1. In software development, the circle-ellipse dilemma demonstrates different drawbacks that could arise from using subtype polymorphism (a data type that is related to another datatype or supertype). The issues are most encountered while the usage of object-oriented programming (OOP). This trouble violates the Liskov substitution principle, one of the principles of object-oriented programming. Suppose we need to define the class of ellipse and the class of circles. Ellipse has three data elements. If Circle inherits from Ellipse, then it will inherit these data variables, but a circle is a special case of the ellipse which only needs two data elements. So the problem is how to declare these two classes, should one of them be derived from the other or should they be independent?

   The problem worries which inheritance relationship must exist between classes which represent circles and ellipses (or, similarly, squares and rectangles). The problem illustrates the difficulties that can arise. At the same time, a base magnificence consists of techniques that change an object in a way that could invalidate invariants located in a derived class, causing the Liskov substitution principle to be violated. The circle–ellipse problem is used to criticize object-oriented programming. It could additionally suggest that hierarchical classifications are challenging to make conventional, that implies situational type structures may be extra practical.

2. The principle of class design state desired properties of class designs such as a class should have only one responsibility. This property of a class design can be related to one of the SOLID principles of object-oriented programming, the single responsibility principle. This principle states that a class should only have a single responsibility, and the specification of the class should be affected if there is a change only to one part of the software specification. We can also relate the data abstraction principle of class design with the interface segregation principle of the SOLID principle. These both principles state to separate the logical representation of an object's range of values from their implementation. It states to have many client-specific interfaces than one general-purpose interface. Class design principles serve as a basic guide to good design, not as a criterion to judge the quality of designs as SOLID principles.

3. Most of the projects/assignments I have done so far will fall in the lower right of the graph. I have only built projects where everything was in one package. This means there is no incoming dependency: that is there is no class outside the package that depends upon the class inside the package. In this case, *I* (the measure of instability) will be one, and the package is instable. If the package is instable, then it requires less work to make a change, and it is a concrete package. As the measure of instability increases, *I*, then the measure of abstractness, *A*, decrease. Because we only need to make a highly stable package more abstract so that it won't be difficult to extend, but if the package is instable, there is no need to make it abstract; since it is not difficult to change.

Reference

1. Martin, R. C. (n.d.). Design Principles and Design Patterns - cvc.uab.es. Retrieved from http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/materials_adicionals/principles_and_patterns.pdf