Interface Segregation Principle

Rule: Interfaces should not force classes to implement what they can’t do. Large interfaces should be divided into small ones.

Consider we add decoding feature to the PasswordHasher interface.

```
public interface PasswordHasher
{
   String hashPassword(String password);
   String decodePasswordFromHash(String hash);
}


public class Sha256Hasher implements PasswordHasher {

   /*
    * CAN't endode and decode password using SHA256.
    *
    * SHA-256 isn’t an encryption algorithm, so there is nothing to decrypt.
    * It is a cryptographic hash algorithm, which allows you to verify that
data matches what is expected but cannot be reversed to produce the
original data.
    */
}
```

SHA256 : It is completely impossible to decrypt SHA-256 in the same way that it is impossible to decrypt milk. SHA-256 isn’t an encryption algorithm, so there is nothing to decrypt. It is a cryptographic hash algorithm, which allows you to verify that data matches what is expected but cannot be reversed to produce the original data.

Problem: This would break this law since one of our algorithms, the SHA256 is not decryptable practically, (it’s a one-way function).

Solution:

Break down the responsibilities into two interfaces .

1. PasswordHasher interface would have hashPassword feature/method.

```
public interface PasswordHasher
{
    String hashPassword(String password);
}
```

2. Add decodePasswordFromHash feature to another interface 'Decryptable'

```
public interface Decryptable
{
   String decodePasswordFromHash(String hash);
}
```

Bas64 is an ending and decoding algorithm so it implements both actions stated above by implementing PasswordHasher and Decryptable interfaces.

```
public class Base64Hasher implements PasswordHasher, Decryptable
{
   @Override
   public String hashPassword(String password)
   {
      return "hashed with base64";
   }

   @Override
   public String decodePasswordFromHash(String hash)
   {
      return "decoded from base64";
   }
}
```

Example 2

Interface Segregation

Within the scope of this principle, the phrase “forced beauty” always comes to my mind. So imagine an interface and assume that all the responsibilities are in this interface, if I implement this interface in a small class to use only a few methods, I will have to override all the remaining methods and these methods will pollute my class unnecessarily. Instead, it is necessary to divide interfaces into more meaningful parts, avoiding unnecessary methods.

In summary, this principle proposes not to assign responsibilities to a single interface, but to divide them by customized meaningful interfaces.

```
public interface PrinterTasks {
    void print(String printContent);
    void scan(String scanContent);
    void fax(String faxContent);
    void printDuplex(String printDuplexContent);
}
```

Let’s assume that we have an interface that hosts processes belonging to the printer. In this case, there will be no problem for the full-featured HPLaserJetPrinter implementation:

```
public class HPLaserJetPrinter implements  PrinterTasks {
    public void print(String printContent) {}
    public void scan(String scanContent) {}
    public void fax(String faxContent) {}
    public void printDuplex(String printDuplexContent) {}
}
```

But for LiquidInkjetPrinter model printer we will have fax() and printDuplex() methods which are not used:

```
public class LiquidInkjetPrinter implements PrinterTasks {
    public void print(String printContent) {}
    public void scan(String scanContent) {}
    public void fax(String faxContent) {
        throw new UnsupportedOperationException();
    }
    public void printDuplex(String printDuplexContent) {
        throw new UnsupportedOperationException();
```

```
    }
}
```

As you can see, we have overridden unused methods unnecessarily. This situation broke the Interface Segregation Principle.

To solve this problem, it would be logical to divide the non-common features into different interfaces. Namely;

```
public interface PrinterTasks {
    void print(String printContent);
    void scan(String scanContent);
}
public interface FaxTasks {
    void fax(String faxContent);
}
public interface PrintDuplexTasks {
    void printDuplex(String printDuplexContent);
}
```

The division above seems perfectly logical. When we examine their use...

```
public class HPLaserJetPrinter implements PrinterTasks, FaxTasks,
PrintDuplexTasks {
    public void print(String printContent) {}
    public void scan(String scanContent) {}
    public void fax(String faxContent) {}
    public void printDuplex(String printDuplexContent) {}
}
public class LiquidInkjetPrinter extends PrinterTasks {
    public void print(String printContent) {}
    public void scan(String scanContent) {}
}
```

As you can see, each class implements the abstraction, which has its own characteristics, so that other dependencies are not unnecessarily forced.