**The solid principles**

**S**ingle Responsibility Principle

**O**pen/closed principle

**L**iskov substitution principle

**I**nterface segregation principle

**D**ependency inversion principle

**Single Responsibility Principle**

Each class is responsible for its role in the functionality of the system, and the responsibilities of a class must be encapsulated by the class and its attributes must be consistent with the responsibilities of the class. This principle is related to cohesion, which means that responsibilities are aligned and focused on a single purpose.

**Open/closed principle**

Software entities should be open for extension but closed to modification, open if it is still available for extension, closed if it is available for use by other class, and therefore should not be modified If the programmer makes modifications, the principle is broken.

**Liskov substitution principle**

Product a base class, with one virtual method, called save, whose intent is to save an object to a file. When implementing a specialization, Widget, of some Product: The implementation of save in Widget adheres to the purpose of save in Product don’t have it do some unrelated thing, like re-load the object from a file instead doesn’t rely on stronger assumptions than programmatically implement any special conditions that required and handle exceptions appropriately Ensure that accomplishes, as minimum, all that is supposed to accomplish. If is supposed to save the x attribute to a file, then must do at least this much.

**Interface segregation principle**

An interface is a ”window” or “portal” into the functionality of a component An interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces, an interface represents public methods of a component Java does support interfaces directly Core Idea: No client (user of a component) should be forced to depend on methods that it does not use The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes

**Dependency inversion principle**

Organize the system in layers, some reusable libraries will be abstract or detail oriented. Reusable layer components do not depend on detail layers. In addition, they depend on the abstraction that the detailed components implement. The abstractions must not depend on the details and the implementation must depend on the abstraction