Table of Contents

[SOLID Principles 3](#_Toc108388840)

[Single Responsibility Principle (SRP) 3](#_Toc108388841)

[Open Closed Principle (OCP) 3](#_Toc108388842)

[Liskov Substitutional Principle (LSP) 3](#_Toc108388843)

[Interface Segregation Principle (ISP) 3](#_Toc108388844)

[Dependency Inversion Principle 3](#_Toc108388845)

[Design Patterns 3](#_Toc108388846)

[Creational Design Pattern 3](#_Toc108388847)

[Abstract Factory 3](#_Toc108388848)

[Builder Pattern 3](#_Toc108388849)

[Builder 3](#_Toc108388850)

[Directory 4](#_Toc108388851)

[Product 4](#_Toc108388852)

[Factory Pattern 6](#_Toc108388853)

[Case Study 6](#_Toc108388854)

[Reference 8](#_Toc108388855)

[Prototype 9](#_Toc108388856)

[Singleton 9](#_Toc108388857)

[Structural Design Pattern 9](#_Toc108388858)

[Adaptor 9](#_Toc108388859)

[Bridge 9](#_Toc108388860)

[Composite 9](#_Toc108388861)

[Decorator 9](#_Toc108388862)

[Façade 9](#_Toc108388863)

[Flyweight 9](#_Toc108388864)

[Proxy 9](#_Toc108388865)

[Behavioral Design Pattern 9](#_Toc108388866)

[Chain of Resp 9](#_Toc108388867)

[Commanda 9](#_Toc108388868)

[Interpreter 9](#_Toc108388869)

[Iterator 9](#_Toc108388870)

[Mediator 9](#_Toc108388871)

[Memento 9](#_Toc108388872)

[Observer 9](#_Toc108388873)

[State 9](#_Toc108388874)

[Strategy 9](#_Toc108388875)

[Template Method 9](#_Toc108388876)

[Visitor 9](#_Toc108388877)

[Reference 10](#_Toc108388878)

# SOLID Principles

## Single Responsibility Principle (SRP)

## 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.

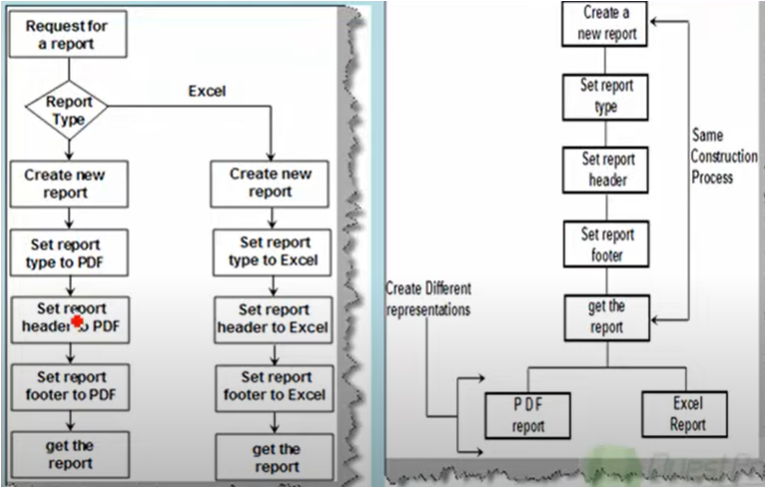
### Directory

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 coordinations.

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();  }  } |

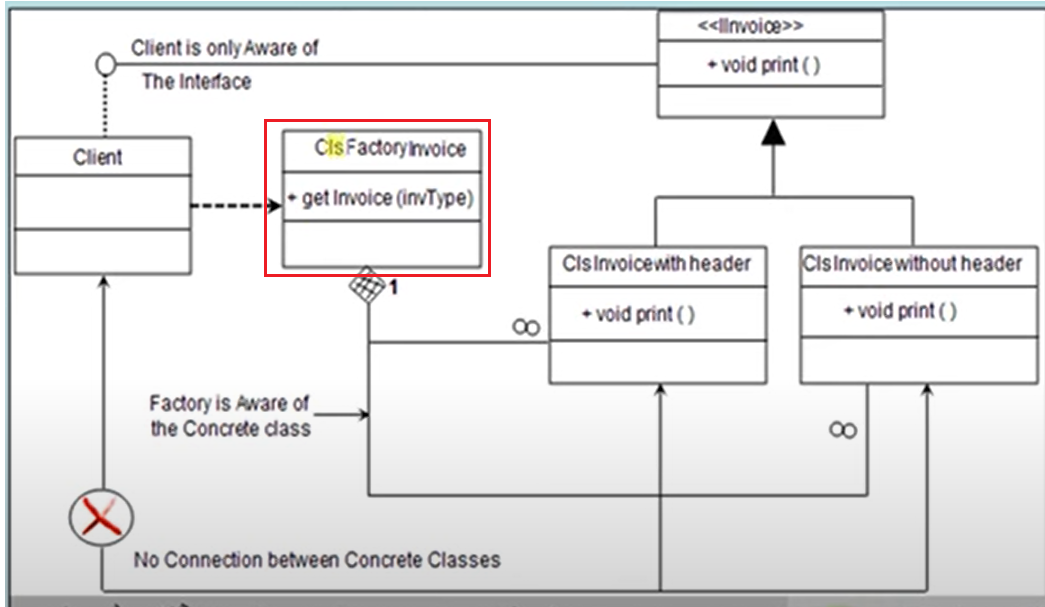
1. From the above code it is clear that we are creating and object based on address type id.

Problems:

* “new” keyword: use new keyword on each type of address 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 address 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> |
|  |
|  |