Builder Pattern

LCSCI5202: Object Oriented Design Week 10

Learning Outcomes

- By the end of this lecture, student would be able to:
  - Understand issues with traditional Object Oriented Design when dealing with different constructors
  - Understand need of Builder Patterns.
  - Implement Builder Patterns with different variations

What was the most complex object you had to create?

Class Pizza

```
public class Pizza
{
private string _size;
private string _crust;
private bool _hasCheese;
private bool _hasPepperoni;
private bool _hasMushrooms;
private bool _hasOlives;
private bool _hasSausage;
private string _sauceType;


public string Size
{
get { return _size; }
set { _size = value; }
}


public string Crust
{
get { return _crust; }
set { _crust = value; }
}

}
```

```
public bool HasCheese
{
get { return _hasCheese; }
set { _hasCheese = value; }
}


public bool HasPepperoni
{
get { return _hasPepperoni; }
set { _hasPepperoni = value; }
}


public bool HasMushrooms
{
get { return _hasMushrooms; }
set { _hasMushrooms = value; }
}


public bool HasOlives
{
get { return _hasOlives; }
set { _hasOlives = value; }
}
```

```
public bool HasSausage
{
get { return _hasSausage; }
set { _hasSausage = value; }
}


public string SauceType
{
get { return _sauceType; }
set { _sauceType = value; }
}
```

Class Pizza

```
public Pizza(string size)
{
        Size = size;
        Crust = "Regular";
        HasCheese = true;
        HasPepperoni = false;
        HasMushrooms = false;
        HasOlives = false;
        HasSausage = false;
        SauceType = "Tomato";

}

public Pizza(string size, string crust, bool hasCheese)
        Size = size;
        Crust = crust;
        HasCheese = hasCheese;

}
```

```
public Pizza(string size, string crust, bool hasCheese, bool hasPepperoni,
bool hasMushrooms, bool hasOlives, bool hasSausage, string sauceType)

{

        Size = size;
        Crust = crust;
        HasCheese =hasCheese;
        HasPepperoni = hasPepperoni;
        HasMushrooms = hasMushrooms;
        HasOlives = hasOlives;
        HasSausage = hasSausage;
        SauceType = sauceType;
}
```

Constructor Overloading !!!

How would YOU solve this problem?

Mutability

```
Pizza myPizza = new Pizza("Large", "Thin", true, true, false,
false, false, "BBQ");

Console.WriteLine($"I ordered: {myPizza.Size} pizza with BBQ
sauce");


Pizza kitchenPizza = customerPizza;
Pizza managerPizza = customerPizza;

kitchenPizza.Size = "Small";
kitchenPizza.SauceType = "Ranch";


Console.WriteLine($"Customer sees: {customerPizza.Size}
{customerPizza.SauceType}");

Console.WriteLine($"Manager sees: {managerPizza.Size}
{managerPizza.SauceType}");
```

Inconsistent State problem

```
public class Pizza
{
public string Size { get; set; }
public int Slices { get; set; }
public decimal Price { get; set; }


public Pizza(string size, int slices, decimal price)

{


            if (size == "Small" && slices > 6)
                        throw new Exception("Small pizzas can't have more
                        than 6 slices!");
            Size = size;
            Slices = slices;
            Price = price;
}

}

Pizza pizza = new Pizza("Small", 4, 12.99m);
pizza.Slices = 12;
```

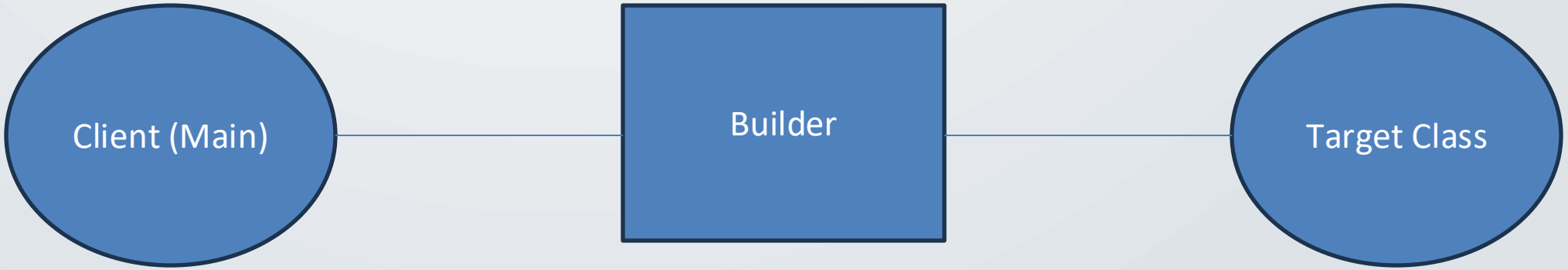
How would YOU solve this problem?

Scattered Validation

```
public class Pizza
{
public string Size { get; set; }
public int Slices { get; set; }
public decimal Price { get; set; }


public Pizza(string size, int slices, decimal price)
{

        if (size == "Small" && slices > 6)
                throw new Exception("Small pizzas can't have more
                than 6 slices!");
        Size = size;
        Slices = slices;
        Price = price;
}
{
public void ValidateOrder(Pizza pizza)
{

        if (pizza.Size == "Small" && pizza.Slices > 6)
                throw new Exception("Small pizzas can't have more
                than 6 slices!");
}
Pizza pizza = new Pizza("Small", 4, 12.99m);
pizza.Slices = 12;
ValidateOrder(Pizza pizza):
```

Solution to all these issues?

Builder Pattern



Builder Pattern

- **The Builder Pattern** separates the construction of a complex object from its representation, allowing step-by-step creation with optional parameters.

- Blocks:
  - Product (the thing you build)
  - Builder (interface for steps)
  - ConcreteBuilder(s) (actual step logic)
  - optional Director (orchestrates step order) (optional)

Basic Work flow of Builder Pattern

- Create the Product Class
  This is the complex object we want to build - our Computer
- Create the Builder Class
  This will have methods to set each property of our Computer
- Add Builder Methods
  Each method sets a property and returns 'this' for chaining
- Add the Build() Method
  - This creates and returns the final Computer object

Computer Builder (Product Class)

```
public class Computer
{
        public string Processor { get; set; }
        public string Memory { get; set; }
        public string Storage { get; set; }
        public string Graphics { get; set; }
        public bool HasWiFi { get; set; }
        public bool HasBluetooth { get; set; }
        public string OperatingSystem { get; set; }
        public double Price { get; set; }


public void DisplaySpecs()
{
        Console.WriteLine($"Computer Specifications:");
        Console.WriteLine($"Processor: {Processor ?? "Not specified"}");
        Console.WriteLine($"Memory: {Memory ?? "Not specified"}");
        Console.WriteLine($"Storage: {Storage ?? "Not specified"}");
        Console.WriteLine($"Graphics: {Graphics ?? "Not specified"}");
        Console.WriteLine($"WiFi: {(HasWiFi ? "Yes" : "No")}");
        Console.WriteLine($"Bluetooth: {(HasBluetooth ? "Yes" : "No")}");
        Console.WriteLine($"OS: {OperatingSystem ?? "Not specified"}");
        Console.WriteLine($"Price: ${Price:F2}");
}
}
```

Computer Builder (Builder Class)

```
public class ComputerBuilder
{
        private Computer _computer;

public ComputerBuilder(){
        _computer = new Computer();
}

public ComputerBuilder SetProcessor(string
processor){
        _computer.Processor = processor;
        return this;
}

public ComputerBuilder SetMemory(string memory){
        _computer.Memory = memory;
        return this;
}

public ComputerBuilder SetStorage(string storage){
        _computer.Storage = storage;
        return this;
}
```

```
public ComputerBuilder SetGraphics(string graphics)
{
_computer.Graphics = graphics;
return this;
}

public ComputerBuilder EnableWiFi()
{
_computer.HasWiFi = true;
return this;
}

public ComputerBuilder EnableBluetooth()
{
_computer.HasBluetooth = true;
return this;
}

public ComputerBuilder SetOperatingSystem(string os)
{
_computer.OperatingSystem = os;
return this;
}
```

```
public ComputerBuilder SetPrice(double price)
{
_computer.Price = price;
return this;
}

public Computer Build()
{
return _computer;
}
}
```

Client Code (Main)

```
class Program
{
static void Main(string[] args)
{
Computer gamingComputer = new
ComputerBuilder()
.SetProcessor("Intel Core i9-
13900K")
.SetMemory("32GB DDR5")
.SetStorage("1TB NVMe SSD")
.SetGraphics("NVIDIA RTX 4080")
.EnableWiFi()
.EnableBluetooth()
.SetOperatingSystem("Windows 11")
.SetPrice(2499.99)
.Build();


gamingComputer.DisplaySpecs();

Computer officeComputer = new
ComputerBuilder()
.SetProcessor("Intel Core i5-12400")
.SetMemory("16GB DDR4")
.SetStorage("500GB SSD")
.EnableWiFi()
.SetOperatingSystem("Windows 11")
.SetPrice(899.99)
.Build();

Console.WriteLine("\n" + new string('-', 40)
+ "\n");
officeComputer.DisplaySpecs();
}
}
```

Builder Pattern Variations

- Simplified Builder
- Classic Builder
  - Director + Abstract Builder + Concrete Builders
- Fluent Builder
- Step Builder

- Choose the Right Variation: Simple builders for most cases, classic for complex scenarios, step builders when order matters.

Director Class (Classic Builder)

```
public static class ComputerDirector
{
public static Computer BuildGamingPC()
{
return new ComputerBuilder()
.SetProcessor("Intel Core i9-13900K")
.SetMemory("32GB DDR5")
.SetStorage("1TB NVMe SSD")
.SetGraphics("NVIDIA RTX 4080")
.EnableWiFi()
.EnableBluetooth()
.SetOperatingSystem("Windows 11")
.SetPrice(2499.99)
.Build();
}
public static Computer BuildOfficePC(){
return new ComputerBuilder()
.SetProcessor("Intel Core i5-12400")
.SetMemory("16GB DDR4")
.SetStorage("500GB SSD")
.EnableWiFi()
.SetOperatingSystem("Windows 11")
.SetPrice(899.99)
.Build();
}
public static Computer BuildBudgetPC(){
return new ComputerBuilder()
.SetProcessor("Intel Core i3-12100")
.SetMemory("8GB DDR4")
.SetStorage("256GB SSD")
.EnableWiFi()
.SetOperatingSystem("Windows 11")
.SetPrice(499.99)
.Build();
}
}
```

Client Code

```
static void Main(string[] args)
{
        Computer gamingComputer = ComputerDirector.BuildGamingPC();
        gamingComputer.DisplaySpecs();


        Console.WriteLine("\n" + new string('-', 40) + "\n");


        Computer officeComputer = ComputerDirector.BuildOfficePC();
        officeComputer.DisplaySpecs();
}
```

Fluent Builder

- Same as Classic Builder and Simplified Builder
- Natural Language Flow

```
QueryBuilder.Select("name",
"email").From("users").Where("age").IsGreaterThan(18)
.OrderBy("name").Build();
```

Step Builder

- It prevents skipping required steps at compile time by narrowing the return type after each step
- Applied usually through Null Exceptions.

```
public Computer Build()
{
        if (string.IsNullOrWhiteSpace(_c.Processor))
                throw new InvalidOperationException("Processor is required.");

        if (string.IsNullOrWhiteSpace(_c.Memory))
                throw new InvalidOperationException("Memory is required.");

        if (string.IsNullOrWhiteSpace(_c.Storage))
                throw new InvalidOperationException("Storage is required.");

        if (string.IsNullOrWhiteSpace(_c.OperatingSystem))
                throw new InvalidOperationException("OS is required.");
return _c;
}
```

Benefits of Builder Pattern

Constructor Overloading (“telescoping constructors”)

- Instead of many overloaded constructors for optional params, Builder exposes small, named steps (e.g., WithX(...)) and a single Build() at the end.
- This removes long/ambiguous constructor lists and the risk of mixing up parameters.

Mutability

- Builder is a natural fit for producing immutable objects: set all data once, validate, then return a finished object that cannot change.

Inconsistent State (half-built/invalid objects leaking out)

- Clients don’t touch the product until Build(); the builder “doesn’t expose the unfinished product while running construction steps.” That prevents fetching a partially configured object.

Scattered Validation

- Put all cross-field checks in one place—Build()—so rules aren’t duplicated across multiple constructors/setters.

Summary

- Builder separates construction from representation, so the same build steps can yield different products. Use it when objects have many options or interdependent fields.
- Eliminates telescoping constructors (dozens of overloads / long parameter lists) and makes creation readable with named steps.
- Fluent chaining: Builder methods return the builder (not the product) to support method chaining; readable, step-by-step configuration before a single Build() call.
- Centralized validation: Put cross-field rules in Build() so invalid combinations are caught once, instead of scattering checks across constructors/setters.
- Prevent inconsistent state: Clients don’t touch the product until Build() returns it, reducing the risk of half-built objects leaking into the system.
- Reuse common recipes: A Director encapsulates repeatable build sequences (e.g., “Gaming PC”, “Office PC”), keeping client code short and consistent.

Activity

Change Class Pizza code from Slide 4 into a builder pattern. Use Classical Builder pattern to implement the code.