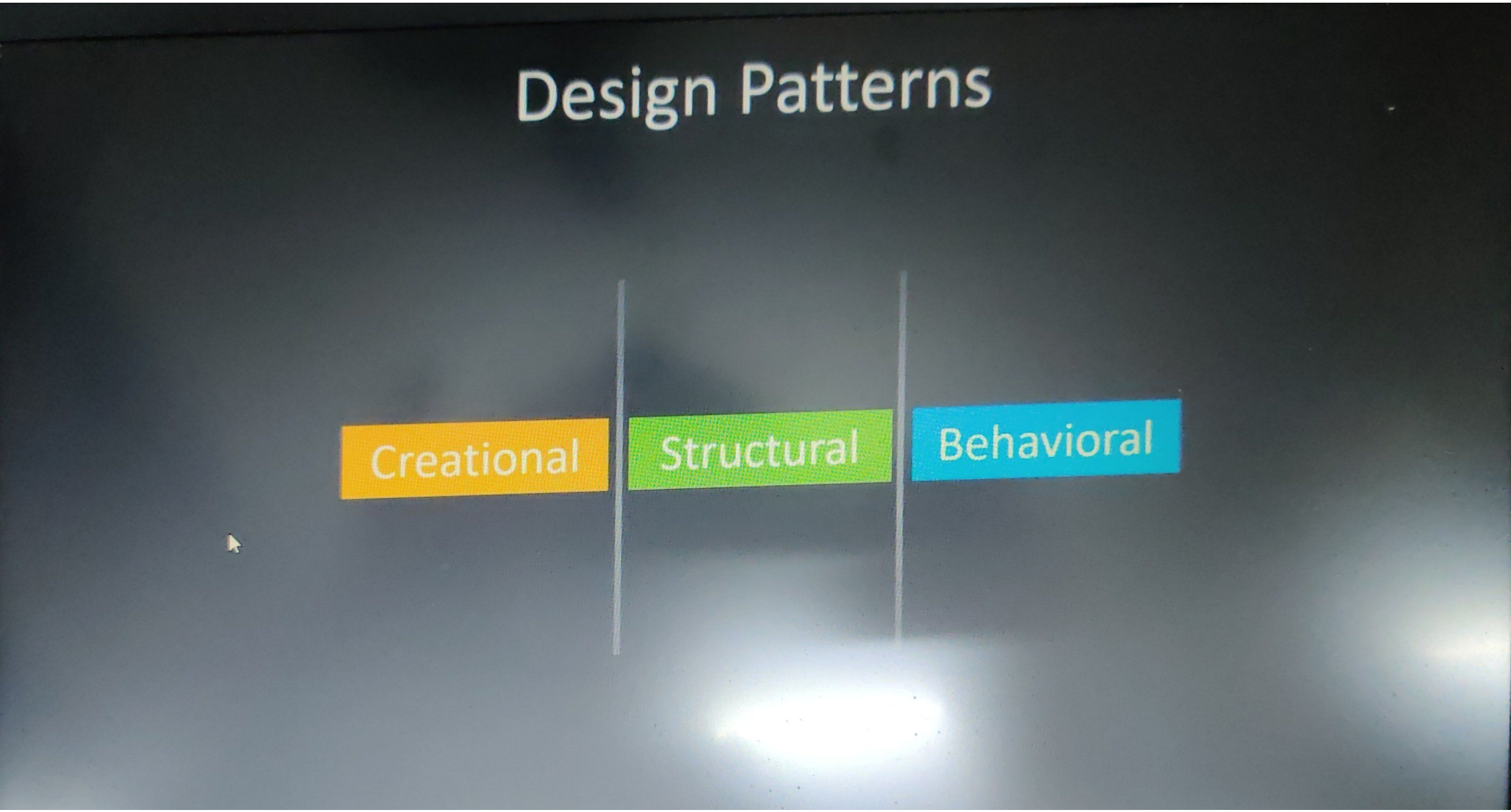

Design Patterns
Creational
Structural
Behavioral

# Design Patterns

**Creational** — Creational patterns deal with the process of creation of objects of classes.

# Design Patterns

**Structural**

**Structural patterns deal with how classes and objects are arranged or composed**

# Design Patterns

**Behavioral**

**Behavioral patterns describe how classes and objects interact & communicate with each other**

# Design Patterns

Creational

- Builder
- Simple Factory
- Factory Method
- Prototype
- Singleton
- Abstract Factory
- Object Pool

## What problem builder design pattern solves?

Class constructor requires a lot of information

```
//Product instances are immutable
class Product {

        public Product(int weight, double price, int shipVolume, int shipCode) {

                //initialize

        }

        //other code

}
```

# What problem builder design pattern solves?

Objects that need other objects or “parts” to construct them.

```
class Address {
        public Address(String houseNumber, String street,....) {
                //initialize
        }
}
```

```
class User {
        public User(String name, Address address, LocateDate birthdate, List<Role> roles ){
                //initialize
        }
        //other code
}
```

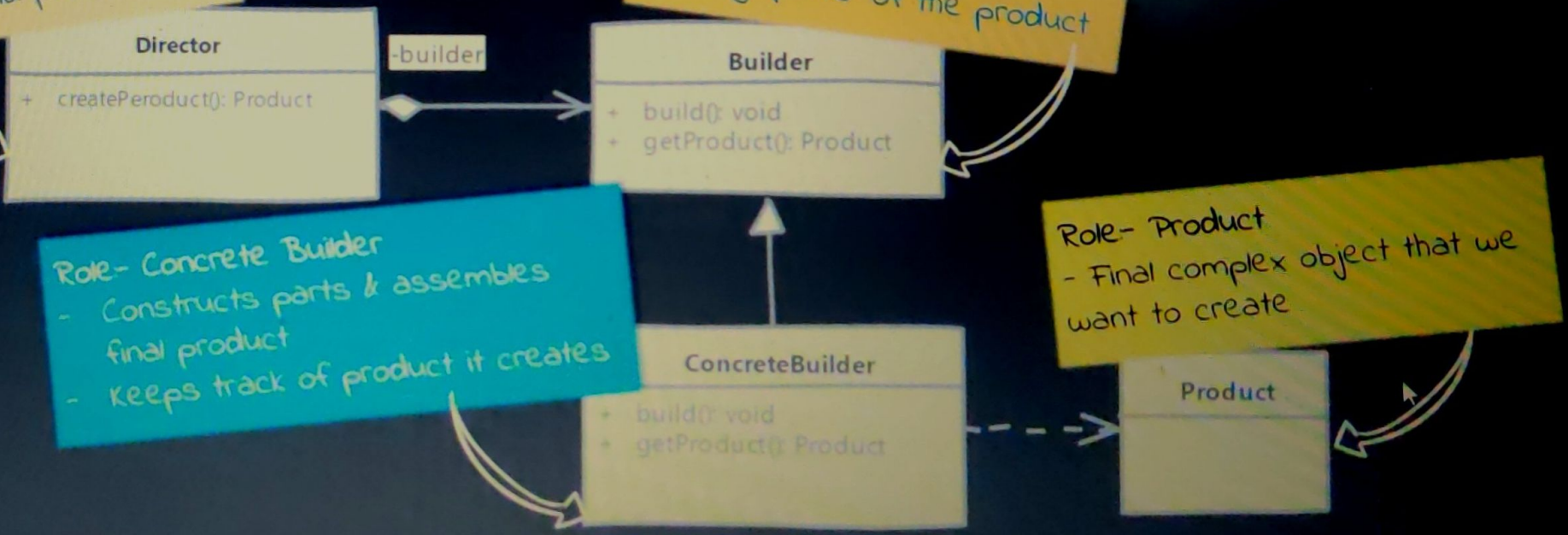
# What is a Builder?

- We have a complex process to construct an object involving multiple steps, then builder design pattern can help us.
- In builder we remove the logic related to object construction from "client" code & abstract it in separate classes.

# UML

Role- Director
- uses builder to construct object
- Knows the steps & their sequence to build product

Role- Builder
- Provides interface for creating "parts" of the product



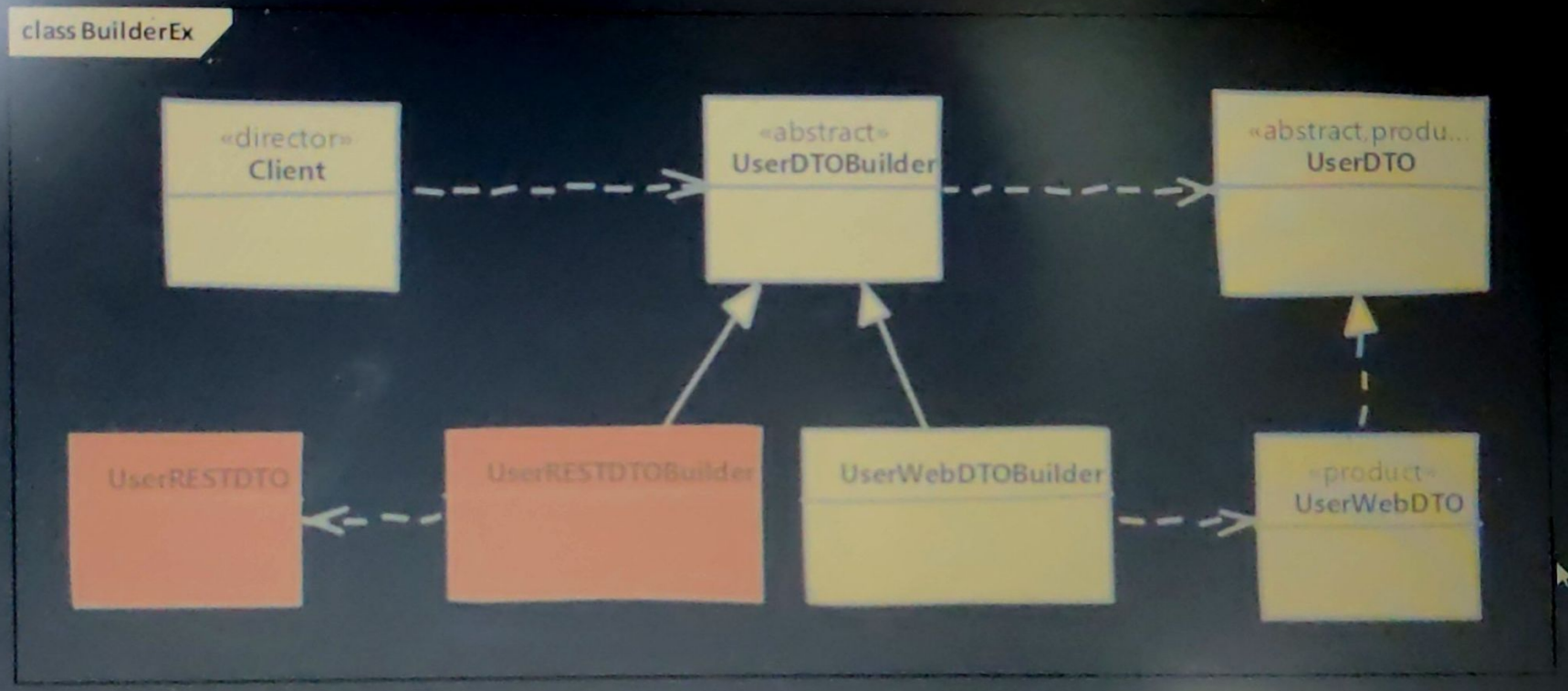

Role- Concrete Builder
- Constructs parts & assembles final product
- Keeps track of product it creates

Role- Product
- Final complex object that we want to create

# Implement a Builder

- We start by creating a builder
  - Identify the "parts" of the product & provide methods to create those parts
  - It should provide a method to "assemble" or build the product/object
  - It must provide a way/method to get fully built object out. *Optionally* builder can keep reference to an product it has built so the same can be returned again in future.
- A director can be a separate class or client can play the role of director.

# Example: UML


class BuilderEx
«director»
Client
«abstract»
UserDTOBuilder
«abstract,produ...
UserDTO
UserRESTDTO
UserRESTDTOBuilder
UserWebDTOBuilder
«product»
UserWebDTO

# Implementation Considerations

- You can easily create an immutable class by implementing builder as an inner static class. You'll find this type of implementation used quite frequently even if immutability is not a main concern.

# Design Considerations

- The director role is rarely implemented as separate class, typically the consumer of the object instance or the client handles that role.
- Abstract builder is also not required if “product” itself is not part of any inheritance hierarchy. You can directly create concrete builder.
- If you are running into a “too many constructor arguments” problem then it’s a good indication that builder pattern *may* help.

# Example of a Builder Pattern

- The java.lang.StringBuilder class as well as various buffer classes in java.nio package like ByteBuffer, CharBuffer are often given as examples of builder pattern.
- In my humble opinion they can be given as examples of builder pattern, but with an understanding that they don't match 100% with GoF definition. These classes do allow us to build final object in steps, providing only a part of final object in one step. In this way they can be thought of as an implementation of builder pattern.
- So a StringBuilder satisfies the intent/purpose of builder pattern. However as soon we start looking at structure of the StringBuilder things start to fall apart. GoF definition also states that, builder has potential to build different representations of product interface using same steps.

# Example of a Builder Pattern

- There is another great example of builder pattern in Java 8. The java.util.Calendar.Builder class is a builder.

```
public static class Builder {
    private static final int NFIELDS = FIELD_COUNT + 1; // +1
    private static final int WEEK_YEAR = FIELD_COUNT;

    private long instant;
    // Calendar.stamp[] (lower half) and Calendar.fields[] (upper half) combined
    private int[] fields;
    // Pseudo timestamp starting from MINIMUM_USER_STAMP.
```

Code from Calendar$Builder.class in rt.jar

```
public Builder setWeekDate(int weekYear, int weekOfYear, int dayOfWeek) {
    allocateFields();
    internalSet(WEEK_YEAR, weekYear);
    internalSet(WEEK_OF_YEAR, weekOfYear);
    internalSet(DAY_OF_WEEK, dayOfWeek);
    return this;
}
```

```
public Calendar build() {
    if (locale == null) {
        locale = Locale.getDefault();
    }
    if (zone == null) {
        zone = TimeZone.getDefault();
    }
    Calendar cal;
    if (type == null) {
```

# Compare & Contrast with Prototype

## Builder

- We have complex constructor and builder allows us to work with that.
- We can create a builder as separate class and so it can work with legacy code.

## Prototype

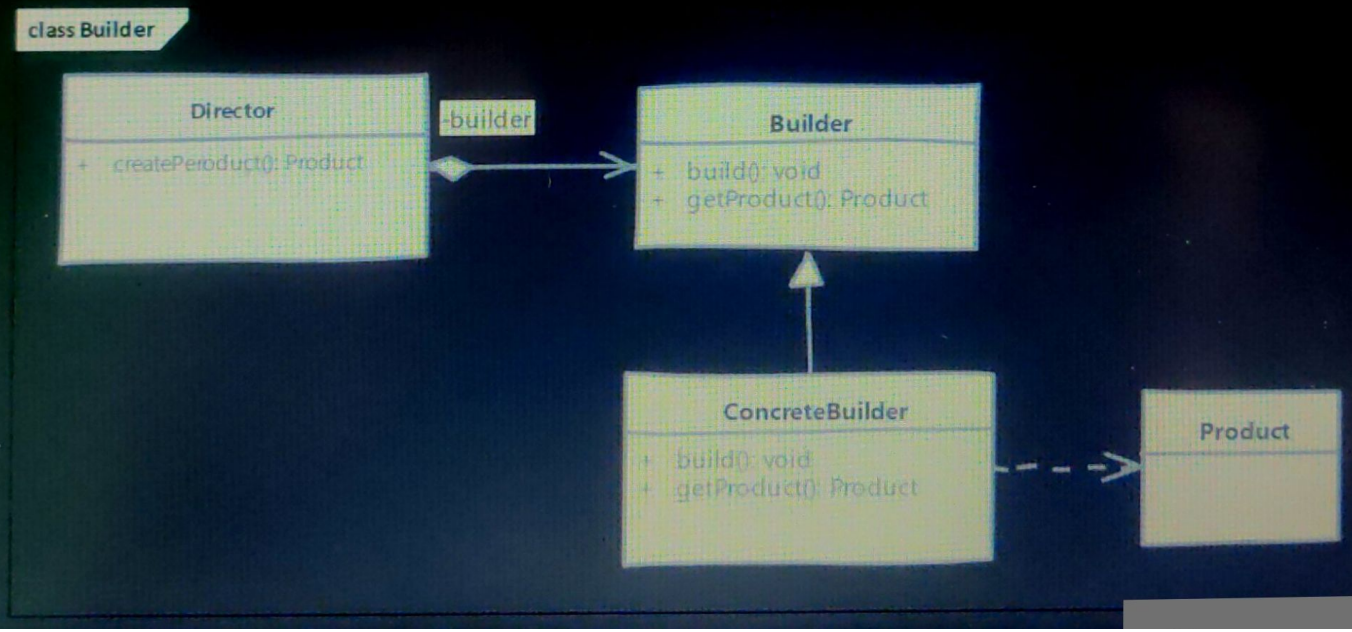
- Prototype allows to altogether skip using constructor.
- In java this pattern works using clone method, and needs to modify existing code so may not work with legacy code.

# Pitfalls

- A little bit complex for new comers mainly because of 'method chaining', where builder methods return builder object itself.
- Possibility of partially initialized object; user code can set only a few or none of properties using withXXX methods and call build(). If required properties are missing, build method should provide suitable defaults or throw exception.

# In-A-Hurry Summary

- **Think of builder pattern when you have a complex constructor or an object is built in multiple steps.**



# In-A-Hurry Summary

## Builder

```
//The concrete builder for UserWebDTO
public class UserWebDTOBuilder implements UserDTOBuilder {

    private String firstName;

    private String lastName;

    public UserWebDTOBuilder withFirstName(String fname) {
        this.firstName = fname;
        return this;
    }

    public UserWebDTOBuilder withLastName(String lname) {
        this.lastName = lname;
        return this;
    }

    public UserWebDTO build() {
        Period age = Period.between(birthday, LocalDate.now
        this.userDTO = new UserWebDTO(firstName+" " + lastN
        return this.userDTO;
    }

    public UserWebDTO getUserDTO() {
        return this.userDTO;
    }
```

## Client

```
public static void main(String[] args) {
    User user = createUser();
    UserDTOBuilder builder = new UserWebDTOBuilder();
    //Client has to provide director with concrete builder
    UserDTO dto = directBuild(builder, user);
    System.out.println(dto);
}
```

## Director (Role played by Client)

```
/**
 * This method serves the role of director in builder pattern.
 */
private static UserDTO directBuild(UserDTOBuilder builder, User user) {
    return builder.withFirstName(user.getFirstName())
    .withLastName(user.getLastName())
    .withBirthday(user.getBirthday())
    .withAddress(user.getAddress())
    .build();
}
```