CIS/IM 452/552 – Assignment #4 – **Due Friday** **2/14 before 11:55pm**

Objective: Create a mini-game implementing the Decorator Design Pattern.

**Step 1 - Prepare:**

**Required:**

* Read chapter 3 in the *Head First Design Patterns* textbook
* Review the textbook’s example code for the Decorator Pattern in Java at <https://resources.oreilly.com/examples/9780596007126/tree/master/examples>
* Review the example code for implementing the Decorator Pattern with C# in Unity that we went through in class posted on Sakai under Resources -> Week 4 – Decorator Pattern
* When you have completed each step above, you can continue to Step 2.

**Recommended** (if you do not fully understand the strategy pattern yet)**:**

* For an additional review of the Decorator Pattern, including walking through another example, watch this video: <https://www.youtube.com/watch?v=j40kRwSm4VE&list=PLF206E906175C7E07&index=11>
* The Java code for the video above can be reviewed at <http://www.newthinktank.com/2012/09/decorator-design-pattern-tutorial/>

**Step 2 – Make UML Diagram and Mini-Game**:

Create a mini-game with Unity and C# that uses an implementation of the Decorator Design Pattern. **Your mini-game must create a new instance of an object and add decorators to it in response to something the player does while the game is running.**

Model the object-oriented design you use to implement the design pattern in your mini-game with a UML diagram using a tool such as [draw.io](https://www.draw.io/). You can have other scripts and classes that are not part of the design pattern in your mini-game, and you do **not** need to include those scripts and classes in the UML diagram.

Be sure to include the relevant variables as well as the methods in each class. The design pattern must be implemented correctly according to its definition. **Look at the example UML diagram for the design pattern in the textbook for a template or starting place. Do not use the example code I show in class as a starting place – you need to decide for yourself how you are going apply the design pattern to your mini-game.**

**Deliverable 1:** A UML diagram of the object-oriented design in your mini-game that implements the Decorator Design Pattern (and only diagram the parts that implement the design pattern), pasted in from whatever program you used to create it ([draw.io](https://www.draw.io/) is recommended) to the single .docx or .pdf document you will submit.

The object-oriented design shown in your UML diagram must be implemented in your mini-game using C# and Unity. **If you change your OOP design for the parts implementing the design pattern while creating your mini-game, you must update your diagram to reflect the changes.**

What makes your unity project a mini-game and not a tech demo, a non-game toy, or a passive experience? **There are 6 requirements for a mini-game listed in the rubric at the end of this document – be sure your mini-game meets all of these requirements.**

**Deliverable 2:** a URL web address linking to a Unity project repository on GitHub containing your mini-game.

* You must put the .gitignore file under Resources on Sakai in your git repository and commit and push the .gitignore file to your repo **before adding your project files or creating a unity project in your git repository**. You will lose points if you do not properly use the .gitignore file.
* You must set your github repo to public – be sure to **set your repo to public** so I can access your files.
* Include the **name of the scene** in Assets/Scenes/ you want me to open, and the path to them if the scene is not there.
* If your script files for the assignment are not in Assets/Scripts/, then note the path to your script files.
* To receive credit for this assignment, all script files that you write or change **must** include the header comment below with your name on it:

/\*

\* (Student Name)

\* (File Name)

\* (Assignment)

\* (Describe, in general, the code contained.)

\*/

**Step 3 – Answer Reflection Questions:**

**Deliverable 3:** Include each of the following reflection questions and your answers to them in your single document you will submit on Sakai.

1. What does the player do in the game that creates an object of a type that can be decorated?
2. What does the player do that adds a decorator to that object (with the decorator design pattern) while the game is running?
3. What does the decorator do to change the object it decorates? For example, the Starbuzz Coffee condiments added to the beverage price, and adding a profession added to the character’s damage.
4. What were the benefits of using the Decorator Pattern to make your mini-game?
5. Did you find any drawbacks to using the Decorator Pattern? If so, what were they?
6. What is the player’s goal in your mini-game and what makes it challenging?
7. How does the game communicate its goal(s) to the player?
8. How can the player fail at the game and how does the game detect it?
9. How does the game give players feedback about how well they are doing?

Submit the deliverables listed above as a **single** .docx or .pdf document on Sakai under Assignments -> Assignment 3 before it is due.

**Grading Rubric/Checklist**

* All parts included:
  + Deliverable 1: UML Diagram (-40 points if missing)
  + Deliverable 2: Link to GitHub repo with your Mini-game (-40 points if missing or not working or the .gitignore file was not used properly). **All code in your mini-game script files must be written by you and not copied from code examples shown in class.**
  + Deliverable 3: Reflection questions and your answers to them (-20 points if missing)
* All requirements met:
  + Mini-game **creates a new instance of an object and add decorators to it in response to something the player does while the game is running** using the Decorator Pattern (15 points)
  + UML Diagram and mini-game correctly implements the Decorator Pattern (15 points) and UML Diagram is complete (10 points)
    - Include all variables and methods relevant to the design pattern
    - Arrows point from concrete sub-classes to super-types like interfaces, abstract classes, and super-classes. To remember the arrow direction, remember the Java keywords A extends B means A -> B and A implements B means A -> B.
    - Add arrows to **show any HAS-MANY relationships**. Remember that X HAS-MANY Y means X -> Y. **You can click the arrow in draw.io and write HAS-MANY on the middle of the arrow’s line to show this relationship.**
  + Mini-game meets all requirements for a mini-game listed below (10 points for each of the 6 requirements)
    - Tutorial elements that teach the player what controls to use (do **not** assume players know WASD=move or use the mouse to look, but use those standard controls if they make sense for your game)
    - A challenging goal is clearly communicated to the player
    - Achieving the goal of the game has a medium level of difficulty – neither too hard nor too easy (or the game has difficulty settings the player can choose or uses dynamic difficulty adjustment)
    - The game has success and failure conditions, meaning the player can win or lose each time they try.
    - Feedback about whether the player succeeded or failed each try is clearly communicated
    - There is a game loop, meaning the player has the ability to retry without resetting the game