6.12 Mẫu thiết kế BUILDER

6.12.1 Mẫu thiết kế Builder

Nói chung, chi tiết việc cấu tạo đối tượng, như khởi tạo và khởi động các thành phần mà tạo nên đối tượng, được giữ bên trong đối tượng, như một phần của hàm tạo. Kiểu thiết kế này gắn chặt quá trình tạo đối tượng với các thành phần mà làm nên đối tượng. Cách tiếp cận này là phù hợp khi việc tạo đối tượng là đơn giản và quá trình tạo đối tượng là xác định và luôn sinh ra cùng một đại diện của đối tượng.

Tuy nhiên, thiết kế này có thể không hiệu quả khi đối tượng được tạo là phức tạp và một dãy các bước làm nên quá trình tạo đối tượng có thể được cài đặt theo các cách khác nhau, do đó tạo ra các đại diện khác nhau của đối tượng. Vì các cài đặt khác nhau của quá trình tạo này được giữ bên trong đối tượng, đối tượng có thể trở nên đồ sộ và kém có tính cấu phần. Hệ quả là bổ sung một cài đặt mới hay tạo một số thay đổi cho cài đặt hiện tại yêu cầu thay đổi đến *code* hiện tại.

Sử dụng mẫu *Builder*, quá trình tạo một đối tượng như vậy có thể được thiết kế hiệu quả hơn. Mẫu *Builder* đề xuất chuyển *logic* tạo ra khỏi lớp đối tượng thành một lớp riêng được tham chiếu như *class Builder*. Ở đây có thể có nhiều hơn một lớp *Builder*, mỗi lớp với cài đặt khác nhau cho dãy các bước tạo nên đối tượng. Mỗi cài đặt *Builder* cho kết quả là đại diện khác nhau của đối tượng.

Để sử dụng mẫu *Builder*, thử trợ giúp công ty ôtô mà triển lãm các ôtô khác nhau của nó sử dụng mô hình đồ họa cho khách hàng. Công ty có công cụ đồ họa mà trưng bày ôtô trên màn hình. Yêu cầu của công cụ này là cung cấp đối tượng *car* cho nó. Đối tượng *car* cần chứa đặc tả của *car*. Công cụ đồ họa sẽ sử dụng đặc tả để hiển thị ôtô. Công ty cần phân loại các ôtô của nó thành các loại khác nhau như ôtô *Sedan* hay ôtô *Sports*. Chỉ có một đối tượng *car* và công việc của chúng ta là tạo đối tượng *car* này tuân theo phân lớp đó. Chẳng hạn, đối với ôtô *Sedan*, một đối tượng *car* tuân theo đặc tả phân loại *Sedan* cần được xây dựng hoặc nếu ôtô *Sport* được yêu cầu, thì đối tượng *car* tuân theo đặc tả ôtô *Sport* cần được xây dựng. Hiện tại, Công ty muốn chỉ hai kiểu ôtô này, nhưng nó có thể yêu cầu các kiểu ôtô khác trong tương lai.

Chúng ta sẽ tạo hai *Builder* khác nhau, mỗi một của một phân loại, tức là *Sedan* và *Sport*. Hai *builder* này sẽ giúp xây dựng đối tượng ôtô tuân theo đặc tả của nó. Nhưng trước hết, chúng ta sẽ bàn chi tiết về mẫu *Builder*.

6.12.2 Mẫu Builder là gì

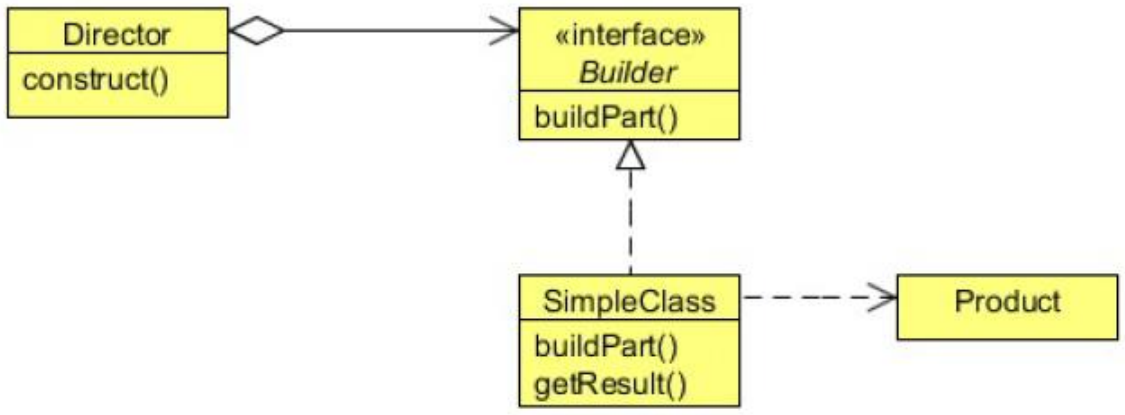
Mục đích của mẫu *Builder* là tách việc tạo một đối tượng phức tạp ra khỏi biểu diễn của nó, sao cho cùng một quá trình tạo có thể tạo được các biểu diễn khác nhau. Kiểu tách này sẽ giảm kích thước của đối tượng. Thiết kế này trở nên có cấu trúc hơn với mỗi cài đặt được chứa trong một đối tượng *builder* khác nhau. Bổ sung cài đặt mới (tức là bổ sung *builder* mới) trở nên dễ dàng. Quá trình tạo đối tượng trở nên độc lập khỏi các thành phần mà tạo nên đối tượng. Nó cung cấp kiểm soát tốt hơn quá trình tạo đối tượng.

Theo thuật ngữ cài đặt, mỗi bước khác nhau trong quá trình tạo có thể được khai báo như các phương thức của một giao diện chung để được cài đặt bằng các *builder* cụ thể khác nhau.

Một đối tượng *client* có thể tạo một khởi tạo của một *builder* cụ thể và triệu gọi tập các phương thức được yêu cầu để xây dựng các phần khác nhau của đối tượng cuối cùng. Cách tiếp cận này yêu cầu mỗi đối tượng client sẽ quan tâm đến logic tạo đó. Mỗi khi logic tạo trải qua thay đổi, mọi đối tượng *client* cần được thay đổi phù hợp.

Mẫu *Builder* đưa ra mức độ tách bạch khác để giải quyết vấn đề này. Thay vì các đối tượng *client* triệu gọi các phương thức tạo khác nhau trực tiếp, mẫu *Builder* đề xuất sử dụng một đối tượng được chỉ định được tham chiếu như *Director* mà có trách nhiệm triệu gọi các phương thức *builder* khác nhau được yêu cầu để xây dựng đối tượng cuối cùng. Các đối tượng *client* khác nhau có thể sử dụng đối tượng *Director* để tạo đối tượng được yêu cầu. Một khi một đối tượng được tạo, đối tượng client có thể yêu cầu trực tiếp từ *builder* đối tượng được xây dựng đầy đủ. Để làm thuận tiện quá trình này, một phương thức mới *getObject* có thể được khai báo trong giao diện *Builder* chung để được cài đặt bởi các *builder* cụ thể khác nhau.

Thiết kế mới này loại bỏ sự không cần thiết cho đối tượng *client* phải làm việc với các phương thức tạo dựng trong quá trình tạo đối tượng và đóng gói chi tiết cách mà đối tượng được tạo khỏi client.



Builder

- Đặc tả giao diện tổng quát cho các phần khác nhau của đối tượng *Product*

ConcreteBuilder

- Tạo và lắp ghép các phần của *Product* bằng cách cài đặt giao diện *Builder*
- Định nghĩa và theo dõi các thể hiện mà nó tạo ra
- Cung cấp giao diện để truy cập *Product*

Director

- Tạo một đối tượng sử dụng giao diện *Builder*

Product

- Biểu diễn đối tượng phức tạp trong quá trình xây dựng. *ConcreteBuilder* xây dựng biểu diễn bên trong của *Product* và định nghĩa quá trình mà nó được lắp ghép.
- Bao gồm các lớp mà định nghĩa các thành phần tạo nên, chứa cả giao diện để lắp ghép các phần thành kết quả cuối cùng.

6.12.3 Cài đặt mẫu Builder

Dưới đây là lớp *Car (Product)* mà chứa một số thành phần quan trọng của ôtô mà được yêu cầu để xây dựng đối tượng *car* hoàn chỉnh.

```
package com.javacodegeeks.patterns.builderpattern;

public class Car {

	private String bodyStyle;
	private String power;
	private String engine;
	private String breaks;
	private String seats;
	private String windows;
	private String fuelType;
	private String carType;

	public Car (String carType){
		this.carType = carType;
	}

	public String getBodyStyle() {
		return bodyStyle;
	}
	public void setBodyStyle(String bodyStyle) {
		this.bodyStyle = bodyStyle;
	}
	public String getPower() {
		return power;
	}
	public void setPower(String power) {
		this.power = power;
	}
	public String getEngine() {
		return engine;
	}
	public void setEngine(String engine) {
		this.engine = engine;
	}
	public String getBreaks() {
		return breaks;
	}
	public void setBreaks(String breaks) {
		this.breaks = breaks;
	}
	public String getSeats() {
		return seats;
	}
	public void setSeats(String seats) {
		this.seats = seats;
	}
	public String getWindows() {
		return windows;
	}
	public void setWindows(String windows) {
		this.windows = windows;
	}
	public String getFuelType() {
		return fuelType;
	}
```

```
        public void setFuelType(String fuelType) {
                this.fuelType = fuelType;
        }

        @Override
        public String toString(){
                StringBuilder sb = new StringBuilder();
                sb.append("--------------"+carType+"---------------------- \\n");
                sb.append(" Body: ");
                sb.append(bodyStyle);
                sb.append("\\n Power: ");
                sb.append(power);
                sb.append("\\n Engine: ");
                sb.append(engine);
                sb.append("\\n Breaks: ");
                sb.append(breaks);
                sb.append("\\n Seats: ");
                sb.append(seats);
                sb.append("\\n Windows: ");
                sb.append(windows);
                sb.append("\\n Fuel Type: ");
                sb.append(fuelType);

                return sb.toString();
        }
}
```

CarBuilder là giao diện *Builder* chứa tập các phương thức được sử dụng để xây dựng đối tượng *car* và các thành phần của nó.

```
package com.javacodegeeks.patterns.builderpattern;

public interface CarBuilder {

        public void buildBodyStyle();
        public void buildPower();
        public void buildEngine();
        public void buildBreaks();
        public void buildSeats();
        public void buildWindows();
        public void buildFuelType();
        public Car getCar();
}
```

Phương thức *getCar* được sử dụng để trả về đối tượng car cuối cùng sau khi tạo.

Ta xem hai cài đặt của giao diện *CarBuilder*, một cho mỗi loại *car*, tức là *Sedan* và *Sport.*

```
package com.javacodegeeks.patterns.builderpattern;

public class SedanCarBuilder implements CarBuilder{

        private final Car car = new Car("SEDAN");

        @Override
        public void buildBodyStyle() {
                car.setBodyStyle("External dimensions: overall length (inches): 202.9, " +
                                "overall width (inches): 76.2, overall height (inches):  ↩
                                    60.7, wheelbase (inches): 112.9," +
                                " front track (inches): 65.3, rear track (inches): 65.5 and ↩
                                     curb to curb turning circle (feet): 39.5");
        }
```

```
        @Override
        public void buildPower(){
                car.setPower("285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm");
        }

        @Override
        public void buildEngine() {
                car.setEngine("3.5L Duramax V 6 DOHC");
        }

        @Override
        public void buildBreaks() {
                car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake  ↩
                    distribution");
        }

        @Override
        public void buildSeats() {
                car.setSeats("Front seat center armrest.Rear seat center armrest.Split- ↩
                    folding rear seats");
        }

        @Override
        public void buildWindows() {
                car.setWindows("Laminated side windows.Fixed rear window with defroster");

        }

        @Override
        public void buildFuelType() {
                car.setFuelType("Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and  ↩
                    437 mi. range");

        }

        @Override
        public Car getCar(){
                return car;
        }

}
```

```
package com.javacodegeeks.patterns.builderpattern;

public class SportsCarBuilder implements CarBuilder{

        private final Car car = new Car("SPORTS");

        @Override
        public void buildBodyStyle() {
                car.setBodyStyle("External dimensions: overall length (inches): 192.3," +
                                " overall width (inches): 75.5, overall height (inches):
                                    54.2, wheelbase (inches): 112.3," +
                                " front track (inches): 63.7, rear track (inches): 64.1 and
                                    curb to curb turning circle (feet): 37.7");
        }

        @Override
        public void buildPower(){
                car.setPower("323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm");
        }
```

```
        @Override
        public void buildEngine() {
                car.setEngine("3.6L V 6 DOHC and variable valve timing");
        }

        @Override
        public void buildBreaks() {
                car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake
                    distribution. StabiliTrak stability control");

        }

        @Override
        public void buildSeats() {
                car.setSeats("Driver sports front seat with one power adjustments manual
                    height, front passenger seat sports front seat with one power
                    adjustments");
        }

        @Override
        public void buildWindows() {
                car.setWindows("Front windows with one-touch on two windows");

        }

        @Override
        public void buildFuelType() {
                car.setFuelType("Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and
                    380 mi. range");
        }

        @Override
        public Car getCar(){
                return car;
        }

}
```

Đây là lớp Director, lưu giữ các bước tương ứng tạo nên các thành phần để tạo ra sản phẩm car.

```
public class CarDirector {

    private CarBuilder carBuilder;

    public CarDirector(CarBuilder carBuilder){
        this.carBuilder = carBuilder;
    }

    public void build(){
        carBuilder.buildBodyStyle();
        carBuilder.buildPower();
        carBuilder.buildEngine();
        carBuilder.buildBreaks();
        carBuilder.buildSeats();
        carBuilder.buildWindows();
        carBuilder.buildFuelType();
    }
}
```

Hai lớp *Builde*r trên tạo và xây dựng *Product*, tức là đối tượng *car* tuân theo đặc tả được yêu cầu. Bây giờ ta kiểm tra *Builder*.

```
package com.javacodegeeks.patterns.builderpattern;

public class TestBuilderPattern {

        public static void main(String[] args) {
                CarBuilder carBuilder = new SedanCarBuilder();
                CarDirector director = new CarDirector(carBuilder);
                director.build();
                Car car = carBuilder.getCar();
                System.out.println(car);

                carBuilder = new SportsCarBuilder();
                director = new CarDirector(carBuilder);
                director.build();
                car = carBuilder.getCar();
                System.out.println(car);
        }

}
```

Code trên cho kết quả sau:

```
--------------SEDAN---------------------
 Body: External dimensions: overall length (inches): 202.9, overall width (inches): 76.2,  ↩
     overall height (inches): 60.7, wheelbase (inches): 112.9, front track (inches): 65.3,  ↩
     rear track (inches): 65.5 and curb to curb turning circle (feet): 39.5
 Power: 285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm
 Engine: 3.5L Duramax V 6 DOHC
 Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution
 Seats: Front seat center armrest.Rear seat center armrest.Split-folding rear seats
 Windows: Laminated side windows.Fixed rear window with defroster
 Fuel Type: Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and 437 mi. range
--------------SPORTS---------------------
 Body: External dimensions: overall length (inches): 192.3, overall width (inches): 75.5,  ↩
     overall height (inches): 54.2, wheelbase (inches): 112.3, front track (inches): 63.7,  ↩
     rear track (inches): 64.1 and curb to curb turning circle (feet): 37.7
 Power: 323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm
 Engine: 3.6L V 6 DOHC and variable valve timing
 Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution. StabiliTrak ↩
      stability control
 Seats: Driver sports front seat with one power adjustments manual height, front passenger  ↩
     seat sports front seat with one power adjustments
 Windows: Front windows with one-touch on two windows
 Fuel Type: Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and 380 mi. range
```

Trong lớp trên, trước hết chúng ta tạ *SedanBuilder* và *CarDirector*. Sau đó chúng ta yêu cầu *CarDirector* xây dựng car tuân theo *builder* mà truyền cho nó. Cuối cùng, chúng ta trực tiếp yêu cầu *builder* cung cấp đối tượng cả được tạo ra.

Chúng ta làm như vậy với *SportBuilder* mà trả về đối tượng car tuân theo đặc tả *Sport car*.

Cách tiếp cận sử dụng mẫu *Builder* là linh hoạt đủ để bổ sung bất cứ loại ôtô mới nào trong tương lai mà không thay đổi bất cứ *code* nào đã tồn tại. Tất cả những gì chúng ta cần là tạo ra *builder* mới tuân theo đặc tả của ôtô mới và cung cấp nó cho *Director* để xây dựng.

6.12.4 Dạng khác của mẫu Builder

Có dạng khác của mẫu *Builder* khác với những gì mà chúng ta đã nhìn thấy. Đôi khi có một đối tượng với danh sách dài các tính chất, và hầu hết các tính chất này đều là tùy chọn. Xét form trực tuyến mà cần phải điền vào để trở thành thành viên của Trang. Bạn cần điền vào mọi ô bắt buộc, nhưng bạn có thể bỏ qua các ô tùy chọn hoặc đôi khi có thể đáng để nhập một vài ô tùy chọn.

Hãy kiểm tra lớp *Form* dưới đây, mà chứa danh sách dài các tính chất và một số tính chất là tùy chọn. Các ô bắt buộc trong *Form* có *firstName*, *lastName*, *userName* và *password*, nhưng các ô khác là tùy chọn.

```
package com.javacodegeeks.patterns.builderpattern;

import java.util.Date;

public class Form {

        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private String address;
        private Date dob;
        private String email;
        private String backupEmail;
        private String spouseName;
        private String city;
```

```
        private String state;
        private String country;
        private String language;
        private String passwordHint;
        private String secuirtyQuestion;
        private String securityAnswer;

}
```

Câu hỏi là, dạng hàm tạo như thế nào chúng ta cần viết cho lớp này. Viết hàm tạo với danh sách tham số dài không là lựa chọn tốt, nó làm cho client thất vọng đặc biệt nếu các trường quan trọng thì có ít. Điều này làm tăng phạm vi lỗi, *client* có thể cung cấp lỗi một giá trị cho một trường sai chỗ.

Cách khác là sử dụng lồng ghép hàm tạo, ở đó bạn cần cung cấp cho một hàm tạo với các tham số bắt buộc, cái khác với một tham số tùy chọn, cái thứ ba với hai tham số tùy chọn, và cứ như vậy và đến cuối cùng là hàm tạo với mọi tham số tùy chọn.

Lồng ghép hàm tạo có thể trông như sau:

```
public Form(String firstName,String lastName){
        this(firstName,lastName,null,null);
}

public Form(String firstName,String lastName,String userName,String password){
        this(firstName,lastName,userName,password,null,null);
}

public Form(String firstName,String lastName,String userName,String password,String address ↩
    ,Date dob){
        this(firstName,lastName,userName,password,address,dob,null,null);
}

public Form(String firstName,String lastName,String userName,String password,String address ↩
    ,Date dob,String email,String backupEmail){
        ...
}
```

Khi bạn muốn tạo một khởi tạo, bạn sử dụng hàm tạo với danh sách ngắn tham số chứa mọi tham số bạn muốn nhập.

Hàm tạo lồng nhau hoạt động, nhưng khó viết *code client* khi có nhiều tham số, và ngay cả rất khó đọc nó. Người đọc cần xem xét, các giá trị có ý nghĩa gì và đếm số tham số để tìm ra cái phù hợp. Các dãy dài các tham số kiểu xác định có thể gây ra các lỗi, Nếu *client* vô tình đảo ngược hai tham số, chương trình dịch sẽ không có vấn đề gì, nhưng chương trình sẽ có hành vi không đúng tại thời gian chạy.

Cách thứ hai, khi bạn đối mặt với nhiều tham số hàm tạo, là mẫu JavaBeans, ở đó bạn gọi hàm tạo ít tham số để tạo đối tượng và sau đó gọi các phương thức để nhập mỗi tham số bắt buộc và mỗi tham số quan tâm tùy chon.

Không may mắn, mẫu JavaBeans có nhược điểm nghiêm trọng của chính nó. Vì hàm tạo được tách ra nhiều lời gọi, JavaBeans có thể ở trong trạng thái nửa vời không đúng đắn qua hàm tạo của nó. Lớp này không có tùy chọn áp đặt tính đúng đắn bằng việc kiểm tra tính đúng đắn của các tham số hàm tạo. Thử sử dụng đối tượng khi nó ở trong trạng thái không đúng đắn có thể gây ra lỗi mà đã được loại bỏ khỏi code chứa lỗi, vì vậy rất khó bắt lỗi.

Có cách thứ ba mà kết hợp an toàn của hàm tạo lồng nhau với tính đọc được của mẫu *JavaBeans*. Nó là một dạng *form* của mẫu *Builder*. Thay vì tạo trực tiếp đối tượng mong muốn, *client* gọi hàm tạo với mọi tham số bắt buộc và nhận được đối tượng *builder*. Sau đó *client* gọi các hàm giống *set* trên đối tượng *builder* để nhập mỗi giá trị tham số quan tâm. Cuối cùng, *client* gọi phương thức ít tham số để tạo đối tượng.

```
package com.javacodegeeks.patterns.builderpattern;

import java.util.Date;
```

```
public class Form {

        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private String address;
        private Date dob;
        private String email;
        private String backupEmail;
        private String spouseName;
        private String city;
        private String state;
        private String country;
        private String language;
        private String passwordHint;
        private String securityQuestion;
        private String securityAnswer;

        public static class FormBuilder {

                private String firstName;
                private String lastName;
                private String userName;
                private String password;
                private String address;
                private Date dob;
                private String email;
                private String backupEmail;
                private String spouseName;
                private String city;
                private String state;
                private String country;
                private String language;
                private String passwordHint;
                private String securityQuestion;
                private String securityAnswer;

                public FormBuilder(String firstName, String lastName, String userName,  ↩
                    String password){
                        this.firstName = firstName;
                        this.lastName = lastName;
                        this.userName = userName;
                        this.password = password;
                }

                public FormBuilder address(String address){
                        this.address = address;
                        return this;
                }

                public FormBuilder dob(Date dob){
                        this.dob = dob;
                        return this;
                }

                public FormBuilder email(String email){
                        this.email = email;
                        return this;
                }

                public FormBuilder backupEmail(String backupEmail){
```

```
                this.backupEmail = backupEmail;
                return this;
        }

        public FormBuilder spouseName(String spouseName){
                this.spouseName = spouseName;
                return this;
        }

        public FormBuilder city(String city){
                this.city = city;
                return this;
        }

        public FormBuilder state(String state){
                this.state = state;
                return this;
        }

        public FormBuilder country(String country){
                this.country = country;
                return this;
        }

        public FormBuilder language(String language){
                this.language = language;
                return this;
        }

        public FormBuilder passwordHint(String passwordHint){
                this.passwordHint = passwordHint;
                return this;
        }

        public FormBuilder securityQuestion(String securityQuestion){
                this.securityQuestion = securityQuestion;
                return this;
        }

        public FormBuilder securityAnswer(String securityAnswer){
                this.securityAnswer = securityAnswer;
                return this;
        }

        public Form build(){
                return new Form(this);
        }
}

private Form(FormBuilder formBuilder){

        this.firstName = formBuilder.firstName;
        this.lastName = formBuilder.lastName;
        this.userName = formBuilder.userName;
        this.password = formBuilder.password;
        this.address = formBuilder.address;
        this.dob = formBuilder.dob;
        this.email = formBuilder.email;
        this.backupEmail = formBuilder.backupEmail;
        this.spouseName = formBuilder.spouseName;
        this.city = formBuilder.city;
        this.state = formBuilder.state;
```

```
            this.country =  formBuilder.country;
            this.language = formBuilder.language;
            this.passwordHint = formBuilder.passwordHint;
            this.securityQuestion = formBuilder.securityQuestion;
            this.securityAnswer = formBuilder.securityAnswer;
    }

    @Override
    public String toString(){
            StringBuilder sb = new StringBuilder();
            sb.append(" First Name: ");
            sb.append(firstName);
            sb.append("\\n Last Name: ");
            sb.append(lastName);
            sb.append("\\n User Name: ");
            sb.append(userName);
            sb.append("\\n Password: ");
            sb.append(password);

            if(this.address!=null){
                    sb.append("\\n Address: ");
                    sb.append(address);
            }
            if(this.dob!=null){
                    sb.append("\\n DOB: ");
                    sb.append(dob);
            }
            if(this.email!=null){
                    sb.append("\\n Email: ");
                    sb.append(email);
            }
            if(this.backupEmail!=null){
                    sb.append("\\n Backup Email: ");
                    sb.append(backupEmail);
            }
            if(this.spouseName!=null){
                    sb.append("\\n Spouse Name: ");
                    sb.append(spouseName);
            }
            if(this.city!=null){
                    sb.append("\\n City: ");
                    sb.append(city);
            }
            if(this.state!=null){
                    sb.append("\\n State: ");
                    sb.append(state);
            }
            if(this.country!=null){
                    sb.append("\\n Country: ");
                    sb.append(country);
            }
            if(this.language!=null){
                    sb.append("\\n Language: ");
                    sb.append(language);
            }
            if(this.passwordHint!=null){
                    sb.append("\\n Password Hint: ");
                    sb.append(passwordHint);
            }
            if(this.securityQuestion!=null){
                    sb.append("\\n Security Question: ");
                    sb.append(securityQuestion);
```

```
                }
                if(this.securityAnswer!=null){
                        sb.append("\\n Security Answer: ");
                        sb.append(securityAnswer);
                }

                return sb.toString();
        }

        public static void main(String[] args) {
                Form form = new Form.FormBuilder("Dave", "Carter", "DavCarter", "DAvCaEr123 ↩
                    ").passwordHint("MyName").city("NY").language("English").build();
                System.out.println(form);
        }

}
```

Code trên tạo kết quả đầu ra như sau:

```
First Name: Dave
Last Name: Carter
User Name: DavCarter
Password: DAvCaEr123
City: NY
Language: English
Password Hint: MyName
```

Như bạn đã thấy rõ, bay giờ *client* chỉ cần cung cấp các trường bắt buộc và các trường mà quan trọng với anh ta. Để tạo đối tượng f*orm*, bây giờ chúng ta triệu gọi hàm tạo *FormBuilder* gồm tất cả các trường bắt buộc và sau đó chúng ta cần gọi tập các phương thức được yêu cầu trên nó và cuối cùng phương thức *build* để nhận đối tượng form.

6.12.5 Khi nào sử dụng mẫu Builder

Sử dụng mẫu *Builder* khi

- Thuật toán để tạo đối tượng phức tạp cần là độc lập khỏi các phần mà tạo nên đối tượng và chúng được lắp ghép như thế nào
- Quá trình tạo cần cho phép các thể hiện khác nhau cho các đối tượng mà được tạo.