***S O L I D***

**Tim Verhees**

*07-10-2022   
V0.1*

# **Versions**

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Description** |
| **07-09-2022** | 0.1 | Initial version |

# **Distribution list**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Name** | **Function** |
| **07-10-2022** | 0.1 | Tim Verhees | Project Handler |
|  | 0.1 | Ali Odaci | Semester Coach |
|  | 0.1 | Maja Pesci | Technical Teacher |
|  | 0.1 | Jacco Snoeren | Technical Teacher |

**Introduction**

SOLID is a set of design principles followed by developers all around the world. It helps keep your application within the best practises for coding and design. This means that it is optimized in terms of how it is structured, written and designed. This helps especially when other people need to start working on the code or if something were to go wrong.

Here’s how the SOLID principles go:

* **S**ingle-responsibility principle: "A module should be responsible to one, and only one, actor." spacer More literally seen as “A class should only have one reason to change.”
* **O**pen–closed principle: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." Meaning that if you were to have to implement a spacer function and change a class, it should only be extended on, not modified.
* **L**iskov substitution principle (LSP): Is a principle based on “substitutability”. Substitutability is spacer the principle that a class and a sub-class should be interchangeable without breaking spacer the program.
* **I**nterface segregation principle (ISP): This principle simply states that no code should depend spacer on methods it does not use. Meaning that, for example, you keep interfaces small spacer and concise. So that when you implement them into and implementation, all spacer acer methods are used.
* **D**ependency inversion principle: This principle is build on two important parts;
  + High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
  + Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

This means that when we look at the high level modules, they are fully independent of lower level modules and instead rely on an abstraction like an interface. This then also rings through for the lower level modules. This then also means that these abstractions should not rely on other objects to provide details.

**SOLID integration**

These strong principles are best practice when it comes to object-oriented design of your application. This means that I will also be using these principles within my individual project’s application. Here’s how that’s going to look for each principle:

* **S**ingle-responsibility principle: I will achieve this principle by separating each classes into the use cases they perform. And example will be the CRUD operations for my Card class. In this case each part of CRUD is separate into different classes and interfaces, with the read being separated into filtered and unfiltered. This ensures that if something is wrong with any of the CRUD operations, I know exactly where to go.
* **O**pen–closed principle: In this case I’m going to achieve this through the aforementioned separation of classes where anything new will be an extension of an existing class (e.g. accepting more fields for the update method) or a new class entirely.
* **L**iskov substitution principle (LSP): This principle will be achieved through not working with subclasses.
* **I**nterface segregation principle (ISP): I will be achieving this principle through not only a separation of classes, but also a separation of interfaces, where each interfaces corresponds to one use case, similar to the classes. This means that nothing within the interface goes unused.
* **D**ependency inversion principle: This will be achieved by making the interfaces simple and concise, making it so they won’t rely on details. Furthermore, both higher and lower level modules will not rely on each other and will always have an interface in between.