OOAD

sean moloney

January 2020

1 SOLID Principles

- `Single Responsibility Principle`
- `Open/Closed Principle`
- `Liskov Substitution Principle`
- `Interface Segregation Principle`
- `Dependency Inversion Principle`

1.1 Single Responsibility Principle

The Single Responsibility Principle states that there should never be more than one reason for a class to change.

When a class has multiple responsibilities, the likelihood that it will need to be changed increases. Each time a class is modified the risk of introducing bugs grows. By concentrating on a single responsibility, this risk is limited.

1.2 Open/Closed Principle

The Open/Closed Principle states that entities should be open to extension but closed to modification.

As with the Single Responsibility Principle, this principle reduces the risk of new errors being introduced by limiting changes to existing code.

1.3 Liskov Substiturion Principle

The Liskov Substitution Principle states that functions that use pointers or references to base classes should be able to use objects of derived classes without knowing it.

If you create a class with a dependency of a given type, you should be able to provide an object of that type or any of its subclasses without introducing unexpected results and without the dependent class knowing the actual type of the provided dependency. If the type of the dependency must be checked so that behaviour can be modified according to type, or if subtypes generated unexpected rules or side effects, the code may become more complex, rigid and fragile.

1.4 Interface Segregation Principle

The Interface Segregartion Principle states that clients should not be forced to depend on an interface that they do not use.

Often when you create a class with a large number of methods and properties, the class is used by other types that only require access to one or two members. The classes are more tightly coupled as the number of members they are aware of grows. When you follow the ISP, large classes implement multiple smaller interfaces that group functions according to their usage. The dependents are linked to these for looser coupling, increasing robustness, flexibility and the possibility of reuse.

1.5 Dependency Inversion Principle

The Dependency Inversion Principle states that high level modules should not depend on low level modules, they should depend on abstractions. These abstractions should not depend upon details.

The DIP primarily relates to the concept of layering within applications, where lower level modules deal with very detailed functions and higher level modules use lower level classes to achieve larger tasks. The principle specifies that where dependencies exist between classes, they should be defined using abstractions, such as interfaces, rather than by referencing classes directly. This reduces fragility caused by changes in low level modules introducing bugs in the higher layers. The DIP is often met with the use of dependency injection.