SOLID PRINCIPLES

1)Single Responsibility Principle (SRP): This principle states that each class should have one responsibility, one single purpose. This means that a class will do only one job, which leads us to conclude it should have only one reason to change.

Violating SRP:

```
Class Signup {
Validate (String username, String email, String password)
{
//logic
}
Authenticate (String email, String password)
{
//logic
}
```

Not violating SRP:

```
Class UserValidation{
Validate (String username, String email, String password) {
//logic
}
}
Class Authentication {
Authenticate (String username, String email, String password)
{
///logic
}}
```

Here, if both validate and authenticate functions are in the same class, SRP is violated as a single class performs two distinct functions. Hence to follow SRP two separate classes with unique responsibility are made.

2)Open Close Principle (OCP): This principle states that objects or entities should be open for extension but closed for modification.

Violating OCP:Here the function to validate email, name and password is inside a single class. So, if we want to validate new field then this class must be modified, which is violation of OCP,to avoid this

```
Class Validation {
Emailvalidate(String email)
{
  //logic
}
Namevalidate(String name)
{
 //logic
}
Passwordvalidate(String password)
{
//logic
}}
```

Not violating OCP:

```
Interface Validation{
validate(data);
}
Class EmailValidator implements Validation {
@Override
Validate(data) {
//logic to validate email
}}
Class NameValidator implements Validation {
@Override
Validate(data) {
//logic to validate name
}}
Class PasswordValidator implements Validation {
@Override
Validate(data) {
//logic to validate password
}}
```

3)Liskov's Substitution principle (LSP): This principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Violating LSP:

```
class AuthenticationService{
 SigninAuthenticate () {}
 SignupAuthenticate () {}
}
Class SigninService extends AuthenticationService{
 SigninAuthentication () {}
// problem: this class extends function SignupService also. But it doesn't need that function. So, object of parent class can't substitute child class and vice versa
}
Class SignupService extends AuthenticationService{
  SignupAuthentication () {}
//problem
}
AuthenticationService signInService = new SigninService();
        SigninService.authenticate;
```

Not violating LSP

```
Class AuthenticationService{
  Authenticate () {}
}
Class SigninService extends AuthenticationService{
  SigninAuthentication () {}
}
Class SignupService extends AuthenticationService{
   SignupAuthentication () {}
}
AuthenticationService signInService = new SigninService();
        SigninService.authenticate;
```

4)Interface Segregation Principle (ISP): This principle states that states that no code should be forced to depend on methods it does not use. Many client specific interfaces are better than one general purpose interface.

Using ISP: Here if both signUp and signIn functions are in the same interface, then as the class SigninServices and SignupServices implements the interface, both the classes would also get the functions that they won't use, which is violation of ISP. So, to avoid this separate small interface is created and then the required classes implement them.

```
interface SignupServices {

        void signUp(String username, String password);
}
interface SigninServices {

        void signIn(String useremail, String password);
}
class SignupAuthentication implements SignupServices {

        @Override
        public void signUp(String username, String password) {
          ///logic
        }
}
class SigninAuthentication implements SigninServices {

        @Override
        public void signIn(String useremail, String password) {
        }
}
```

5)Dependency Inversion Principle (DIP): This principle states that any higher classes should always depend upon the abstraction of the class rather than the detail. High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Using DIP:Here the object of the interface(abstract) is used to call the implemented classes

```
Interface Validation{
validate(data);
}
Class EmailValidator implements Validation {
@Override
Validate(data) {
//logic to validate email
}}
Class NameValidator implements Validation {
@Override
Validate(data) {
//logic to validate name
}}
 Class PasswordValidator implements Validation {
@Override
Validate(data) {
//logic to validate password
}}
Validation Name = new NameValidator();
                Validation Email = new EmailValidator();
                Validation Password = new PasswordValidator();

                Password.validate(password);
                Email.validate(email);
                Name.validate(name);
```