HW1 – Abstraction and Modularity

Estimated time: 8-12 hours

Objectives

- Become familiar with the principles of abstraction and modularity, while extending the functionality of an existing a software system.
- Become familiar with using UML class diagrams to communicate the structural aspects of a system's design.
- Become familiar with basic unit testing techniques.

Overview

In this assignment, you will extend the initial Minimal version of the Shapes class library to include triangles, squares, rectangles, and ellipses by following the principles of **abstraction** and **modularity**. This will involve considering their respective adherence criteria and any supporting principles and practices discussed in class prior to the assignment's due date. You should also try to follow the principle of encapsulation using your intuition, the initial introduction of encapsulation provided in class, and the good-poor comparison examples available in the course's Git repositories.

A starting point for this code is available in the "okay design" variation directory of the initial Minimal Shapes class library. You can access all the variations of this class library by cloning the following git repository:

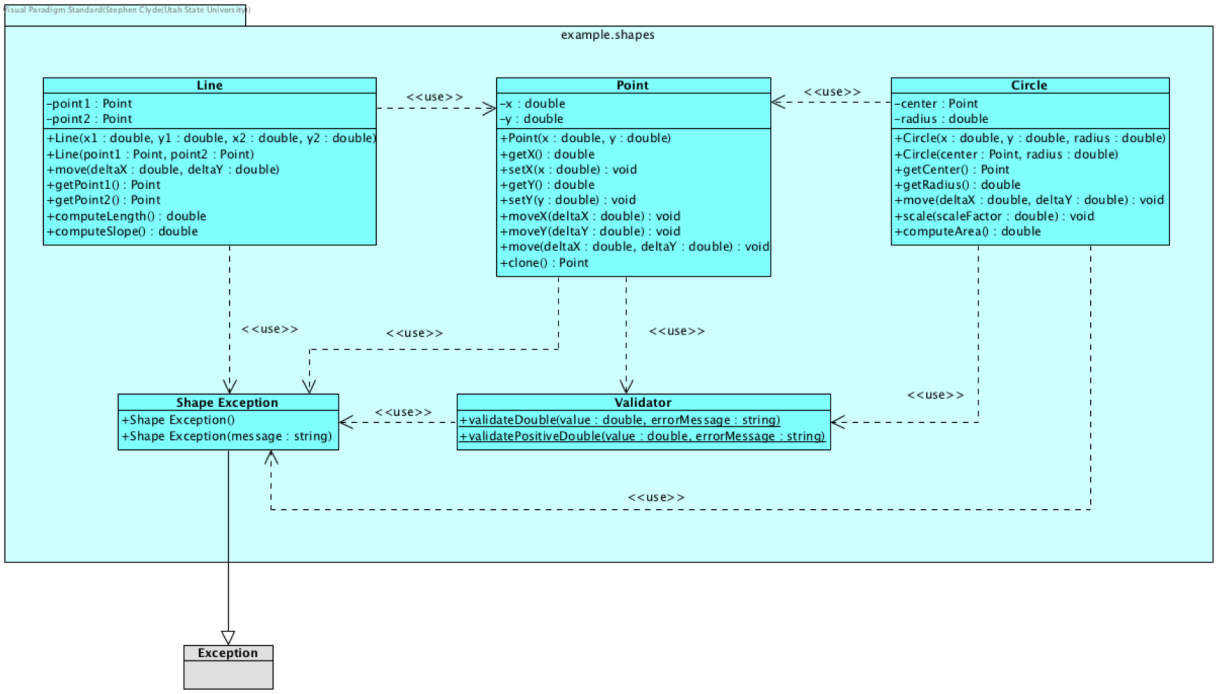
https:// bitbucket.org/usucssedevelopment/shapes.git.

For this assignment, you may use any class-based OO language that supports inheritance and at least some basic encapsulation, e.g., private properties. C#, Java, Typescript, Python, and C++ all meet these criteria. JavaScript does not, although some future version promises to do so. Also, note that the starting point is only available in Java and C#. If you choose to work in one of the other acceptable OO languages, you must first port the starting point to your language of choose.

Figure 1 shows the classes and their relationships in the starting point code. Note that is only an "okay design" consisting of the five initial classes. As you extend the design for triangles, squares, rectangles, and ellipses, you must do so in a way that improves follows the principles of abstraction and modularity. Simply adding the four new classes to the existing design won't do. Think about how you can localize design decisions and how you can create meaningful abstractions for shapes. Think about commonalities and differences between the different kinds of shapes.

Keep in mind that what you are building is a class library, not a program with a "main" function. In other words, there is nothing to "run" in the traditional sense. The class library is intended to be used in other programs and not by itself. Nevertheless, you will exercise all the components in the library through

Figure 1 – Initial Design for Shapes Library



unit test cases. In-fact, collectively, your unit test cases should ideally execute every line of code, exercise every branching statement in each possible direction, and test the bounds of every loop. Doing so should give you confidence that all your components are all working as expected.

You will be graded on the quality of your design, implementation, and testing, especially with respect to the principles of abstraction and modularity. In this class (and in your profession as a computer scientist or software engineer), it is not sufficient to have code that simple "runs". Your software needs to possess desirable characteristics, such as reliability, maintainability, reusability, extensibility, testability, etc. Those characteristics come by following principles and by applying proven practices and patterns.

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. Clone the initial Shapes Library Git repository to a local directory and examine its contents.
3. Copy or port the "okay design" variation to your own repository. Be sure to copy the hidden folders, if you want them. For example, for the Java version, copy the .idea and .gradle folders if you are going to use Java, IntelliJ, and Gradle.
4. Study the functional and non-functional requirements listed below.

5. Design your extension to the class library to satisfy the requirements and by following the principles of abstraction and modularity. Capture the structure of that design in one or more UML Class Diagrams. Add these work artifacts to your Git repository, commit, and push.
6. Implement your design, committing and pushing your Git repository frequently. Also, be sure to update your design diagrams if your design changes. It most like will.
7. 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.
8. Write up a short report that includes your UML Class Diagrams and insights uncovered during the design, implementation, or testing. Not counting the diagrams, the report should be **no more than 1 page, with 1.5 spacing**. Save your report as PDF file called HW1-Report.pdf in the root directory your Git repository. Commit and push.
9. Share your repository with cs5700-hw1@usu.edu. (Note this may change. Watch for announcements).
10. 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.

Note: committing and pushing your git repository not only provides you with a great backup of your work but provides at least some document of your effort over time. It documents when you start working on the assignment, how frequently worked on the assignment, the progress made during, and much more.

Requirements

For the purposes of expressing the requirements, the name *Shapes* refers to the Shapes Library that you are constructing. A *user* in this context is any piece of software that uses *Shapes*.

Functional Requirements

1. General Requirements
   - 1.1. All components in Shapes should be usable in a 2-dimensional space with X and Y axes. There is no constraint on direction or meaning of axes.
   - 1.2. The library needs to support Points, Lines, Ellipses, Circles, Rectangles, and Squares.
   - 1.3. All classes need to encapsulate the states of their objects, such that an object's state cannot be changed except by executing method on that object.
2. Points
   - 2.1. Shapes should not allow the construction of a bad point, e.g., one with an infinite x or y coordinate.
   - 2.2. A user should be able to access the x and y coordinates that define the point's position.
   - 2.3. A user should be able to move a point relative to either axis or both axes at the same time.
   - 2.4. A user should be to clone a point.
3. Line

3.1. Shapes should not allow the construction of a bad line, i.e., one with a bad point for either terminus. A line cannot have a zero length.
3.2. A user should be able to access the points that define the termini of a line.
3.3. A user should be able to move a line.
3.4. A user should be able to compute the length of a line.
3.5. A user should be able to compute the slope of a line.

4. Triangle
   4.1. Shapes should not allow the construction of a bad triangle: no edge of the triangle should have a zero length, the vertices cannot all be in the same line, and the length of an edge cannot be greater than the sum of the other two.
   4.2. A user should be able to access the attributes that comprise the definition of a triangle, e.g., the vertices.
   4.3. A user should be able to move a triangle.
   4.4. A user should be able to compute the area of a triangle.
   4.5. Optional: a user should be able to rotate a triangle.
5. Ellipse
   5.1. Shapes should not allow the construction of a bad ellipse. For example, its area cannot be zero.
   5.2. A user should be able to access the attributes that comprise the definition of an ellipse, e.g., the foci.
   5.3. A user should be able to move an ellipse.
   5.4. A user should be able to compute its area.
   5.5. Optional: A user should be able to rotate an ellipse.
6. Circle
   6.1. A circle in Shapes should behave similar to an Ellipse, except that two foci must be the same and rotation (if implemented) has not affect.
7. Rectangle
   7.1. Shapes should not allow the construction of a bad rectangle: no edge can have a zero length and adjacent edges must form right angle.
   7.2. A user should be able to access the attributes of the define the rectangle, e.g., vertices, height, width, etc.
   7.3. A user should be able to move a rectangle
   7.4. A user should be able to compute its area
   7.5. Optional: A user should be able to rotate a rectangle
8. Square
   8.1. A square in Shapes should behave similar to a rectangle, except all its sides must be the same length.

Non-Functional Requirements

1. All work artifacts must be kept under version control in a private Git repository.
   1.1. The work artifacts include your design, implementation, test cases, and report.
   1.2. The developer must make frequent pushes to the Git repository.

2. Shapes must be tested using executable unit test cases that are include in the software solution and that can be run using standard testing harness available in the selected development environment.

Submission Instructions

Zip up your entire solution, including test cases and sample input files, in a zip archive file called CS5700_hw1_<*fullname*>.zip, where fullname is your first and last names. Then, submit the zip file to the Canvas system.

Grading Criteria

Criteria	Max Points
A clear and concise design, described in UML class diagrams, that reflects good abstraction and modularity, as well as reasonable encapsulation.	40
A working implement with good abstraction and modularity, as well as reasonable encapsulation.	40
Meaningful, executable unit test cases that provide good coverage from a path-testing perspective.	30
A short report that discusses your design and insights that you uncovered during the design, implementation, or testing. The report should also include your class diagrams.	10