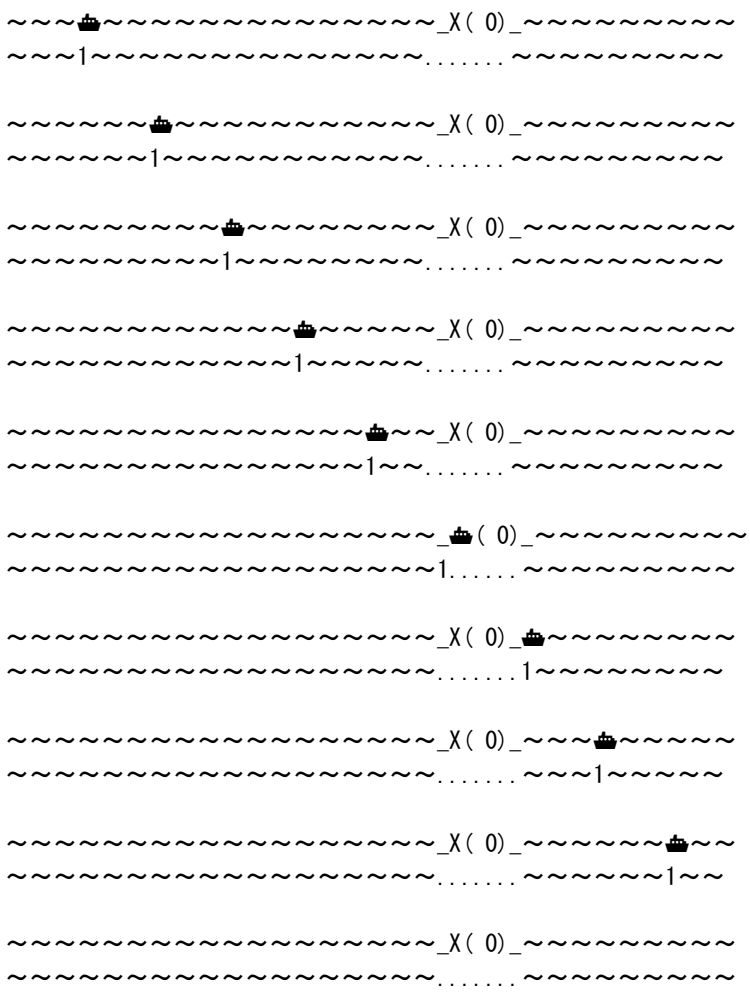
Tag ONCE, the for-each loop for this task with GRADING: LOOP_ALL.

The grader will be putting break points at these lines (including inside __iter__ and __next__) to ensure they are hit.

Moving the Section Multiple Times (Menu Option 3)

This is the same as the above, but the user is asked how many times the update is run. Between each update, output the system, and when done, reprint the menu. For example, if starting with the default system with one boat added, the following is output (you must follow this format):

```
Choice:> 3
How many updates:> 10
```




```
1) Add Default Boat
2) Update One Tick
3) Update X Ticks
4) Show Section Details
5) Add Boat
6) Make Tester
7) Make New Simulator
0) Quit

Choice:>
```

If a user inputs something other than an integer, output "`Please, input a positive integer`" and reprint the menu.

Show River Section Details (Menu Option 4)

Sections can print more details. Output each section as found, left to right, in the following format:

```
Section <num>
Boats: <num> Flow: <num>
//new line here
```

<num> refers to the number of the lock starting at 1, the number of boats in this section, and the rate of flow of the river in this section. For example, consider the below state using the default river:

```
🚢~~~~~~~~~~~~~🚢~~🚢~~_🚢( 0)_🚢~~~~~~~~~
5~~~~~~~~~~~~~4~~3~~2......1~~~~~~~~~
```

This option would output:

```
Choice:> 4
Section 1
Boats: 3 Flow: 0

Section 2
Boats: 1 Flow: 1
```

You must use an iterator to return only the sections.

Tag ONCE, the __iter__(self) for this task with GRADING: ITER_RESTRICT.

Tag ONCE, the for-each loop for this task with GRADING: LOOP_RESTRICT.

Add Boat (Menu Option 5)

Add a boat of a user set engine and travel behavior to the beginning of the river, then reprint the system and the menu. Asking for the boat parameters must follow the following format:

```
Choice:> 5
What engine power:> 2
What travel method. (1) Steady or (2) Max :> 1
🚢~~~~~~~~~~~~~🚢~~🚢~~_🚢( 0)_🚢~~~~~~~~~
5~~~~~~~~~~~~~4~~3~~2......1~~~~~~~~~
```

If a user inputs something other than an integer, output "`Please, input a positive integer`" and reprint the main menu. If a user inputs an integer out of range, output "`Input an option in the range 1-2`" and reprint the main menu.

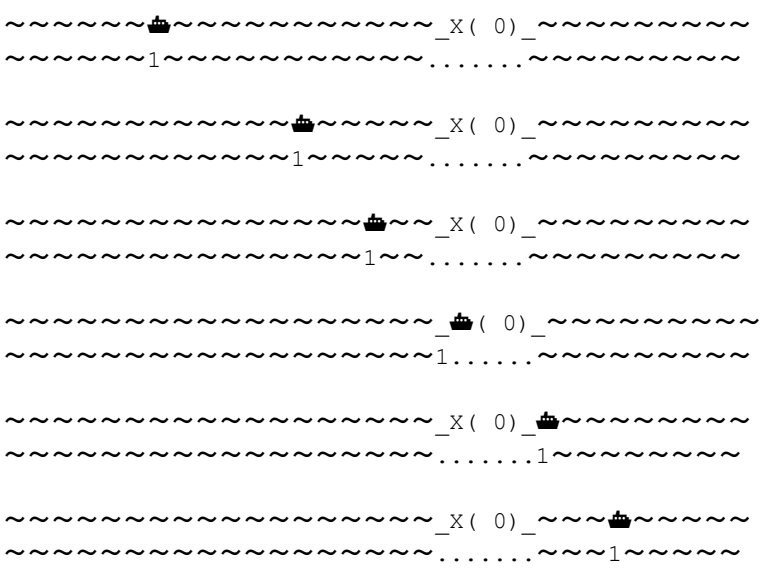
Boat behavior

You must implement the strategy pattern for the boat's travel behaviors. This MUST be a set of function(s), although these functions may be in a class. You MUST call a function to perform the behavior. The behavior types are

1. Steady:
    a. Updates one unit length every update regardless of engine power
    b. A boat cannot pass a boat in front of it
2. Max:
    a. Update distance is based engine power and river flow. Specifically, it is at minimum one unit length, and up to "engine – flow" lengths.
    b. A boat cannot pass a boat in front of it
    c. A boat always needs to wait at least one update for the lock gate to open or to exit the river. Therefore, if the distance traveled puts the boat past the end of a river section,

place the boat at the end of the section, and then in the lock for the next update if open.

As an example, if a boat with 2 engine power and max behavior is added to the initial river, a multi update of 6 would show:



Notice the boat stopped for the lock, and slowed down in the second part of the river.

- Tag ONCE, the basic function\class with GRADING: STEADY_TRAVEL.
- Tag ONCE, the Restricted function\class with GRADING: MAX_TRAVEL.

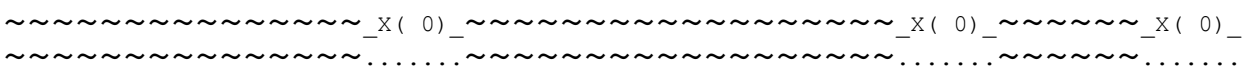
You may code this either in class-based format or first-class function format.

Making a New Tester River (Menu Option 6, Part A)

Make a new river with the following properties to test the next strategy pattern. Details on the options for the behaviors for the strategy pattern are in the following sections.

- A section of length 5, and flow of 0
- A lock with none\pass-through behavior, and depth 0
- A section of length 6, and a flow of 2
- A lock with a basic filling behavior, and a depth of 2
- A section of length 3, and a flow of 3
- A lock with a fast-emptying behavior, and a depth of 5

It should look like this:



Fancier Locks (Menu Option 6, Part B)

You must implement the strategy pattern for the behaviors other than the default pass through. This

MUST be a set of functions, although these functions may be in a class. You MUST call a function to perform the behavior. The behavior types are

1. None:
    a. Pass through (no filling or emptying)
2. Basic:
    a. Each update fills and empties with 1 unit
3. Fast Empty
    a. Each update fills at 1 unit
    b. Each update empties at 2 units

- Tag ONCE, the basic function\class with GRADING: BASIC_FILL.
- Tag ONCE, the percentage function\class with GRADING: FAST_EMPTY.

A lock can only accept a boat when its level is 0, and a lock can only release a boat if its level is equal to its depth. Fill before trying to release on an update. A lock empties if it has no boat. The number in the lock string is its current level. This number may not go outside the range [0, depth]. Here is an example with a boat just before the second lock run with a multi update:

```
~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~2~~.......~~~~~~~~~~.......

~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~_( 0)_~~~~~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~~~~~2.....~~~~~~~~~~.......

~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~_( 1)_~~~~~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~~~~~2.....~~~~~~~~~~.......

~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~_X( 2)_~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~~~~~.......2~~~~~~~~.......

~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~_X( 1)_~~~~~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~~~~~.......~~~2~~~~~~.......

~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~_~~~( 0)_
~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~~~~~~.......~~~~~~~2~~~.....
```

I'll be putting break points on these to ensure they are hit.

You may code this either in class-based format or first-class function format.

Making a New River (Menu Option 7)

This task clears the current simulator set-up and then makes a custom version. The main purpose of this task is to ensure you meet the D in SOLID in your earlier work. A section after a section is legal, as is a lock after a lock, just like in real life.

Ask the user whether to add a section or a lock to the end of the current lock, using the following format (there is a space at the end):

```
Section (1) or Lock (2):>
```

If the user asks for a section, ask for the length of section and its flow with the following format (there is a space at the end of :>):

```
Length:> <num>
Flow:> <num>
```

If the user asks for a lock, ask for the loading behavior and then the packaging behavior with the following format (there is a space at the end of :>):

```
Fill behavior: None (1), Basic (2), or Fast Empty (3):> <num>
Depth:> <num>
```

After asking for a section or lock, ask if the user wants another section. Continue until the user types in 'n.' You must use the following format (there is a space at the end):

```
//new line here
Add another component (n to stop):>
```

When done print the new system, and reprint the menu.

All out of range integers or improper input (e.g. 2.5) must output "`Cannot accept value`," and then jump the remaining options to restart the at "add another component" line. **TIP:** Consider a second try-catch for this task. **TIP 2**: Remember how to raise an exception.

If someone asks, I am not doing a "sanity" check on boat sizes, section sizes, broken locks, etc. in the tests as I thought the range and non-integer cases were sufficient to show understanding, although this extra check would be easy to add.

Example Creation of a Custom Simulator That Would Rebuild the Texting Version

```
Choice:> 7
Section (1) or Lock (2):> 1
Length:> 5
Flow:> 0

Add another component (n to stop):> y
Section (1) or Lock (2):> 2
Fill behavior: None (1), Basic (2), or Fast Empty (3):> 1
Depth:> 0

Add another component (n to stop):> Y
Section (1) or Lock (2):> 1
Length:> 6
Flow:> 1

Add another component (n to stop):> m
Section (1) or Lock (2):> 2
Fill behavior: None (1), Basic (2), or Fast Empty (3):> 1
Depth:> 2

Add another component (n to stop):> y
Section (1) or Lock (2):> 1
Length:> 3
Flow:> 3

Add another component (n to stop):> 1
Section (1) or Lock (2):> 2
Fill behavior: None (1), Basic (2), or Fast Empty (3):> 3
Depth:> 5

Add another component (n to stop):> n
~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~~~~~~~~~~~_X( 0)_~~~~~~~~~_X( 0)_
~~~~~~~~~~~~~~~~~.......~~~~~~~~~~~~~~~~~~~.......~~~~~~~~~.......
```

If I add a boat and then show the lock details at this point, I would see the following:

```
Section 1
Boats: 1 Flow: 0

Section 2
Boats: 0 Flow: 1

Section 3
Boats: 0 Flow: 3
```

Additional restrictions

- You must copy in the checklist and complete it.
- You must display a system by overriding the **__str__()** function. This must be a single call to the system.
- You MUST put your code into a package named your lastName_firstName (lowercase with no prefixes or suffixes.
- There should not be a getter function for ANY list. You must use an iterator. This will cause you to lose all points for the iterator.
- You must use Python's commenting structure for functions. The autoformatting (epytext or restructured) version is up to you.
- The iterators must be done using Python's iterator structure. You must override __iter__() and __next__(). While this will depend on the class diagram, it is *highly* likely you will have two iterators for one class.
- You must use the OOP diagram developed in class.
- Major classes must be in their own file.
- Copy and paste the grading tags since we look for an exact match on those.

Grading Tiers

These tiers start with the simplest tasks, and go to the most involved. You must "reasonably" complete the lower tiers before gaining points for the later tiers. By "reasonably," I can reasonably assume you honestly thought it was complete. OOP line items fall into this category, since you cannot check those on your end. Not passing an automatic test is not reasonable. Not having __iter__ or __next__ for the iterator is also not reasonable. For partial credit on a tier, you **must** put a comment in the main file header comment on how to test for it.

For tiers with sub parts (e.g., 2a, 2b, etc.), the sub tests may be completed in any order. ***However***, the tests are run in order and stop on the first failed. There is a line that starts with "`tests =`" that you may rearrange to change the order. Since the tests are overwritten on our side, if you complete a higher number subtier and do not complete a lower one, **you must let the grader know** in the checklist in order to have those graded.

Submission instructions

1. If given a file that is not supposed to be changed, I will be overwriting the file with the original when I grade.
2. Check that you are not in violation of the additional deductions in the main tab.
3. Complete the checklist and paste into the top of MainStarter.
4. All of your Python files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes).
5. Delete the out folders, then zip your package folder into ***ONE*** zip folder named your lastName_firstName (lowercase with no prefixes or suffixes). Make sure this matches your package name!

If you are on Linux, make sure you see "hidden folders" and you grab the ".idea" folder. That folder is what actually lists the files included in the project!

6. Submit to D2L. The dropbox will be under the associated topic's content page.
7. *Check* that your submission uploaded properly. No receipt, no submission.

You may upload as many times as you please, but only the last submission will be graded and stored

If you have any trouble with D2L, please email me (with your submission if necessary).

Rubric

All of the following will be graded all or nothing, unless indicated by a multilevel score.

Item	Points
Other deductions	
Tierless str meets D in SOLID*	10
1. Initial Show System\Compiling	16
Got it compiling	10
Bad input handled (-33% per error)	6
2. Add Default Item	8
Added and shown properly	4
Second+ item ignored	4
3. Basic Update (one item)	26
Moves along sections (-50% per error)	12
String format correct	4
Iterator used*	10
4. Basic Update (multiple items)	8
Same as prior tier at 50%, without iterator line	←See note to left
5. Multi Update	8
Updates correct amount	3
Bad input handled\error handled	5
6. Show Details	14
Shows details properly (-50% per minor error)	6
Iterator used*	8
7. Add User Specified Item with or without Errors(50% per error)	6
8. User Specified Boat Updates Properly	10
Basic movement still works	3
Powered movement works (-50% per error)	4
Doesn't pass other boats	3
9. Tester part A	10
Boats works up to second lock (-25% per error)	8
Formatting correct	2
10. Tester part B	20
Boats works to end (-50% per error)	6
Strategy pattern for basic fill*	7
Strategy pattern for fast empty*	7
11. Custom System\Stress Test **	14
String formatting correct	3
Everything still works (-25% per error)	8
Bad input handled	3
Total	150

*Has a tag associated with it.

** This tier has 2 tests associated with it. A tests all section/lock orderings. B tests error checking.