HW3 – Composite, Factories, and Flyweight

Estimated time: 16-20 hours

Objectives

- Become familiar with using the Composite pattern.
- Become familiar with using at least one of the following: Simply Factor Idiom, Factor Method, or Abstract Factory.
- Become familiar with using the Flyweight Pattern.
- Become more familiar with using UML.
- Become more familiar with basic unit testing techniques.

Overview

In this assignment, you will extend the Shapes library with

- two new kinds of Shapes: Composite Shapes and Embedded Pictures
- three new features: the ability to 1) load a shape from a script, 2) save a shape to script, and 3) render a shape to an image.

A Composite Shape is a special kind of shape may contain zero or more other shapes. User of the library should be able to anything to Composite Shape that they can do to shapes in general. They also should able be able to

- add a shape to Composite Shape
- remove a shape from Composite Shape
- remove all shapes from a Composite Shape

A Composite Shape cannot include itself, either directly or indirectly. Also, a Shape can only belong to at most one Composite Shape. In other words, a Shape cannot be shared among multiple Composite Shapes.

An Embedded Picture is a special kind of shape that is a bitmap-type image. It can be loaded from file or resource and can be any size, independent of the size of the image in the file or resource. If the original image dimension doesn't evenly scale to the Embedded Picture's dimensions, you can decide which to skew or crop the image. (That choice is not relevant to the assignment. So, do whichever option is easiest.) Many Embedded Picture objects may be created from the same file or resource.

Since Composite Shape and Embedded Pictures are shapes, and are in fact 2-dimensional shapes, they need to support move and compute-area behaviors. The area of the Composite Shape is the sum of the all its sub-shapes. The area of an Embedded Picture is the area of its width times height.

There are no constraints on the spatial overlapping of shapes.

You have the freedom of deciding how to integrate the three new features into the library and what abstractions you will provide to the user. It is important, however, that your abstractions provide for good reuse and extensibility. Specifically, your abstractions, need to enable the following for the user of the library:

- The ability to create specializations of new Shape.
- The ability to save a shape as a script to a file or any kind of output stream.
- The ability to create a shape from a file, resource, or other kind of input stream.
- The ability to render a Shape to GUI object, a file, or any other kind of graphics device.

You are free to design your own format for the scripts. A single script is for one shape. Of course, that one shape may be a composite shape, which may include other shapes and those some of those may be composite shapes that include yet other shapes.

Even though, you may use XML, JSON, or some other standard markup language for your scripts, you may not use an existing serializer or de-serializer framework or component. Such are factories, at least in part, and would eliminate the need for you to write your own factory.

You will be graded on 1) the quality of your design, implementation, and testing, especially with respect to the principles of abstraction, modularity, encapsulation, and 2) your appropriate use of the design patterns. In this assignment, you should find opportunities to use the Composite, at least one type of Factor, and the Flyweight pattern. You may find opportunities to use other patterns, as well.

The starting point for this assignment is your completed HW1. However, you may want to refactor and improve that implementation now that you know more patterns and more about the principles of good OO design.

Instructions

To complete this assignment, you should do the following:

1. Create your own private Git repository for this homework assignment on BitBucket, GitHub, or with some other Git service provide.
2. Copy your HW1 solution into the Git repository as the starting point for this assignment. Be sure to copy the hidden files and folders, except the .git folder.
3. Design your new class library with the necessary extensions and improvements. Capture the structure of that design in one or more UML Class and Interaction Diagrams.

4. Implement your design. Be sure to update your design diagrams if your design changes and to commit/push your Git repository frequently.
5. Create unit test cases for each component such that your test cases collectively provide good coverage from a path-testing perspective. Again, commit and push your Git repository frequently.
6. Share your repository with cs5700@aggies.usu.edu
7. Zip your Git repository and submit it to Canvas by the due date/time. Include your git repository URL as a note in the submission.

Hints

- With respect to loading a shape from a file or resource, instead of passing the filename or resource name into some method, pass a general input stream. Let the user of the library decide what specific kind of input stream to create and how to create it. In doing so, you will be applying the Strategy Pattern, even your you are not building the strategies. The general input stream abstraction is the abstract strategy and the whatever concrete input stream the user constructs is the concrete strategy.
- Similarly, for saving a shape to a file or resource, instead of passing the filename or resource into some method, pass a general output stream.
- Likewise, for rendering a shape to an image, have the library accept a graphics object of some kind. Both Java's JDK and C#'s .net frameworks provide Graphics classes. The user of the library can create a concrete Graphics object many different ways depending on the needs of the application.
- Keep your library and its abstractions as simple as possible.
- Keep you scripting language simple and easy to parse.

Grading Criteria

Criteria	Max Points
A clear and concise design documented with UML class and interaction diagrams	20
A working implementation, with good abstraction modularity, encapsulation	25
Effective use of the Composite pattern	15
Effective use of at least one Factory	15
Effective use of the Flyweight pattern	15
Reasonable unit-test cases	30