COMP2511

Creational Pattern:
Builder Pattern

Prepared by

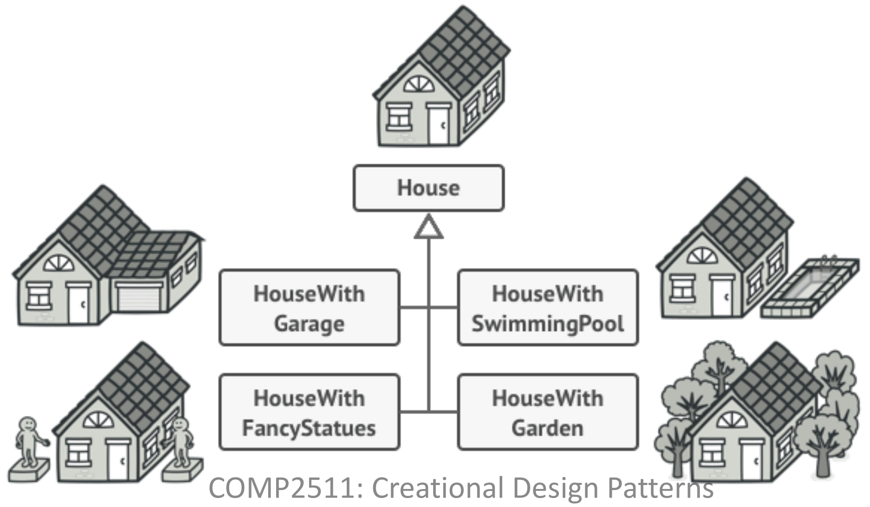
Dr. Ashesh Mahidadia

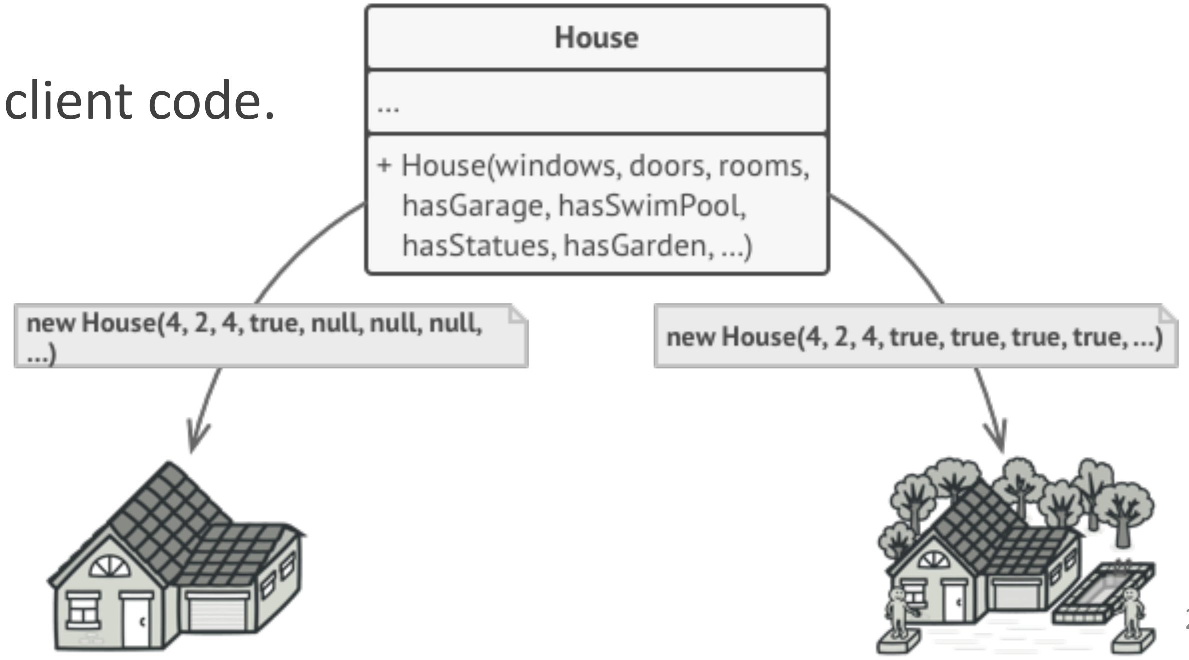
Builder Pattern

Intent: Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem:

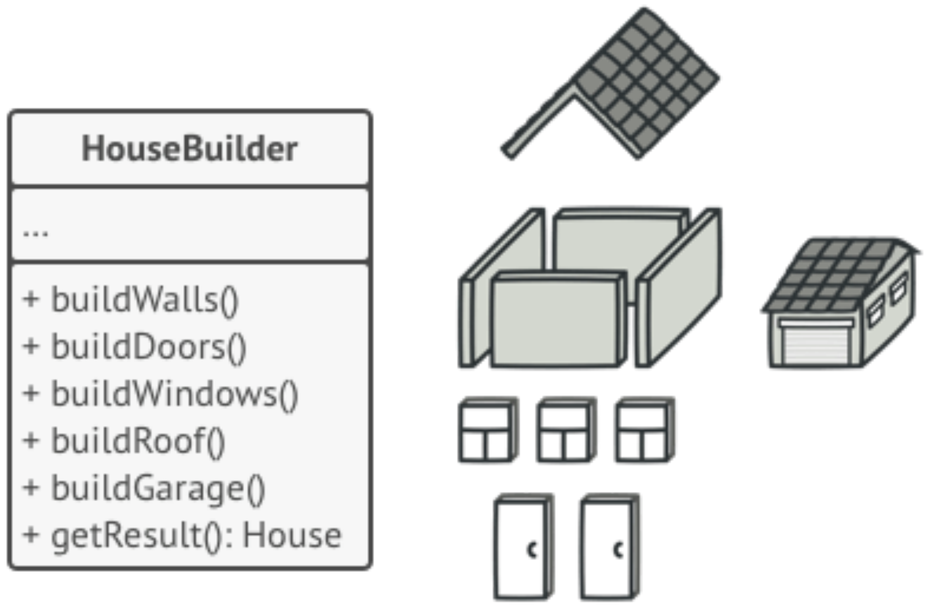
- Imagine a complex object that requires laborious, step-by-step initialization/construction of many fields and nested objects.
- Such initialization/construction code is usually buried inside a monstrous constructor with lots of parameters.
- Or even worse: scattered all over the client code.




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, ...)

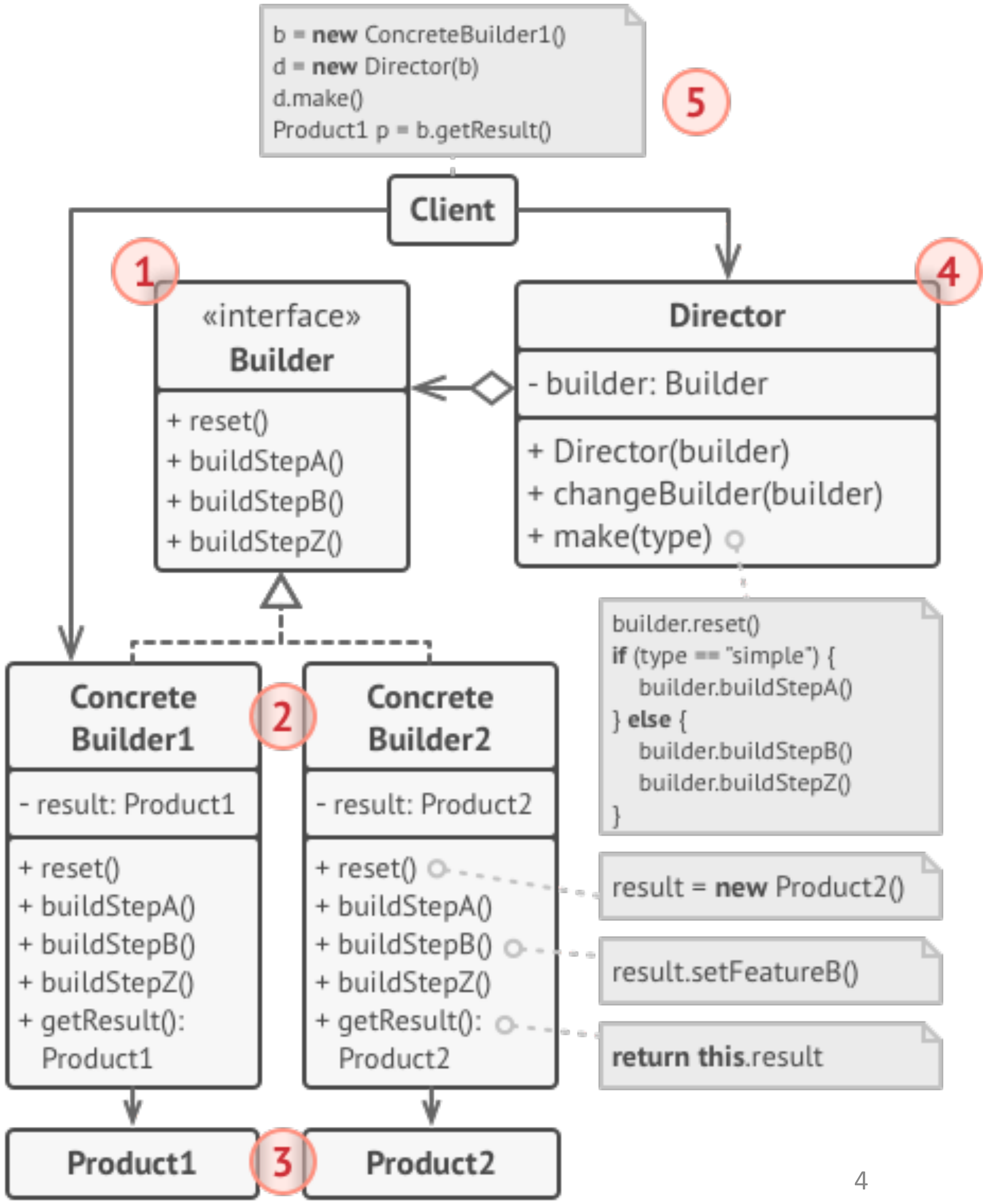
Builder Pattern

- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called **builders**.
- 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.
- **Director**: The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



Builder Pattern: Structure

- The **Builder** interface declares product construction steps that are common to all types of builders.
- **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
- **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
- The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- The **Client** must associate one of the builder objects with the director.



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

```
builder.reset()
if (type == "simple") {
   builder.buildStepA()
} else {
   builder.buildStepB()
   builder.buildStepZ()
}
```

```
result = new Product2()
```

```
result.setFeatureB()
```

```
return this.result
```

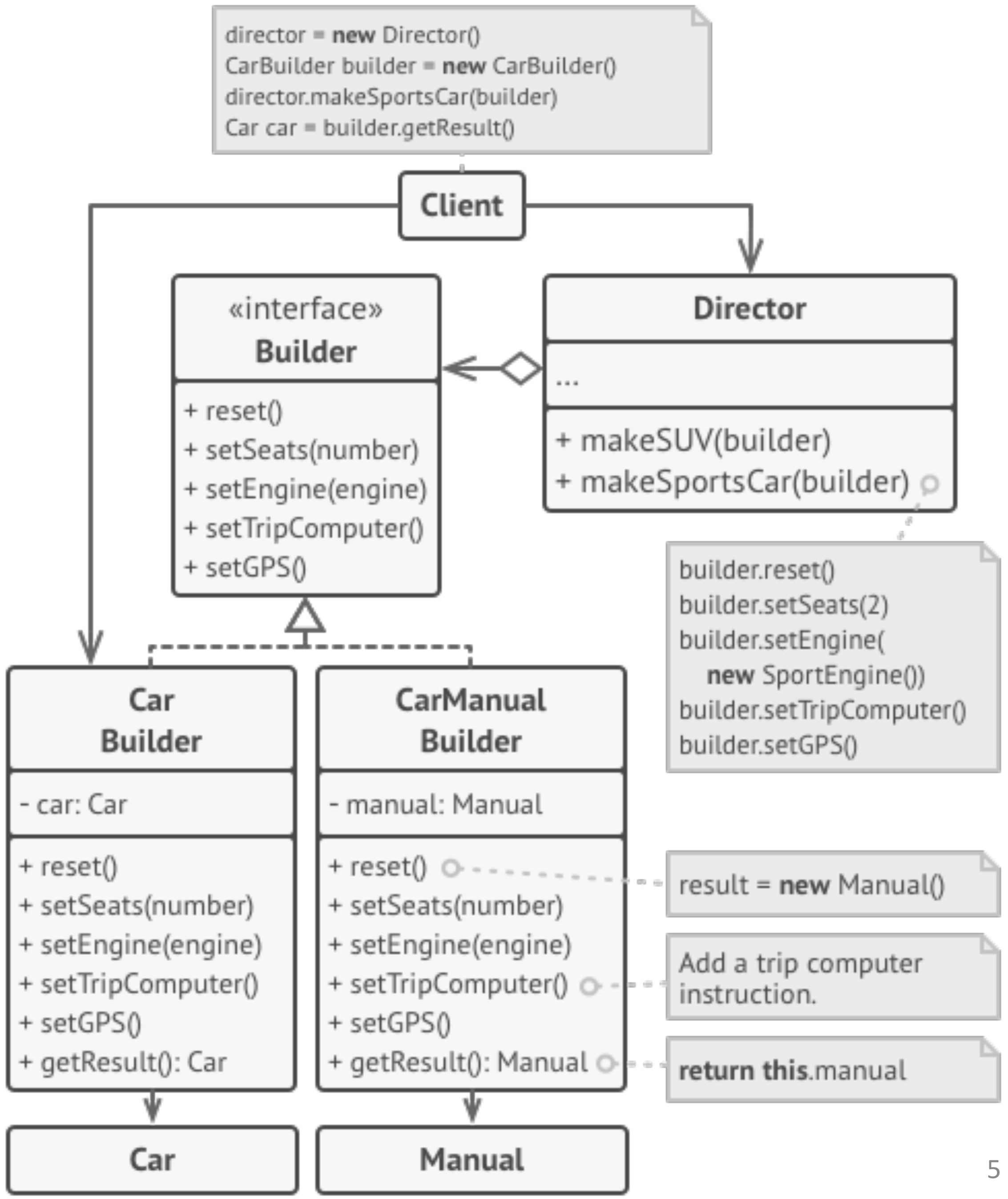
Builder Pattern: Example

This example illustrates how you can reuse the same object construction code when,

- ❖ building different types of cars, and
- ❖ creating the corresponding manuals for them.

Example in Java **(MUST read)**:

https://refactoring.guru/design-patterns/builder/java/example



Relations with Other Patterns

- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, or Builder (more flexible, but more complicated).
- Builder focuses on constructing complex objects step by step.
- Abstract Factory specializes in creating families of related objects.
- Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.

Builder Pattern

For more information, read:

https://refactoring.guru/design-patterns/builder

End