FREE eBook

LEARNING solid-principles

Free unaffiliated eBook created from
Stack Overflow contributors.

#solid-principles

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: solid-principles

It is an unofficial and free solid-principles ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official solid-principles.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with solid-principles

Remarks

This section provides an overview of what solid-principles is, and why a developer might want to use it.

It should also mention any large subjects within solid-principles, and link out to the related topics. Since the Documentation for solid-principles is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

You can use any IDE and OOP language to implement **S.O.L.I.D Principles**. In the sample code I have used C# as it is the most widely used language in .NET word and is closely resembles Java and C++.

Liskov Substitution Principle

Why to use LSP

Scenario: Suppose we have 3 databases (Mortgage Customers, Current Accounts Customers and Savings Account Customers) that provide customer data and we need customer details for given customer's last name. Now we may get more than 1 customer detail from those 3 databases against given last name.

Implementation

BUSINESS MODEL LAYER:

```
public class Customer
{
    // customer detail properties...
}
```

DATA ACCESS LAYER:

```
public interface IDataAccess
{
    Customer GetDetails(string lastName);
```

```
}
```

Above interface is implemented by the abstract class

```
public abstract class BaseDataAccess : IDataAccess
{
    /// <summary> Enterprise library data block Database object. </summary>
    public Database Database;
    public Customer GetDetails(string lastName)
    {
        // use the database object to call the stored procedure to retirve the customer
detials
    }
}
```

This abstract class has a common method "GetDetails" for all 3 databases which is extended by each of the database classes as shown below

MORTGAGE CUSTOMER DATA ACCESS:

```
public class MortgageCustomerDataAccess : BaseDataAccess
{
    public MortgageCustomerDataAccess(IDatabaseFactory factory)
    {
        this.Database = factory.GetMortgageCustomerDatabase();
    }
}
```

CURRENT ACCOUNT CUSTOMER DATA ACCESS:

```
public class CurrentAccountCustomerDataAccess : BaseDataAccess
{
    public CurrentAccountCustomerDataAccess(IDatabaseFactory factory)
    {
        this.Database = factory.GetCurrentAccountCustomerDatabase();
    }
}
```

SAVINGS ACCOUNT CUSTOMER DATA ACCESS:

```
public class SavingsAccountCustomerDataAccess : BaseDataAccess
{
    public SavingsAccountCustomerDataAccess(IDatabaseFactory factory)
    {
        this.Database = factory.GetSavingsAccountCustomerDatabase();
    }
}
```

Once these 3 data access classes are set, now we draw our attention to the client. In the Business layer we have CustomerServiceManager class that returns the customer detials to its clients.

BUSINESS LAYER:

```
public class CustomerServiceManager : ICustomerServiceManager, BaseServiceManager
{
   public IEnumerable<Customer> GetCustomerDetails(string lastName)
   {
        IEnumerable<IDataAccess> dataAccess = new List<IDataAccess>()
        {
            new MortgageCustomerDataAccess(new DatabaseFactory()),
            new CurrentAccountCustomerDataAccess(new DatabaseFactory()),
            new SavingsAccountCustomerDataAccess(new DatabaseFactory())
        };

        IList<Customer> customers = new List<Customer>();

       foreach (IDataAccess nextDataAccess in dataAccess)
       {
            Customer customerDetail = nextDataAccess.GetDetails(lastName);
            customers.Add(customerDetail);
       }

        return customers;
   }
}
```

I havent shown the Dependency Injection to keep it simple as its already getting complicated now.

Now if we have a new customer detail database we can just add a new class that extends BaseDataAccess and provides its database object.

Of course we need identical stored procedures in all participating databases.

Lastly, the client for CustomerServiceManagerclass will only call GetCustomerDetails method, pass the lastName and should not care about how and where the data is coming from.

Hope this will give you a practical approach to understand LSP.

Read Getting started with solid-principles online: https://riptutorial.com/solid-principles/topic/9700/getting-started-with-solid-principles

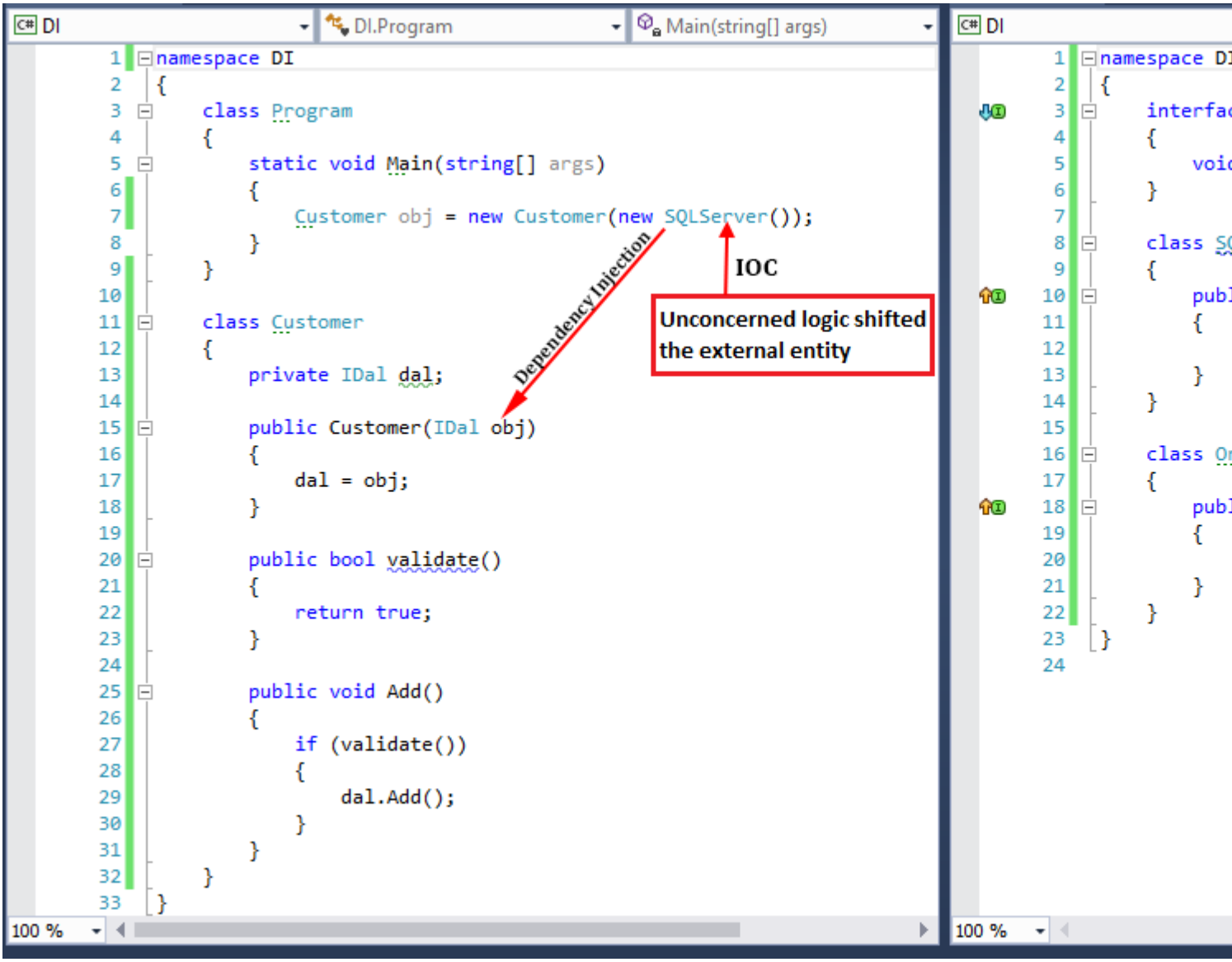
Chapter 2: Dependency Inversion Principle (DIP)

Introduction

The principle basically says, `Class` should depend on abstractions (e.g interface, abstract classes), not specific details (implementations). That means, You should let the caller create the dependencies instead of letting the class itself create the dependencies.

Examples

Dependency Inversion Principle C#



To understand *Dependency Inversion Principle (DIP)* we need to clear concept about *Inversion Of Control(IOC)* and *Dependency Injection(DI)*. So here we discuss all about the terms with

Dependency Inversion Principle (DIP).

Inversion Of Control(IOC) : The control or logic which is not the part of that entity is taken care by someone else. Or, IOC is a programming technique where the unconcerned LOGIC is delegated to some other entity.

Dependency Injection(DI) : Dependency injection is a technique which helps to inject dependent objects of a class that makes architecture more loosely coupled.

There are lots of IOC container for .NET framework that helps us to resolved dependency. Some are listed here.

Read Dependency Inversion Principle (DIP) online: https://riptutorial.com/solid-principles/topic/9730/dependency-inversion-principle--dip-

Chapter 3: Interface Segregation Principle (ISP)

Introduction

The principle states that no client should be forced to depend on methods that it doesn't use. A client should never be forced to implement an interface that it doesn't use or client shouldn't be forced to depend on methods that they don't use.

Examples

Interface Segregation Principle C#

Here we give an example of **ISP** violation and then refactor that violation. Without talking unnecessary things let's jump into the code.

ISP violation :

```
public interface IMessage{
 IList<string> ToAddress {get; set;}
 IList<string> BccAddresses {get; set;}
 string MessageBody {get; set;}
 string Subject {get; set;}
 bool Send();
}

public class SmtpMessage : IMessage{
 public IList<string> ToAddress {get; set;}
 public IList<string> BccAddresses {get; set;}
 public string MessageBody {get; set;}
 public string Subject {get; set;}
 public bool Send(){
  // Code for sending E-mail.
 }
}

public class SmsMessage : IMessage{
 public IList<string> ToAddress {get; set;}
 public IList<string> BccAddresses {
  get { throw new NonImplementedException(); }
  set { throw new NonImplementedException(); }
 }
 public string MessageBody {get; set;}
 public string Subject {
  get { throw new NonImplementedException(); }
  set { throw new NonImplementedException(); }
 }
 public bool Send(){
  // Code for sending SMS.
 }
}
```

In the **SmsMessage** we don't need **BccAddresses and Subject**, but we forced to implement it because of **IMessage** interface . So it's violate the ISP principle.

Remove violation according to *ISP*:

```
public interface IMessage{
 bool Send(IList<string> toAddress, string messageBody);
}

public interface IEmailMessage : IMessage{
 string Subject {get; set;}
 IList<string> BccAddresses {get; set;}
}

public class SmtpMessage : IEmailMessage{
 public IList<string> BccAddresses {get; set;}
 public string Subject {get; set;}
 public bool Send (IList<string> toAddress, string messageBody){
  // Code for sending E-mail.
 }
}

public class SmsMessage : IMessage{
 public bool Send (IList<string> toAddress, string messageBody){
  // Code for sending SMS.
 }
}
```

SmsMessage need only *toAddress* and *messageBody*, so now we can use **IMessage** interface to avoid unnecessary implementations.

Read Interface Segregation Principle (ISP) online: https://riptutorial.com/solid-principles/topic/9731/interface-segregation-principle--isp-

Chapter 4: Open Closed Principle (OCP)

Introduction

Software entities (class, modules, functions etc) should be open for extension but closed for modification.

Examples

Open Closed Principle C#

Here, we try to explain OCP using codebase. First we'll show a scenario that violate OCP and then we'll remove that violation.

Area Calculation (OCP violation Code) :

```
public class Rectangle{
 public double Width {get; set;}
 public double Height {get; set;}
}

public class Circle{
 public double Radious {get; set;}
}

public double getArea (object[] shapes){
 double totalArea = 0;

 foreach(var shape in shapes){
  if(shape is Rectangle){
   Rectangle rectangle = (Rectangle)shape;
   totalArea += rectangle.Width * rectangle.Height;
  }
  else{
   Circle circle = (Circle)shape;
   totalArea += circle.Radious * circle.Radious * Math.PI;
  }
 }
}
```

Now if we need to calculate another another type of object (say, Trapezium) then we've to add another condition. But from the rule's of OCP we know **Software entities should be closed for modification**. So it violates OCP.

Ok. Let's try to solve this violation implementing OCP.

```
public abstract class shape{
 public abstract double Area();
}

public class Rectangle : shape{
```

```
 public double Width {get; set;}
 public double Height {get; set;}

 public override double Area(){
  return Width * Height;
 }
}

public class Circle : shape{
 public double Radious {get; set;}

 public override double Area(){
  return Radious * Radious * Math.PI;
 }
}

public double getArea (shape[] shapes){
 double totalArea = 0;

 foreach(var shape in shapes){
  totalArea += shape.Area();
 }

 return totalArea;
}
```

Now if we need to calculate another type of object, we don't need to change logic (in getArea()), we just need to add another class like *Rectangle or Circle*.

Read Open Closed Principle (OCP) online: https://riptutorial.com/solid-principles/topic/9727/open-closed-principle--ocp-

Chapter 5: Single Responsibility Principle (SRP)

Introduction

There should never be more than one reason for change anything in software entities (class, function, file etc). A class, function, file etc should have only one reason to change.

Remarks

Just because you can, doesn't mean you should.

Examples

Single Responsibility Principle C#

Let's go through the problem first. Have a look on the code below:

```
public class BankAccount
{
    public BankAccount()  {}

    public string AccountNumber { get; set; }
    public decimal AccountBalance { get; set; }

    public decimal CalculateInterest()
    {
        // Code to calculate Interest
    }
}
```

Here, **BankAccount** class contains the properties of account and also *calculate the interest* of account. Now look at the few change Request we received from business:

1. Please add a new Property *AccountHolderName* .
2. Some new rule has been introduced to calculate interest.

This are totally different type of change request. One is changing on features; where as other one is impacting the functionality. We have 2 different types of reason to change one class. This violates Single Responsibility Principle.

Now let's try to implement **SRP** to resolved this violation. Look at the code below:

```
public interface IBankAccount
{
    string AccountNumber { get; set; }
    decimal AccountBalance { get; set; }
```

```
}

public interface IInterstCalculator
{
    decimal CalculateInterest();
}

public class BankAccount : IBankAccount
{
    public string AccountNumber { get; set; }
    public decimal AccountBalance { get; set; }
}

public class InterstCalculator : IInterstCalculator
{
    public decimal CalculateInterest(IBankAccount account)
    {
        // Write your logic here
        return 1000;
    }
}
```

Now our **BankAccount** class is just responsible for properties of the bank account. If we want to add any new *business rule* for the *Calculation of Interest*, we don't need to change **BankAccount** class.

And also **InterestCalculator** class requires no changes, in case we need to add a new Property *AccountHolderName*. So this is the *implementation of Single Responsibility Principle*.

We have also used Interfaces to communicate between *InterestCalculator and BankAccount* class. This will help us to manage dependencies between classes.

Single Responsibility Principle

Introduction

SRP can be defined as “a class handles only one responsibility”. This is a very short definition for something influential on to the other principles of S.O.L.I.D. I believe that if we get this right, it will have a positive knock-on effect on the upcoming principles, so let’s get started! Practical Example Let’s say we have an online store where people can order some items or products and in the code base we have a class OrderProcessor that processes the new orders when people click on Pay Now button.

The reason `OrderProcessor` is written is to carry out the following tasks, in other words `OrderProcessor` class has the following responsibilities:

1. Check the credit card has been accepted – finance
2. Check the money has been charged – finance
3. Check the item is in stock – inventory
4. Request the item for reservation
5. Get the estimated delivery time
6. Email the confirmation to the customer

Here is the definition of `OrderProcessor` class

```
public class OrderProcessor
{
    public bool ProcessOrder(Order orderToProcess)
    {
        // 1)    Check the credit card has been accepted.
        int creditCardId = orderToProcess.CreditCardId;

        CreditCard creditCardDetails = new CreditCard();

        using (SqlConnection connect = new SqlConnection())
        {
            using (SqlCommand command = new SqlCommand())
            {
                command.Connection = connect;
                command.CommandText = "<schema>.<spName>";
                command.CommandType = CommandType.StoredProcedure;
                SqlParameter idParam = new SqlParameter();
                idParam.Direction = ParameterDirection.Input;
                idParam.Value = creditCardId;
                idParam.ParameterName = "@Id";
                idParam.DbType = DbType.Int32;
                command.Parameters.Add(idParam);
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        creditCardDetails.CardId = int.Parse(reader["Id"].ToString());
                        creditCardDetails.CardLongNumber =
reader["CardLongNumber"].ToString();
                        creditCardDetails.CvcNumber =
int.Parse(reader["CvcNumber"].ToString());
                        creditCardDetails.ExpiryDate =
DateTime.Parse(reader["ExpiryDate"].ToString());
                        creditCardDetails.NameOnTheCard = reader["NameOnTheCard"].ToString();
                        creditCardDetails.StartDate =
DateTime.Parse(reader["StartDate"].ToString());
                    }
                }

            }
        }

        // charge the total amount using the credit card details..

        decimal amountToCharge = orderToProcess.OrderTotal;
        using (WebClient webClient = new WebClient())
        {
            string response =
webClient.DownloadString($"https://CreditCardProcessor/api/ProcessesPayments?amount={amountToCharge}&Cr

            // processes response: check if its been successful or failure then proceed
further....
        }



        // check the item is in the stock
```

```
        Dictionary<int, bool> productAvailability = new Dictionary<int, bool>();

        foreach (int productId in orderToProcess.ProductIds)
        {
            using (SqlConnection connection = new SqlConnection())
            {
                using (SqlCommand command = new SqlCommand())
                {
                    command.Connection = connection;
                    command.CommandText = "<schema>.<spName>";
                    command.CommandType = CommandType.StoredProcedure;
                    SqlParameter idParam = new SqlParameter();
                    idParam.Direction = ParameterDirection.Input;
                    idParam.Value = productId;
                    idParam.ParameterName = "@Id";
                    idParam.DbType = DbType.Int32;
                    command.Parameters.Add(idParam);

                    object resultObject = command.ExecuteScalar();
                    bool prductAvailable = bool.Parse(resultObject.ToString());
                    if (prductAvailable)
                        productAvailability.Add(productId, true);
                }
            }
        }

        // request item for reservation

        ReservationServiceClientProxy client = new ReservationServiceClientProxy();

        foreach (KeyValuePair<int, bool> nextProduct in productAvailability)
        {
            ReservationRequest requst = new ReservationRequest() { ProductId = nextProduct.Key
};
            ReservationResponse response = client.ReserveProduct(requst);
        }

        // calculate estimated time of delivery...
        DeliveryService ds = new DeliveryService();
        int totalMinutes = 0;
        foreach (KeyValuePair<int, bool> nextProduct in productAvailability)
        {
            totalMinutes += ds.EstimateDeliveryTimeInMinutes(nextProduct.Key);
        }


        // email customer


        int customerId = orderToProcess.CustomerId;
        string customerEmail = string.Empty;

        using (SqlConnection connection = new SqlConnection())
        {
            using (SqlCommand command = new SqlCommand())
            {
                command.Connection = connection;
                command.CommandText = "<schema>.<spName>";
                command.CommandType = CommandType.StoredProcedure;
                SqlParameter idParam = new SqlParameter();
                idParam.Direction = ParameterDirection.Input;
```

```
                idParam.Value = customerId;
                idParam.ParameterName = "@customerId";
                idParam.DbType = DbType.Int32;
                command.Parameters.Add(idParam);

                object resultObject = command.ExecuteScalar();
                customerEmail = resultObject.ToString();
            }
        }

        MailMessage message = new MailMessage(new
MailAddress("Some.One@SuperCheapStore.co.uk"), new MailAddress(customerEmail));
        message.Body = $"You item has been dispatched and will be delivered in {totalMinutes /
1440} days";
        message.Subject = "Your order update!";

        SmtpClient smtpClient = new SmtpClient("HostName/IPAddress");
        smtpClient.Send(message);

        return true;

    }
}
```

As we can see in the class definition that `OrderProcessor` has taken more than one responsibility. Lets turn our attention to the version 2 of `OrderProcesser` class that is written by keeping SRP in mind.

```
public class OrderProcessorV2
{
    public bool ProcessOrder(Order orderToProcess)
    {
        // 1)    Check the credit card has been accepted.
        CreditCardDataAccess creditCardAccess = new CreditCardDataAccess();
        CreditCard cardDetails =
creditCardAccess.GetCreditCardDetails(orderToProcess.CreditCardId);

        // 2)    Check the money has been charged - finance

        PaymentProcessor paymentProcessor = new PaymentProcessor();
        paymentProcessor.DebitAmount(orderToProcess.OrderTotal, cardDetails);

        // 3)    Check the item is in stock - inventory

        InventoryService inventory = new InventoryService();

        Dictionary<int, bool> productAvailability =
inventory.CheckStockAvailability(orderToProcess.ProductIds);

        foreach (int nextProductId in orderToProcess.ProductIds)
        {
            inventory.CheckStockAvailability(nextProductId);
        }


        // 4)    Request the item for reservation
        ReservationService reservation = new ReservationService();

        foreach (int nextProductId in orderToProcess.ProductIds)
```

```
        {
            reservation.ReserveProduct(nextProductId);
        }


        // 5)    Get the estimated delivery time
        // calculate estimated time of delivery...
        DeliveryService ds = new DeliveryService();
        int totalMinutes = 0;
        foreach (KeyValuePair<int, bool> nextProduct in productAvailability)
        {
            totalMinutes += ds.EstimateDeliveryTimeInMinutes(nextProduct.Key);
        }


        // 6)    Email the confirmation to the customer

        CustomerDataAccess customerDataAccess = new CustomerDataAccess();

        Customer cust = customerDataAccess.GetCustomerDetails(orderToProcess.CustomerId);

        EmailService mailService = new EmailService();
        mailService.NotifyCustomer(cust.Email);

        // if everything step is successful then return true..

    }
}
```

The amount of code lines in `OrderProcessorV2` has radically changed so does the readability. The overall responsibility of `OrderProcessorV2` can be understood within the amount of time that has taken to read this line. This results in more productivity.

Read Single Responsibility Principle (SRP) online: https://riptutorial.com/solid-principles/topic/9726/single-responsibility-principle--srp-

Credits

S. No	Chapters	Contributors
1	Getting started with solid-principles	Community, Yawar Murtaza
2	Dependency Inversion Principle (DIP)	Arif
3	Interface Segregation Principle (ISP)	Arif
4	Open Closed Principle (OCP)	Arif
5	Single Responsibility Principle (SRP)	Arif, Yawar Murtaza