Date 07/06/2019

A in function do something, there should not emerge any problem & all processes should work smoothly as previously.

Example:

```
class VideoPlayer
{
    public function play($file)
    {
        //play the video
    }
}
class AviVideoPlayer extends VideoPlayer
{
    public function play($file)
    {
        if(pathinfo($file, PATHINFO_EXTENSION) !==
            'avi')
        {
            throw new Exception; //violates the LSP
        }
    }
}
```

Here highlighted code is responsible to violate the LSP. As VideoPlayer class play method has same execution & return type of all types of video players but it's child class AviVideoPlayer play method throw exception for non-avi video players.

Date
07/06/2017

Example:

```
interface LessonRepositoryInterface
{
    /**
     * Fetch all records
     *
     * @return array
     */
    public function getAll();
}
class FileLessonRepository implements LessonRepositoryInterface
{
    public function getAll()
    {
        //return through filesystem
        return [];
    }
}
class DbLessonRepository implements LessonRepositoryInterface
{
    public function getAll()
    {
        return Lesson::all(); //violates the LSP
    }
}
function foo(LessonRepositoryInterface $lesson)
{
}
```

return Lesson::all()->toArray();

Here we can use both class object i.e. either FileLessonRepository or DbLessonRepository.

Date
07/06/2019

Here highlighted code in DbLessonRepository class violates the L.S.P. As first child class FileLessonRepository return array which is ok but second child class DbLessonRepository of LessonRepositoryInterface returns Collection class object which is wrong. All child classes of same interface must return same type of output.

Principle

4. Interface Segregation: A client should never be forced to implement an interface that it does not use.

or

Clients should not be forced to depend on methods they do not use.

Example:

```
interface ManageableInterface
{
    public function beManaged();
}
interface WorkableInterface
{
    public function work();
}
interface SleepableInterface
{
    public function sleep();
}
```

Date
07/06/2017

```
class HumanWorker implements WorkableInterface,
    SleepableInterface, ManageableInterface
{
    public function work()
    {
        //implement work method
    }
    public function sleep()
    {
        return 'human sleeping';
    }
    public function beManaged()
    {
        $this->work();
        $this->sleep();
    }
}
class AndroidWorker implements WorkableInterface,
    ManageableInterface
{
    public function work()
    {
        return 'android working';
    }
    public function beManaged()
    {
        $this->work();
    }
}
```

Date 07/06/2017

decouple - डिकपल - अलग करना

```
class Captain
{
    public function manage(ManageableInterface
                                $worker)
    {
        $worker->beManaged();
    }
}
```

5. Dependency Inversion Principle: Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module; but they should depend on abstractions.
This principle allows for decoupling, following example seems like the best way to explain this principle.

```
class PasswordReminder
{
    private $dbConnection;
    public function __construct(MySQLConnection
        $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }
}
```

First the MySQLConnection is the low level module while the PasswordReminder is high level, but according to the definition of D in S.O.L.I.D. which states that 'Depend on Abstraction not on concreti