Final Project
Christopher Columbus Adventure!

Objective: The purpose of this project is for you to put into practice the principles and practices of Object-Oriented design that you have learned during SE350. The only way to accomplish this is to design and implement a system that has a non-trivial number of classes. The final project will therefore take some significant effort on your part, but this is the only realistic way to learn to program!! Each group/person is encouraged to design and construct a final product that challenges their own skill level. The requirements described here specify the minimum requirements of the class, and this is a project for which you are encouraged to go above and beyond! This project option builds upon the work you have done in Homework 2. For the final project form groups of 4 (in –class students), not more not less, online students can either work in groups or individually. Below is the guided approach to complete the project:

Starting Point: Fully working Christopher Columbus solution from Homework 2.

New functionality to be added will include:

- A larger grid.
- A search for **treasure** with a **"Win" notification** if CC finds the treasure before being caught by the pirates and/or sea monsters.
- **Sea monsters** swimming around one (or more) areas of the ocean.
- **Improved algorithm for moving the pirate ship**. Create multiple strategies for pirate ships to move. Try to make them smarter than the current "chase" strategies.
- Write **JUnit tests** for two classes.
- Use **four design patterns** (see below).

Optional-- Ideas for Extensions (in lieu of final project)

- Movable grid (so that you don't see all of the terrain at the same time).
- Design mode – in which you design the game (in much the same way as Gardenlayout in Lab 3) by dragging islands etc. Also use other widgets to specify number of islands etc.
- Whirlpools which randomly appear. If CC goes down a whirlpool he appears at the exit point. (A bit like snakes and ladders).
- Better polish around the overall game.
- More challenging adventure.
- Use more design patterns (see below).

Design Patterns

The requirements specify that you must use four design patterns in addition to Observer which has already been implemented in the second assignment. Here are some suggestions:

- **Observer:** This is already in use for the Pirate ships as they observe Christopher Columbus.
- **Singleton**: You can create the map grid as a singleton. Advantage – you can get access to the map grid from any class without having to pass the object around! (Will be covered in class on Week 8).
- **Factory Method**: Use the factory method to create pirate ships. You could have two types of Pirate ship. They could have different images, different origins etc. Remember the factory method pattern (as described in class on week 7) must have an abstract creation method which defers object creation to concrete subclasses. However, the superclass defines other shared methods which act on the created object.
- **Strategy:** You could create more than one strategy for how the pirate ships move around the map. Perhaps some of them chase Christopher Columbus (using the observer pattern), while others just patrol the waters in different ways.
- **Decorator:** You could use the decorator to add special protective powers to Christopher Columbus ship. You could use it to add functionality to the whirlpool.
- **Composite:** You could quite easily use the composite to create groupings of monsters/sharks. The composite node would define the boundaries in which they could swim. Individual animals are added to a 'container'. All containers are added to the game (or ocean). You can then stop or start the animals. Whirlpools could be also added as leaf nodes to the ocean. When CC's ship moves you can find out which container (if any) it is currently in.

Deliverables:

Points computed out of 100 but then scaled to 40 percent of the final grade.

Element:	Points:
1. Video demos of the game (May 22nd)	10
2. UML Diagram submitted in advance (May 16th)	10
3. All prescribed functionality.	25
4. Five design patterns implemented correctly	25
5. Two classes have good unit test coverage	10
6. Java docs or regular comments	10
7. Overall elegant design.	10