Design patterns end up being a way to describe a solution, but you don't **have** to adapt your classes to the solution. Think of them more as a guide that suggest a good solution to a wide array of problems.

Let's talk about SOLID:

1. **Single responsibility**.

   Just because u can doesn’t mean u should.

   A class should have only one responsibility. That means that for example, a Person class should only worry about the domain problem regarding the person itself, and not for example, its persistence in the database. For that, you may want to use a PersonDAO for example. A Person class may want to keep its responsibilities the shortest it can. If a class is using too many external dependencies (that is, other classes), that's a symptom that the class is having too many responsibilities. This problem often comes when developers try to model the real world using objects and take it too far. Loosely coupled applications often are not very easy to navigate and do not exactly model how the real world works.

   Ur classes should be small not more than screen full of code, avoid GOD classes, split bog classes into smaller classes.

2. **Open Closed**.

   Open chest surgery is not needed when putting on a coat.

   Classes should be extendible, but not modifiable. That means that adding a new field to a class is fine, but changing existing things are not. Other components on the program may depend on said field.

   Ur classes should be open for extension, but closed for modification.

3. **Liskov substitution**.

   If it looks like a duck, quacks like a duck, but need batteries you probably have the wrong abstraction

   A class that expects an object of type animal should work if a subclass dog and a subclass cat are passed. That means that Animal should NOT have a method called bark for example, since subclasses of type cat won't be able to bark. Classes that use the Animal class, also shouldn't depend on methods that belong to a class Dog. Don't do things like "If this animal is a dog, then (casts animal to dog) bark. If animal is a cat then (casts animal to cat) meow".

   Any violation will fail the “is a” test eg-

   A square is a rectangle

   But a rectangle is not a square.

4. **Interface segregation principle**.

You want me to plug this in but where?

Keep your interfaces the smallest you can. A teacher that also is a student should implement both the IStudent and ITeacher interfaces, instead of a single big interface called IStudentAndTeacher.

Many client specific interfaces are better than one general purpose interface.

Keep ur components focused and minimise dependencies between them.

5. **Dependency inversion principle**.

Would u solder a lamp directly to the electrical wiring in a wall?
Objects should not instantiate their dependencies, but they should be passed to them. For example, a Car that has an Engine object inside should not do engine = new DieselEngine(), but rather said engine should be passed to it on the constructor. This way the car class will not be coupled to the DieselEngine class.
High-level modules should not depend on low-level modules. Both should depend on abstractions.'

'Abstractions should not depend upon details. Details should depend upon abstractions.'

This is not the same as dependency injection- which is how objects obtain dependent objects.

- Dependency Injection is an implementation technique for populating instance variables of a class.
- Dependency Inversion is a general design guideline which recommends that classes should only have direct relationships with high-level abstractions.