# The Builder Pattern

## Chapter Content

- The Builder Pattern – Overview
- The Builder Pattern UML Diagram
- Builder Pattern Example
- Builder Pattern Discussion
- Exercise

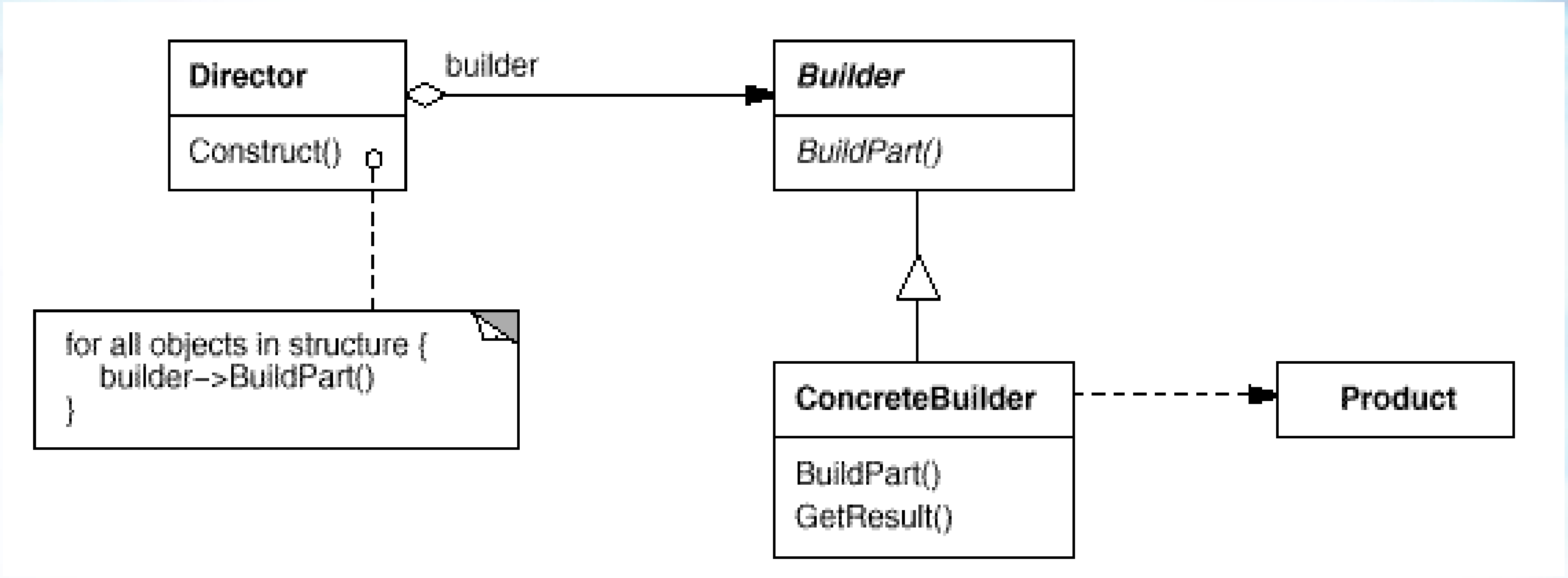
## The Builder Pattern Overview

- When constructing **complex objects**, one may wish to use the same **construction process** (algorithm) regardless of how the complex object is represented internally.

## Builder Pattern Overview (cont.)

- A **Director** decides on a general construction algorithm: What parts to put together.
- The creation of each part is delegated to a **Builder** that decides how each part is represented and assembled with other parts. Note that the assembly depends on the internal representation of the complex object!.

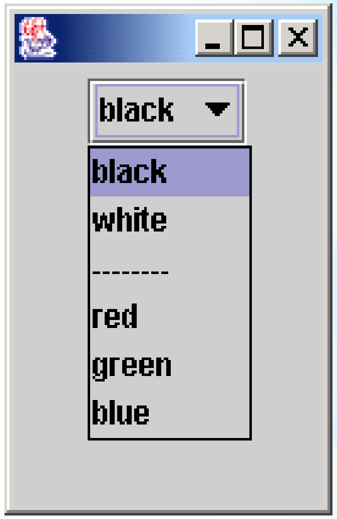
# The Builder Pattern UML Diagram



**Director holds the general algorithm for constructing a complex object. However, the construction & assembly of <u>each</u> part is assigned to the builder, which depends on the internal representation of the constructed complex object.**

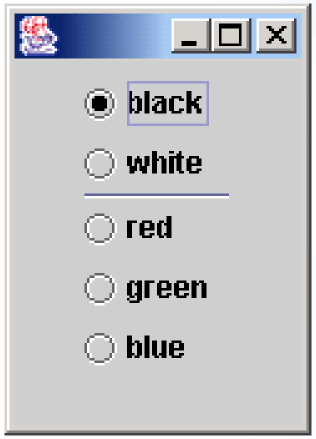
## Builder Pattern Example

```
abstract class Builder {
     public abstract void addPart(String choices);
     public abstract void addSeparator();
     public abstract Component getResult();
}
class ComboBuilder extends Builder {
     private JComboBox combo = new JComboBox();
     public void addPart(String choices){
          combo.addItem(choices);
     }
     public void addSeparator(){
          combo.addItem("--------");
     }
     public Component getResult(){ // Return the Complex object
          return combo;
     }
}
```



# Builder Pattern Example (cont.)

```
class RadioButtonBuilder extends Builder {
    private Box panel = Box.createVerticalBox();
    private ButtonGroup group = new ButtonGroup();
    public void addPart(String choices){
     JRadioButton bt = new JRadioButton(choices);
     group.add(bt);
     panel.add(bt);
    }
    public void addSeparator(){
     panel.add(new JSeparator());
    }
    public Component getResult(){ // Return the Complex object
        // select first radio button
    ((JradioButton)group.getElements().nextElement())
   .setSelected(true);
        return panel;
    }
}
```



## Builder Pattern Example (cont.)

```
class ListDirector {
    public Component create(Builder builder, String[] choices){
         for(String choice : choices){
              if (choice==null)
                        builder.addSeparator();
              else
                        builder.addPart(choice);
         }
         return builder.getResult();
    }
}
Usage:
ListDirector director = new ListDirector();
String[] choices = { "black", "white", null, "red", "green", "blue" };
Component comp = director.create( new RadioButtonBuilder(), choices);
Component comp2 = director.create( new ComboBuilder(), choices);
```

## Builder Pattern Discussion

- **General:**
  - Easy to add new kinds of complex objects.
  - Nice demonstration of isolating the minimal factor that changes, avoiding code duplication.
- **Is it always required?**
  - We wouldn’t need the builder if all complex classes (in our example, ComboBox and RadioButton panel) had a uniform interface for adding a part, such as addPart(String).

## Builder Pattern Discussion (cont.)

- **How does the Builder Pattern differ from the Factory Pattern?**
  - The builder element is similar to a factory-of-parts (Factory Pattern), BUT it has more responsibility – it also assembles the different parts to a single complex object.
  - This means that the builder usually does not return ***independent*** parts for general use.
  - The director element is a private case of the Factory Pattern as it creates complex objects.

## Consequences of the Builder DP

### Consequences of the Builder Pattern

1. Vary the internal representation of the product + hide the details of product assembly.
2. Each builder is independent of the others elements. Improves modularity, easy to add.
3. Step-by-step construction of products – more control.
4. Builder resembles Abstract Factory – Both return classes with methods and objects:
   - **Abstract Factory** returns a <u>family</u> of related classes
   - **Builder** constructs a complex object <u>step by step</u> depending on the  data presented to it.