“Heaven’s Light is Our Guide”

Department of Computer Science & Engineering

RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

Lab Report

Submitted By:

1. **Name: Khandoker Sefayet Alam**
   Roll:2003121
2. **Name: Siam Reza**
   Roll: 2003122
3. **Name: Md. Omar Hossain**
   Roll:2003123

Department: Computer Science & Engineering
Section-C

Course code: CSE 3206
Course name: Software Engineering Sessional

Submitted To:

Farjana Parvin
Lecturer,
Dept of Computer Science & Engineering,RUET

Problem 01:

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.

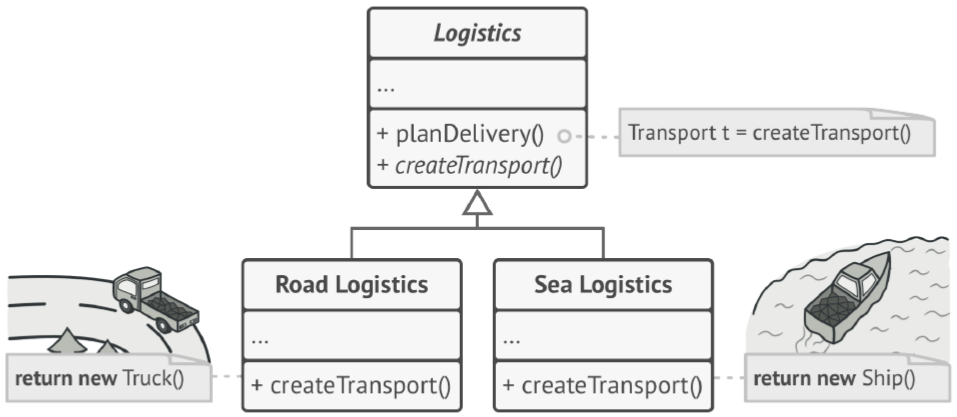
After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

Great news, right? But how about the code? At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

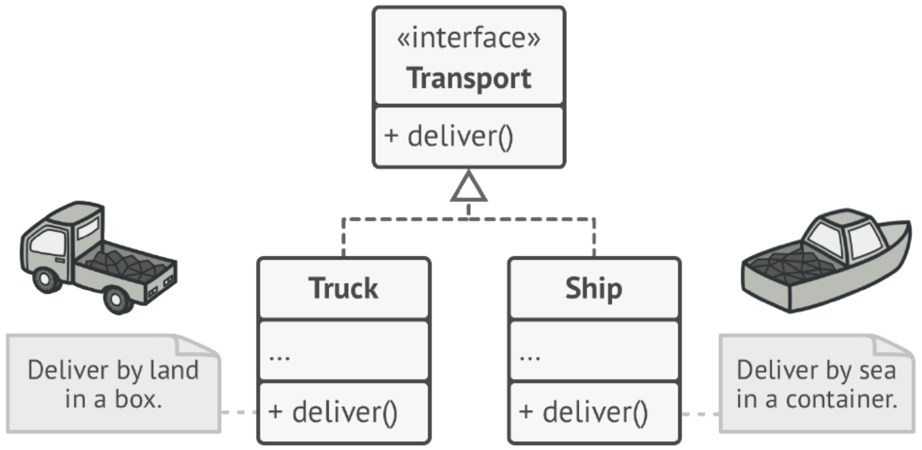
As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

Solution:

UML Diagrams:



Subclasses can alter the class of objects being returned by the factory method.



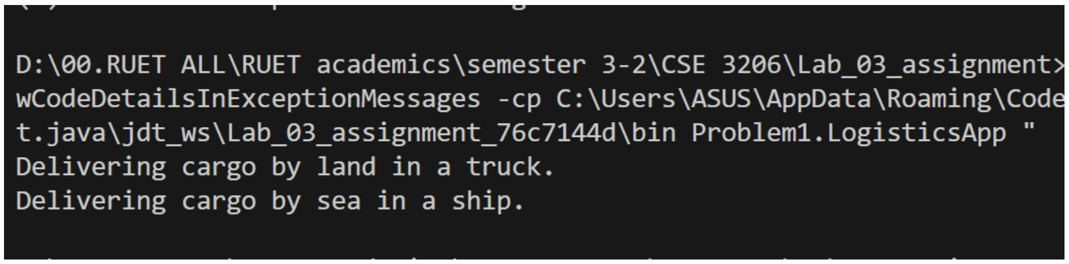
All products must follow the same interface.

Code:

```
Code:

package Problem1;


interface Transport {
    void deliver();
}

class Truck implements Transport {
    @Override
    public void deliver() {
        System.out.println("Delivering cargo by land in a truck.");
    }
}

class Ship implements Transport {
    @Override
    public void deliver() {
        System.out.println("Delivering cargo by sea in a ship.");
    }
}

abstract class Logistics {

    public abstract Transport createTransport();
```

```java
    public void planDelivery() {

        Transport transport = createTransport();
        transport.deliver();
    }
}

class RoadLogistics extends Logistics {
    @Override
    public Transport createTransport() {
        return new Truck();
    }
}

class SeaLogistics extends Logistics {
    @Override
    public Transport createTransport() {
        return new Ship();
    }
}

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

        Logistics roadLogistics = new RoadLogistics();
        roadLogistics.planDelivery();


        Logistics seaLogistics = new SeaLogistics();
        seaLogistics.planDelivery();
    }
}
```

OUTPUT:



```
D:\00.RUET ALL\RUET academics\semester 3-2\CSE 3206\Lab_03_assignment>
wCodeDetailsInExceptionMessages -cp C:\Users\ASUS\AppData\Roaming\Code
t.java\jdt_ws\Lab_03_assignment_76c7144d\bin Problem1.LogisticsApp "
Delivering cargo by land in a truck.
Delivering cargo by sea in a ship.
```

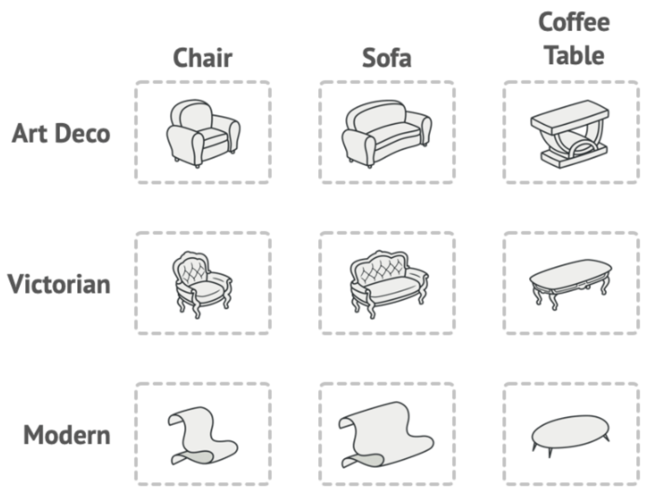
Problem 02:

☹ Problem

Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: `Chair` + `Sofa` + `CoffeeTable`.

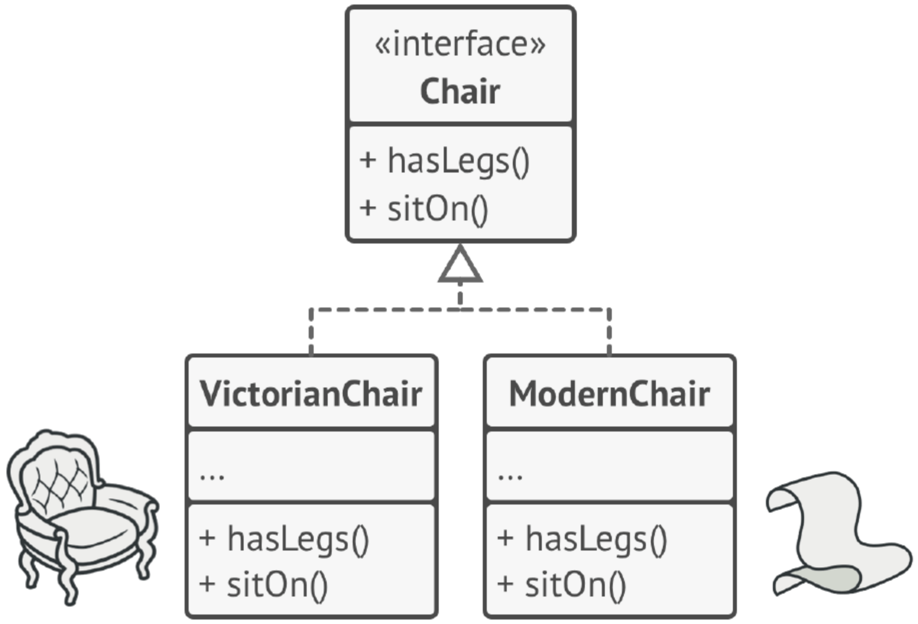
2. Several variants of this family. For example, products `Chair` + `Sofa` + `CoffeeTable` are available in these variants: `Modern`, `Victorian`, `ArtDeco`.



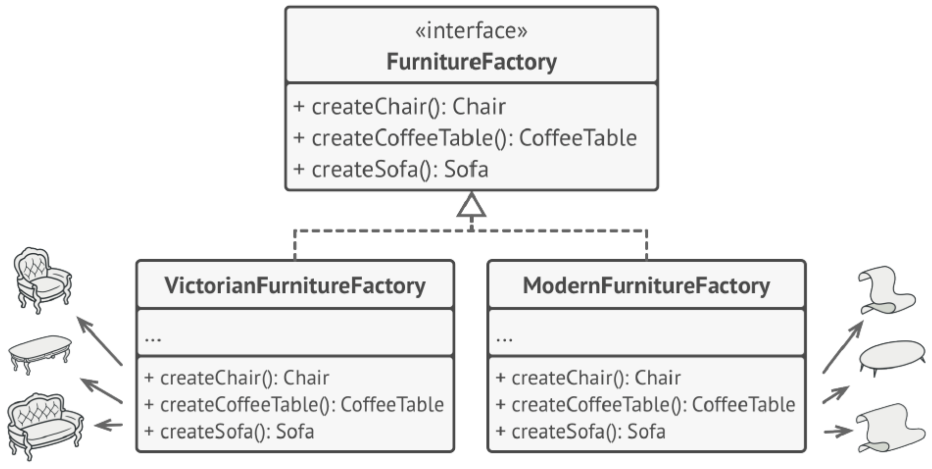
Product families and their variants.

You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

UML Diagrams:



All variants of the same object must be moved to a single class hierarchy.

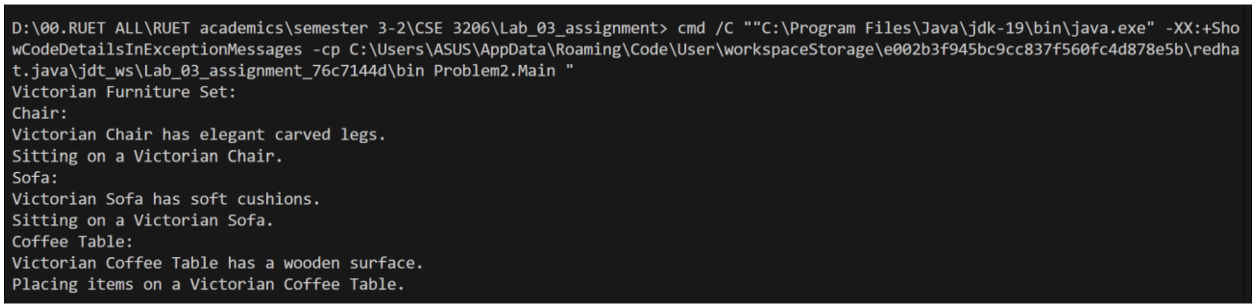


Each concrete factory corresponds to a specific product variant.

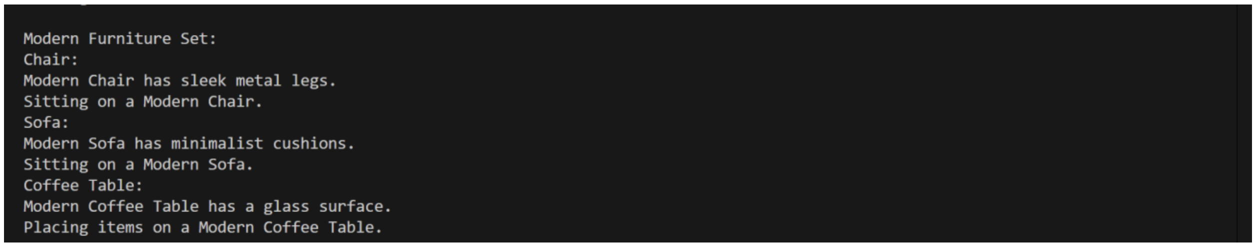
Code:

```java
package Problem2;


interface Chair {
    void hasLegs();
    void sitOn();
}

interface Sofa {
    void hasCushions();
    void sitOn();
}

interface CoffeeTable {
    void hasSurface();
    void placeItemsOn();
}


class VictorianChair implements Chair {
    public void hasLegs() {
        System.out.println("Victorian Chair has elegant carved legs.");
    }

    public void sitOn() {
        System.out.println("Sitting on a Victorian Chair.");
    }
}

class VictorianSofa implements Sofa {
    public void hasCushions() {
        System.out.println("Victorian Sofa has soft cushions.");
    }

    public void sitOn() {
        System.out.println("Sitting on a Victorian Sofa.");
    }
}

class VictorianCoffeeTable implements CoffeeTable {
    public void hasSurface() {
        System.out.println("Victorian Coffee Table has a wooden surface.");
    }
    public void placeItemsOn() {
        System.out.println("Placing items on a Victorian Coffee Table.");
    }
}

class ModernChair implements Chair {
    public void hasLegs() {
        System.out.println("Modern Chair has sleek metal legs.");
    }

    public void sitOn() {
        System.out.println("Sitting on a Modern Chair.");
    }
}

class ModernSofa implements Sofa {
    public void hasCushions() {
        System.out.println("Modern Sofa has minimalist cushions.");
    }

    public void sitOn() {
        System.out.println("Sitting on a Modern Sofa.");
    }
}

class ModernCoffeeTable implements CoffeeTable {
    public void hasSurface() {
        System.out.println("Modern Coffee Table has a glass surface.");
    }

    public void placeItemsOn() {
        System.out.println("Placing items on a Modern Coffee Table.");
    }
}

interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
    CoffeeTable createCoffeeTable();
}

class VictorianFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new VictorianChair();
    }
    public Sofa createSofa() {
        return new VictorianSofa();
    }

    public CoffeeTable createCoffeeTable() {
        return new VictorianCoffeeTable();
    }
}

class ModernFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ModernChair();
    }

    public Sofa createSofa() {
        return new ModernSofa();
    }

    public CoffeeTable createCoffeeTable() {
        return new ModernCoffeeTable();
    }
}

class FurnitureShop {
    private Chair chair;
    private Sofa sofa;
    private CoffeeTable coffeeTable;

    public FurnitureShop(FurnitureFactory factory) {
        chair = factory.createChair();
        sofa = factory.createSofa();
        coffeeTable = factory.createCoffeeTable();
    }

    public void showFurniture() {
        System.out.println("Chair: ");
        chair.hasLegs();
        chair.sitOn();
        System.out.println("Sofa: ");
        sofa.hasCushions();
        sofa.sitOn();
        System.out.println("Coffee Table: ");
        coffeeTable.hasSurface();
        coffeeTable.placeItemsOn();
    }
}

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

        System.out.println("Victorian Furniture Set:");
        FurnitureShop victorianShop = new FurnitureShop(new
VictorianFurnitureFactory());
        victorianShop.showFurniture();

        System.out.println("\nModern Furniture Set:");
        FurnitureShop modernShop = new FurnitureShop(new
ModernFurnitureFactory());
        modernShop.showFurniture();
    }
}
```

OUTPUT:



```
D:\00.RUET ALL\RUET academics\semester 3-2\CSE 3206\Lab_03_assignment> cmd /C ""C:\Program Files\Java\jdk-19\bin\java.exe" -XX:+Sho
wCodeDetailsInExceptionMessages -cp C:\Users\ASUS\AppData\Roaming\Code\User\workspaceStorage\e002b3f945bc9cc837f560fc4d878e5b\redha
t.java\jdt_ws\Lab_03_assignment_76c7144d\bin Problem2.Main "
Victorian Furniture Set:
Chair:
Victorian Chair has elegant carved legs.
Sitting on a Victorian Chair.
Sofa:
Victorian Sofa has soft cushions.
Sitting on a Victorian Sofa.
Coffee Table:
Victorian Coffee Table has a wooden surface.
Placing items on a Victorian Coffee Table.
```

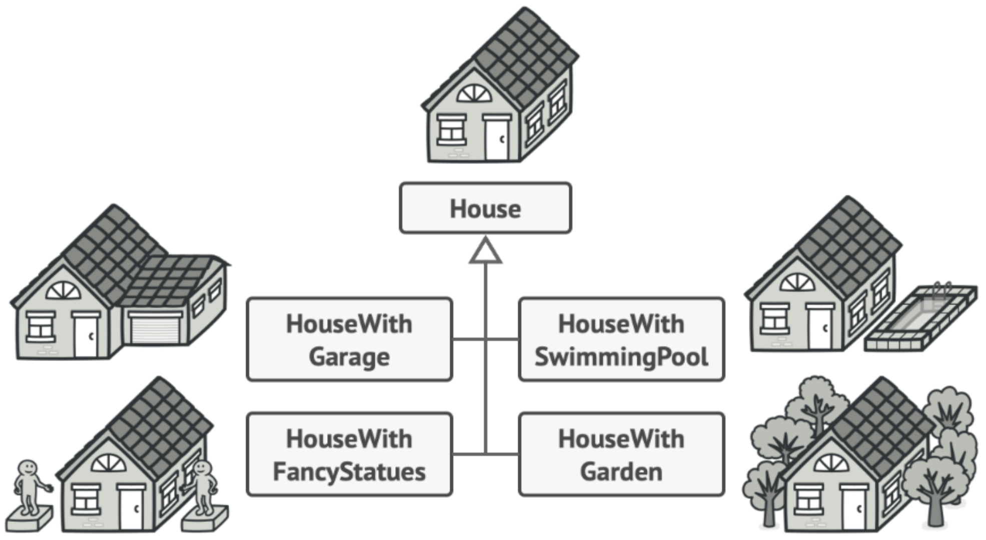


```
Modern Furniture Set:
Chair:
Modern Chair has sleek metal legs.
Sitting on a Modern Chair.
Sofa:
Modern Sofa has minimalist cushions.
Sitting on a Modern Sofa.
Coffee Table:
Modern Coffee Table has a glass surface.
Placing items on a Modern Coffee Table.
```

Problem 03:

☹ Problem

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.



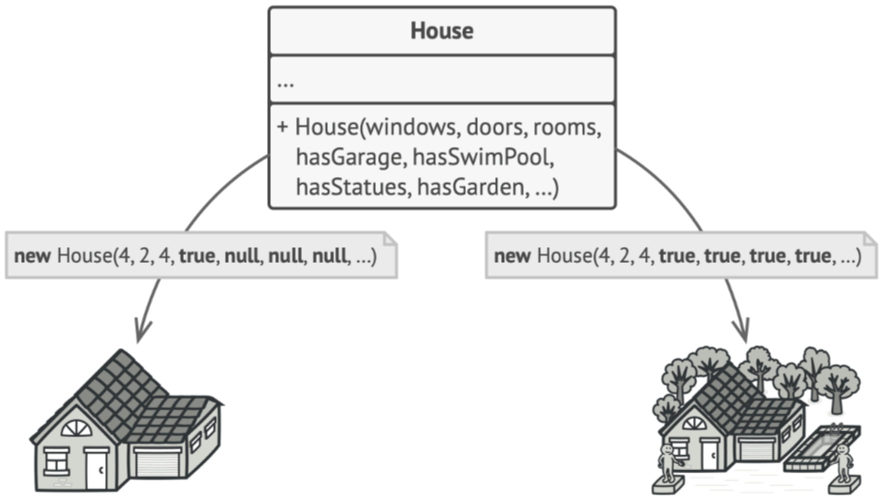
You might make the program too complex by creating a subclass for every possible configuration of an object.

For example, let's think about how to create a `House` object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

The simplest solution is to extend the base `House` class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

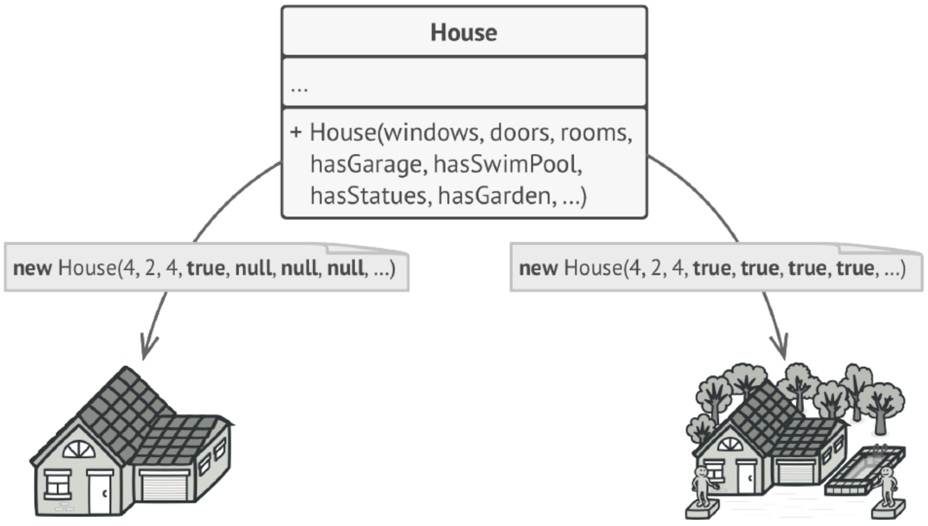
UML Diagrams:

There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base `House` class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

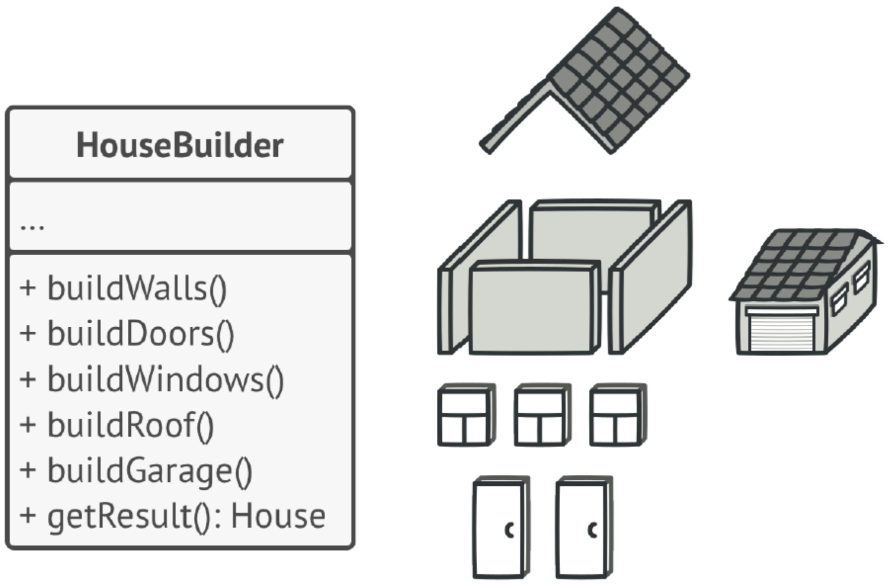

House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

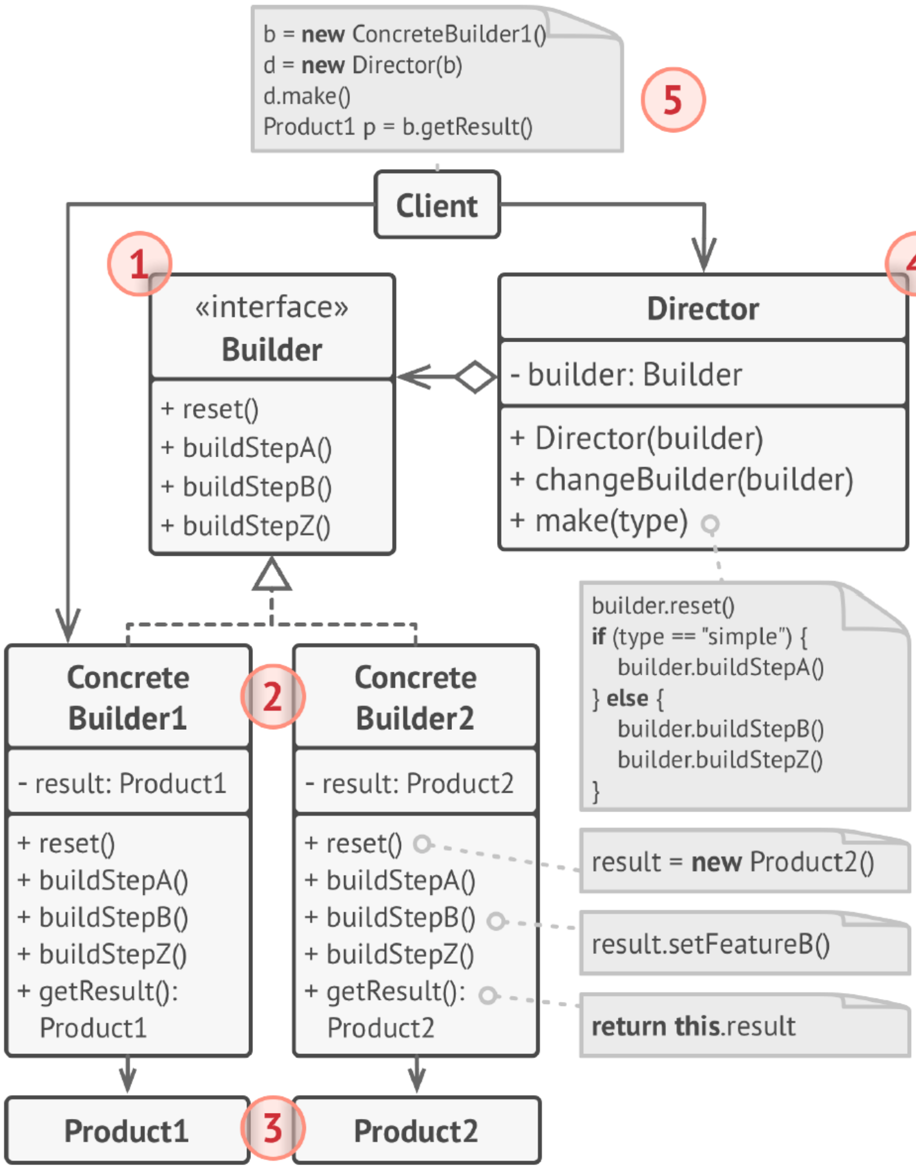
In most cases most of the parameters will be unused, making **the constructor calls pretty ugly**. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

The constructor with lots of parameters has its downside: not all the parameters are needed at all times.



The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.



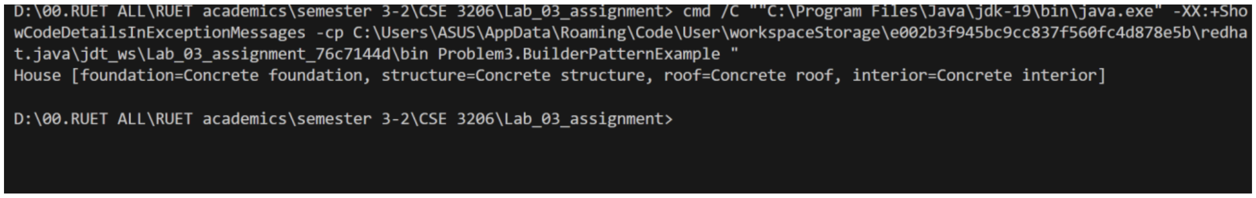
Code:

```
package Problem3;

class House {
    private String foundation;
    private String structure;
    private String roof;
```

```java
    private String interior;

    public void setFoundation(String foundation) {
        this.foundation = foundation;
    }

    public void setStructure(String structure) {
        this.structure = structure;
    }

    public void setRoof(String roof) {
        this.roof = roof;
    }

    public void setInterior(String interior) {
        this.interior = interior;
    }

    @Override
    public String toString() {
        return "House [foundation=" + foundation + ", structure=" + structure +
", roof=" + roof + ", interior=" + interior + "]";
    }
}

interface HouseBuilder {
    void buildFoundation();
    void buildStructure();
    void buildRoof();
    void buildInterior();
    House getHouse();
}

class ConcreteHouseBuilder implements HouseBuilder {
    private House house;

    public ConcreteHouseBuilder() {
        this.house = new House();
    }

    @Override
    public void buildFoundation() {
        house.setFoundation("Concrete foundation");
    }
    @Override
    public void buildStructure() {
        house.setStructure("Concrete structure");
    }

    @Override
    public void buildRoof() {
        house.setRoof("Concrete roof");
    }

    @Override
    public void buildInterior() {
        house.setInterior("Concrete interior");
    }

    @Override
    public House getHouse() {
        return house;
    }
}

class Engineer {
    private HouseBuilder houseBuilder;

    public Engineer(HouseBuilder houseBuilder) {
        this.houseBuilder = houseBuilder;
    }

    public House constructHouse() {
        houseBuilder.buildFoundation();
        houseBuilder.buildStructure();
        houseBuilder.buildRoof();
        houseBuilder.buildInterior();
        return houseBuilder.getHouse();
    }
}

// Main class
public class BuilderPatternExample {
    public static void main(String[] args) {
        HouseBuilder concreteHouseBuilder = new ConcreteHouseBuilder();
        Engineer engineer = new Engineer(concreteHouseBuilder);
        House house = engineer.constructHouse();
        System.out.println(house);
    }
```

```
}
```

OUTPUT:



```
D:\00.RUET ALL\RUET academics\semester 3-2\CSE 3206\Lab_03_assignment> cmd /C ""C:\Program Files\Java\jdk-19\bin\java.exe" -XX:+Sho
wCodeDetailsInExceptionMessages -cp C:\Users\ASUS\AppData\Roaming\Code\User\workspaceStorage\e002b3f945bc9cc837f560fc4d878e5b\redha
t.java\jdt_ws\Lab_03_assignment_76c7144d\bin Problem3.BuilderPatternExample "
House [foundation=Concrete foundation, structure=Concrete structure, roof=Concrete roof, interior=Concrete interior]

D:\00.RUET ALL\RUET academics\semester 3-2\CSE 3206\Lab_03_assignment>
```