# GoF Design Patterns – Builder

The purpose of this exercise is to implement a builder design pattern, and to find out what this pattern is useful for.

## UML

# 

## Definition

*Separate the construction of a complex object from its representation so that the same construction process can create different representations.*

## SOLID

Most GoF design patterns follow the SOLID principles, and although not mentioned in the definition, the decorator pattern also helps support the following principles.

1. Single responsibility principle – A builder has the single responsibility of constructing an object of a certain interface
2. Open/Closed principle – A builder can be easily sub-classed or implemented again following the same interface. In the latter case the director can be reused.
3. Liskov’s substitution principle: When the OC principle is applied, the builder can be replaced and all code should still run without any problems
4. Interface segregation principle. The builder should have a short interface by separating the build methods.
5. Dependency inversion principle. By using the interface of the builder, the director can reuse the same logic

## How to implement

To implement a builder, you want to think about the parts of the object you want to build separately. For each of these parts create a *BuildPart* method. To illustrate this we will build a point, with an X and a Y coordinate that needs to be built.

1. Create a new interface *IPointBuilder*
2. Create the methods *BuildX*():*void*, and *BuildY*():*void*. Also add a *GetResult*():*Point* method.
3. Implement this interface with a *PointBuilder*.
4. In the *PointBuilder*, take the required arguments in the constructor, and store these as instance variables
5. Create a *PointDirector* that takes an *IPointBuilder* in the constructor.
6. In the *PointDirector*, create a method *Construct*():*Point*. The construct method will call the builders build method, and returns the *GetResult*();

## Basic example

In this example, we have created the code for the “How to implement section”. The example is obviously very simple, to illustrate how the pattern works. It would obviously be overkill to write a builder for a point – unless the construction is very complex. Do business

class Point

{

public int X { get; set; }

public int Y { get; set; }

}

interface IPointBuilder

{

void BuildX();

void BuildY();

Point GetResult();

}

class PointBuilder : IPointBuilder

{

private int x;

private int y;

private Point result;

// Takes the required arguments to build the data in the constructor

public PointBuilder(int x, int y)

{

this.x = x;

this.y = y;

this.result = new Point();

}

public void BuildX()

{

this.result.X = this.x;

}

public void BuildY()

{

this.result.Y = this.y;

}

public Point GetResult()

{

return this.result;

}

}

// The director directs the build, and can for instance make sure that

// the build methods are called in the correct order (in case there is dependence)

class PointDirector

{

private IPointBuilder builder;

// Takes the interface of the builder into the constructor, DI

public PointDirector(IPointBuilder builder)

{

this.builder = builder;

}

// Directs the build and returns the result

public Point Construct()

{

this.builder.BuildX();

this.builder.BuildY();

var result = this.builder.GetResult();

return result;

}

}

In the example above, please note the following:

* We can now implement endless builders, and reuse the director to actually construct the object.
* The builder only has a single responsibility, and that is to build the various parts of the point. Note that for more complex objects, you could also use a hierarchy for this. Think for instance of an HTML page, that builds the page, which uses a page builder that creates a header/content/footer, which in turn also use builders to construct more details items like divs and spans.
* If there is any dependence – or there is an order in which the build methods need to be called, we could create another director. In this example it is not implemented in an interface, but if you foresee any of these issues, it is smart to interface the director as well.

## Training Exercise Builder 1

## Create a console application that will build a vehicle. A vehicle has an engine, a number of wheels, a steering wheel, and a brake-system.

* The engine by itself has multiple parts as well. It will result in an IEngine, where the parts are represented by strings. IEngine (Valves, Sparkplug, Piston)
* All the parts can be represented by strings. The result will be an IVehicle, with the various parts represented by strings, and an IEngine.
* All tests are required to be implemented, and also stylecop must be up-to-date
* You will use the following builders
  + IVehicleBuilder
    - BuildEngine()
    - BuildWheels()
    - BuildSteering()
    - BuildBrakeSystem()
  + IEngineBuilder
    - BuildValves()
    - BuildSparkplug()
    - BuildPiston()
* Create a VehicleDirector class that can build the entire vehicle
* Use a separate class that can direct the build for the engine.
* You will implement a builder for building a motorcyle and you will implement a builder that creates a car. You can choose yourself which components the vehicles have.
* You will have a class that knows how to display the various parts of the vehicle.
* The Vehicle class will override the ToString() method and calls this display class to output the result.