# S.O.L.I.D.

HELLO!

- S.O.L.I.D
- Models for the Rescue
- Domain Driven Design
    - DDD Introduction
    - DDD Strategic Design
    - DDD Tactical Design
- Software Architecture (Parts I, II & III)
- Application Lifecycle Management
- Continuous Integration, Deployment, Delivery
- Generic Domain as a Service
- CQRS
- Event Sourcing
- oAuth 2.0 & OIDC
- Lucene
- Hystrix
- Cassandra

# THE CODE

```
using System.IO;
using System.Xml.Linq;
using Newtonsoft.Json;

namespace SOLID
{
    class Program
    {
        static void Main(string[] args)
        {
            var sourceFileName = "InputFile.xml";
            var targetFileName = "OutputFile.json";

            string input;
            using (var stream = File.OpenRead(sourceFileName))
            using (var reader = new StreamReader(stream))
            {
                input = reader.ReadToEnd();
            }

            var xdoc = XDocument.Parse(input);
            var doc = new Document
            {
                Title = xdoc.Root.Element("title").Value,
                Text = xdoc.Root.Element("text").Value
            };

            var serializedDoc = JsonConvert.SerializeObject(doc);

            using (var stream = File.Open(targetFileName, FileMode.Create, FileAccess.Write))
            using (var sw = new StreamWriter(stream))
            {
                sw.Write(serializedDoc);
                sw.Close();
            }
        }
    }
}
```

```
namespace SOLID
{
    public class Document
    {
        public string Title { get; set; }

        public string Text { get; set; }
    }
}
```

# What was wrong in this picture?

```
using System.IO;
using System.Xml.Linq;
using Newtonsoft.Json;

namespace SOLID.Refactored
{
    class Program
    {
        static void Main(string[] args)
        {
            var sourceFileName = "InputFile.xml";
            var targetFileName = "OutputFile.json";

            var input = GetInput(sourceFileName);
            var doc = GetDocument(input);
            var serializedDoc = SerializeDocument(doc);
            PersistDocument(serializedDoc, targetFileName);
        }

        private static string GetInput(string sourceFileName)...

        private static Document GetDocument(string input)...

        private static string SerializeDocument(Document doc)...

        private static void PersistDocument(string serializedDoc, string targetFileName)...
    }
}
```

Is it better?


# Single Responsibility Principle

"A class should have only one reason to change. “

OR

There can be only one requirement that, when changed, will cause a class to change.

```
using System;

namespace SOLID.SRP
{
    class Program
    {
        static void Main(string[] args)
        {
            var sourceFileName = "InputFile.xml";
            var targetFileName = "OutputFile.json";

            var formatConverter = new FormatConverter();

            if (formatConverter.Convert(sourceFileName, targetFileName) == false)
            {
                Console.WriteLine("Conversion failed.");
                Console.ReadLine();
            }
        }
    }
}
```

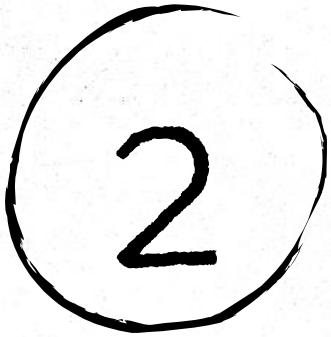
```
using System;
using System.IO;

namespace SOLID.SRP
{
    public class FormatConverter
    {
        private readonly DocumentStorage documentStorage;
        private readonly InputParser inputParser;
        private readonly DocumentSerializer documentSerializer;

        public FormatConverter()...

        public FormatConverter(DocumentStorage documentStorage, InputParser inputParser, DocumentSerializer documentSerializer)
        {
            this.documentStorage = documentStorage;
            this.inputParser = inputParser;
            this.documentSerializer = documentSerializer;
        }

        public bool Convert(string sourceFileName, string targetFileName)
        {
            string input;

            try
            {
                input = documentStorage.GetData(sourceFileName);
            }
            catch (FileNotFoundException)
            {
                return false;
            }

            var doc = inputParser.Parse(input);
            var serializedDoc = documentSerializer.Serialize(doc);

            try
            {
                documentStorage.Persist(serializedDoc, targetFileName);
            }
            catch (AccessViolationException)
            {
                return false;
            }

            return true;
        }
    }
}
```

```
using System.IO;

namespace SOLID.SRP
{
    public class DocumentStorage
    {
        public string GetData(string sourceFileName)
        {
            string input;
            using (var stream = File.OpenRead(sourceFileName))
            using (var reader = new StreamReader(stream))
            {
                input = reader.ReadToEnd();
            }

            return input;
        }

        public void Persist(string serializedDoc, string targetFileName)
        {
            using (var stream = File.Open(targetFileName, FileMode.Create, FileAccess.Write))
            using (var sw = new StreamWriter(stream))
            {
                sw.Write(serializedDoc);
                sw.Close();
            }
        }
    }
}

```

```
using System.Xml.Linq;

namespace SOLID.SRP
{
    public class InputParser
    {
        public Document Parse(string input)
        {
            var xdoc = XDocument.Parse(input);
            var doc = new Document
            {
                Title = xdoc.Root.Element("title").Value,
                Text = xdoc.Root.Element("text").Value
            };

            return doc;
        }
    }
}

```

```
using Newtonsoft.Json;

namespace SOLID.SRP
{
    public class DocumentSerializer
    {
        public string Serialize(Document doc)
        {
            var serializedDoc = JsonConvert.SerializeObject(doc);

            return serializedDoc;
        }
    }
}

```

# Benefits




# Open/Close Principle

"Software entities should be open for extension, but closed for modification. “

OR

Once a class is done, it is DONE!

```
using System;
using System.Configuration;
using System.IO;

namespace SOLID.OCP
{
    public class FormatConverter
    {
        private readonly InputParser inputParser;
        private readonly IDocumentSerializer documentSerializer;

        public FormatConverter()...

        public FormatConverter(InputParser inputParser, IDocumentSerializer documentSerializer)...

        public bool Convert(string sourceFileName, string targetFileName)
        {
            string input;

            var documentStorage = GetDocumentStorageForFileName(sourceFileName);

            try
            {
                input = documentStorage.GetData(sourceFileName);
            }
            catch (FileNotFoundException)
            {
                return false;
            }

            var doc = inputParser.Parse(input);
            var serializedDoc = documentSerializer.Serialize(doc);

            try
            {
                documentStorage.Persist(serializedDoc, targetFileName);
            }
            catch (AccessViolationException)
            {
                return false;
            }

            return true;
        }

        private DocumentStorage GetDocumentStorageForFileName(string fileName)...

        private bool IsBlobStorageUrl(string str)...
    }
}

```

```
namespace SOLID.OCP
{
    public interface IDocumentSerializer
    {
        string Serialize(Document doc);
    }
}

```

```
using Newtonsoft.Json;

namespace SOLID.OCP
{
    public class PascalCaseJsonSerializer : IDocumentSerializer
    {
        public string Serialize(Document doc)
        {
            return JsonConvert.SerializeObject(doc);
        }
    }
}
```

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

namespace SOLID.OCP
{
    public class CamelCaseJsonSerializer : IDocumentSerializer
    {
        public string Serialize(Document doc)
        {
            var settings = new JsonSerializerSettings
            {
                ContractResolver = new CamelCasePropertyNamesContractResolver()
            };

            return JsonConvert.SerializeObject(doc, settings);
        }
    }
}

```

```
namespace SOLID.OCP
{
    public abstract class DocumentStorage
    {
        public abstract string GetData(string sourceFileName);
        public abstract void Persist(string serializedDoc, string targetFileName);
    }
}
```

```
using System.IO;

namespace SOLID.OCP
{
    public class FileDocumentStorage : DocumentStorage
    {
        public override string GetData(string sourceFileName)
        {
            string input;
            using (var stream = File.OpenRead(sourceFileName))
            using (var reader = new StreamReader(stream))
            {
                input = reader.ReadToEnd();
            }

            return input;
        }

        public override void Persist(string serializedDoc, string targetFileName)
        {
            using (var stream = File.Open(targetFileName, FileMode.Create, FileAccess.Write))
            using (var sw = new StreamWriter(stream))
            {
                sw.Write(serializedDoc);
                sw.Close();
            }
        }
    }
}

```

```
using System;
using System.Net;

namespace SOLID.OCP
{
    public class HttpInputRetriever : DocumentStorage
    {
        public override string GetData(string sourceFileName)
        {
            if (sourceFileName.StartsWith("http", StringComparison.Ordinal) == false)
            {
                throw new InvalidOperationException();
            }

            var client = new WebClient();

            var input = client.DownloadString(sourceFileName);

            return input;
        }

        public override void Persist(string serializedDoc, string targetFileName)
        {
            throw new NotImplementedException();
        }
    }
}

```

# Benefits


# Liskovs Substitution Principle

"Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T. “

– Barbara Liskov

OR

A subclass should behave in such a way that it will not cause problems when used instead of the superclass.

– Normal People

1. Contra variance of method arguments in sub class is allowed.
2. Covariance of return types in sub class is not allowed.
3. No new exception types are allowed to be thrown, unless they are sub classes of previously used ones.
4. Preconditions cannot be strengthened in a subtype. Post conditions cannot be weakened in a subtype.
5. The history constraint(not allowed to change mutable to immutable and vice versa.)

```
using System.Drawing;
using NUnit.Framework;

namespace SOLID.LSP
{
    [TestFixture]
    public class CartTests
    {
        [Test]
        public void Make_sure_car_can_start()
        {
            var car = new Car(Color.Red);
            //var car = new BrokenCar(Color.Red);
            //var car = new CrimeBossCar(Color.Black, true);
            //var car = new Prius(Color.Red);
            //var car = new StolenCar(Color.Red);

            try
            {
                car.StartEngine();
            }
            catch (OutOfFuelException)
            {
                Assert.Fail("Car had no gas.");
            }

            Assert.IsTrue(car.IsEngineRunning);
        }

        [Test]
        public void Make_sure_car_is_painted_correctly()
        {
            var car = new Car(Color.Red);
            //var car = new PimpedCar(Color.Red);

            Assert.AreEqual(Color.Red, car.Color);
        }
    }
}

```

```
using System.Drawing;

namespace SOLID.LSP
{
    public class Car
    {
        private bool hasFuel = true;

        public Car(Color color)
        {
            Color = color;
        }

        public Color Color { get; protected set; }

        public bool IsEngineRunning { get; private set; }

        public virtual void StartEngine()
        {
            if (hasFuel)
                IsEngineRunning = true;
            else
                throw new OutOfFuelException();
        }

        public virtual void StopEngine()
        {
            IsEngineRunning = false;
        }
    }
}
```

```
using System;
using System.Drawing;

namespace SOLID.LSP
{
    public class BrokenCar : Car
    {
        public BrokenCar(Color color)
            : base(color)
        {
        }

        public override void StartEngine()
        {
            throw new NotImplementedException();
        }
    }
}
```

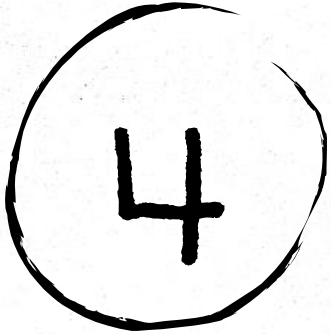
```
using System.Drawing;

namespace SOLID.LSP
{
    public class CrimeBossCar : Car
    {
        private readonly bool boobyTrapped;

        public CrimeBossCar(Color color, bool boobyTrap)
            : base(color)
        {
            this.boobyTrapped = boobyTrap;
        }

        public override void StartEngine()
        {
            if (boobyTrapped)
                throw new MetYourMakerException("Boom! You are dead!");

            base.StartEngine();
        }
    }
}
```

```
using System;
using System.Drawing;

namespace SOLID.LSP
{
    public class Prius : Car
    {
        private int acceleration;

        public Prius(Color color)
            : base(color)
        {
            acceleration = 0;
        }

        /// <summary>
        /// Sets the acceleration of the car depending on how hard the accelerator pedal is pressed.
        /// </summary>
        /// <param name="acceleration">This value should be between 0 and 100</param>
        public void SetAcceleration(int acceleration)
        {
            if (acceleration < 0 || acceleration > 100)
                throw new ArgumentException("Acceleration value should be between 0 and 100 percent.");

            this.acceleration = acceleration;

            if (acceleration >= 50)
                base.StartEngine();
            else
                base.StopEngine();
        }

        public override void StartEngine()
        {

        }

        public override void StopEngine()
        {

        }
    }
}

```

```
1   using System.Drawing;
2
3   namespace SOLID.LSP
4   {
5       public class StolenCar : Car
6       {
7           private bool ignitionWiresStripped;
8
9           public StolenCar(Color color)
10              : base(color)
11          {
12          }
13
14          public void StripIgnitionWires()
15          {
16              ignitionWiresStripped = true;
17          }
18
19          public override void StartEngine()
20          {
21              if (ignitionWiresStripped == false)
22                  return;
23
24              base.StartEngine();
25          }
26      }
27  }
28
```

```
using System.Drawing;

namespace SOLID.LSP
{
    public class PimpedCar : Car
    {
        private int temp;

        public PimpedCar(Color color, int temp = 0)
            : base(color)
        {
            this.temp = temp;

            SetColor(color);
        }

        public void SetTemprature(int temp)
        {
            this.temp = temp;
            SetColor(Color);
        }

        private void SetColor(Color color)
        {
            if (this.temp > 20)
                Color = color;
            else
                Color = Color.Black;
        }
    }
}
```

# Benefits




# Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use.“

```
namespace SOLID.OCP
{
    public abstract class DocumentStorage
    {
        public abstract string GetData(string sourceFileName);
        public abstract void Persist(string serializedDoc, string targetFileName);
    }
}
```

```
1  namespace SOLID.ISP
2  {
3      public interface IDocumentPersister
4      {
5          void Persist(string serializedDoc, string targetFileName);
6      }
7  }
8
```

```
namespace SOLID.ISP
{
    public interface IInputRetriever
    {
        string GetData(string sourceFileName);
    }
}

```

```
namespace SOLID.ISP
{
    public abstract class DocumentStorage : IInputRetriever, IDocumentPersister
    {
        public abstract string GetData(string sourceFileName);
        public abstract void Persist(string serializedDoc, string targetFileName);
    }
}

```

```
using System.IO;

namespace SOLID.ISP
{
    public class FileDocumentStorage : DocumentStorage
    {
        public override string GetData(string sourceFileName)
        {
            string input;
            using (var stream = File.OpenRead(sourceFileName))
            using (var reader = new StreamReader(stream))
            {
                input = reader.ReadToEnd();
            }

            return input;
        }

        public override void Persist(string serializedDoc, string targetFileName)
        {
            using (var stream = File.Open(targetFileName, FileMode.Create, FileAccess.Write))
            using (var sw = new StreamWriter(stream))
            {
                sw.Write(serializedDoc);
                sw.Close();
            }
        }
    }
}

```

```
using System.Configuration;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.StorageClient;

namespace SOLID.ISP
{
    public class BlobDocumentStorage : DocumentStorage
    {
        private readonly CloudBlobClient cloudBlobClient;
        private readonly string container;

        public BlobDocumentStorage()
        {
            this.container = ConfigurationManager.AppSettings["blobStorageContainer"];
            var account = CloudStorageAccount.FromConfigurationSetting(ConfigurationManager.AppSettings["DataConnectionString"]);
            this.cloudBlobClient = account.CreateCloudBlobClient();
        }

        public override string GetData(string sourceFileName)
        {
            var cloudBlobContainer = cloudBlobClient.GetContainerReference(container);
            var blob = cloudBlobContainer.GetBlobReference(sourceFileName);

            return blob.DownloadText();
        }

        public override void Persist(string serializedDoc, string targetFileName)
        {
            var cloudBlobContainer = cloudBlobClient.GetContainerReference(container);
            cloudBlobContainer.CreateIfNotExist();

            var blob = cloudBlobContainer.GetBlobReference(targetFileName);
            blob.UploadText(serializedDoc);
        }
    }
}

```

```
using System;
using System.Net;

namespace SOLID.ISP
{
    public class HttpInputRetriever : IInputRetriever
    {
        public string GetData(string sourceFileName)
        {
            if (sourceFileName.StartsWith("http", StringComparison.Ordinal) == false)
            {
                throw new InvalidOperationException();
            }

            var client = new WebClient();

            var input = client.DownloadString(sourceFileName);

            return input;
        }
    }
}
```

# Benefits


# Dependency Inversion Principle

# Dependency Inversion

$\neq$

# Dependency Injection

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.“

OR

By making sure classes don't depend on specific implementations, it becomes easy to change things around.

```
using System;
using System.IO;

namespace SOLID.DIP
{
    public class FormatConverter
    {
        private readonly IInputParser inputParser;
        private readonly IDocumentSerializer documentSerializer;

        public FormatConverter()...

        public FormatConverter(IInputParser inputParser, IDocumentSerializer documentSerializer)...

        public bool Convert(string sourceFileName, string targetFileName)
        {
            string input;

            var inputRetriever = InputRetriever.ForFileName(sourceFileName);

            try
            {
                input = inputRetriever.GetData(sourceFileName);
            }
            catch (FileNotFoundException)
            {
                return false;
            }

            var doc = inputParser.Parse(input);
            var serializedDoc = documentSerializer.Serialize(doc);

            try
            {
                var documentPersister = DocumentPersister.ForFileName(targetFileName);
                documentPersister.Persist(serializedDoc, targetFileName);
            }
            catch (AccessViolationException)
            {
                return false;
            }

            return true;
        }
    }
}
```

```
using System;
using System.Configuration;

namespace SOLID.DIP
{
    class Program
    {
        static void Main(string[] args)
        {
            var sourceFileName = "InputFile.xml";
            var targetFileName = "OutputFile.json";

            var documentSerializer = new CamelCaseJsonSerializer();
            var inputParser = new XMLInputParser();

            var formatConverter = new FormatConverter(inputParser, documentSerializer);
            if (formatConverter.Convert(sourceFileName, targetFileName) == false)
            {
                Console.WriteLine("Conversion failed.");
                Console.ReadLine();
            }
        }

        private static void ConfigureStorage()
        {
            var blobStorage = new BlobDocumentStorage();
            var fileStorage = new FileDocumentStorage();
            var httpInputRetriever = new HttpInputRetriever();

            InputRetriever.RegisterInputRetriever(x => x.StartsWith("http", StringComparison.Ordinal), httpInputRetriever);
            InputRetriever.RegisterInputRetriever(IsBlobStorageUrl, blobStorage);
            InputRetriever.RegisterInputRetriever(x => true, fileStorage);

            DocumentPersister.RegisterDocumentPersister(IsBlobStorageUrl, blobStorage);
            DocumentPersister.RegisterDocumentPersister(x => true, fileStorage);
        }

        private static bool IsBlobStorageUrl(string str)...
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SOLID.DIP
{
    public static class InputRetriever
    {
        private static readonly Dictionary<Func<string, bool>, IInputRetriever> inputRetrievers = new Dictionary<Func<string, bool>, IInputRetriever>();

        public static void RegisterInputRetriever(Func<string, bool> evaluator, IInputRetriever retriever)
        {
            inputRetrievers.Add(evaluator, retriever);
        }

        public static IInputRetriever ForFileName(string fileName)
        {
            return inputRetrievers.First(x => x.Key(fileName)).Value;
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SOLID.DIP
{
    public static class DocumentPersister
    {
        private static readonly Dictionary<Func<string, bool>, IDocumentPersister> documentPersisters = new Dictionary<Func<string, bool>, IDocumentPersister>();

        public static void RegisterDocumentPersister(Func<string, bool> evaluator, IDocumentPersister persister)
        {
            documentPersisters.Add(evaluator, persister);
        }

        public static IDocumentPersister ForFileName(string fileName)
        {
            return documentPersisters.First(x => x.Key(fileName)).Value;
        }
    }
}

```

# Benefits

# thanks!

## Any questions?

You can find me at
blagovest.vp@gmail.com