HW3 – Reducing Complexity with Appropriate Use Design Patterns

Estimated time: 24-30 hours

Objectives

- Become more familiar with UML modeling
- Become familiar with applying the Singleton, Factory, Flyweight, Command, and Undo patterns
- Improve unit testing skills to reduce complexity and improve performance
- Strengthen programming skills in area of user interfaces

Overview

In this assignment, you will build a basic UML drawing program that allows a user to create simple class diagrams. Specifically, the users should be able to draw symbols for classes, binary associations, aggregations, compositions, generalization/specializations, and dependencies.

Functional Requirements

1. The user should be able to create a new class diagram at any time, through either a menu option or a hotkey.
   1.1. The user should be able to give the diagram a title
   1.2. The user should be able to modify or clear the diagram title at any time

2. The user should be able to load a previously saved diagram at any time, through a menu option or hotkey.
   2.1. If the user has unsaved work in a current diagram, the program should warn the user before loading a file and give the user a chance to either save the current diagram or explicitly discard unsaved changes.
   2.2. The program should allow the user to browse for a file to load

3. The user should be able to save the current drawing to file, anytime there are unsaved changed.
   3.1. If the drawing has not been saved before, then the program should ask the use for the path name of the file to create
   3.2. The program should allow the user to browse to a directory and specify a file name in the directory.

4. The program should present the user with a palette containing the following types of class-diagram symbols: class, binary association, aggregation, composition, generalization/specialization, and dependency.

5. The user should able to select any component-type from a palette and place it on the drawing, in a syntactically correct place and with appropriate labels.
    5.1. A class must be represented by a box with its class name inside of the box
    5.2. All relationships (binary associations, aggregations, compositions, generalizations/specializations, and dependency) must be connected to two classes
        5.2.1. A binary association must be represented by a solid line connecting the two boxes representing the participating classes. The line must have a label and a direction arrow on the label that shows how to read the association's name
        5.2.2. An aggregation must be represented by a solid line connecting the two boxes representing the participating classes, with open diamond on the aggregate's end.
        5.2.3. A composition must be represented by a solid line connecting the two boxes representing the participating classes, with solid diamond on the composite's end.
        5.2.4. A generalization/specialization must be represented by a solid line connecting the two boxes representing the participating classes, with open triangle on the generalization's end.
        5.2.5. A dependency must be represented by a dashed line connecting the two boxes representing the participating classes, with open arrowhead on the dependency's end.

6. The user should be able to select an existing class on a diagram and perform an action on it.
    6.1. The user should be able to move the selected class to a new location on the drawing
        6.1.1. All connected relationships should be redrawn so they are still connected to the class.
    6.2. The user should be able to remove the selected class from the drawing
        6.2.1. All connected relationships should be deleted automatically.
    6.3. The user should be able to change the size of the box representing the class
        6.3.1. All connected relationships should be redrawn so they are still connected to the class.
    6.4. The user should be able to change the background color of the box representing the class
    6.5. The user could be able to duplicate a selected class to create a new class, but with a distinct name

7. The user should be able to select an existing relationship and perform an action on it.
    7.1. The user should be able to delete a selected relationship
    7.2. The user should be able to change the label on a binary association
    7.3. The user should be able to change the label's direction arrow on a binary association

8. The UI should allow multiple gestures (keystrokes, mouse movements, and button clicks) for some of the actions. For example, the user should be able to hit the delete key to delete a selected component.

9. The system should keep a history of actions (commands) that the user performs to since the drawing was created or last opened.

10. The user must be able to undo previous drawing actions in reverse order, all the way back to the initial drawing creation or opening of the drawing.

11. Allow the user to modify drawing styles for the component types.
    11.1. For the diagram, the user should be able to configure the background color, which will be the background color for the whole diagram, the default for the boxes that represent classes, the open triangles of generalization/specializations, and open diamonds of aggregations.
    11.2. For the diagram, the user should be able to configure the foreground color, which will be the color of all lines and text.
    11.3. For classes, the user should be able to configure the line thickness and attributes of the class-name font, like the face, size, and weight
    11.4. For relationships, in general, should be able to specify the line thickness
    11.5. For associations, the user should be able configure attributes of the label font, like the face, size, and weight
    11.6. For aggregations, the user should be able configure the size of the diamond
    11.7. For generalizations, the user should be able configure the size of the triangle
    11.8. For dependencies, the user should be able configure the size of the arrowhead

NOTE: For this drawing program, it is not necessary to support data members, methods, multiplicity constraints, roles, *n*-ary relationships, packages, or any other type of class-diagram component not explicitly mentioned in the requirements.

Instructions

To build this system, you will need to do the following:

1. Study the requirements. Ask questions to clarify, if necessary.
2. Design your system.
    2.1. Look for **good** opportunities to apply the singleton, factory, flyweight, command, and undo patterns.
    2.2. Also, don't forget about the other patterns that you've learned, like strategy, decorator, and observer.
    2.3. Use the patterns to help manage the inherent complexity of the problem and eliminate accidental complexity, if it seems appropriate. Don't feel like you must use every pattern. Only use a pattern if makes sense.
    2.4. Be sure to separate your GUI Layer from your Application Logic Layer. The classes that represent diagrams or things on the diagram should in your Application Logic layer. Classes for rendering these objects could be in the GUI layer.
3. Implement and test the classes in your Application Logic Layer. Your unit tests for the components in this layer must be executable and thorough
4. Implement your GUI
5. Test your software at the system level using ad hoc testing methods.

Submission Instructions

Zip up your entire solution, including test cases and sample input files, in an archive file called CS5700_hw3_<*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 designs consisting of UML class, interaction, and state diagrams that describe your system well enough that another developer could implement it from your diagrams.	20
A working implementation, with good encapsulation, abstractions, inheritance, and aggregation; readable code; low coupling and high cohesion.	30
"Good" application of design patterns in a way that helped manage complexity and ensure quality; no obvious missed opportunities; and no forced or awkward uses of design patterns.	30
Thorough executable unit test cases for core application logic	30
Reasonable systems test using ad hoc methods	10