Final CSCD 349 overview

* What’s a design pattern?
  + What benefits design patterns add for developers
* Pillars of OO (4 of them)
  + Description of each
    - **Abstraction** allows for multiple versions of a concrete implementation of the object being told what to do to exist. This allows for an object with varying data types unbeknownst to the client to be used for the same operations, such as an area method for a circle and square.
    - **Encapsulation**: the client should never be able to access data that is uniquely available to the underlying object. All the client needs to be able to do is invoke methods that manipulate the data of an object to produce their required answers. Encapsulation heavily uses abstraction, as it allows the use of an interface to separate the client from the workings of the program
    - **Polymorphism:** allows for even greater generalizations of an object and more functionality. An example of a shape, whether it be a circle, square, triangle or something else. Depending on what measurements a user enters, a different shape is formed. The user can call for the area to be measured and each shape knows how to do just that. Polymorphism also allows for runtime changes in what instance of a shape is being used, making a program even more versatile to the user
    - **Inheritance**: where a child class inherits a set of abilities from the parent class, extending the functionality of the parent but violates the pillar of encapsulation
* 10 – 12 fill in the blank
  + Gives a definition and we put down the pattern
    - **Strategy Pattern**: A structural pattern that defines a family of algorithms, encapsulates each one and makes them interchangeable
    - **Observer Pattern**: A behavioral pattern that defines a one to many dependency between objects so that when one object changes state, all the dependents are notified and updated automatically
    - **Decorator Pattern**: a structural pattern that attaches additional responsibilities to an object dynamically (at run time). Decorators provide a flexible alternative to sub-classing for extending functionality
    - **Façade Pattern**: provides a unified interface to a set of interfaces in a sub-system. Defines a higher-level interface that makes the sub-system easier to use
    - **Factory Method**: a creational pattern that defines an interface for creating an object. But let’s subclasses decide which class to create
    - **Adapter Pattern**: converts the interface of a class into another interface the client expects, lets classes work together that could not otherwise due to incompatible interfaces
* 10 true or false
  + About patterns or OO principals
    - OO principals
      * S.O.L.I.D
* All of quiz 2
  + List 3 OO principals and what each are and a pattern that follows it
    - **Single Responsibility**: the idea that every method of a class should have one responsibility. If a method does too many operations to achieve one goal, split the other operations up into methods of their own, this greatly improves testing of your code and being able to change code without breaking other aspects of a class. Single Responsibility applies to classes as well, if a class shape does work for circle and square, abstract shape and have a class for just circle operations and just square operations to simplify your code.
      * A design pattern that follows this principle is factory method which receives information and creates a concrete implementation of a user specified object. Just like how our factory assignment received measurements and a name and was able to form individual classes for Circle, Square, Rectangle and triangle in response. All of those classes had one responsibility, to perform operations on their shapes
    - **Open/Closed**: The idea of programing with the intent of being open to extensions (forming a new class that inherits the functionality of the super class, extending it past the original classes abilities) and closed to modifications (not having to change code in the class and potentially breaking other aspects of the program, changing everything slightly in the end).
      * A design pattern that follows this principle is decorator, which extends the functionality of an Object to include other Objects that modify the original in some way. With the Christmas tree assignment, we extended the Christmas tree functionality (i.e., price) to include a variety of decorations that did not modify the underlying structure, just wrapped themselves around it. Thus being open for extension, and closed for modification
    - **Liskov Substitution**: A class which extends the functionality of a super class must be completely substitutable for the super class (I.e., shape is super class, circle is a shape. Circle must have all of shapes abilities plus its own)
      * A design pattern that follows this principle is factory method again. The factory creates a shape, the shape can be circle, triangle, etc. All of these different shapes have the same functionality of shape (area) but a triangle could have extended functionality to find the angle given two sides, a circle cannot have that functionality though.
    - **Interface Segregation**: One all-encompassing interface is not OK! Instead use multiple small interfaces so a client doesn’t have functionality they don’t need or want to implement.
      * A design pattern that follows this principle is Observer. instead of having the observer functionality as part of the observable class it was split into two implementations. One that can update all the observers, and one the maintains a list of observers to be updated about the observable object.
    - **Dependency Inversion**: Taking all that is the same between classes and putting it in an abstraction, so that only methods that differ need implementing between classes of similar functionality.
      * A design pattern that follows this principle is Factory method AGAIN. In our assignment the shape abstracted variables and methods shared between all shapes, then the concrete versions of the shapes determined how to do it.
  + List 3 code smells, what each means and how to refactor it
    - Deodorant: Writing comments that describe what something is and not why something is. Solve by renaming methods and variables to be intent revealing
    - Conditional Complexity: using a large if else or switch case to decide what to use in a program. Solve by using strategy design to remove this complexity, having multiple small classes that also will allow for easy runtime changes to what is being used
    - Duplicate code: when multiple classes have identical or almost identical code. Solve by removing all that is the same between classes to an interface that those classes can then extend.
    - Freeloader: a class that doesn’t do enough to justify having it as a separate class. Solve by putting its functionality in another class.
    - Inappropriate Intimacy: classes have too much access to each other’s underlying data. Solve by using interfaces instead of inheritance if possible or change the visibility of data to deny access.
* 1 of 2 (second problem is worth 7 points extra credit) Design problem
  + Pick a pattern and justify why
  + Draw UML that applies pattern
  + Code snippets
* UML based on provided scenario