**SOLID and Clean Code**

**Single responsibility principle**

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

**Pros of using DI**:

* DI makes testing logic in isolation much easier
* TODO

**DI makes testing logic in isolation much easier**

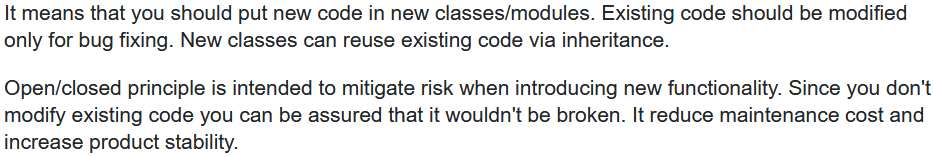
|  |
| --- |
| public class DropboxClient {  private static final String ***KEY\_DROPBOX\_ACCESS\_TOKEN*** = **"DROPBOX\_ACCESS\_TOKEN"**;  private final String dropboxAccessToken;  private DbxClientV2 dropboxClient;   DropboxClient(**Dotenv** env) {  dropboxAccessToken = env.get(***KEY\_DROPBOX\_ACCESS\_TOKEN***); if (StringUtils.*isBlank*(dropboxAccessToken)) {  ***log***.info(**"Cannot instantiate instance because {} is empty"**, ***KEY\_DROPBOX\_ACCESS\_TOKEN***);  } else {  DbxRequestConfig config = DbxRequestConfig.*newBuilder*(**"dropbox/notion-backup"**).build();  dropboxClient = new DbxClientV2(config, dropboxAccessToken);  }  }   public void **upload**(File fileToUpload) { dropboxClient.files().upload(fileToUpload); }  } |

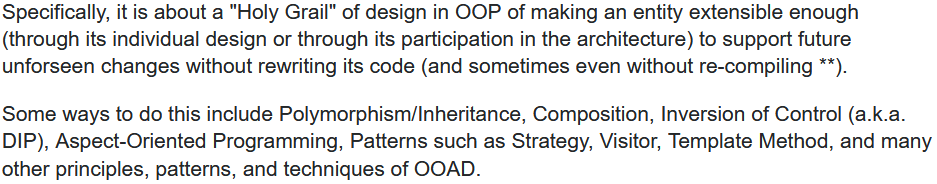
For instance we want to mock the upload() method of this class. The method depends on a dropboxClient field, which is instantiated inside the constructor. Only the dotenv instance is injected from outside.

So when we want to test only the upload() method (unit test) and not the upload() method AND the dropboxClient methods (integration test), we need to be able to mock the dropboxClient (and also any other method calls which goes outside of this method block).

**Open–closed principle**

"Software entities ... should be open for extension, but closed for modification."



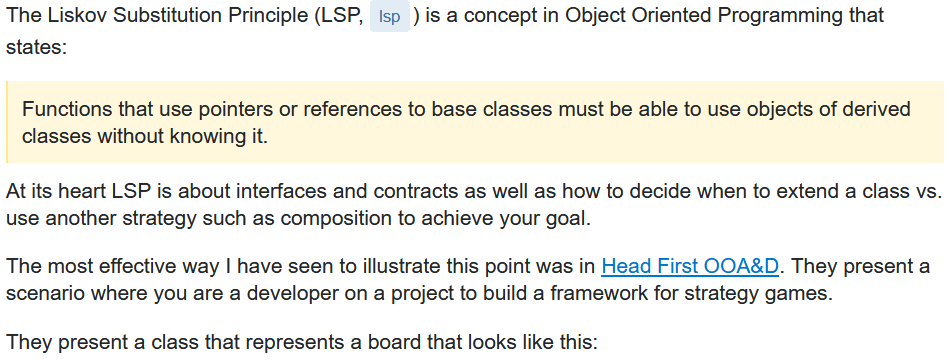


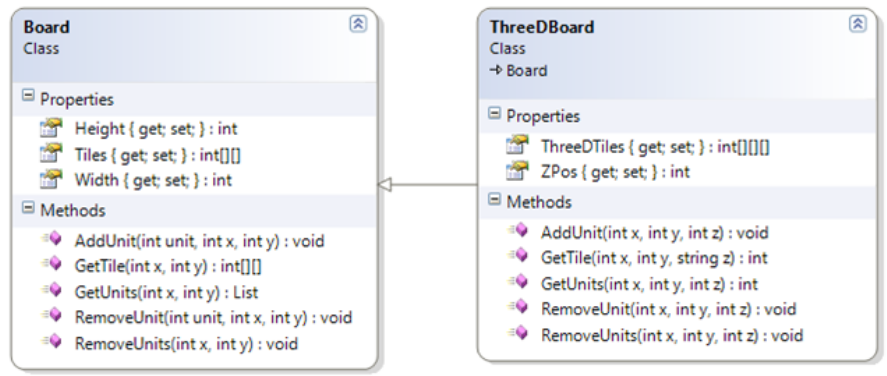
* **we should strive to write code that doesn’t have to be changed every time the requirements change**.

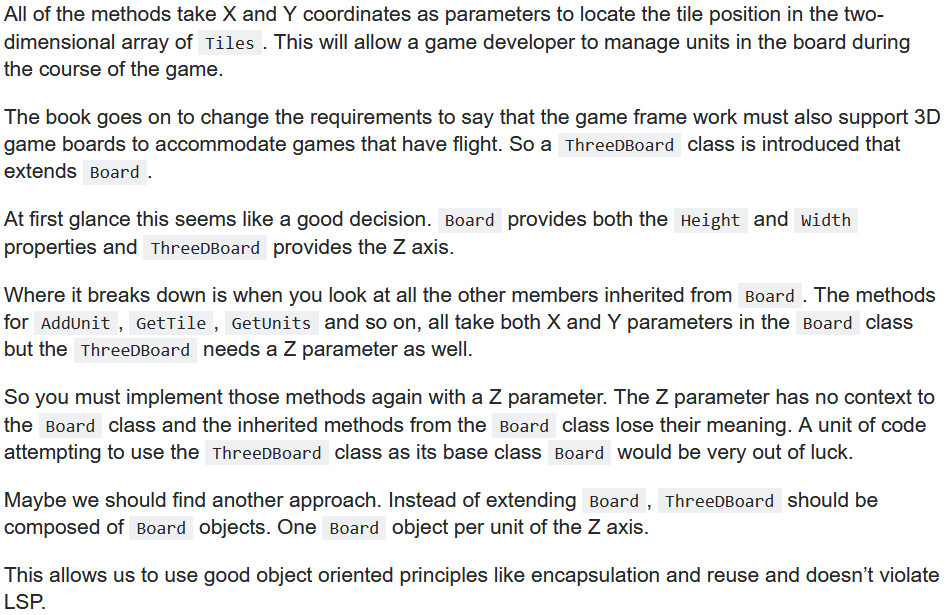
**Liskov substitution principle**

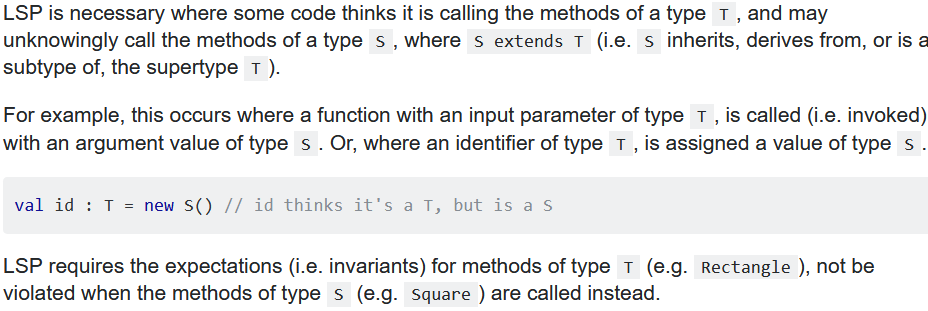
"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

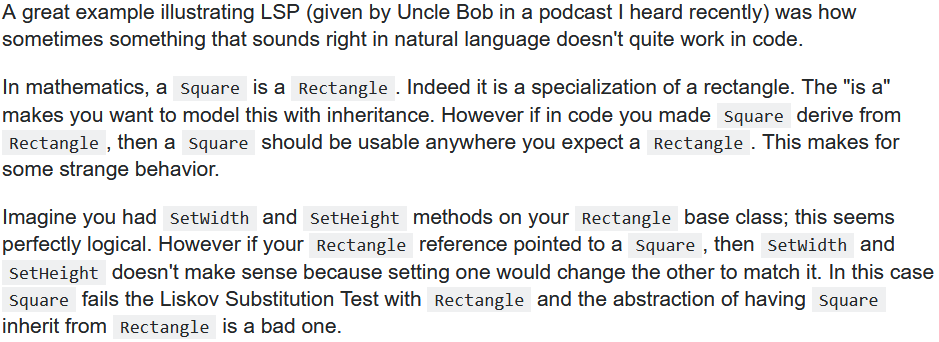
Is it enough for a particular object to inherit from another? In 1987, Barbara Liskov proposed an answer to this question, arguing that an object should only be considered a subtype of another object if it is **interchangeable** with its parent object so far as any interacting function is concerned.









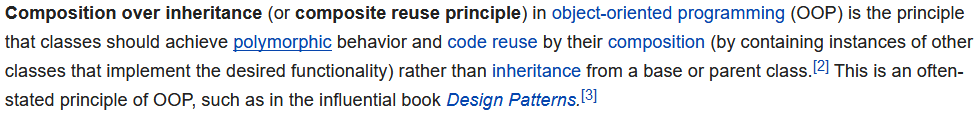




**In simple words**: LSP is about deciding when to and when not to inherit certain classes and make a subtype. Even if it sounds perfectly logical in a real world example, it still might break LSP, resulting in some unexpected behaviour. Basically if a class is a subtype of another, they should be exchangeable with no unexpected side-effects. If it can’t be done then you need to take a look at your inheritance hierarchy, changing it. This doesn’t mean that a subtype can’t extend itself with new functionalities. It means the user wants to see the same result for the common behaviours that both the superclass and subclass has.

* **If it looks like a duck, quacks like a duck but needs batteries, you probably have the wrong abstraction**.
* Explained in the **Board**, **ThreeDBoard** example: if a ‘subtype’ modifies its behaviour so drastically, meaning that it is no more connected to the parent class, then it violates the LSP. We want them to be exchangeable but **ThreeDBoard** introduces a new parameter for every method that it inherits from the **Board** class, if you try to use **ThreeDBoard** in place of **Board** and use that **ThreeDBoard** instance to do your operations as if it were a **Board**, you will not be able to.
* If we say that something(parent) has a property or smth is true for it at the top of the inheritance chain, then everything that comes down the chain must respect that property and the same should be true for those as well. Parent can do something -> all subtypes can do that as well, the same thing.

This is why we talk about composition over inheritance. Most of the time we make simple and logical sounding assumptions like ‘all of the animals can speak’ and the dangerous part is we think we identified all of the subtypes of animals that we will see in our system and later we get new requirements, then if we don’t refactor we might break LSP (apparently not all of the animals can speak).

Use ‘has a’ instead of ‘is a’ because ‘has a’ is much less troublesome. That thing which you ‘have’ has its own inheritance hierarchy, so the LSP is still applied to that.

**Interface segregation principle**

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

In the field of software engineering, the **interface-segregation principle (ISP)** states that no client should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP.

**Dependency inversion principle**

One should "depend upon abstractions, [not] concretions."

In object-oriented design, the dependency inversion principle is a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.

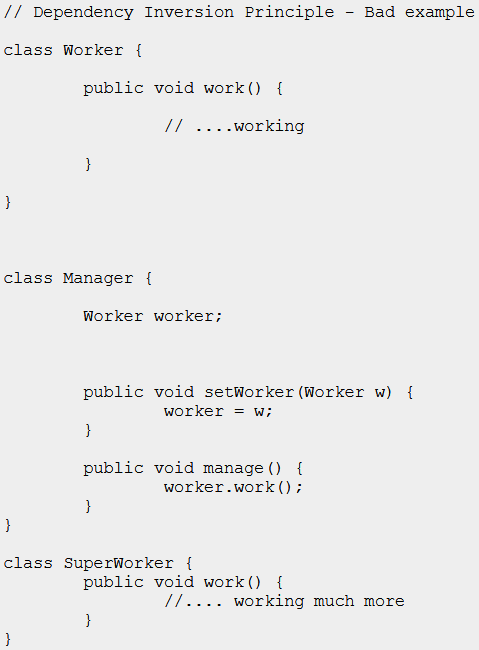
The principle states:

* High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
* Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

**DIP Example**

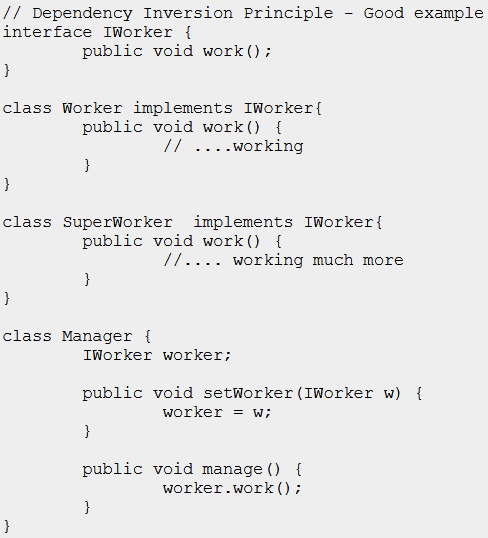
Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.

Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

* we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).
* some of the current functionality from the manager class might be affected.
* the unit testing should be redone.

All those problems could take a lot of time to be solved and they might induce new errors in the old functionlity. The situation would be different if the application had been designed following the Dependency Inversion Principle. It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

* Worker ve SuperWorker farklı classlar oldugu için bu basit örnekte bile Workerın kullanıldıgı yerlerde bütün typeları vs değiştirmek gerek. Aynı şekilde unit testindeki typleları vs...

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved(considering there is no change in the high level logic):

* Manager class doesn't require changes when adding SuperWorkers.
* Minimized risk to affect old functionality present in Manager class since we don't change it.
* No need to redo the unit testing for Manager class.

**Conclusion**

When this principle is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In this case instantiation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort, will result in more classes and interfaces to maintain, in a few words in more complex code, but more flexible. This principle should not be applied blindly for every class or every module. If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.

**Clean Code / Best Practices**

getSomething() vs findSomething() vs fetchSomething() ….

<https://stackoverflow.com/questions/2141818/method-names-for-getting-data>

<https://softwareengineering.stackexchange.com/questions/182113/how-and-why-to-decide-between-naming-methods-with-get-and-find-prefixes>

**Using NotNull Annotation in method argument**

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application

Description automatically generated

**Which @NotNull Java annotation should I use?**

* <https://stackoverflow.com/questions/4963300/which-notnull-java-annotation-should-i-use>

Graphical user interface, text, application, email

Description automatically generatedI'm looking to make my code more readable as well as use tooling like IDE code inspection and/or static code analysis (FindBugs and Sonar) to avoid NullPointerExceptions. Many of the tools seem incompatible with each others' @NotNull/@NonNull/@Nonnull annotation and listing all of them in my code would be terrible to read. Any suggestions of which one is the 'best'? Here is the list of equivalent annotations I've found:

Graphical user interface, text, application, email

Description automatically generated

* <https://projectlombok.org/features/NonNull>
* When and how much to use @Nonnull or @Nullable?