SOLID

Software Development Is not a Jenga game

Mark Nijhof

SOLID

Software development is not a Jenga game

Mark Nijhof

This book is for sale at http://leanpub.com/solid

This version was published on 2014-04-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Mark Nijhof by spreading the word about this book on Twitter!

The suggested hashtag for this book is #solid.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#solid

Contents

Disclaimer

Two things:

- These are guidelines, not rules, use your own judgement when to follow or break the recommendation.
- All code examples are as simplistic as I could make them to still show the main idea.

SOLID

Does developing software feel like you are playing a game of Jenga? Is making a change to your code very complicated? Are you afraid that making this change in your code will break something unexpected elsewhere in the code? If so then that is a clear sign of bad code, bad code will slow you down, not only months from now, but even days from now.

Have you ever written code that you didn't understand the next day? I have been watching many presentations by Uncle Bob and I really like the part where he asks his audience if they have ever been significantly slowndown by bad code, and after everybody sticks up their hand he asks "So why did you write it?".

To address these issues we have to change the way we write code, we have to think ahead and write code that not only allows change, but we have to write code that expects change. Write it assuming you will change it again tomorrow. And to help the developer write such code there are several different patterns and principles available; SOLID is an acronym for 5 different principles which are very important to the design of your code that greatly improves the ability of your software to cope with change.

Apparently Uncle Bob felt to strong about the order of the principles that he didn't see the SOLID acronym himself, Michael Feathers was the one actually proposing it.

Great design is asking to hear the answer to a riddle and then thinking, "duh! of course"

Quote from Tim Barcz

I feel that this is the best order in which to explain the different SOLID principles, mostly because the principles are so integrated with each other. When explaining SOLID to others I always struggled with the SOLID order. TO me a different order makes more sense, so here we go S DOLI.

Single Responsibility Principle - SRP

THERE SHOULD NEVER BE MORE THAN ONE REASON
FOR A CLASS TO CHANGE

Look at the beautiful knife that can do it all, until one item breaks and you will need to replace the whole knife, one broken part affects the whole module.

Single Responsibility Principle is the first principle in SOLID and states that â€œThere should never be more than one reason for a class to changeâ€�. Something with one purpose, one responsibility is always going to be easier to change, purely because you can much clearer see what it does and what it does not.

Also, because there is only one responsibility there is a much smaller chance that the change in the code will affect other

code, behavior perhaps, not code. And it greatly improves reuse of code.

Finally it makes you code so much easier to test, since you can test one responsibility in isolation of the rest.

Violating SRP

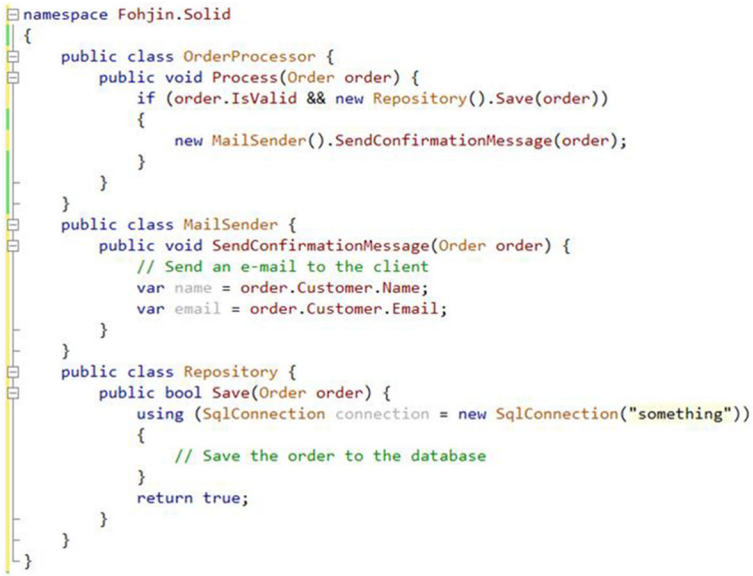
```
namespace Fohjin.Solid
{
    public class OrderProcessor
    {
        public void Process(Order order)
        {
            if (order.IsValid && Save(order))
            {
                SendConfirmationMessage(order);
            }
        }
        private static bool Save(Order order)
        {
            using (SqlConnection connection = new SqlConnection("something"))
            {
                // Save the order to the database
            }
            return true;
        }
        private static void SendConfirmationMessage(Order order)
        {
            // Send an e-mail to the client
            var name = order.Customer.Name;
            var email = order.Customer.Email;
        }
    }
}
```

The different responsibilities

```
namespace Fohjin.Solid
{
    public class OrderProcessor
    {
        public void Process(Order order)
        {
            if (order.IsValid && Save(order))
            {
                SendConfirmationMessage(order);
            }
        }
        private static bool Save(Order order)
        {
            using (SqlConnection connection = new SqlConnection("something"))
            {
                // Save the order to the database
            }
            return true;
        }
        private static void SendConfirmationMessage(Order order)
        {
            // Send an e-mail to the client
            var name = order.Customer.Name;
            var email = order.Customer.Email;
        }
    }
}
```

Look at the different responsibilities in the OrderProcessor class, there is: Composition of the order process, saving the order to the database and sending a confirmation message. These are three different responsibilities. There are potentially three different reasons why this class could change.

Responsibilities nicely separated



```
namespace Fohjin.Solid
{
    public class OrderProcessor {
        public void Process(Order order) {
            if (order.IsValid && new Repository().Save(order))
            {
                new MailSender().SendConfirmationMessage(order);
            }
        }
    }
    public class MailSender {
        public void SendConfirmationMessage(Order order) {
            // Send an e-mail to the client
            var name = order.Customer.Name;
            var email = order.Customer.Email;
        }
    }
    public class Repository {
        public bool Save(Order order) {
            using (SqlConnection connection = new SqlConnection("something"))
            {
                // Save the order to the database
            }
            return true;
        }
    }
}
```

As you can see we have been able to separate the different responsibilities into three different classes. One look upon each class clearly identifies what the class should be doing.

Dependency Inversion Principle - DIP

HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS

ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS

Not as painful as it looks, not even close

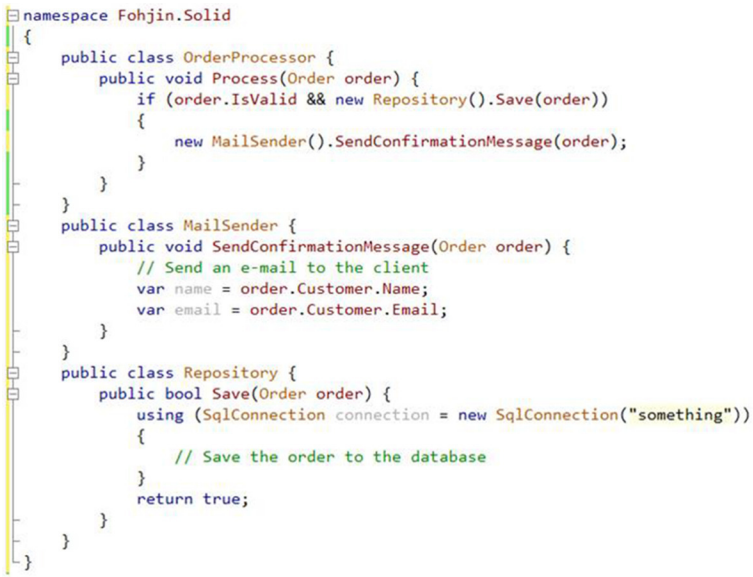
Dependency Inversion Principle is the fifth and last SOLID principle and states that "High level modules should not depend upon low level modules, both should depend upon abstractions" and "Abstractions should not depend upon details, details should depend upon abstractions".

Wow, well basically this means is that nowhere in you code you should depend upon an actual implementation; instead you should only depend upon interfaces or abstract classes. And why you should do this is because this enables you to

just change any actual implementation with another implementation and none of the other code knows or cares about it.

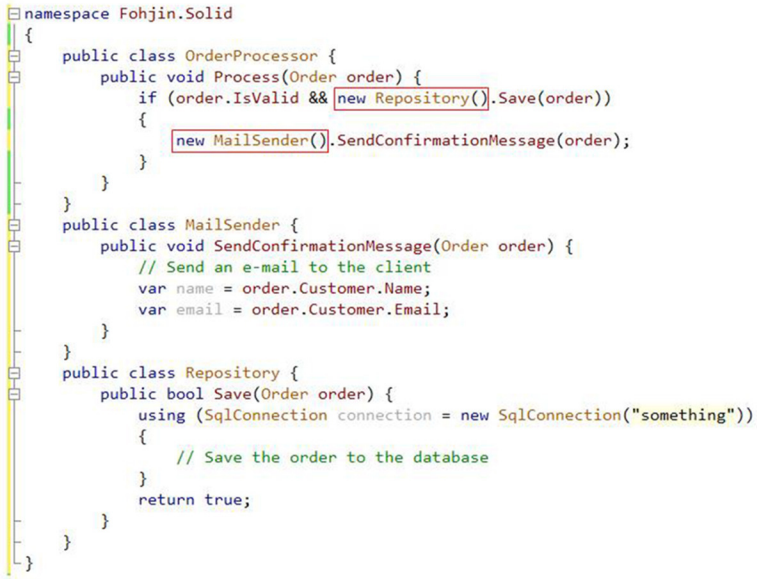
Yes sir, your other code does not care.

Violating DIP



```
namespace Fohjin.Solid
{
    public class OrderProcessor {
        public void Process(Order order) {
            if (order.IsValid && new Repository().Save(order))
            {
                new MailSender().SendConfirmationMessage(order);
            }
        }
    }
    public class MailSender {
        public void SendConfirmationMessage(Order order) {
            // Send an e-mail to the client
            var name = order.Customer.Name;
            var email = order.Customer.Email;
        }
    }
    public class Repository {
        public bool Save(Order order) {
            using (SqlConnection connection = new SqlConnection("something"))
            {
                // Save the order to the database
            }
            return true;
        }
    }
}
```

Depending on specific implementations



```
namespace Fohjin.Solid
{
    public class OrderProcessor {
        public void Process(Order order) {
            if (order.IsValid && new Repository().Save(order))
            {
                new MailSender().SendConfirmationMessage(order);
            }
        }
    }
    public class MailSender {
        public void SendConfirmationMessage(Order order) {
            // Send an e-mail to the client
            var name = order.Customer.Name;
            var email = order.Customer.Email;
        }
    }
    public class Repository {
        public bool Save(Order order) {
            using (SqlConnection connection = new SqlConnection("something"))
            {
                // Save the order to the database
            }
            return true;
        }
    }
}
```

In this example you can see that my OrderProcessor is depending upon actual implementations (details) of the Repository and MailSender. So if I would like to change the way I send notifications to the client I would create another class SmsSender and I would have to change the OrderProcessor as well to be able to use it, which is bad, because I suddenly made the OrderProcessor aware of how it should send messages.

Depending on abstractions

```
namespace Fohjin.Solid
{
    public class OrderProcessor
    {
        private readonly IRepository _repository;
        private readonly IMailSender _mailSender;

        public OrderProcessor(IRepository repository, IMailSender mailSender)
        {
            _repository = repository;
            _mailSender = mailSender;
        }

        public void Process(Order order)
        {
            if (order.IsValid && _repository.Save(order))
            {
                _mailSender.SendConfirmationMessage(order);
            }
        }
    }
}
```

Now the OrderProcessor has no dependencies upon actual implementations anymore, it only relies on interfaces; IRepository and IMailSender. So if I now want to change the way notifications are send I only have to create a new implementation of IMailSender and provide that implementation to the OrderProcessor, OrderProcessor would simply continue to work as it is, its behavior has change, but it doesn't care about that, it only cares about its own specific logic.

Open Closed Principle - OCP

SOFTWARE ENTITIES SHOULD BE OPEN FOR
EXTENSION BUT CLOSED FOR MODIFICATION

This hair style is pretty much closed for modifications, but very easy to extend.

The second SOLID principle is the Open Closed Principle which states that “Software entities should be open for extensions but closed for modifications”.

This basically means that you should be able to change the external behavior or external dependencies of a class without having to physically change the class itself. You would want this kind of behavior because this enables you to make a change in one part of the code without you having to change

other parts of the code as well as long as you can work within the boundaries of the existing contract.

Closed for modification and open for extension

```
namespace Fohjin.Solid
{
    public class OrderProcessor : IOrderProcessor
    {
        private readonly IRepository _repository;
        private readonly IMailSender _mailSender;

        public OrderProcessor(IRepository repository, IMailSender mailSender)
        {
            _repository = repository;
            _mailSender = mailSender;
        }

        public void Process(Order order)
        {
            if (order.IsValid && _repository.Save(order))
            {
                _mailSender.SendConfirmationMessage(order);
            }
        }
    }
}
```

You would recognize this class, it is the result of the Dependency Inversion Principle, and this class is already adhering to the Open Closed Principle, if you want to see a class that doesn't go back to the start of the Dependency Inversion Principle.

So, does that mean that the Open Closed Principle can be achieved with Dependency Inversion? Yes that is correct, but that is not the only way you can make a class extendable.

Using the decorator pattern

```
namespace Fohjin.Solid.SRP
{
    public class DecoratedOrderProcessor : IOrderProcessor
    {
        private readonly IOrderProcessor _orderProcessor;

        public DecoratedOrderProcessor(IOrderProcessor orderProcessor)
        {
            _orderProcessor = orderProcessor;
        }

        public void Process(Order order)
        {
            BeforeProcessing();
            _orderProcessor.Process(order);
            AfterProcessing();
        }
        private static void BeforeProcessing()
        {
            // Do something before Process() was called
        }
        private static void AfterProcessing()
        {
            // Do something after Process() was called
        }
    }
}
```

As you may have seen in the previous slide I added an interface on top of the OrderProcessor, the reason for this is that I can now easily create another implementation with the same contract.

Then I can create a decorator class DecoratedOrderProcessor which adds behavior to my original class without having to modify the code of OrderProcessor. I can even add this behavior during runtime to my class.

Liskov Substitution Principle - LSP

FUNCTIONS THAT USE POINTERS OR REFERENCES
TO BASE CLASSES MUST BE ABLE TO USE OBJECTS
OF DERIVED CLASSES WITHOUT KNOWING IT

The garbage truck doesn't care about the color of the containers, as long as they all open from the top and not from the bottom.

The third SOLID principle is the Liskov Substitution Principle which is described as "Functions that use pointers or references to base classes must be able to use objects or derived classes without knowing it".

That means that when our code uses a specific class or interface it should be able to use a derived class or different implementation of the interface without having to change its internal behavior. This again is to minimize the impact change will have on your code, it feels like déjà vu.

Violating LSP

```
namespace Fohjin.Solid
{
    public class Order {
        private readonly List<Item> _items = new List<Item>();
        public bool IsValid { get { return CheckIsValid(); } }

        private bool CheckIsValid() {
            var isValid = true;
            _items.ForEach(item => {
               if (!item.IsInStock) isValid = false;
            });
            return isValid;
        }
    }
    public class PriorityOrder : Order
    {
        private readonly List<Item> _items = new List<Item>();
        public new bool IsValid { get { return AreItemsInStock(); } }

        private bool AreItemsInStock()
        {
            _items.ForEach(item => {
                if (!item.IsInStock) throw new Exception("No items in stock");
            });
            return true;
        }
    }
}
```

Look at how PriorityOrder is violating the Liskov Substitution Principle because when a client of Order is checking IsValid it expects a Boolean, but when a PriorityOrder is passed to the client suddenly the client needs to try and catch the potential exception that is thrown when the order is not valid. So suddenly the client needs to know with which specific type it is dealing, i.e. Order or PriorityOrder, which is violating the Liskov Substitution Principle.

Another Violation of LSP

```
namespace Fohjin.Solid
{
    public class Rectangle
    {
        public double Width { get; set; }
        public double Height { get; set; }
    }
    public class Square : Rectangle
    {
        public new double Width { get; set; }
        public new double Height
        {
            get { return Width; }
            set { Width = value; }
        }
    }
    public class SomeOtherClassMakingAssumptions
    {
        public void SurfaceTest(Rectangle rectangle)
        {
            rectangle.Width = 4;
            rectangle.Height = 2;
            Assert.That(rectangle.Width * rectangle.Height, Is.EqualTo(8));
        }
    }
}
```

This is an example that Uncle Bob uses himself, and as you can see that the Square class is providing a completely different implementation to setting the Height and thus all clients of Rectangle that assume the surface is calculated by multiplying the Width times the Height are in for a nasty surprise.

Interface Segregation Principle - ISP

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE

The doctor doesn't want to hear your life story, just the parts that will help him help you.

Interface Segregation Principle is the fourth SOLID principle and states that â€œClients should not be forced to depend upon interfaces that they do not useâ€.

Which means that when an object has different usages, not to confuse with different responsibilities, then there should be a specific interface for each of these usages? And I am not going to repeat myself by saying that this is to minimize the impact change will have on your code.

Three different usages

```
namespace Fohjin.Solid
{
    public class Order : IOrder ...

    public interface IShoppingCartOrder {
        void AddItem(Item item);
        void RemoveItem(Item item);
        IEnumerable<Item> GetItems();
    }
    public interface ICheckOutOrder {
        Customer Customer { get; set; }
    }
    public interface IOrder : IShoppingCartOrder, ICheckOutOrder {
        bool IsValid { get; }
        double GetTotalAmount();
    }

    public interface IShoppingCart {
        void UpdateOrder(IShoppingCartOrder order);
    }
    public interface ICheckOut {
        void AttachCustomer(ICheckOutOrder order);
    }
    public interface IOrderProcessor {
        void Process(IOrder order);
    }
}
```

Consider a simple web shop where customers can add and remove items out of their shopping cart and of course inspect the item currently in there. After that they go to check-out and are asked to fill out their personal details, and the final step is a review of the complete order and personal details on which they can submit the order. Now here are three different usages of the order, not every step needs to have or want to have all the information the Order can provide.

Specific interfaces for specific use-cases

```
namespace Fohjin.Solid
{
    public class Order : IOrder ...

    public interface IOrder
    {
        void AddItem(Item item);
        void RemoveItem(Item item);
        IEnumerable<Item> GetItems();
        Customer Customer { get; set; }
        bool IsValid { get; }
        double GetTotalAmount();
    }

    public interface IShoppingCart {
        void UpdateOrder(IOrder order);
    }
    public interface ICheckOut {
        void AttachCustomer(IOrder order);
    }
    public interface IOrderProcessor {
        void Process(IOrder order);
    }
}
```

Now we have created specific interfaces for the specific steps in the order process. The Order object still looks exactly the same as it did in code, but the different clients have a completely different view on it. This enables you to also create different objects one for each step implementing only the needed interface.

Don’t Repeat Yourself - DRY

Donâ€™t Repeat Yourself, I have said it many times before but when you have the same functionality multiple times in your code, then you are risking that when change comes you forget something. And it is not just that code that you have duplicated, you also have duplicate tests; it makes your code fragile.

Donâ€™t Repeat Yourself. Uh just did it again!

You Ain’t Gonne Need It - YAGNI

I’ll add a picture when the requirements tell me too do so

You Ain’t Gonne Need It, it happens so often, you are working on some code and think; hey, this is something they might need later on, I’ll just add it now since I am working on the code anyway.

Then when you get to the point that you have a better understanding of what they actually need you will realize that what you had created is not what is needed.

So instead of saving time by implementing what you thought was needed when you where working on the same code, you actually lost this time because you will have to do it over again.

When you have designed your code properly, adding the functionality is not going to be a problem anyway.

Test Driven Design - TDD

Test Driven Development or Test Driven Design is a technique that really enforces you to do proper design. Writing your tests up front is really helping you think about your design and about what you want to accomplish, so any â€œlostâ€ time in creating these tests is surely won back by the increased quality of your code.

Oh yeah, and of course the fact that your code is now being tested, which will help you deal with change more easily in the future. There really is no good excuse for not doing proper testing.

Testing is done in any other industry, so why not in Software Development?

Clean Code

Always code as if the guy maintaining your code would be a violent psychopath and he knows where you live.

This is so true, and even truer is the fact that you will probably be your own Psychopath.

So, be kind to yourself and start producing quality code.