

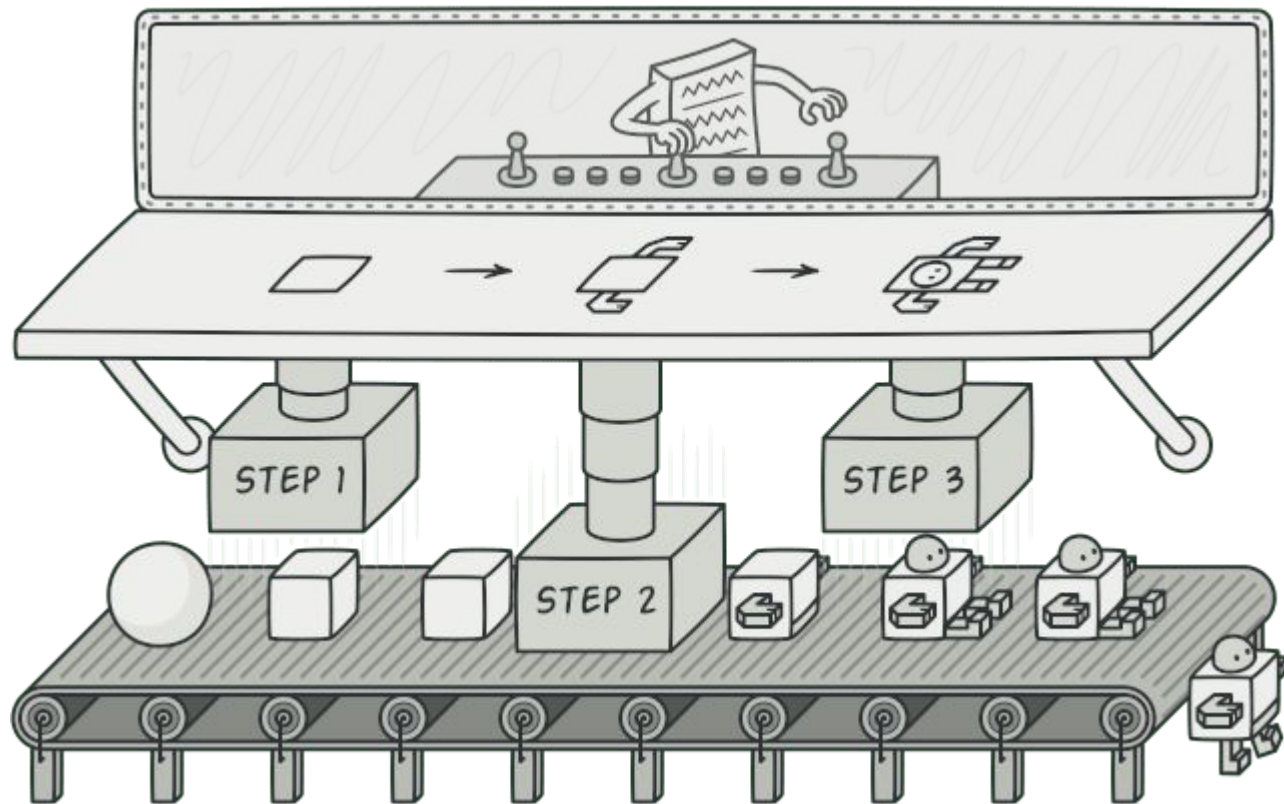
Builder Pattern

Builder Pattern

- Builder pattern builds a complex object using simple objects and using a step by step approach.
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- A Builder class builds the final object step by step. This builder is independent of other objects.
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder Pattern

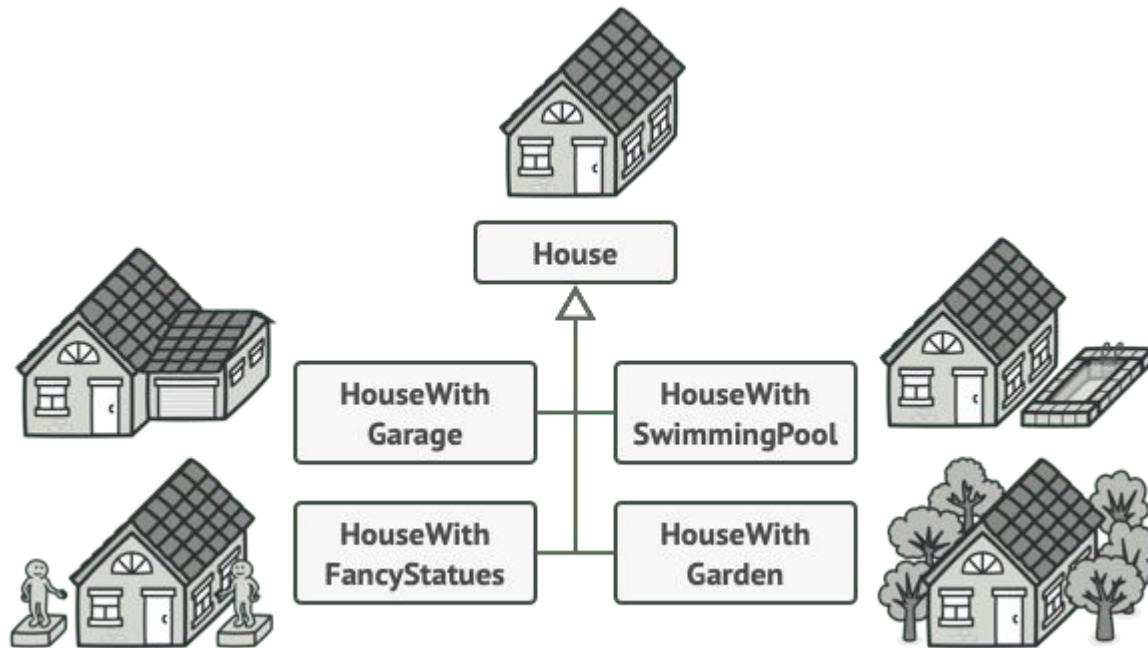
The Builder pattern lets you build objects step by step, using only those steps that you really need.



Builder Pattern

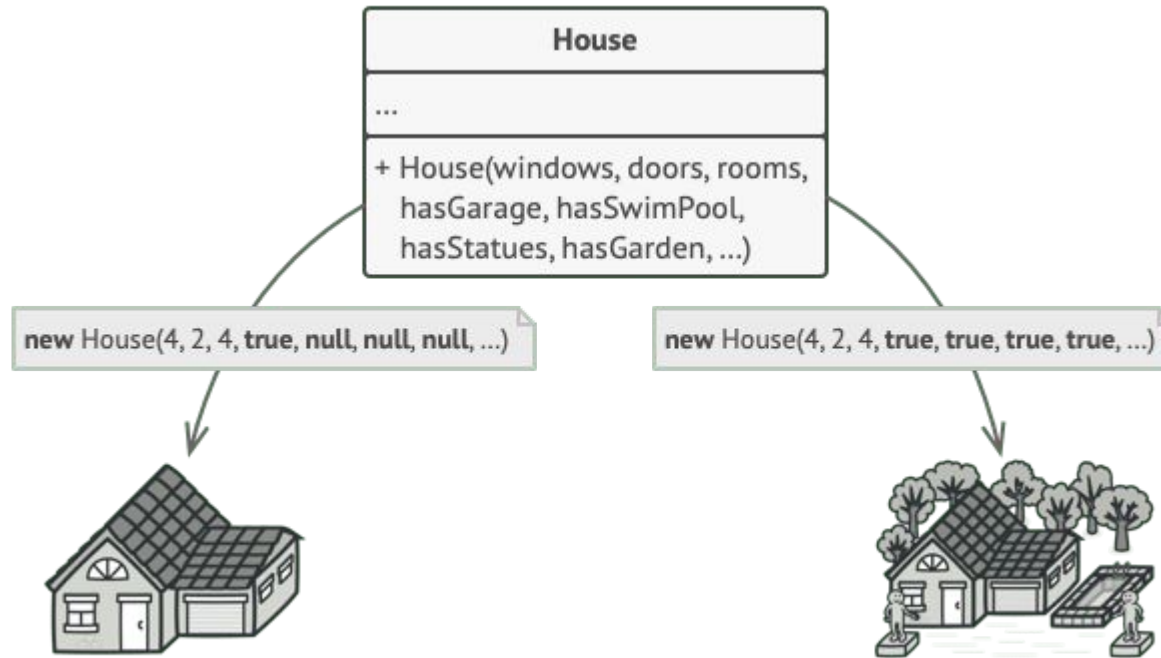
Problem

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects



Builder Pattern

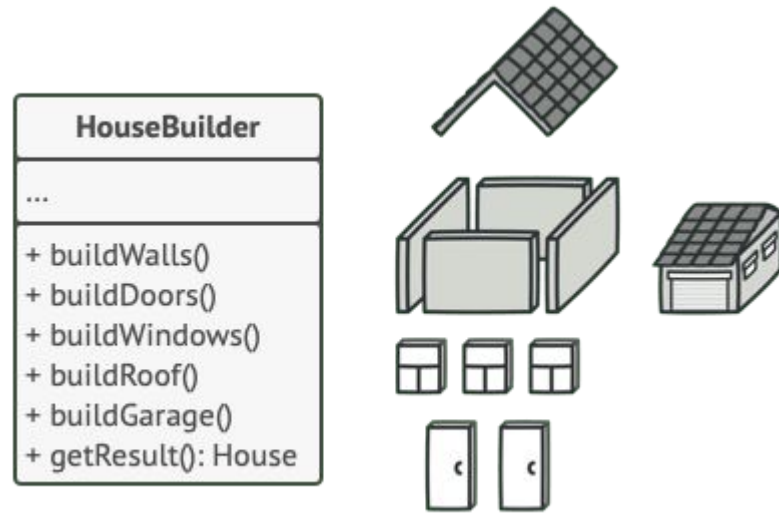
- In most cases most of the parameters will be unused



Builder Pattern

Solution

The Builder pattern suggests that, extract the object construction code out of its own class and move it to separate objects called *builders*



Builder Pattern

Item.java

```
public interface Item {  
    public String name();  
    public Packing packing();  
    public float price();  
}
```

Burger.java

```
public abstract class Burger implements Item {  
  
    @Override  
    public Packing packing() {  
        return new Wrapper();  
    }  
  
    @Override  
    public abstract float price();  
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {  
  
    @Override  
    public Packing packing() {  
        return new Bottle();  
    }  
  
    @Override  
    public abstract float price();  
}
```

Builder Pattern

VegBurger.java

```
public class VegBurger extends Burger {  
  
    @Override  
    public float price() {  
        return 25.0f;  
    }  
  
    @Override  
    public String name() {  
        return "Veg Burger";  
    }  
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {  
  
    @Override  
    public float price() {  
        return 50.5f;  
    }  
  
    @Override  
    public String name() {  
        return "Chicken Burger";  
    }  
}
```


Builder Pattern

Coke.java

```
public class Coke extends ColdDrink {  
  
    @Override  
    public float price() {  
        return 30.0f;  
    }  
  
    @Override  
    public String name() {  
        return "Coke";  
    }  
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {  
  
    @Override  
    public float price() {  
        return 35.0f;  
    }  
  
    @Override  
    public String name() {  
        return "Pepsi";  
    }  
}
```

Builder Pattern

Packing.java

```
public interface Packing {  
    public String pack();  
}
```

Wrapper.java

```
public class Wrapper implements Packing {  
  
    @Override  
    public String pack() {  
        return "Wrapper";  
    }  
}
```

Bottle.java

```
public class Bottle implements Packing {  
  
    @Override  
    public String pack() {  
        return "Bottle";  
    }  
}
```

Builder Pattern

Meal.java

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

public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item){
        items.add(item);
    }

    public float getCost(){
        float cost = 0.0f;

        for (Item item : items) {
            cost += item.price();
        }
        return cost;
    }

    public void showItems(){
        for (Item item : items) {
            System.out.print("Item : " + item.name());
            System.out.print(", Packing : " + item.packing().pack());
            System.out.println(", Price : " + item.price());
        }
    }
}
```

Builder Pattern

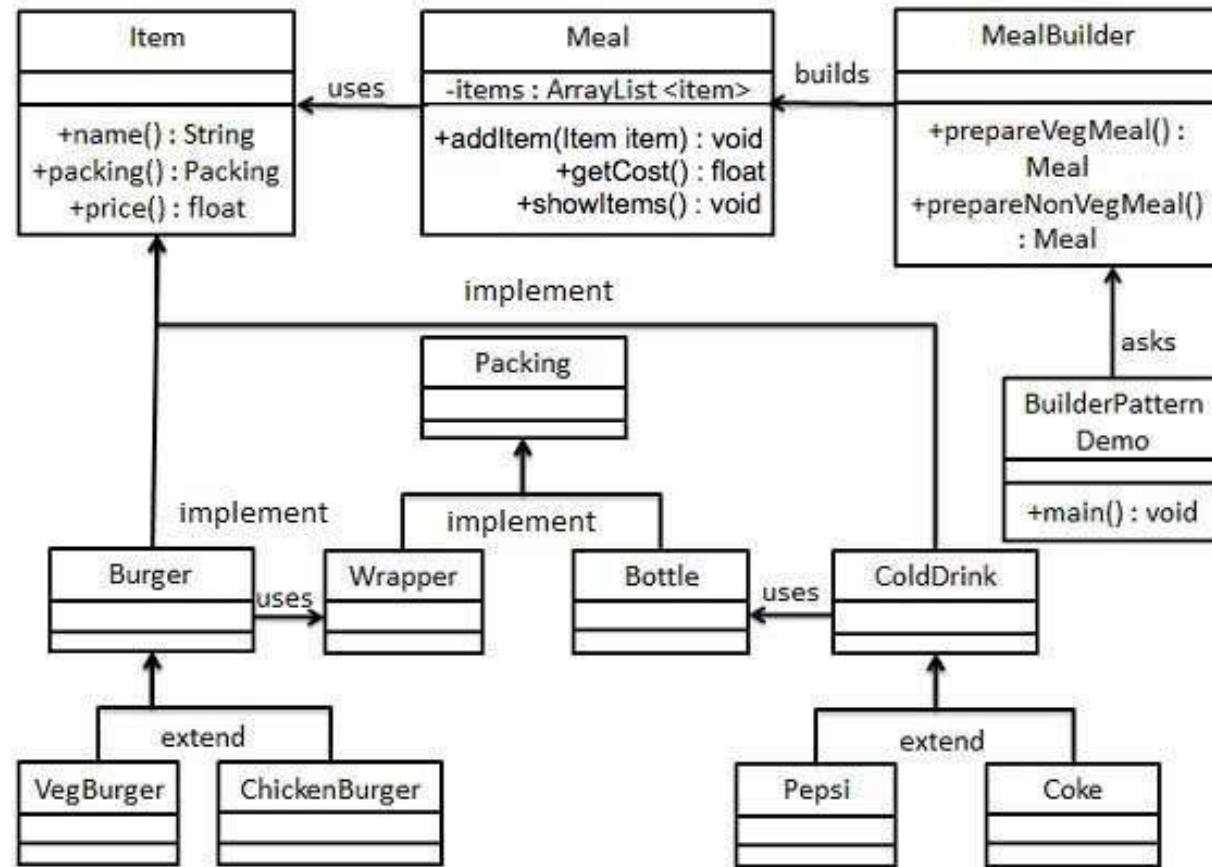
MealBuilder.java

```
public class MealBuilder {  
  
    public Meal prepareVegMeal () {  
        Meal meal = new Meal();  
        meal.addItem(new VegBurger());  
        meal.addItem(new Coke());  
        return meal;  
    }  
  
    public Meal prepareNonVegMeal () {  
        Meal meal = new Meal();  
        meal.addItem(new ChickenBurger());  
        meal.addItem(new Pepsi());  
        return meal;  
    }  
}
```

BuilderPatternDemo.java

```
public class BuilderPatternDemo {  
    public static void main(String[] args) {  
  
        MealBuilder mealBuilder = new MealBuilder();  
  
        Meal vegMeal = mealBuilder.prepareVegMeal();  
        System.out.println("Veg Meal");  
        vegMeal.showItems();  
        System.out.println("Total Cost: " + vegMeal.getCost());  
  
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();  
        System.out.println("\n\nNon-Veg Meal");  
        nonVegMeal.showItems();  
        System.out.println("Total Cost: " + nonVegMeal.getCost());  
    }  
}
```

UML Diagram of Builder Pattern



H/W

- Which patterns are related or similar to Builder?
- Difference between Factory and Builder?
- Think some real life scenario where builder pattern can be used?

Composite Pattern

- Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.
- work with these structures as if they were individual objects.
- Composite creates Parent - Child relations between objects while Builder is used to create group of objects of predefined types.