BÁO CÁO CHỦ ĐỀ TÌM HIỂU

SEMINAR: BUILDER PATTERN

Nhóm: 04

Giảng viên hướng dẫn: Thầy Nguyễn Minh Huy
Thầy Hồ Tuấn Thanh
Thầy Trương Toàn Thịnh
Thầy Mai Anh Tuấn

Sinh viên thực hiện: 18127097 - Hà Thế Hiển
18127105 - Đỗ Quốc Huy
18127146 - Khưu Vĩ Luân
18127175 - Dương Tấn Phát

MỤC LỤC

1. TÓM TẮT

Theo định nghĩa của GOF:

> "The intent of the Builder design pattern is to separate the construction of a complex object from its representation so that the same construction process can create different representations"

Builder Pattern là một mẫu thiết kế thuộc nhóm Creational Pattern, được tạo ra để hỗ trợ trong việc khởi tạo ra các đối tượng có cấu trúc phức tạp. Hướng tiếp cận của Builder là tách biệt quá trình xây dựng phức tạp của đối tượng khỏi thể hiện của đối tượng đó. Quá trình xây dựng đối tượng được chia thành từng bước và thực hiện từng phần độc lập nhau, cuối cùng trả về đối tượng kết quả.

2. ĐẶT VẤN ĐỀ, BÀI TOÁN VÀ ĐỘNG LỰC

Hãy xem xét những đối tượng có cấu trúc phức tạp gồm nhiều trường và các trường có thể lồng nhau. Việc xây dựng hàm constructor của các đối tượng kiểu này thường cần truyền vào rất nhiều tham số. Ví dụ ở đây nếu chúng ta cần tạo một đối tượng từ lớp Pizza với nhiều thuộc tính, constructor của nó sẽ có dạng:

```
class Pizza {
    Pizza(int size, boolean cheese, boolean pepperoni, boolean vegetable, boolean seafood, boolean mushroom,
boolean bean, boolean tomato_sauce, boolean ham, ...)
    // ...
```

1. Hàm khởi tạo lớp Pizza với nhiều tham số

Vấn đề đặt ra ở đây là chúng ta phải truyền tất cả các tham số này vào lúc khởi tạo và không phải đối tượng nào cũng cần tất cả các tham số này. Điều này tất nhiên có thể giải quyết bằng cách truyền vào các đối số có giá trị là **null** hoặc **false**, nhưng lúc này việc khởi tạo sẽ trở nên khó khăn trong việc xác định tham số, mã trở nên xấu xí và khó bảo trì. Ví dụ nếu ta chỉ cần một loại Pizza phô mai thông thường có size 25:

```
//...
new Pizza(25, true, true, false, false, false, false, false, ....)
//...
```

2. Khởi tạo một đối tượng từ lớp Pizza với nhiều tham số không cần thiết

Vấn đề này có thể giải quyết với một số ngôn ngữ cho phép nạp chồng phương thức khởi tạo (như C++, C# hoặc Java):

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }

    // ...
```

3. Nạp chồng phương thức khởi tạo lớp Pizza

Cách tiếp cận này có thể giải quyết vấn đề một hàm khởi tạo có quá nhiều tham số như trên, nhưng cũng mang đến một số bất tiện sau:

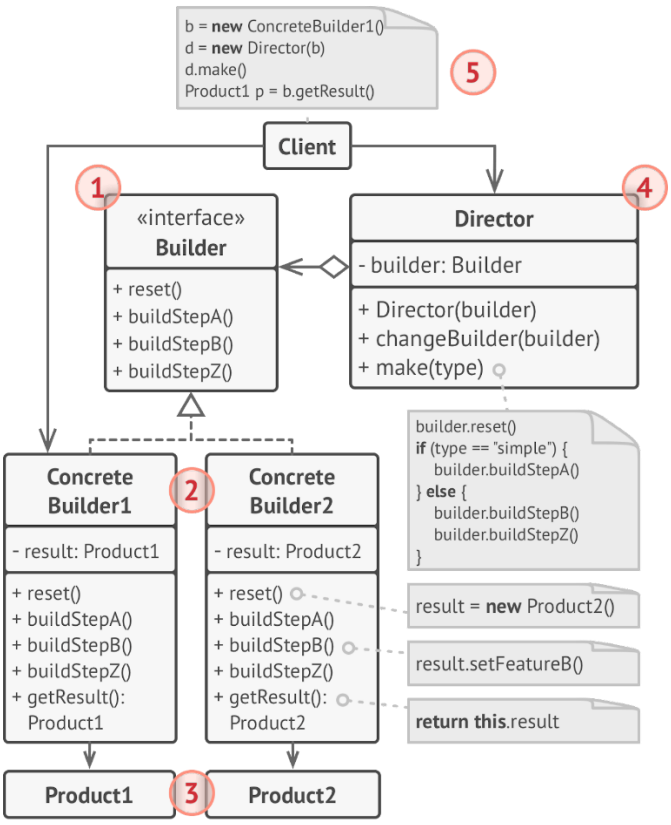
- ❖ Phải tạo nhiều hàm constructor cho các trường hợp tham số khác nhau
- ❖ Khó khăn trong việc xác định các tham số truyền vào

Mẫu Builder được tạo ra nhằm mục đích hỗ trợ trong việc tạo ra các đối tượng phức tạo như thế!

3. LỜI GIẢI

3.1. Sơ đồ lớp

Mẫu Builder gợi ý rằng nên trích xuất mã xây dựng đối tượng ra khỏi lớp của chính đối tượng đó và di chuyển nó đến một đối tượng riêng biệt gọi là Builders.



4. Sơ đồ lớp cài đặt mẫu Builder
Ảnh trích xuất từ tài liệu: [5] Alexander Shvets - Dive Into Design Patterns (2019).

3.2. Giải thích các thành phần của mẫu

Mẫu được cài đặt với các thành phần chính như sau:

3.2.1. Builder (1)

Là một abstract interface, khai báo các bước xây dựng product chung cho tất cả các loại builder.

3.2.2. ConcreteBuilder (2)

Là các lớp được kế thừa từ Builder, chứa các triển khai khác nhau của các bước xây dựng đối tượng. Các ConcreteBuilder có thể tạo ra các product không tuân theo interface chung.

3.2.3. Product (3)

Là các đối tượng kết quả của quá trình xây dựng.

3.2.4. Director (4)

Chứa builder, quyết định sẽ gọi loại builder nào và thứ tự gọi những builder và các bước trong việc tạo final product.
Việc có một lớp director trong chương trình là không hoàn toàn cần thiết. Hoàn toàn có thể gọi các bước thực hiện builder từ client (như một số phần cài đặt mẫu Builder trong thư viện Java được cài đặt mà không cần director).
Thế nhưng việc sử dụng director sẽ giúp ta có thể dễ dàng tái sử dụng mã, giấu các bước xây dựng phức tạp giúp client dễ dàng sử dụng hơn.

3.3. Ngữ cảnh áp dụng

Dùng để tạo một đối tượng phức tạp cấu tạo gồm nhiều phần độc lập với nhau, bằng cách tạo từng phần độc lập và ghép lại để được đối tượng đó. Quá trình tạo cho phép nhiều cách biểu diễn việc tạo đối tượng đó (thứ tự tạo các phần độc lập có thể thay đổi).

3.4. Các dạng Builder

3.4.1. Just Builder (dạng cơ bản)

Để giúp người dùng xây dựng builder pattern một cách dễ dàng và ẩn đi các class không cần thiết

```
var builder = new HtmlDocumentBuilder();
builder.OpenTag("p");
builder.AddText("Text");
builder.CloseTag("p");
```

5. Mẫu Just Builder

3.4.2. Fluent Builder

Xây dựng giống builder bình thường nhưng các hàm sẽ trả về một class hoặc con trỏ dẫn tới một class

```
var document = new
HtmlDocumentBuilder().OpenTag("p").AddText("Text").CloseTag("p").Build();
```

6. Mẫu Fluent Builder

3.4.3. Strict Builder

Hàm build() sẽ được chứa ở một class khác class ban đầu để ngăn việc gọi build() đầu tiên và để biết hàm cần gọi trước khi gọi hàm build(). Thực hiện bằng cách các hàm trả về class hoặc con trỏ dẫn tới một class khác với class chứa hàm đó.

3.4.4. Nested Builder

Lúc tạo object có thể gọi các hàm của các class phía trong của object đó mà class đó vẫn private đối với bên ngoài

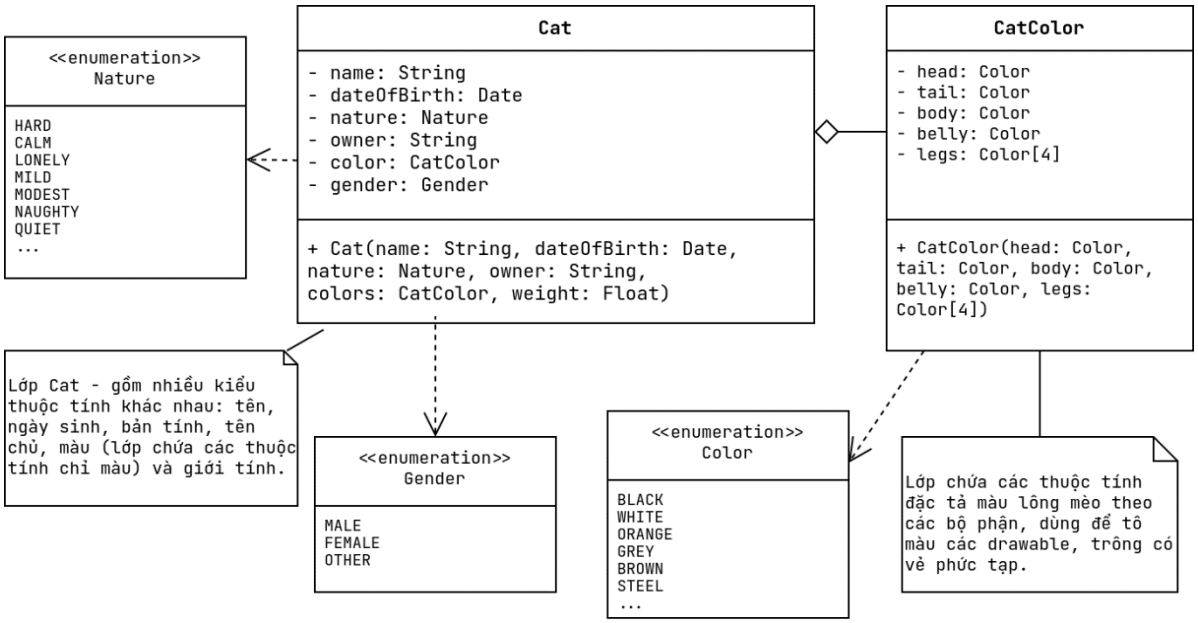
```
var mailMessage = new MailMessage.Builder()
                        .From("from@mail.com")   .To("to@mail.com")
                        .From("from@mail.com")   .To("to@mail.com")
                        .Buil();
```

7. Mẫu Nested Builder

.From .To đang thực hiện việc khởi tạo từng message nhỏ trong tất cả message của toàn bộ MailMessage.

4. CÀI ĐẶT, VÍ DỤ MINH HỌA

Ở phần này, ta sẽ minh họa việc sử dụng mẫu Builder thông qua ví dụ sau đây. Ta sẽ có một lớp Cat và cần tạo ra các đối tượng từ lớp đó. Lớp Cat có sơ đồ lớp như sau:

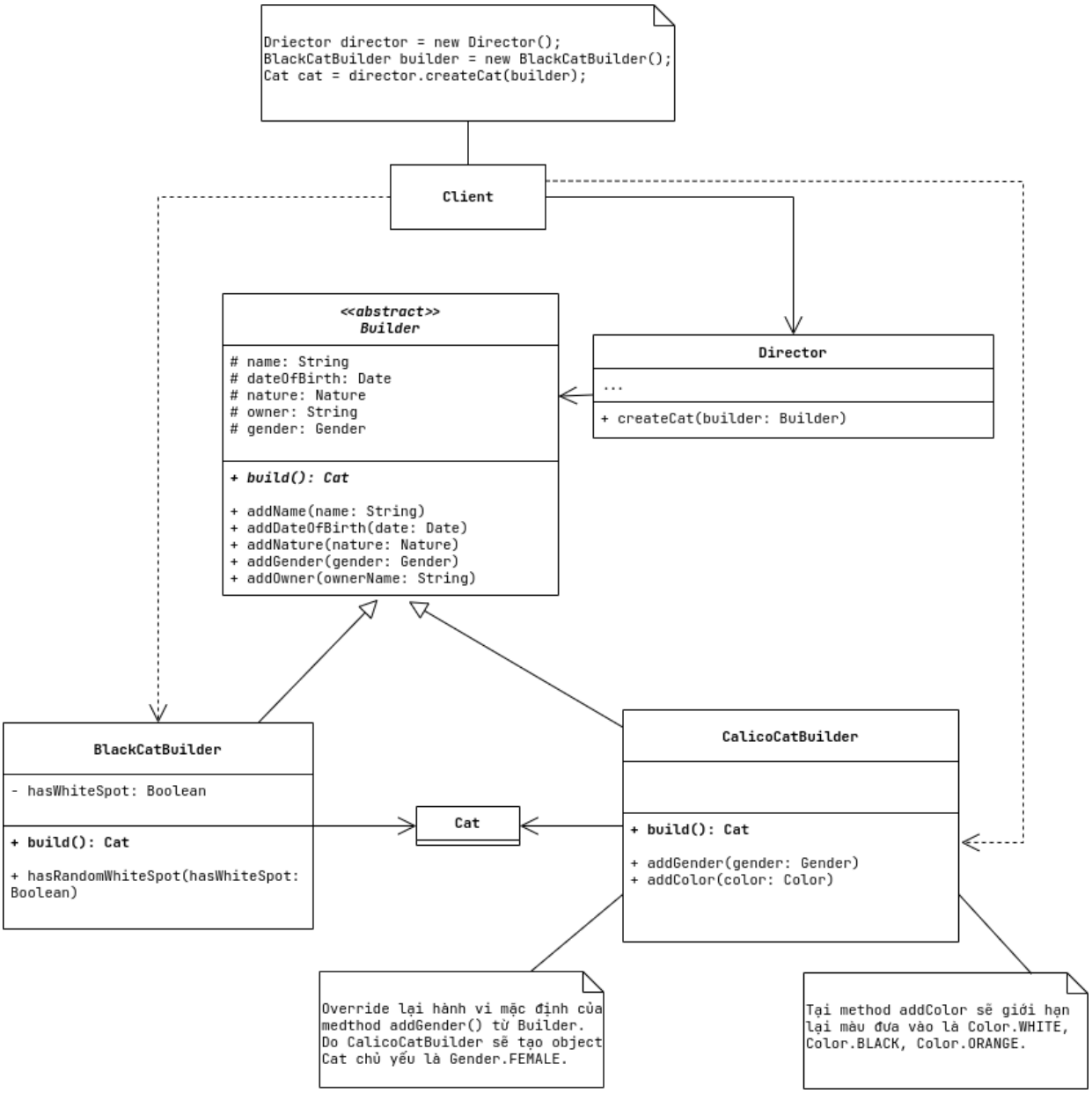


8. Sơ đồ UML lớp Cat

Ta cần tạo đối tượng Cat theo nhiều kiểu khác nhau. Nên ta sẽ tách phần tạo đối tượng Cat ra thành các class Builder. Các class này có các phương thức để xây dựng từng phần của một đối tượng Cat.

Trong ví dụ này sẽ đưa ra 2 builder cho 2 loại đối tượng “Cat” khác nhau: BlackCatBuilder (đối tượng mèo đen), CalicoCatBuilder (đối tượng mèo tam thể).

Cài đặt chi tiết từ ví dụ trên:



9. Sơ đồ UML cài đặt mẫu Builder

```java
import java.util.*;

/**
* Builder class with default implementation of common build methods.
*/


abstract class Builder {
   protected String name;
   protected Date dateOfBirth;
   protected Nature nature;
   protected String owner;
   protected Gender gender = Gender.FEMALE;


    // methods that add variations to the final product
   public void addName(String name) { this.name = name; }
   public void addDateOfBirth(Date date) { this.dateOfBirth = date; }
   public void addNature(Nature nature) { this.nature = nature; }
   public void addOwner(String owner) { this.owner = owner; }
   public void addGender(Gender gender) { this.gender = gender; }
   // Produce Cat object
   public abstract Cat build();
}

/**
* Concrete Builder to produce a Cat object that represents a black cat.
*/
class BlackCatBuilder extends Builder {
   // random white spot in black fur color
   private boolean randomSpot = false;
   private final Random random = new Random();

   /**
   * Determine the final Cat object that has random white spot in its color attribute
   * @param randomSpot true then the Cat object *may* have random white spots.
   */
   public void hasRandomSpot(boolean randomSpot) { this.randomSpot = randomSpot; }


   /**
   * @return the color to be used for constructing the color of the Cat.
   */
   private Color randomWhiteSpot() {
       if (randomSpot) {
           return random.nextInt(100) < 50 ? Color.WHITE : Color.BLACK;
       }
       return Color.BLACK;
   }


   private CatColor produceBlackCatColor() {
       // The cat has a chance of getting random white spot
       // in some body parts when the randomSpot flag == true
       return new CatColor(Color.BLACK, randomWhiteSpot(), Color.BLACK,
               randomWhiteSpot(),
               new Color[]{
                 randomWhiteSpot(), randomWhiteSpot(), randomWhiteSpot(),
       randomWhiteSpot()
               }
       );
   }
   @Override
   public Cat build() {
       return new Cat(name, dateOfBirth, nature, owner, produceBlackCatColor(), gender);
   }
}
```

```java
/**
* Concrete Builder to produce a Cat object that represents a calico cat.
* The produced object will have 3 colors: Color.WHITE, Color.Black, Color.ORANGE,
* and its gender will be mostly Gender.FEMALE.


*/
class CalicoCatBuilder extends Builder {
   // initial body color.
   private Color bodyColor = Color.BLACK;
   // colors other than bodyColor.
   private List<Color> otherColors = Arrays.asList(
           Color.WHITE,
           Color.ORANGE
   );
   private final Random random = new Random();
   private static final List<Color> calicoColors = Arrays.asList(
       Color.BLACK, Color.WHITE, Color.ORANGE
   );
   private static boolean isCalicoColor(Color color) {
       return calicoColors.contains(color);
   }


   // initialize the gender of the produced object.
   public CalicoCatBuilder() { getGender(); }


   // methods to generate a random & valid color for a calico cat object.
   private Color randomCalicoColors() {
       int index = random.nextInt(calicoColors.size() - 1);
       return calicoColors.get(index);
   }
   private Color pickCalicoColorExceptBodyColor() {
       int index = random.nextInt(calicoColors.size() - 1);
       return otherColors.get(index);
   }
   /**
   * Generate the color for the production.
   */
   private CatColor produceCalicoColors() {


       return new CatColor(
               randomCalicoColors(),
               pickCalicoColorExceptBodyColor(),
               bodyColor,
               pickCalicoColorExceptBodyColor(),
               new Color[]{
                       randomCalicoColors(),
                       randomCalicoColors(),
                       randomCalicoColors()
               }
       );
   }
   /**
    * Specify the body color of the calico cat needed to be produced.
    * @param color color to be the body color of the cat.
    *              Invalid color will be ignored.
    */
   public void addBodyColor(Color color) {
       // ignore invalid colors
       if (isCalicoColor(color)) {
           this.bodyColor = color;
           List<Color> otherColors = new ArrayList<>(calicoColors);
           otherColors.remove(color);
           this.otherColors = otherColors;
       }
   }
```

```
    /**
     * Get the gender of the produced object
     * The Cat will have a chance to be Gender.MALE about 1 per 3000 cats.
    */
    private void getGender() {
       this.gender = random.nextInt(3000) == 1 ? Gender.MALE : Gender.FEMALE;
    }
    @Override
    // override the default implementation since calico cats are mostly female
    public void addGender(Gender gender) { getGender();    }
    @Override
    public Cat build() {
        return new Cat(name, dateOfBirth, nature, owner, produceCalicoColors(), gender);
    }
}


// A Cat object that has many attributes.
// It can have many configurations to represent their breed.
class Cat {
    private final String name;
    private final Date dateOfBirth;
    private final Nature nature;
    private final String owner;
    private final CatColor color;
    private final Gender gender;


    public Cat(String name, Date dateOfBirth, Nature nature, String owner, CatColor color,
Gender gender) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
        this.nature = nature;
        this.owner = owner;
        this.color = color;
        this.gender = gender;
    }


    @Override
    public String toString() {
        return "Cat{" + "name='" + name + '\'' + ", dateOfBirth=" + dateOfBirth + ",
nature="
       + nature + ", owner='" + owner + '\'' + ", color=" + color + ", gender=" + gender
'}';
    }
}


class CatColor {
    private final Color head;
    private final Color tail;
    private final Color body;
    private final Color belly;
    private final Color[] legs;
    public CatColor(Color head, Color tail, Color body, Color belly, Color[] legs) {
        this.head = head;
        this.tail = tail;
        this.body = body;
        this.belly = belly;
        this.legs = legs;
    }
    @Override
    public String toString() {
       return "CatColor{" +
               "head=" + head +
               ", tail=" + tail +
                       ", body=" + body +
                 ", belly=" + belly +
                 ", legs=" + Arrays.toString(legs) +
                 '}';
    }
```

```
// Enum constants
enum Color { BLACK, WHITE, ORANGE, GREY, BROWN, STEEL }
enum Gender { MALE, FEMALE, OTHER }
enum Nature { HARDY, CALM, LONELY, MILD, MODEST, NAUGHTY, QUIET }



// Execute the building process in a particular sequence,
// The Director class is optional and it might be omitted in some use-cases.
class Director {
   // Create a cat named Alex from the given builder.
   // In this case, the client will pass a builder object to alter the implementation.
   // So Alex might be a black cat or calico cat.
   public Cat createCat(Builder builder) {
       builder.addName("Alex");
       builder.addDateOfBirth(new Date(1634651257263L));
       builder.addGender(Gender.MALE);
       builder.addNature(Nature.CALM);
       builder.addOwner("Cynthia");
       return builder.build();
   }
}

public class BuilderExample {
   public static void main(String[] args) {
       System.out.println("Demo");


       Director director = new Director();
       Builder builder = new BlackCatBuilder();


       Cat cat = director.createCat(builder);
       System.out.println("Cat object produced from Producer: \n" + cat + "\n----");


       // Create cat object directly from builder
       CalicoCatBuilder calicoBuilder = new CalicoCatBuilder();
       calicoBuilder.addName("Andy");
       calicoBuilder.addOwner("Cynthia");
       System.out.print("Cat object produced from Builder: \n");
       System.out.println(calicoBuilder.build());
   }
}
```

Output mẫu:

```
Demo
Cat object produced from Producer:
Cat{name='Alex', dateOfBirth=Tue Oct 19 20:47:37 ICT 2021, nature=CALM,
owner='Cynthia', color=CatColor{head=BLACK, tail=BLACK, body=BLACK, belly=BLACK,
legs=[BLACK, BLACK, BLACK, BLACK]}, gender=MALE}
----
Cat object produced from Builder:
Cat{name='Andy', dateOfBirth=null, nature=null, owner='Cynthia',
color=CatColor{head=BLACK, tail=WHITE, body=BLACK, belly=WHITE, legs=[WHITE, WHITE,
BLACK]}, gender=FEMALE}
```

5. BÀN LUẬN

5.1. Ưu điểm

- Cho phép thay đổi biểu diễn nội bộ của một lớp. (Ví dụ: Thêm các kiểm tra ràng buộc cho đối tượng khởi tạo trước khi trả về cho người dùng,...)
- Đóng gói code để xây dựng và trình bày. (Dễ đọc, dễ bảo trì khi object được tạo ra với số lượng thuộc tính lớn)
- Cho phép kiểm soát từng bước của quá trình xây dựng đối tượng.
- Giảm thiểu việc tạo nhiều constructor, không cần khởi tạo giá trị null hay false cho các tham số không sử dụng.
- Giảm thiểu việc truyền sai tham số đến các constructor vì người dùng dễ dàng nhận biết khi gọi phương thức tương ứng.

5.2. Hạn chế

- Yêu cầu một ConcreteBuilder cho từng loại đối tượng khác nhau.
- Các thuộc tính của đối tượng không đảm bảo rằng sẽ được khởi tạo.
- Code bị lặp lại nhiều, dài dòng vì cần phải sao chép tất cả thuộc tính từ Product sang Builder.
- Tăng độ phức tạp của code tổng thể do số lượng code tăng lên.

5.3. So sánh với mẫu khác

Builder Pattern	Factory Pattern
Tập trung vào việc tạo các object theo từng bước, truyền vào từng thuộc tính và ghép lại với nhau	Tập trung vào việc phân loại các object theo nhóm và tạo các Creator dành riêng cho từng nhóm
Mỗi builder có thể tạo các object có số lượng, loại thuộc tính khác nhau	Mỗi Creator chỉ có thể tạo ra object thuộc tính tương ứng với Creator đó
Object được trả về sau khi đã hoàn thành các bước	Object được trả về ngay lập tức (Abstract Factory Pattern)

Lưu ý:

- Thông thường, các thiết kế thường bắt đầu bằng Factory Method (độ phức tạp thấp, độ tùy chỉnh cao, nhiều subclass) và dần chuyển thành Abstract

Factory, Prototype hoặc Builder Pattern (độ linh hoạt cao, độ phức tạp cao) tùy theo nhận thức về mức độ cần thiết của độ linh hoạt tại các thời điểm.
- Đôi khi các Creational Pattern có thể bổ trợ cho nhau: Builder Pattern có thể sử dụng một trong số các pattern khác để triển khai các component được xây dựng. Ngoài ra, Abstract Factory, Builder, Prototype Pattern có thể sử dụng Singleton trong quá trình triển khai.

6. TÀI LIỆU KHAM KHẢO

[1] Gamma E., Vlissides J., Johnson R. - Design Patterns - Elements of Reusable Object-Oriented Software, (1997).
[2] Eric Freeman, Elisabeth Robson - Head First Design Patterns - Building Extensible and Maintainable Object-Oriented Software - O'Reilly Media (2020).
[3] Vaskaran Sarcar - Java Design Patterns - A Hands-On Experience with Real-World Examples - Apress (2019).
[4] CSE316_Lecture14a_Creational_Design_Patterns.pdf - CSE316: Fundamentals of Software Development - https://www3.cs.stonybrook.edu/~amione/CSE316_Course/
[5] Alexander Shvets - Dive Into Design Patterns (2019).
[6] 4 Ways to Implement Builder Design Pattern in C# | by Sasha Mathews | Level Up Coding (gitconnected.com) - https://levelup.gitconnected.com/4-ways-to-implement-builder-design-pattern-in-c-dd193e07096c
[7] Builder Pattern - https://refactoring.guru/design-patterns/builder