Table of Contents

[SOLID 3](#_Toc108481079)

[Single Responsibility Principle (SRP) 3](#_Toc108481080)

[Cast Study 3](#_Toc108481081)

[Violation 4](#_Toc108481082)

[Open Closed Principle (OCP) 5](#_Toc108481083)

[Liskov Substitutional Principle (LSP) 6](#_Toc108481084)

[Interface Segregation Principle (ISP) 7](#_Toc108481085)

[Dependency Inversion Principle 8](#_Toc108481086)

[Design Patterns 10](#_Toc108481087)

[Creational Design Pattern 10](#_Toc108481088)

[Abstract Factory 10](#_Toc108481089)

[Builder Pattern 10](#_Toc108481090)

[Builder 10](#_Toc108481091)

[Director 10](#_Toc108481092)

[Product 10](#_Toc108481093)

[Factory Pattern 13](#_Toc108481094)

[Case Study 13](#_Toc108481095)

[Reference 15](#_Toc108481096)

[Prototype 16](#_Toc108481097)

[Singleton 16](#_Toc108481098)

[Structural Design Pattern 16](#_Toc108481099)

[Adaptor 16](#_Toc108481100)

[Bridge 16](#_Toc108481101)

[Composite 16](#_Toc108481102)

[Decorator 16](#_Toc108481103)

[Façade 16](#_Toc108481104)

[Flyweight 16](#_Toc108481105)

[Proxy 16](#_Toc108481106)

[Behavioral Design Pattern 16](#_Toc108481107)

[Chain of Resp 16](#_Toc108481108)

[Commanda 16](#_Toc108481109)

[Interpreter 16](#_Toc108481110)

[Iterator 16](#_Toc108481111)

[Mediator 16](#_Toc108481112)

[Memento 16](#_Toc108481113)

[Observer 16](#_Toc108481114)

[State 16](#_Toc108481115)

[Strategy 16](#_Toc108481116)

[Template Method 16](#_Toc108481117)

[Visitor 16](#_Toc108481118)

[Reference 17](#_Toc108481119)

# SOLID

S.O.L.I.D is the collection of best practices of the Object Oriented Principles which can be applied to your design to achieve the below desire goals

* Loose coupling
* High Maintainability
* Easy to understand

## Single Responsibility Principle (SRP)

THERE SHOULD NOT BE MORE THEN ONE REASON FOR THE CLASS TO CHANGE.



SRP states that**, “An object should only one reason to change”**. If the object should have more than one reason to change, which means it has more than one responsibility and is in violation of SRP.

Applying SRP Principles will leads to high quality of code and better design as it have the below benefits

* More readable, that is easy to understand
* Less error prone
* More robust
* Better testable
* Better maintainable and extendable

## Cast Study

|  |
| --- |
| public abstract class BankAccount  {  public double Balance { get; set; }  void Deposit(double amount) { }  void Withdraw(double amount) { }  void AddInterest(double amount) { }  } |

Let us consider, we use the above BankAccount class for a Person Saving and Current account.

This will cause the class for more than one reason to change. Because saving account have interest added to it, but for Current Account we don’t have Interest.

To make it SRP, we need to refractor the code.

|  |
| --- |
| public abstract class BankingAccount  {  public double Balance { get; set; }  void Deposit(double admount) { }  void Withdraw(double amount) { }  }  public class CurrentAccount : BankingAccount  {    }  public class SavingAccount : BankingAccount  {  void AddInterest(double amount) { }  } |

In the above code we have use an abstract class to carry the common code and created two concrete classes, so that we can isolate the methods which are causing more than one reason to change.

## Violation

When we actually think about the SRP, there are many classes in .NET Framework which violate the SRP all the times. GetHashCode(), .ToString() are some of the examples. We can even come across the many situations where SRP get violated.

***“SRP is one of the simplest of the principle, and one of the hardest to get it right.”***

## Open Closed Principle (OCP)

## Liskov Substitutional Principle (LSP)

## Interface Segregation Principle (ISP)

## Dependency Inversion Principle

# Design Patterns

# Creational Design Pattern

## Abstract Factory

## Builder Pattern

It is from the family of creational pattern.

Builder pattern helps us to **separate the construction of complex objects from its representation**, so that the same construction process can create the different representations.

Builder process is useful when the construction of the object is very complex.

Way to choose the Builder Pattern

1. Is the process of construction is complex?
2. Do we need to decouple the representation from its construction process?

If the answer yes, then go for Builder Patter otherwise we can achieve using factory pattern or using “new” keyword

### Builder

Builder is responsible for defining the construction process for individual parts. Builder has those individual processes to initialize and configure the product.

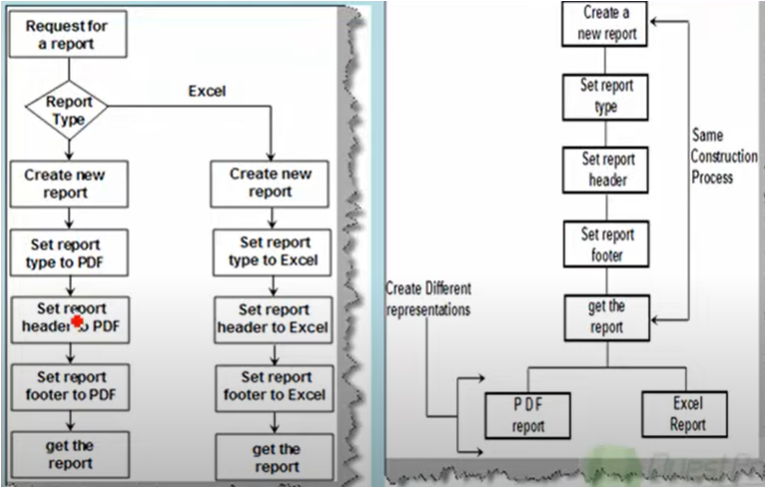
### Director

Director takes those individual processes from the builder and defines the sequence to build the product

### Product

Product is the final product which is produced from the builder and director coordination.

Below process help us to understand the Behavior and creational process.



|  |
| --- |
| public class ReportBuilder  {  public string ReportType;  public void SetHeader(string header)  {  Console.WriteLine("Header set {0}",header);  }  public void SetFooter(string footer)  {  Console.WriteLine("Footer set {0}",footer);  }  public void DisplayReport()  {  Console.WriteLine("Hello from Display Report");  }  }  public abstract class ReportBase  {  public ReportBuilder reportObj;  public abstract void setHeader();  public abstract void setFooter();  public abstract void setReportType();  public void CreateNewReport()  {  reportObj = new ReportBuilder();  }  public ReportBuilder GetReport()  {  return reportObj;  }  }  public class PdfReport : ReportBase  {  public override void setHeader()  {  reportObj.SetHeader("Type PDF Header");  }  public override void setFooter()  {  reportObj.SetFooter("Type PDF Footer");  }  public override void setReportType()  {  reportObj.ReportType = "PDF";  }  }  public class ExcelReport : ReportBase  {  public override void setHeader()  {  reportObj.SetHeader("Type Excel Footer");  }  public override void setFooter()  {  reportObj.SetFooter("Type Excel Footer");  }  public override void setReportType()  {  reportObj.ReportType = "Excel";  }  }  public class Director  {  public ReportBuilder MakeReport(ReportBase reportBase)  {  reportBase.CreateNewReport();  reportBase.setReportType();  reportBase.setHeader();  reportBase.setFooter();  return reportBase.GetReport();  }  } |

|  |
| --- |
| public static void Main(string[] args)  {  ReportBuilder();  }  public static void ReportBuilder()  {  ReportBuilder report = new ReportBuilder();  Director director = new Director();  PdfReport pdf = new PdfReport();  report = director.MakeReport(pdf);  report.DisplayReport();  ExcelReport excel = new ExcelReport();  report = director.MakeReport(excel);  report.DisplayReport();  } |

## Factory Pattern

It is from the family of creational patterns, which means to construct or create something.

## Case Study

|  |
| --- |
| public static void Main(string[] args)  {  GetInvoice\_bad(1);  }  public static void GetInvoice\_bad(int invoiceTypeId)  {  if (invoiceTypeId== 1)  {  Models.InvoiceWithHeader invoiceWithHeader = new();  invoiceWithHeader.Print();  }  else if (invoiceTypeId== 2)  {  Models.InvoiceWithoutHeader invoiceWithoutHeader = new();  invoiceWithoutHeader.Print();  }  } |

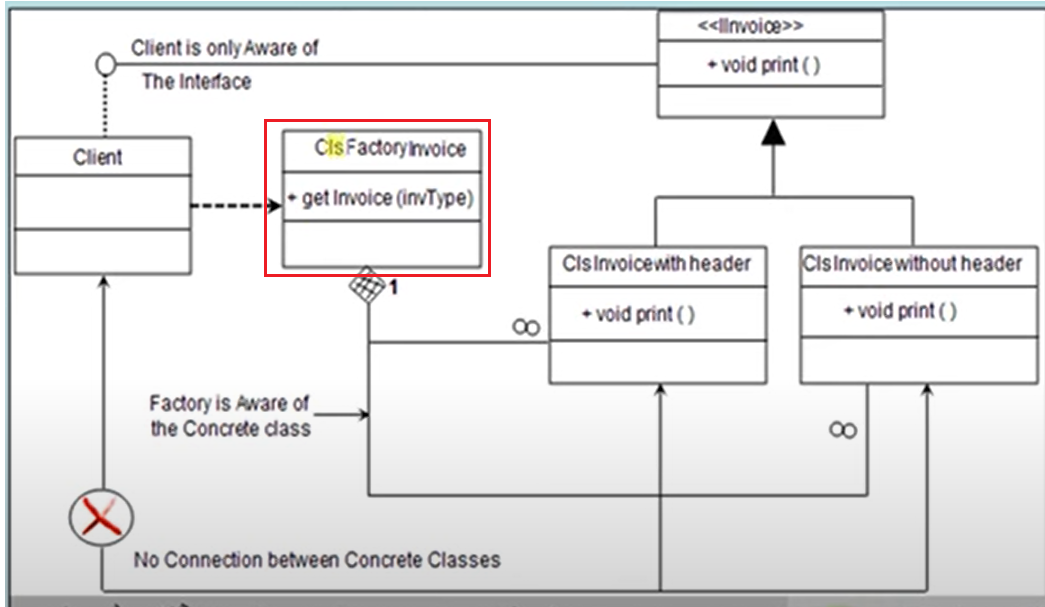
1. From the above code it is clear that we are creating and object based on invoice type id.

Problems:

* “new” keyword: use new keyword on each type of invoice creation. Which may results lot of instance of object. (Lot of scattered new keywords)
* Condition: creating objects based on conditions may result to enhance controller logic every time whenever a new address type added which result in creating a “new” object and code duplication. (Client is aware of all invoice types)

Resolution:

* We can overcome the above problem, which the help of Factory pattern.
* “Factory Class” – Creating a Factory class, and it is responsible for creating a object based on invoice type (which will help to resolve the dependency with new key word)
* Creating an Interface, and concrete class implements that interface, will resolve a problem of adding a new client.



|  |
| --- |
| public interface IInvoice  {  void Print();  }  public class InvoiceWithHeader : IInvoice  {  public void Print()  {  Console.WriteLine("Hello from Invoice with header");  }  }  public class InvoiceWithoutHeader : IInvoice  {  public void Print()  {  Console.WriteLine("Hello from Invoice without header");  }  }  public class InvoiceFactory  {  public static IInvoice GetInvoice(int invoiceType)  {  IInvoice invoiceObj = null;  if (invoiceType == 1)  invoiceObj = new InvoiceWithHeader();  else if (invoiceType == 2)  invoiceObj = new InvoiceWithoutHeader();  return invoiceObj;  }  } |

|  |
| --- |
| public static void GetInvoice\_factory()  {  IInvoice invoice = null;  int invoiceTypeId = 0;  Console.WriteLine("Enter Invoice Type, Supported (1 & 2)");  invoiceTypeId = Convert.ToInt32(Console.ReadLine());  invoice = InvoiceFactory.GetInvoice(invoiceTypeId);  invoice.Print();  }  public static void Main(string[] args)  {  GetInvoice\_factory();  } |

## Reference

<https://github.com/saihari-lgr/Patterns>

## Prototype

## Singleton

# Structural Design Pattern

## Adaptor

## Bridge

## Composite

## Decorator

## Façade

## Flyweight

## Proxy

# Behavioral Design Pattern

## Chain of Resp

## Commanda

## Interpreter

## Iterator

## Mediator

## Memento

## Observer

## State

## Strategy

## Template Method

## Visitor

# Reference

|  |
| --- |
| <https://www.youtube.com/watch?v=agkWYPUcLpg&list=PLqF02iWwxMP5sp42wwR_UZ8aB7MTEqGPh> |
|  |
|  |