Design Patterns Builder

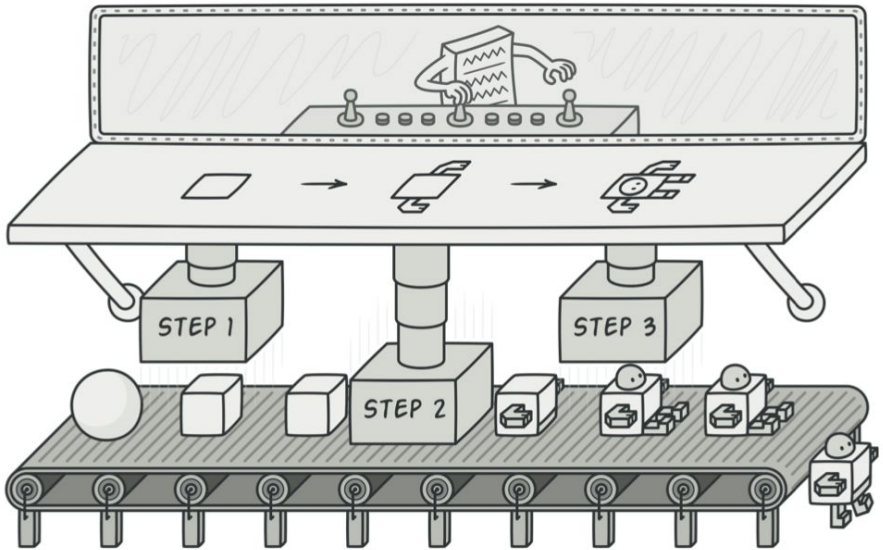
...

Announcements

- Due Date of Project 1
  - April 24 at midnight
  - This ensures we will have time to get through all design patterns
  - But you can start now with the Features List! In fact, I recommend it :)
- See Class Schedule for up-to-date lecture schedule

Builder Design Pattern

- Creational design pattern
- Lets you construct complex objects step by step.
- Allows you to produce different types and representations of an object using the same construction code.



Builder: Problem

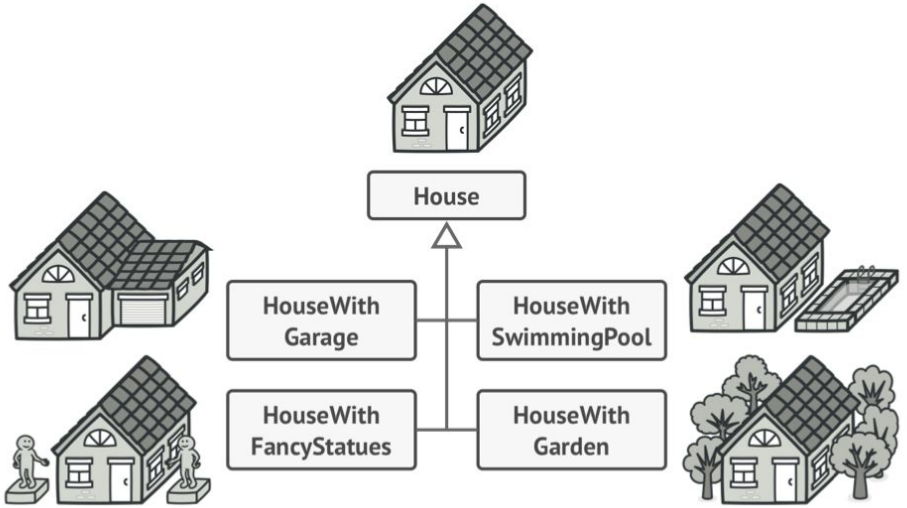
- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.
- Such initialization code is usually buried inside a monstrous constructor with lots of parameters.
- Or even worse: scattered all over the client code.

Builder: Problem

- 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)?

Builder: Problem

- 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.



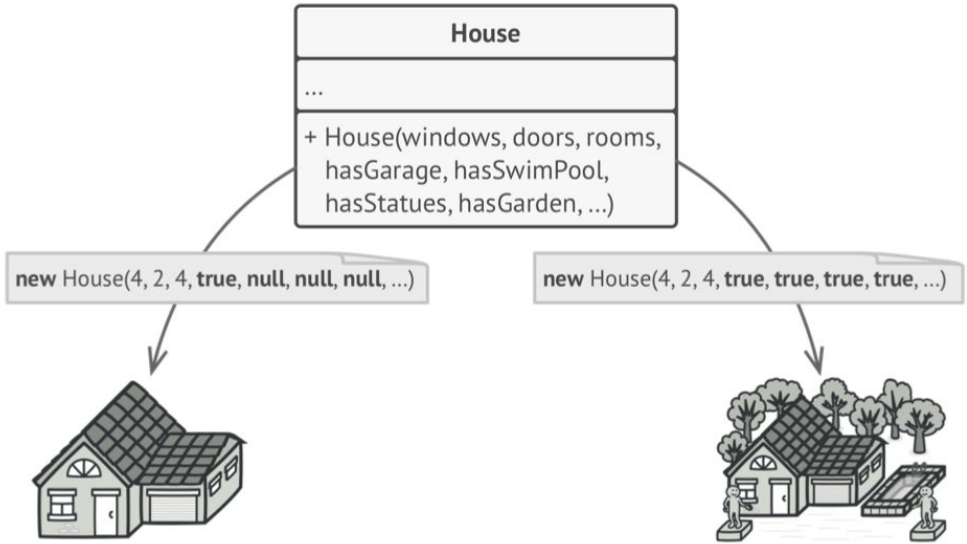
You might make the program too complex by creating a subclass for every possible configuration of an object.

Builder: Problem

- There’s another approach that doesn’t involve creating 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.

Builder: Problem

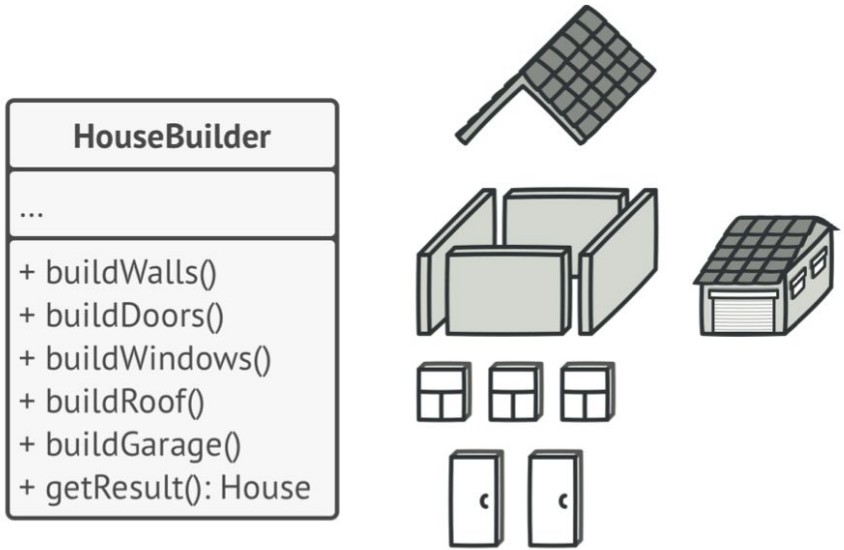
- In most cases most of the parameters will be unused, making the constructor calls pretty ugly.
- Only a fraction of houses have swimming pools, so the parameters related to pools will usually be useless
  - Example: a BERT model



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

Builder: Solution

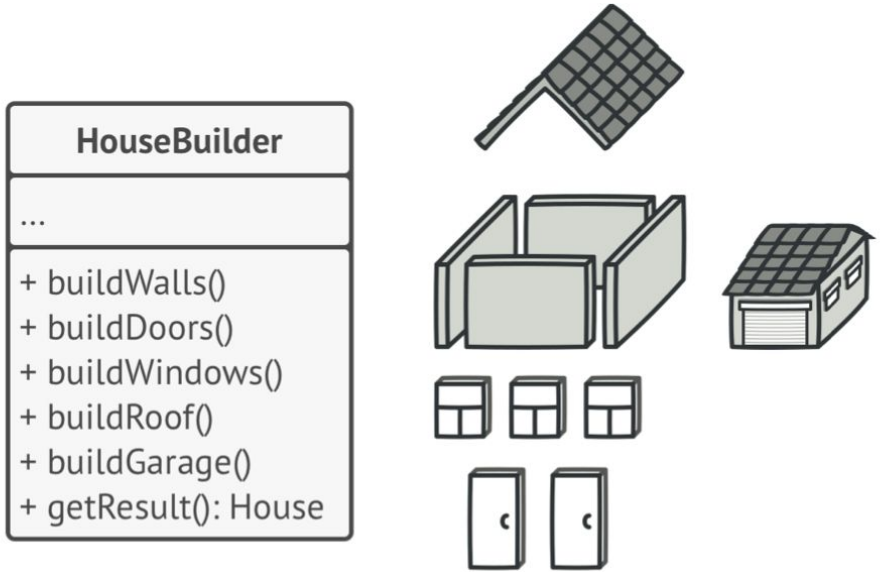
- The Builder pattern:
  - Extract the object construction code out of its own class
  - Move it to separate object, called a Builder



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.

Builder: Solution

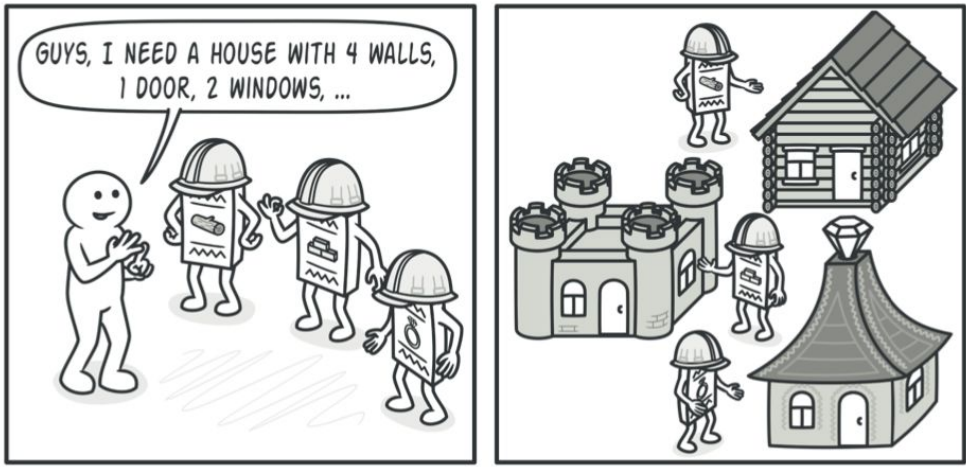
- The pattern organizes object construction into a set of steps
  - build_walls, build_door , etc.
- To create an object, you execute a series of these steps on a builder object.
- You don't need to call all of the steps
  - You can call **only** those steps that are necessary for producing a particular configuration of an object.



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.

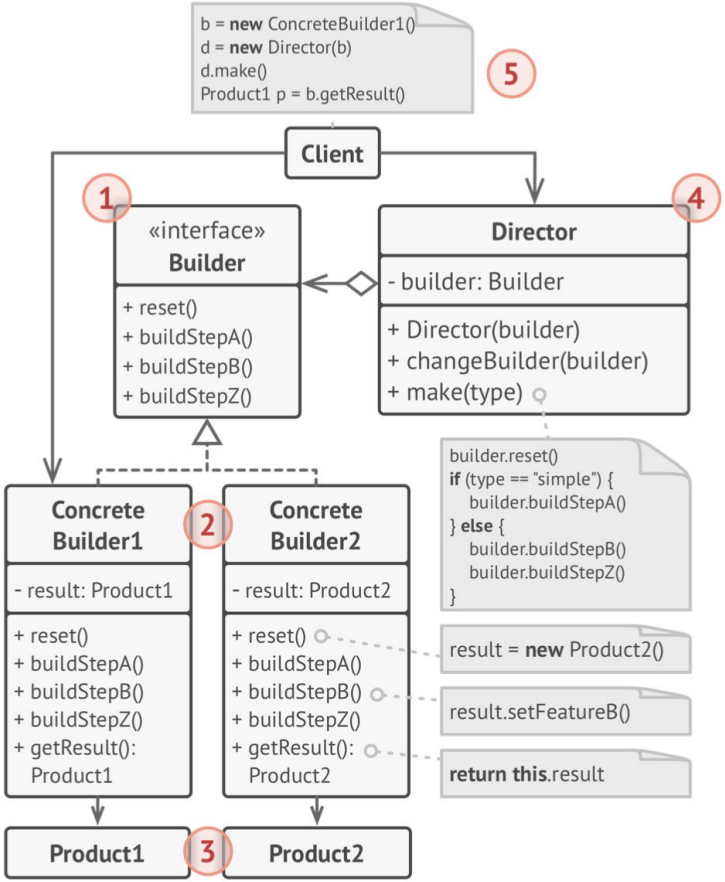
Builder: More Advanced Solution

- Some of the construction steps might require different implementation when you need to build various representations of the product.
  - For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.
- In this case, you can create several different builder classes
  - implement the same set of building steps, but in a different manner.
- Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.



Different builders execute the same task in various ways.

Builder: Structure



Builder: Parts

1. The Builder interface declares product construction steps that are common to all types of builders.

Builder: Parts

2. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don’t follow the common interface.

Builder: Parts

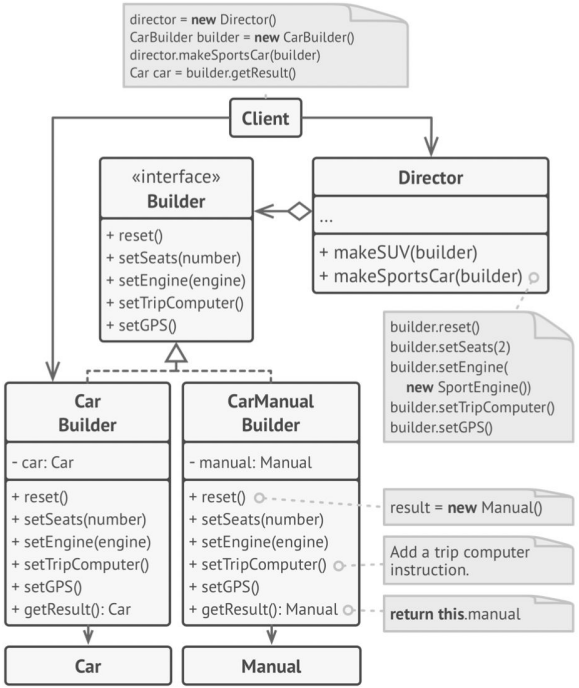
3. Products are resulting objects. Products constructed by different builders don’t have to belong to the same class hierarchy or interface.

Builder: Concrete Example

We want to create a CarBuilder.

Want to ensure that the CarManual and the Car correspond to each other

Builder: Concrete Example



The example of step-by-step construction of cars and the user guides that fit those car models.

When to Use Builder Pattern

Use the Builder pattern to get rid of a “telescopic constructor”.

If you have a giant constructor (init method) with a lot of specific arguments, you can use a builder instead!

When to Use the builder pattern

- Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).
- The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.
- A real-world example of this is varied machine learning pipelines!

Builder: Pros

- You can construct objects step-by-step, defer construction steps, or run steps recursively.
- You can reuse the same construction code when building various representations.
- Single Responsibility Principle.
    - You can isolate complex construction code from the business logic of the product.

Builder: Cons

- The overall complexity of the code increases
- Sometimes it’s confusing or unintuitive to have the constructor code live in a different class!

Builder: In-class exercise

You find some [bad] code for a PizzaMaker that looks like this:

```
class Pizza:

    def __init__(size: int, cheese: bool, pepperoni: bool, chile: bool, basil: bool, extra_sauce: bool, ham: bool, extra_cheese: bool, pineapple: bool, mushroom: bool)
```

Change this code into a builder pattern! You will need two classes:

- Pizza
- PizzaBuilder