**Encapsulation**

This principle consists of two main concepts:

1. The ability of classes to hide their internal implementation, forcing users to interact with objects (instances of the class) via public interface methods and preventing users from accessing the object's state directly. For example, we can’t put food directly into a human’s stomach because it’s dangerous, but we can do that via the mouth (which is a public interface to the stomach in this context). Not all languages implement this principle strictly. For example, in Python, there are no built-in mechanisms to restrict access to the internals of a particular class, and the programmer is responsible for avoiding bad practices.
2. The ability of classes to connect both state and behavior together in one entity.

**Inheritance**

This concept makes it possible to extend the functionality of a base class without duplicating its code. A derived class inherits the properties and behaviors of the base class and can add its own functionality or override inherited methods.

**Polymorphism**

This concept allows for different behaviors under the same method name. For example, the *toString()* method, which is present in most high-level languages, prints an object’s state in a human-readable format, but its behavior depends on the object type, so different code will be executed. In this way, polymorphism often results from inheritance, but this is not the only available form. In many languages like C++, C#, and Java, it’s possible to define methods with the same name but with a different number or types of parameters, and this is called parametric polymorphism (or method overloading). However, there are also languages, such as Python, that don’t support parametric polymorphism in the same way.

**Aggregation and Composition**

Aggregation and Composition are both forms of association between objects, but they differ in the strength of the relationship and how they manage the lifecycle of the associated objects.

Aggregation is a "has-a" relationship where one object contains another, but the contained object can exist independently of the container. This means that if the container object is destroyed, the contained object can still exist on its own. Aggregation represents a weaker form of association between the objects. For example, imagine a Car and a Driver. A car has a driver, but the driver can exist independently without the car. If the car is destroyed, the driver remains unaffected. In this case, the driver is aggregated into the car, but they are not tightly coupled.

Composition is a stronger form of association, also a "has-a" relationship, but with a strict dependency between the objects. In composition, the contained object cannot exist independently of the container. If the container object is destroyed, all the objects it contains are also destroyed. This represents a stronger ownership, where the lifecycle of the contained object is controlled by the container.

Think of a House and its Rooms. A house is composed of rooms, and a room cannot exist without the house. If the house is destroyed, all the rooms are destroyed as well. In this case, the rooms are tightly coupled to the house and cannot exist on their own.