NAME: TOCHI VICTOR IBEBULAM

COURSE: COMP 3220

TITLE: OBJECT ORIENTED ANALYSIS AND DESIGN

PROJECT ASSIGNMENT:

IDENTIFY 5 DESIGN PATTERNS OF YOUR CHOICE THAT YOU NEED TO DESCRIBE IN DETAILS

CREATOR

It defines the scenario for when an object should create another object. The scenarios could be any of the following below.

Object A should create object B if:

- Object A **contains** Object B
- Object A **saves** Object B to a file or database
- Object A **uses** Object B in someway
- Object A **has all the data needed to instantiate** Object B

In most situations, more than one of these 4 rules would be true.

PROBLEM SCENARIO:

A Bike comprises Wheels and a Frame. What or Who should create the Wheel and Frame Objects?

SOLUTION:

The Bike Object decomposition = 2 Wheel Objects + 1 Frame Object. There are different ways of doing this. The Wheels and Frame make up a Bike, the rule specifies if the Bike Object should contain the 2 Wheel Objects And Frame Objects so therefore, Bike Object should **create and instantiate** the 2 Wheel Objects And Frame Objects directly in itself (constructor)

USEFULNESS:

- It encourages encapsulation
- It encourages low-coupling
- It increases clarity of code
- It fosters reusability

INFORMATION EXPERT

The general principle says if Object A is going to instantiate Object B, it must have the info needed to create it. Object A must be the expert on how to create Object B since it has the information so therefore it is assigned the responsibility.

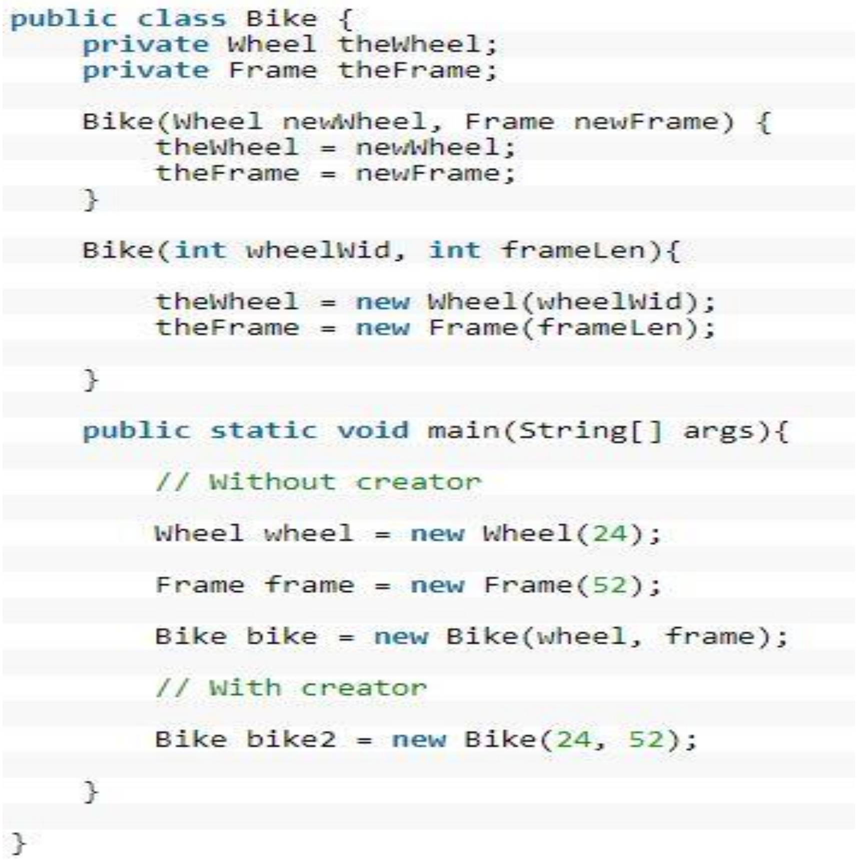
PROBLEM SCENARIO:

In order to create the Wheel Object and Frame Object in the Bike class, what information about the Wheels and Frame would the Bike Object need?

SOLUTION:

The answer could be anything that describes the Wheels and Frame. For example: The Wheels and Frame have dimensions. The Creator Object needs all the pieces to make the final Object.

Below is a code that describes the Creator and Information Expert



```
public class Bike {
     private Wheel theWheel;
     private Frame theFrame;

     Bike(Wheel newWheel, Frame newFrame) {
          theWheel = newWheel;
          theFrame = newFrame;
     }

     Bike(int wheelWid, int frameLen){

          theWheel = new Wheel(wheelWid);
          theFrame = new Frame(frameLen);

     }

     public static void main(String[] args){

          // Without creator

          Wheel wheel = new Wheel(24);

          Frame frame = new Frame(52);

          Bike bike = new Bike(wheel, frame);

          // With creator

          Bike bike2 = new Bike(24, 52);

     }

}
```

USEFULNESS:

- Encapsulation
- It allows the end user to provide parameters instead of creating parts of the object

LOW COUPLING

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

A class with high coupling relies on many other classes to work properly. A class with high coupling **is not going to be reusable, hard to understand in isolation and easily broken if other classes it relies on changes.**

LEVELS OF COUPLING:

- **Dependency:** This is when a Class A depends on Class B but it isn’t a member of Class A. It uses Class B. This is an example of very low coupling
- **Association:** When a class has a reference to another class. It has a **“Has A”** relationship with another class
- **Composition & Aggregation:** A class holds an instance of another class
- **Inheritance:** This is the highest level of coupling. A class implements or extends another class. It is an “Is A” relationship

Low coupling can be achieved by:

- designing classes that are independent so that changes in other classes have no effect
- Avoid creating sub classes and if you do subclass, make sure to subclass interfaces or abstract classes
- Add flexibility and encapsulation to classes to avoid major problems from high coupling if you have to use it

USEFULNESS:

- Low coupled classes are less affected by changes
- Simple to understand in isolation
- Class can be reused

HIGH COHESION

Cohesion is a measure of how strongly related and focused the responsibilities of an Object are. An Object with highly related responsibilities that does not do a tremendous amount of work has high cohesion. It is very important to keep classes very easy to comprehend, maintained and reusable. Create classes that handle few responsibilities for one part of the system; have many classes that handle all the many pieces of your system.

A class should perform what it is designed to do, if not sure, delegate it to another class.

PROBLEM SCENARIO:

You have a class that adds two numbers, but the same class creates a window displaying the result. This is a low cohesive class because the window and the adding operation don't have much in common.

SOLUTION:

The window is the visual part of the program and the adding function is the logic behind it.

To create a high cohesive solution, you would have to create a class Window and a class Sum. The window will call Sum's method to get the result and display it. This way you will develop separately the logic and the GUI of your application.

USEFULNESS:

- Reduced module complexity
- Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
- Increased module reusability

CONTROLLER

When creating a controller ask yourself **what object beyond the UI is going to coordinate all of the system operations?** Most of the cases, the controller is going to represent the device that the software runs in. To easily figure out which object inside of your system is the controller, look at the sequence diagram. The Object that connects to most of the other objects in the system is the controller.

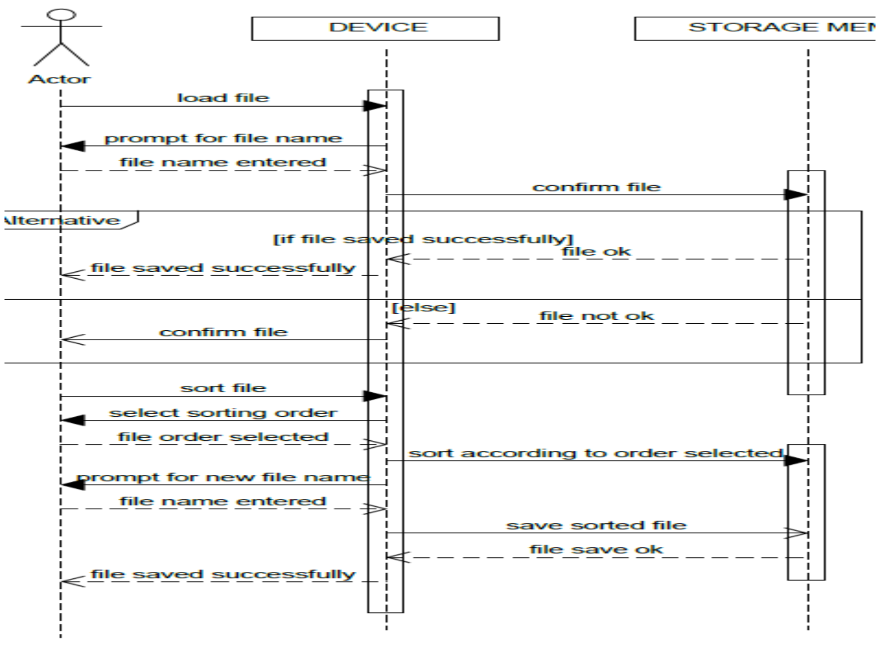
When creating a system using object oriented design, make sure you have just one controller

PROBLEM SCENARIO:

What object in assignment 1 receives and coordinates the system operation.

SOLUTION:

The device is the interface between the User and the Storage Memory. It receives messages and coordinates the sequence of events in the system.



USEFULNESS:

- The main job of the controller is to receive messages or requests from the UI
- The controller forwards the requests to the proper system objects
- It lowers the coupling of the UI to system objects
- It allows UI to change without affecting the system
- It prevents the UI from handling system events
- It encourages reusability