Builder

The design problem

Imagine a pleasant situation that you are in a restaurant and making an order. The restaurant has all these signature dishes, each of them consisting of a set of ingredients.
It can happen that a guest wants their dish modified: suppose, you don't eat onion or are allergic to peanuts or some other specific ingredient.
The chef then would have to change the properties of a dish dynamically. In situations like that, the **Builder** design pattern really saves the day.

Imagine you work on some GUI library. As a developer, you do your best to make your library convenient for application developers who will use it later. The library has an alert dialog element, which contains title, text, apply and cancel buttons, footer, and picture. You decided to create a class describing it:

```
class AlertDialog {
    private String title;
    private String text;
    private String applyButton;
    private String cancelButton;
    private String footer;
    private String picture;

    AlertDialog(String title, String text, String applyButton, String cancelButton,
String footer, String picture) {
        this.title = title;
        this.text = text;
        this.applyButton = applyButton;
        this.cancelButton = cancelButton;
        this.footer = footer;
        this.picture = picture;
    }
}
```

To create an instance, just call the constructor:

```
AlertDialog alertDialog = new AlertDialog("title", "text", "applyButton",
"cancelButton", "footer", "pathToPicture");
```

Suppose an application developer needs an element with the title and apply button only. In such a case, your constructor doesn't look friendly and affects the code readability:

```
AlertDialog alertNotification = new AlertDialog("Completed successfully", null,
"Ok", null, null, null);
```

To avoid such constructor calls in application code, you create the

corresponding constructor:

```
class AlertDialog {
    ...

    AlertDialog(String title, String applyButton) {
        this(title, null, applyButton, null, null, null);

    ...
}
```

The problem seems to be solved, but only until an application developer decides to create an element with another combination of elements. In this case, you need to create a new constructor as well. Thus the number of constructors grows again. Such a situation is called constructor pollution.

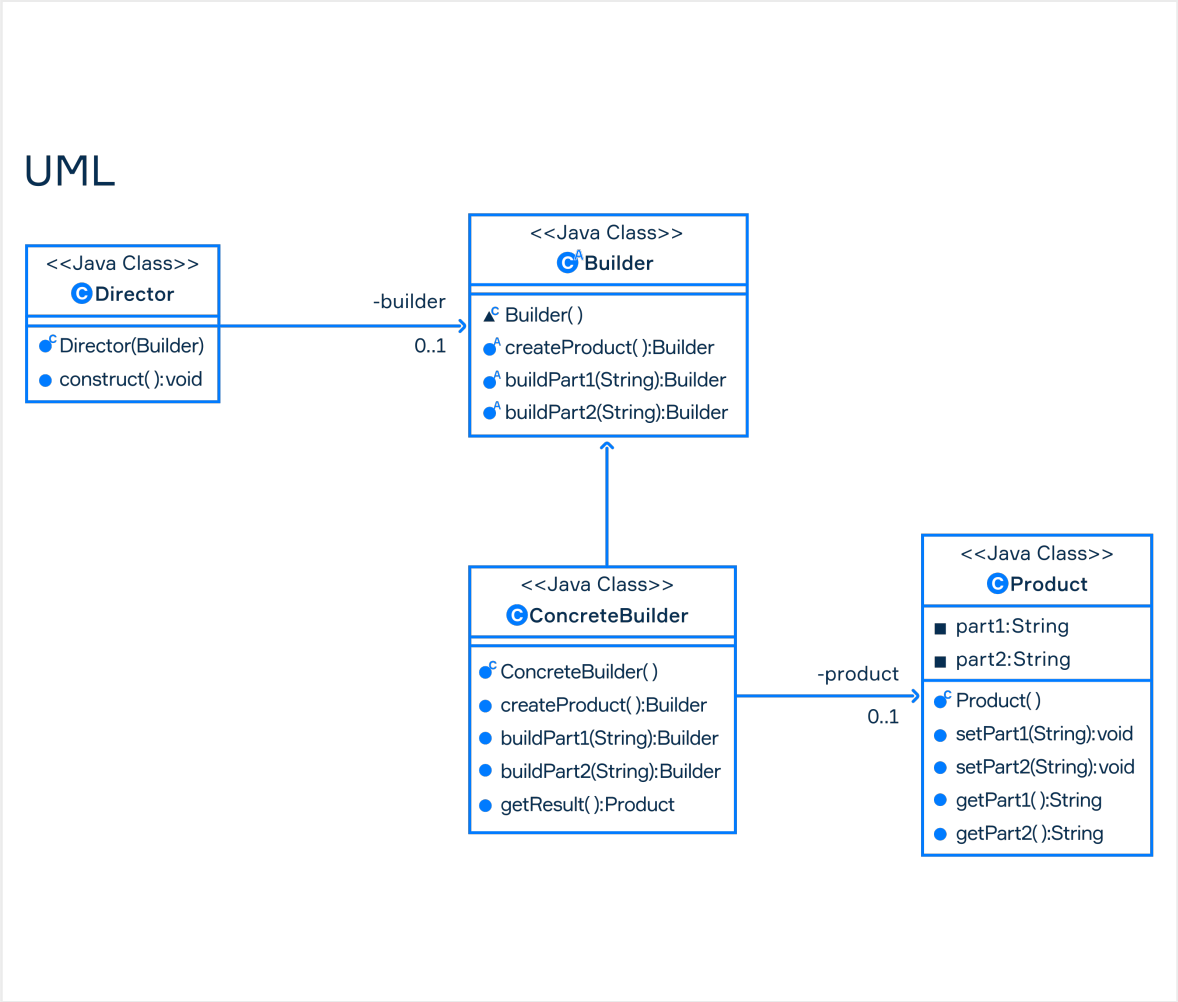
An alternative way is adding one default constructor AlertDialog() {...} and setters for each field:

```
AlertDialog alertDialog = new AlertDialog();
alertDialog.setTitle("Completed successfully");
alertDialog.setApplyButton("Ok");
```

It solves the problem, but in case if AlertDialog instances must be immutable, the solution doesn't work.

The Builder pattern

The **Builder** pattern is a **creational** design pattern used to separate complex object construction from its representation. It can be used to create objects with a specified structure step-by-step. The main benefit is that you can avoid the so-called constructor pollution.



The builder pattern has the following components:

- **Builder** interface describes the steps of product construction. Each complex object requires the service of a **Builder** class to generate object instances.
- **ConcreteBuilder** implements **Builder** to make the required representation of the product. It will construct and assemble the parts of the final product and provide the interface to retrieve it. This is the main component that keeps track of the specific representation of the product.
- **Director** manages the object creation process using the **Builder** class, and it will not directly create and assemble the final complex object.
- **Product** is the complex object constructed using the concrete builder class which contains the final user-requested representation.

In our example, we will create a simple *AlertDialog* with a fixed structure: *Title, Text, ApplyButton,* and *CancelButton.* Note that the Builder pattern allows us to avoid the fixed properties, but it can't allow extending them.

Example
There are only two steps, but stay vigilant as the first step is capacious and powerful. The implementation of *AlertDialog* contains its Builder:

```
class AlertDialog {
    private String title;
    private String text;
    private String applyButton;
    private String cancelButton;

    private AlertDialog(String title, String text, String applyButton, String
cancelButton) {
        this.title = title;
        this.text = text;
        this.applyButton = applyButton;
        this.cancelButton = cancelButton;
    }

    @Override
    public String toString() {
        String str = "";
        if (title != null) {
            str += "The title is: \"" + title + "\"\n";
        }
        if (text != null) {
            str += "The text is: \"" + text + "\"\n";
        }
        if (applyButton != null) {
            str += "The applyButton is: \"" + applyButton + "\"\n";
        }
        if (cancelButton != null) {
            str += "The cancelButton is: \"" + cancelButton + "\"\n";
        }

        return str;
    }

    static class Builder {
        private String title;
        private String text;
        private String applyButton;
        private String cancelButton;

        Builder() {}

        Builder setTitle(String title) {
            this.title = title;
            return this;
        }

        Builder setText(String text) {
```

```
            this.text = text;
            return this;
        }

        Builder setApplyButton(String applyButton) {
            this.applyButton = applyButton;
            return this;
        }

        Builder setCancelButton(String cancelButton) {
            this.cancelButton = cancelButton;
            return this;
        }

        AlertDialog build() {
            return new AlertDialog(title, text, applyButton, cancelButton);
        }
    }
}
```

For simplicity, the example doesn't contain **Builder** and **Director**. Builder is a **ConcreteBuilder** here. AlertDialog is a **Product**. Note that we made the constructor private to prevent direct usage.

The last step is *TestDrive*. As you can see above, method *build()* returns the object we needed. This is our trigger starting object creation and works as a **Director**.

```
class TestDrive {
    public static void main(String[] args) {

        AlertDialog twoButtonsDialog = new AlertDialog.Builder()
                .setTitle("Two buttons dialog")
                .setText("You can use either `Okay` or `Cancel`")
                .setApplyButton("Okay")
                .setCancelButton("Cancel")
                .build();

        System.out.println(twoButtonsDialog);

        AlertDialog oneButtonsDialog = new AlertDialog.Builder()
                .setTitle("One button dialog")
                .setText("You can use `Close` only")
                .setCancelButton("Close")
                .build();

        System.out.println(oneButtonsDialog);
    }
```

}
And that's it. Although the example pattern code structure differs from a pattern structure described in UML, this is a common builder implementation. T he pattern looks small and easy to implement, but it is extremely helpful in combination with other patterns, such as *Composite pattern,* any *Factory pattern,* and others. Finally, the output is:

The title is: "Two buttons dialog"
The text is: "You can use either `Okay` or `Cancel`"
The applyButton is: "Okay"
The cancelButton is: "Cancel"

The title is: "One button dialog"
The text is: "You can use `Close` only"
The cancelButton is: "Close"

Conclusion
We have created an encapsulated step-by-step creational process. Also, the builder provides an opportunity to create an object with a variable set of steps and consequently instantiated properties (unlike "one-step" Factory Patterns).

One More example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

/**
 * ConcreteComponent - Geek.
 **/
class Geek {

    private String type;
    private List<String> languages;
    private int experience;

    public Geek(String type, List<String> languages, int experience) {
        this.type = type;
        this.languages = languages;
        this.experience = experience;
    }

    @Override
    public String toString() {
```

```
        return "Type : " + type + "\n" + "Languages : " + languages + "\n" +
"Experience : " + experience + " years";
    }

}

/**
 * Builder interface describe step of object creation.
 **/
interface Builder {
    void setType(String type);

    void setLanguages(List<String> languages);

    void setExperience(int experience);
}

/**
 * Concrete Builder build Geek component.
 **/
class GeekBuilder implements Builder {

    private String type;
    private List<String> languages;
    private int experience;

    @Override
    public void setType(String type) {
        this.type = type;
    }

    @Override
    public void setLanguages(List<String> languages) {
        this.languages = languages;
    }

    @Override
    public void setExperience(int experience) {
        this.experience = experience;
    }

    public Geek getResult() {
        // write your code here ...
        return new Geek(type,languages,experience);
    }
}
```

```
/**
 * Builder Director.
 */
class GeekDirector {
    public void buildAdmin(Builder builder) {
        builder.setType("Admin");
        ArrayList<String> languages = new ArrayList<>();
        languages.add("Perl");
        languages.add("PowerShell");
        builder.setLanguages(languages);
        builder.setExperience(10);
    }
}

class Main {
    public static void main(String[] args) {
        final Scanner scanner = new Scanner(System.in);
        final String geekName = scanner.nextLine();
        scanner.close();
        System.out.println("Geek " + geekName + " created.");
        GeekDirector director = new GeekDirector();
        GeekBuilder builder = new GeekBuilder();
        director.buildAdmin(builder);
        Geek geek = builder.getResult();
        System.out.println(geek);
    }
}
```

What is a disadvantage of the Builder pattern?

It complicates the code due to the introduction of additional classes...