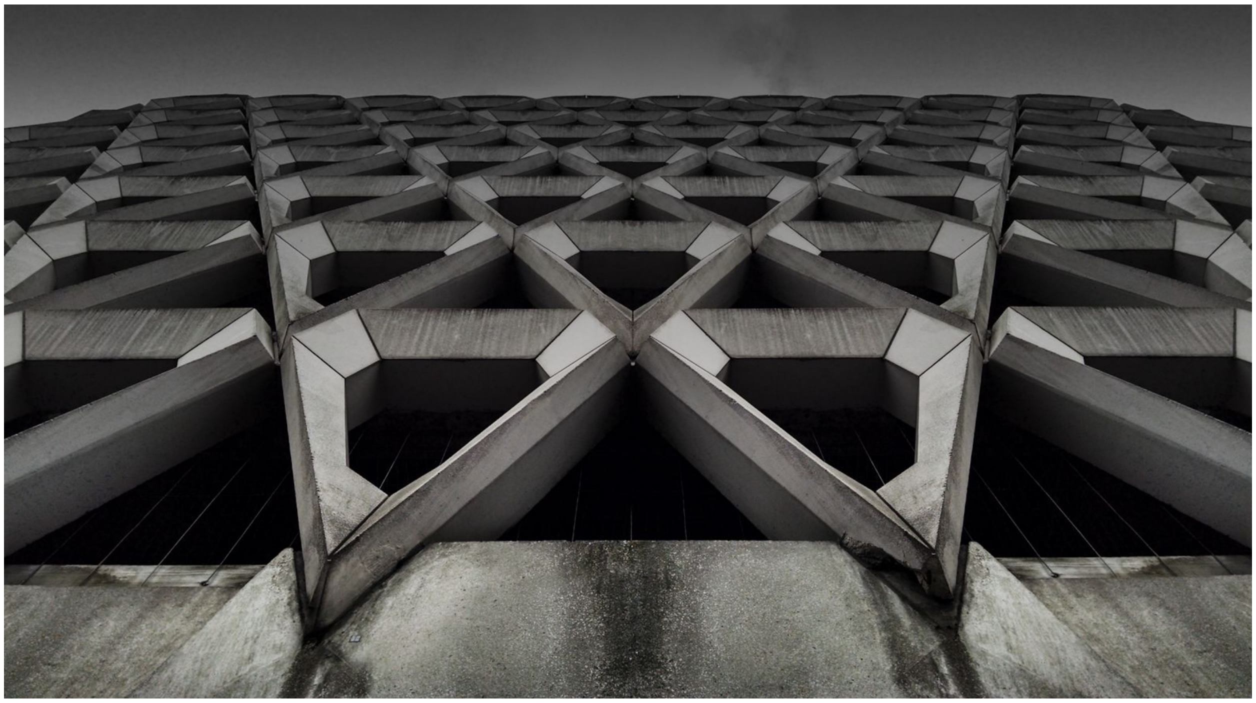
SOLID PRINCIPLES

By: David Gulde, Dubs Lasley, Haley Smith, and Matt Webb



05.02.2020

4390 Software System Development

INTRODUCTION

SOLID is a coding standard used to help developers to avoid a bad design. SOLID is used to help make code more extensible, logical, and easier to read.

SRP: The Single Responsibility Principle

The single responsibility principle applies to classes and methods. The responsibility it is referring to is that of the actor. Only certain methods and classes should affect one actor in the program at a time. For example, if one method affected multiple actors at once but the only one knew about the change then there could be some complications on the way the unknowing actors are using the program after the change. Merges may also become a problem if more than one developer is working on the same class that is used by multiple departments when they try to deploy their separate changes to the same class, there is some risk.

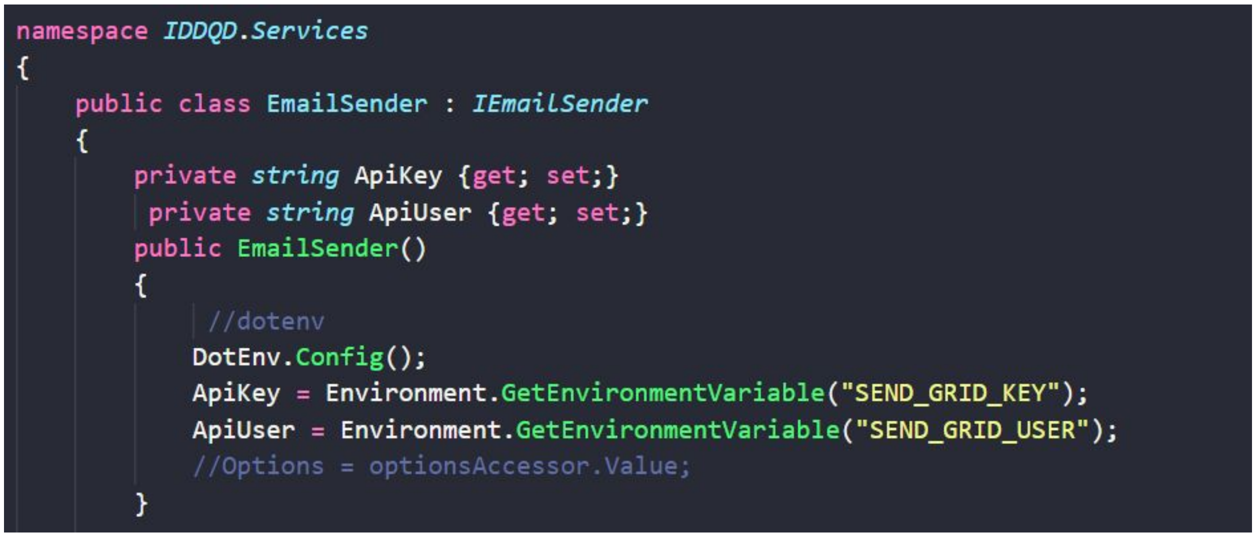
“A class should have one and only one reason to change.” [1]

Martin, Robert C. Clean Architecture: a Craftsman's Guide to Software Structure and Design. Prentice Hall, 2018.

This principle is all about cohesion, which is a measure of how closely related two things are, and consequently whether they belong together. I've always liked the way Uncle Bob described it: "A class should have one, and only one, reason to change." Achieving this principle is difficult, but the benefits are huge, because when you need to make a change to a behavior, you can always identify just the right place to make it. Also, because each class does only one thing, you can feel confident that you won't accidentally change some other behavior.

SRP Code Example:

The SendGrid emailing service we used in our project would apply to this principle due to the fact that it’s only responsibility in our application is to send emails.



```
namespace IDDQD.Services
{
    public class EmailSender : IEmailSender
    {
        private string ApiKey {get; set;}
         private string ApiUser {get; set;}
        public EmailSender()
        {
            //dotenv
            DotEnv.Config();
            ApiKey = Environment.GetEnvironmentVariable("SEND_GRID_KEY");
            ApiUser = Environment.GetEnvironmentVariable("SEND_GRID_USER");
            //Options = optionsAccessor.Value;
        }
```

OCP: The Open-Closed Principle

The open-closed principle refers to the changes that are made to an existing program. When the program is done but needs some modifications or enhancements the main design should not have to be changed. The modifications should be something completely new that can just be added on. This would mean that good practices such as no hard coding and useful variables would need to have been implemented in the original program.

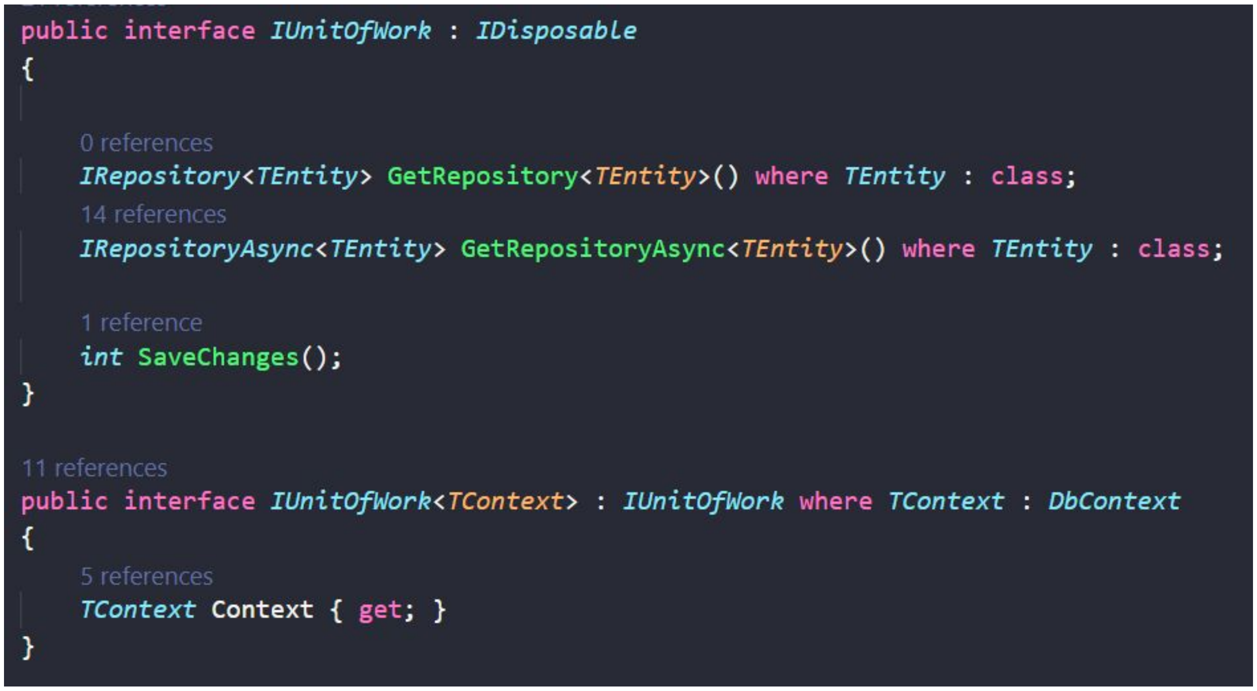
"A software artifact should be open for extension, but closed for modification."[1]

Martin, Robert C. Clean Architecture: a Craftsman's Guide to Software Structure and Design. Prentice Hall, 2018.

This principle was originally described by Bertrand Meyer in his book "Object-Oriented Software Construction" (Prentice Hall PTR, 1997), and it can be confusing. I prefer to describe it in a simplified way: "You should be able to extend or change the behavior of a class without changing the class itself." While you probably don't want to apply a principle like this to every behavior in a class, you should be mindful of reasonable places that consuming code may want to change the behavior. To achieve this, you need to introduce abstractions and leverage polymorphism. The result will make it easier for derived classes to extend the behavior in response to new requirements.

OCP Code Example:

The data file our team created was designed to be able to be used not only inside of our program but could be carried to another project as well. If another developer would like to use this data file all they would have to do is build onto the existing code.



```
public interface IUnitOfWork : IDisposable
{

    0 references
    IRepository<TEntity> GetRepository<TEntity>() where TEntity : class;
    14 references
    IRepositoryAsync<TEntity> GetRepositoryAsync<TEntity>() where TEntity : class;

    1 reference
    int SaveChanges();
}

11 references
public interface IUnitOfWork<TContext> : IUnitOfWork where TContext : DbContext
{
    5 references
    TContext Context { get; }
}
```

LSP: The Liskov Substitution Principle

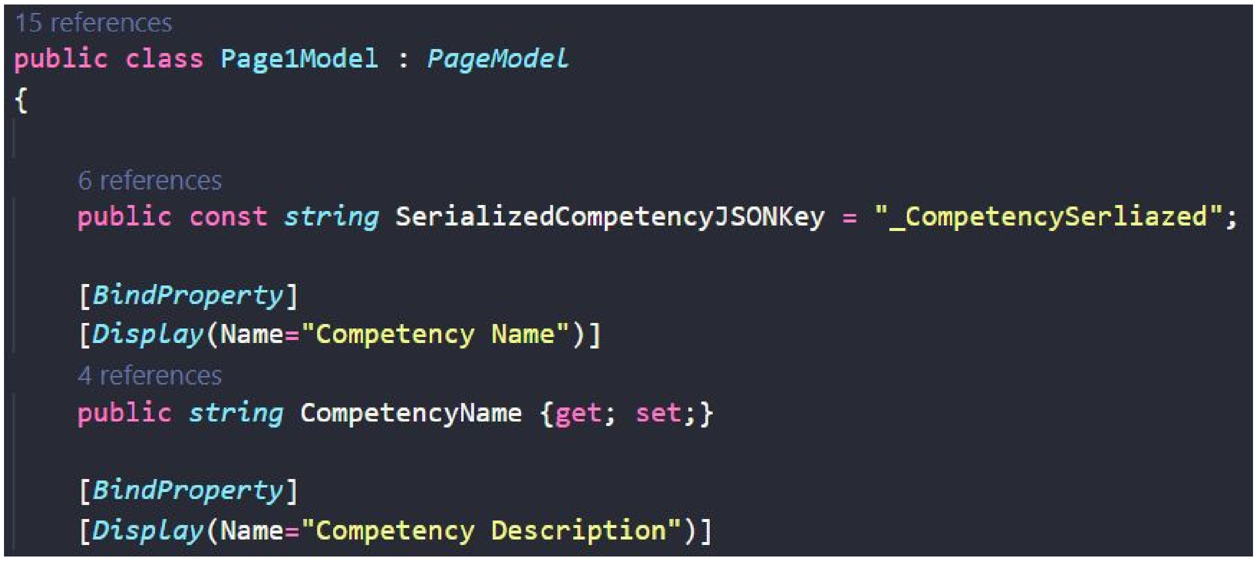
This principle focuses on inheritance. One method should be able to be used for the subtypes of a class. All subtypes should be able to use the methods of the original class if they are made correctly. In the taxi example in the book demonstrated the use of a REST interface. The purpose of the example was to explain that it is not worth making changes to a major part of the program or multiple changes for one element. Choose the simple change to fix the problem.

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then 2 is a subtype of T."[2]

This principle is one I've seen many developers get wrong. In essence, it says that if you create a derived class, you should be able to provide it to anything that consumes the base class, and it should still work. This principle is closely related to the Open/Closed Principle, as in object-oriented languages the primary means of extending a class is by deriving from it. Without this rule, all clients who consume classes in an inheritance hierarchy have to know about every derived class, which is clearly bad from a maintenance perspective. When I talk to developers about this, I often say: "You don't inherit from another class to reuse some code it has. You inherit because it is the other class, but with some changes."

LSP Code Example:

Our wizard builder inherited all of its pages from PageModel in our data folder.



```
15 references
public class Page1Model : PageModel
{

    6 references
    public const string SerializedCompetencyJSONKey = "_CompetencySerliazed";

    [BindProperty]
    [Display(Name="Competency Name")]
    4 references
    public string CompetencyName {get; set;}

    [BindProperty]
    [Display(Name="Competency Description")]
```

ISP: The Interface Segregation Principle

The interface segregation principle addresses the issues of using one system for multiple components. As the name says, different interfaces should be segregated so that the main component does not have to be altered every time a change needs to be done for one component. Having them separated avoids the issue of having to deploy and rebuild other sections of the program when a change is only needed for one piece.

“Depending on something that carries baggage that you don’t need can cause you troubles that you didn’t expect.”[1]

The essence of this principle is twofold: Many fine-grained interfaces are good, and interfaces should be designed for their clients. It's important to note that this isn't only applicable to Microsoft .NET Framework interfaces, but should be thought of more broadly and applied to the public interface of a class. Like the other SOLID principles, this one is related to the others in that it asserts each interface should have a single responsibility and not be overloaded. Also, you should be careful about inheritance for your interfaces: Just because two interfaces happen to share some methods, it doesn't follow that one inherits from another, or even from a common base. Think about it this way: "Clients shouldn't be forced to depend on interfaces they don't use."

ISP Code Example:

Our interface for our application stayed consistent for every page. We did not segregate any layouts for different actors. Our actors were all considered users and did not require a separate interface.

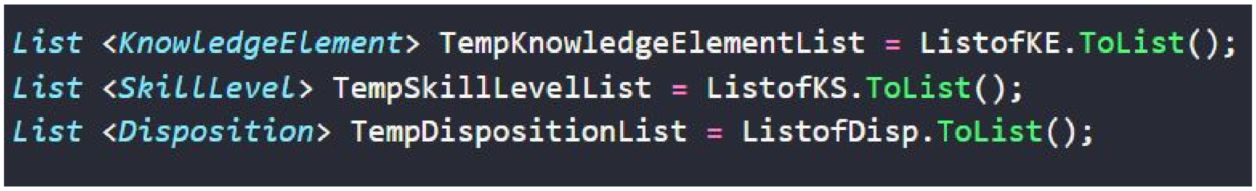
DIP: The Dependency Inversion Principle

This principle explains that statements should not depend on static modules and only abstract declarations such as interfaces and classes. In the book, it refers to the static modules as volatile elements. They are considered volatile because they are constantly undergoing change or are still being developed. It makes sense to not rely on these elements because of frequent changes to they will cause bugs or breaks somewhere else. It would be a pain to have to figure out all the things that rely on an element after it has been changed and go into the program and change each one of the dependent elements. Over time there will be more and more elements built and that would mean more to keep up with and change each time.

“The most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.”[1]
Dependency inversion isn't just the last one on the list, it's also the one that brings them all together for the win. It's very natural to create layers of components where higher layers depend on lower layers. This principle challenges that assumption, and holds that higher-level things should depend on abstractions, and not on the lower-level things themselves. Uncle Bob likes to say, "Depend on abstractions. Do not depend on concretions." Creating malleable, maintainable code requires finding the right abstractions and using them as the code upon which you depend. These abstractions become the "seams" where you can change the code in response to new or changing requirements.

DIP Code example:

If any pages of our wizard needed modifications the change would only need to happen to the desired page. There is no dependency on any particular page in order to make minor changes to any part of the wizard builder.



```
List <KnowledgeElement> TempKnowledgeElementList = ListofKE.ToList();
List <SkillLevel> TempSkillLevelList = ListofKS.ToList();
List <Disposition> TempDispositionList = ListofDisp.ToList();
```

CONCLUSION

Applying all these principles to one program can be challenging but it is worth it after the program is up and running. The main purpose of these principles is to keep errors, redundancy, and reconfiguring of your application to a minimum. They are meant to make the editing and enhancements to your program easier and more manageable. If you were to ignore these principles while building the architecture of your application problems are likely to arise during your development and after.

References

1. Martin, R. C. (2018). *Clean Architecture: a craftsman's guide to software structure and design*. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Prentice Hall.
2. Barbara Liskov, “Data Abstraction and Hierarchy,” SIGPLAN Notices 23, 5 (May 1998).