**S.O.L.I.D** is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin.

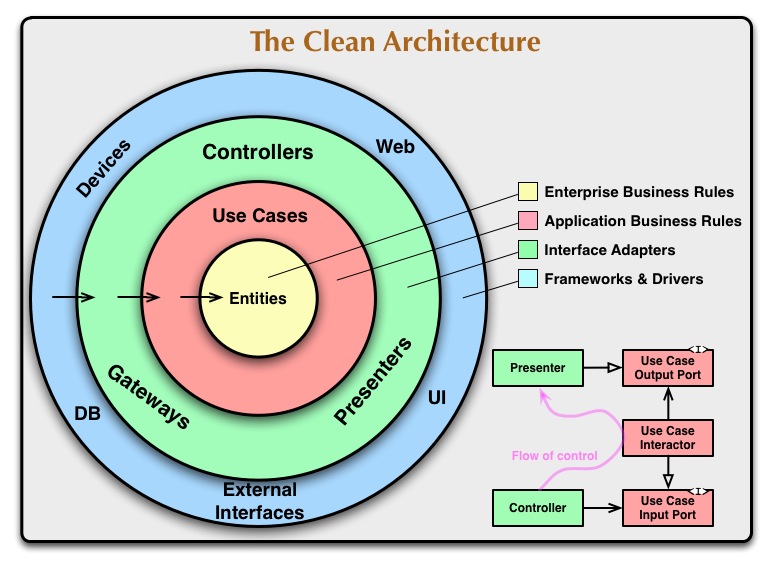
* **S** - *Single-responsibility principle*. A class should have one and only one reason to change, meaning that a class should have only one job.
* **O** - *Open-closed principle*. Objects or entities should be open for extension, but closed for modification.
* **L** - *Liskov substitution principle*. Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

. The **principle** defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass

* **I** - *Interface segregation principle*. A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
* **D** - *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.

*ACID* is a acronym which is commonly used to define the properties of a relational database system, it stand for following terms

* **Atomicity** - This property guarantees that if one part of the transaction fails, the entire transaction will fail, and the database state will be left unchanged.
* **Consistency** - This property ensures that any transaction will bring the database from one valid state to another.
* **Isolation** - This property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially.
* **Durable** - means that once a transaction has been committed, it will remain so, even in the event of power loss.



What is DDD?

Domain-driven design is not a technology or a methodology. DDD provides a structure of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains.

- Wikipedia

Concepts to cover in this article:

1. Understanding the Domain.
2. Ubiquitous Language.
3. Contexts and Bounded Contexts.
4. Entities and Value Objects.
5. Aggregates and Aggregate Roots.
6. Persistence Ignorance.
7. Repository.
8. Domain Service.

In this article I will try to avoid becoming too technical, rather I will try to go through different concepts of DDD being close to the real world.  I will try not to show any code here. Because I believe if you understand the concept and starts thinking in DDD way, implementation is easy. The toughest part is to tune your thinking process!

Understanding the Domain<https://github.com/cx525v/aspnetcore/upload/master>

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.  
- Wikipedia

* You have a defined amount of land
* Your building will have 6 floors.
* Each floor will have 4 apartments.

What is your domain here?

The domain is Building(?). It could be. But note that, if you consider Building as your domain you may miss few granular details for your requirement. The building you are going to design must have design for apartments where people will live. So, a general term “Building” can make us miss few details. So, we may narrow down our domain to “Residential Building”.

|  |  |
| --- | --- |
| **Creational Patterns** | |
| [Abstract Factory](https://www.dofactory.com/net/abstract-factory-design-pattern) | Creates an instance of several families of classes |
| [Builder](https://www.dofactory.com/net/builder-design-pattern) | Separates object construction from its representation |
| [Factory Method](https://www.dofactory.com/net/factory-method-design-pattern) | Creates an instance of several derived classes |
| [Prototype](https://www.dofactory.com/net/prototype-design-pattern) | A fully initialized instance to be copied or cloned |
| [Singleton](https://www.dofactory.com/net/singleton-design-pattern) | A class of which only a single instance can exist |

|  |  |
| --- | --- |
| **Structural Patterns** | |
| [Adapter](https://www.dofactory.com/net/adapter-design-pattern) | Match interfaces of different classes |
| [Bridge](https://www.dofactory.com/net/bridge-design-pattern) | Separates an object’s interface from its implementation |
| [Composite](https://www.dofactory.com/net/composite-design-pattern) | A tree structure of simple and composite objects |
| [Decorator](https://www.dofactory.com/net/decorator-design-pattern) | Add responsibilities to objects dynamically |
| [Facade](https://www.dofactory.com/net/facade-design-pattern) | A single class that represents an entire subsystem |
| [Flyweight](https://www.dofactory.com/net/flyweight-design-pattern) | A fine-grained instance used for efficient sharing |
| [Proxy](https://www.dofactory.com/net/proxy-design-pattern) | An object representing another object |

|  |  |
| --- | --- |
| **Behavioral Patterns** | |
| [Chain of Resp.](https://www.dofactory.com/net/chain-of-responsibility-design-pattern) | A way of passing a request between a chain of objects |
| [Command](https://www.dofactory.com/net/command-design-pattern) | Encapsulate a command request as an object |
| [Interpreter](https://www.dofactory.com/net/interpreter-design-pattern) | A way to include language elements in a program |
| [Iterator](https://www.dofactory.com/net/iterator-design-pattern) | Sequentially access the elements of a collection |
| [Mediator](https://www.dofactory.com/net/mediator-design-pattern) | Defines simplified communication between classes |
| [Memento](https://www.dofactory.com/net/memento-design-pattern) | Capture and restore an object's internal state |
| [Observer](https://www.dofactory.com/net/observer-design-pattern) | A way of notifying change to a number of classes |
| [State](https://www.dofactory.com/net/state-design-pattern) | Alter an object's behavior when its state changes |
| [Strategy](https://www.dofactory.com/net/strategy-design-pattern) | Encapsulates an algorithm inside a class |
| [Template Method](https://www.dofactory.com/net/template-method-design-pattern) | Defer the exact steps of an algorithm to a subclass |
| [Visitor](https://www.dofactory.com/net/visitor-design-pattern) | Defines a new operation to a class without change |