**Першин Илья ПИ-15-1**

**Порождающий паттерн – Строитель**

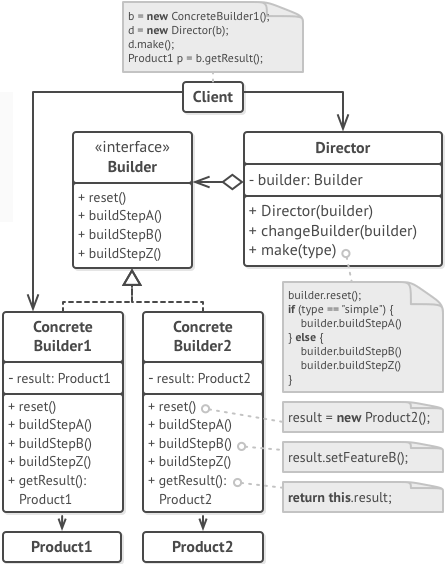
# Определение

Строитель (Builder) – порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

# Назначение и цель

1. избавиться от «телескопического конструктора»
2. когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.
3. когда вам нужно собирать сложные составные объекты, например, деревья Компоновщика.

# UML-диаграмма



Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.  
Builder: определяет интерфейс для создания различных частей объекта Product  
ConcreteBuilder: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему  
Director: распорядитель - создает объект, используя объекты Builder

# Особенности

* Позволяет создавать продукты пошагово.
* Позволяет использовать один и тот же код для создания различных продуктов.
* Изолирует сложный код сборки продукта от его основной бизнес-логики.
* Усложняет код программы за счёт дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

# Пример 1 – хлеб

Хлеб может иметь различную комбинацию компонентов: ржаной и пшеничной муки, соли, пищевых добавок. И нам надо обеспечить выпечку разных сортов хлеба. Для разных сортов хлеба может варьироваться конкретный набор компонентов, не все компоненты могут использоваться. В данном случае с помощью конкретных строителей RyeBreadBuilder и WheatBreadBuilder создаются объекты Bread с определенным набором. В роли распорядителя выступает класс пекаря Baker, который вызывает методы конкретных строителей для построения нового объекта.

Код примера взят с <https://metanit.com/sharp/patterns/2.5.php>

class Program

{

    static void Main(string[] args)

    {

        // содаем объект пекаря

        Baker baker = new Baker();

        // создаем билдер для ржаного хлеба

        BreadBuilder builder = new RyeBreadBuilder();

        // выпекаем

        Bread ryeBread = baker.Bake(builder);

        Console.WriteLine(ryeBread.ToString());

        // оздаем билдер для пшеничного хлеба

        builder = new WheatBreadBuilder();

        Bread wheatBread = baker.Bake(builder);

        Console.WriteLine(wheatBread.ToString());

        Console.Read();

    }

}

// абстрактный класс строителя

abstract class BreadBuilder

{

    public Bread Bread { get; private set; }

    public void CreateBread()

    {

        Bread = new Bread();

    }

    public abstract void SetFlour();

    public abstract void SetSalt();

    public abstract void SetAdditives();

}

// пекарь

class Baker

{

    public Bread Bake(BreadBuilder breadBuilder)

    {

        breadBuilder.CreateBread();

        breadBuilder.SetFlour();

        breadBuilder.SetSalt();

        breadBuilder.SetAdditives();

        return breadBuilder.Bread;

    }

}

// строитель для ржаного хлеба

class RyeBreadBuilder : BreadBuilder

{

    public override void SetFlour()

    {

        this.Bread.Flour = new Flour { Sort = "Ржаная мука 1 сорт" };

    }

    public override void SetSalt()

    {

        this.Bread.Salt = new Salt();

    }

    public override void SetAdditives()

    {

        // не используется

    }

}

// строитель для пшеничного хлеба

class WheatBreadBuilder : BreadBuilder

{

    public override void SetFlour()

    {

        this.Bread.Flour = new Flour { Sort = "Пшеничная мука высший сорт" };

    }

    public override void SetSalt()

    {

        this.Bread.Salt = new Salt();

    }

    public override void SetAdditives()

    {

        this.Bread.Additives = new Additives { Name = "улучшитель хлебопекарный" };

    }

}

//мука

class Flour

{

    // какого сорта мука

    public string Sort { get; set; }

}

// соль

class Salt

{}

// пищевые добавки

class Additives

{

    public string Name { get; set; }

}

class Bread

{

    // мука

    public Flour Flour { get; set; }

    // соль

    public Salt Salt { get; set; }

    // пищевые добавки

    public Additives Additives { get; set; }

    public override string ToString()

    {

        StringBuilder sb = new StringBuilder();

        if (Flour != null)

            sb.Append(Flour.Sort + "\n");

        if (Salt != null)

            sb.Append("Соль \n");

        if (Additives != null)

            sb.Append("Добавки: "+Additives.Name+" \n");

        return sb.ToString();

    }

}

# Пример 2 – автомобиль и руководство

Автомобиль — это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того чтобы настраивать автомобиль через конструктор, мы вынесем его сборку в отдельный класс-строитель, предусмотрев методы для конфигурации всех частей автомобиля. Но к каждому автомобилю нужно ещё и руководство, совпадающее с его конфигурацией. Для этого мы создадим ещё один класс строителя, который вместо конструирования автомобиля, будет печатать страницы руководства к той детали, которую мы встраиваем в продукт.

Код примера взят с <https://refactoring.guru/ru/design-patterns/builder>   
Реализацию можно посмотреть <https://refactoring.guru/ru/design-patterns/builder/java/example>

// Строитель может создавать различные продукты, используя

// один и тот же процесс строительства.

**class** **Car** **is**

Can have GPS, trip computer **and** various numbers of seats.

Can be a city car, a sports car, **or** a cabriolet.

**class** **Manual** **is**

Textual representation of a car.

// Интерфейс строителя объявляет все возможные этапы и шаги

// конфигурации продукта.

**interface** **Builder** **is**

**method** reset()

**method** setSeats(...)

**method** setEngine(...)

**method** setTripComputer(...)

**method** setGPS(...)

// Все конкретные строители реализуют общий

// интерфейс по-своему.

**class** **CarBuilder** **implements** Builder **is**

**private** **field** car:Car

**method** reset()

Put a **new** Car instance into the "car" field.

**method** setSeats(...) **is**

Set the number of seats in car.

**method** setEngine(...) **is**

Install a given engine.

**method** setTripComputer(...) **is**

Install a trip computer.

**method** setGPS(...) **is**

Install a global positioning system.

**method** getResult(): Car **is**

**Return** the current car object.

// В отличие от других создающих паттернов, строители могут

// создавать совершенно разные продукты, не имеющие

// общего интерфейса.

**class** **CarManualBuilder** **implements** Builder **is**

**private** **field** manual:Manual

**method** reset()

Put a **new** Manual instance into the "manual" field.

**method** setSeats(...) **is**

Document car seats features.

**method** setEngine(...) **is**

Add an engine instruction.

**method** setTripComputer(...) **is**

Add a trip computer instruction.

**method** setGPS(...) **is**

Add GPS instruction.

**method** getResult(): Manual **is**

**Return** the current manual object.

// Директор знает в какой последовательности заставлять

// работать строителя. Он работает с ним через общий

// интерфейс строителя. Из-за этого, он может не знать какой

// конкретно продукт сейчас строится.

**class** **Director** **is**

**method** constructSportsCar(builder: Builder) **is**

builder.reset()

builder.setSeats(2)

builder.setEngine(**new** SportEngine())

builder.setTripComputer(**true**)

builder.setGPS(**true**)

// Директор получает объект конкретного строителя от клиента

// (приложения). Приложение само знает какой строитель

// использовать, чтобы получить нужный продукт.

**class** **Application** **is**

**method** makeCar **is**

director = **new** Director()

CarBuilder builder = **new** CarBuilder()

director.constructSportsCar(builder)

Car car = builder.getResult()

CarManualBuilder builder = **new** CarManualBuilder()

director.constructSportsCar(builder)

// Готовый продукт возвращает строитель, так как

// директор чаще всего не знает и не зависит от

// конкретных классов строителей и продуктов.

Manual manual = builder.getResult()