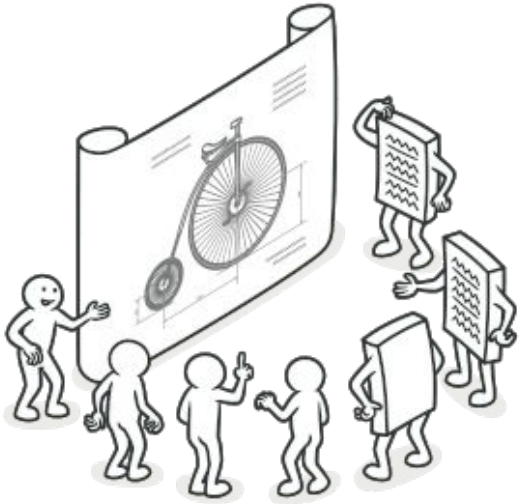
The Builder Design Pattern

Gowtham Rajendra

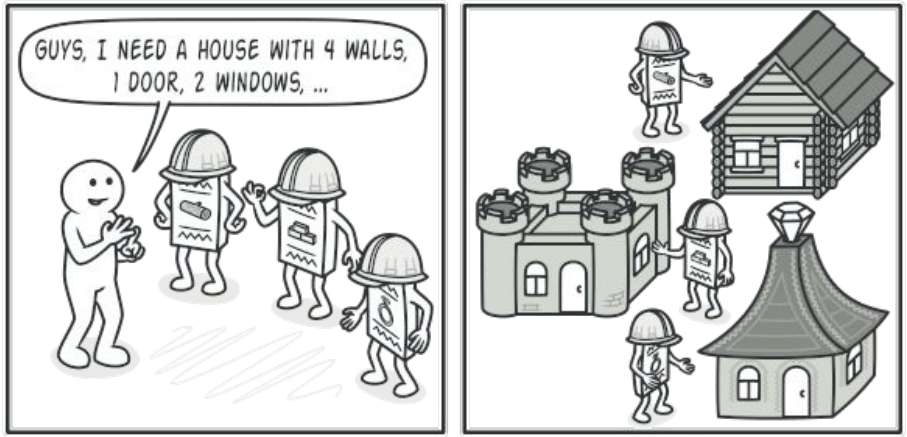
What are Design Patterns?

- Best practices used for software development
  - Commonly used in object-oriented programming
- Acts as a template and offers solutions to common problems during software development
- Three main types of design patterns
  - Creational
  - Structural
  - Behavioral



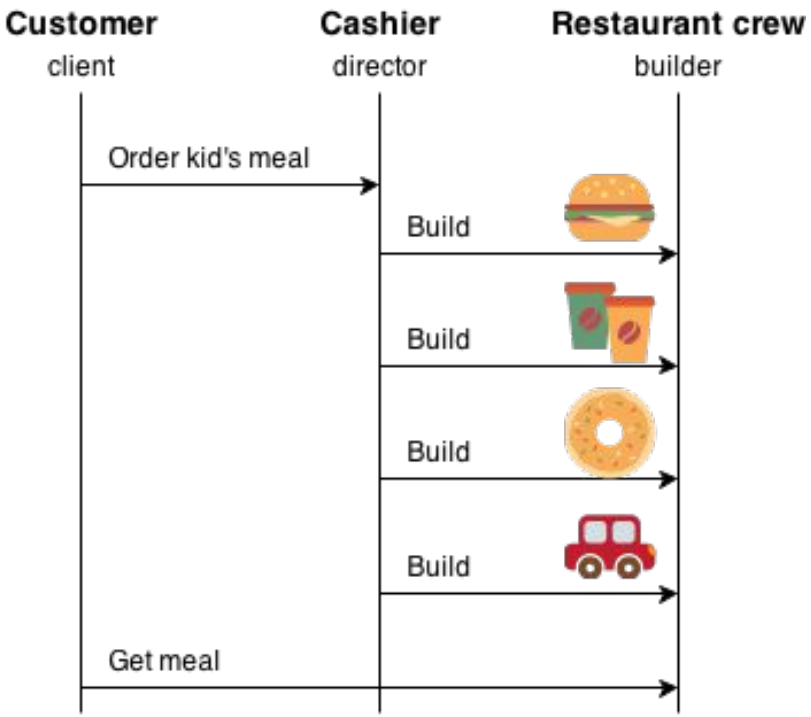
Overview and Usage of the Builder Pattern

- Creational design pattern
- Used to create a complex object that is made up of other smaller objects
  - The complex object can vary while using the same builder code
- Creation of builder objects are independent and is hidden from the client



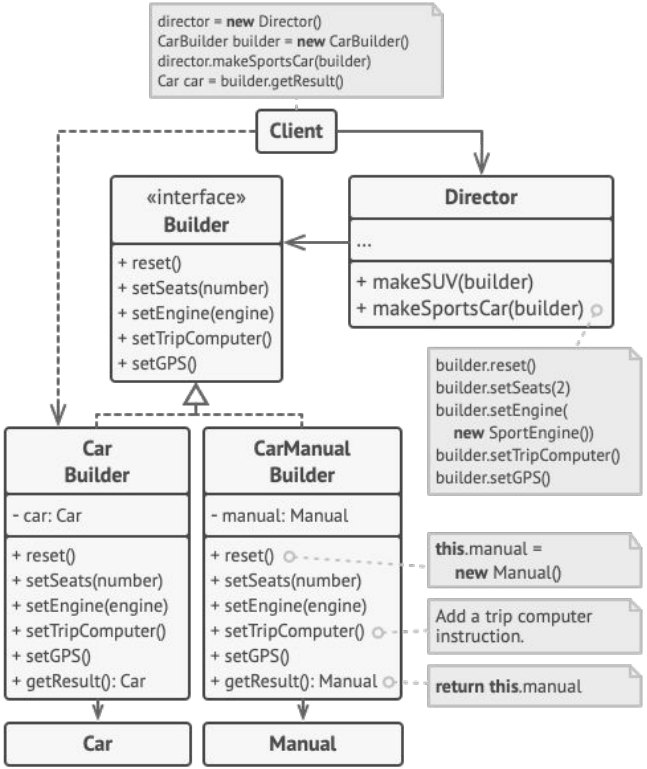
Simple Example of Builder Pattern

- Building a kids meal at a restaurant
- Each item in the meal can differ but building the meal uses the same process
    - E.g. drink can be Pepsi, Coke, Sprite etc.
- The director calls the builder to build each item
- The builder puts the items together and returns the built meal



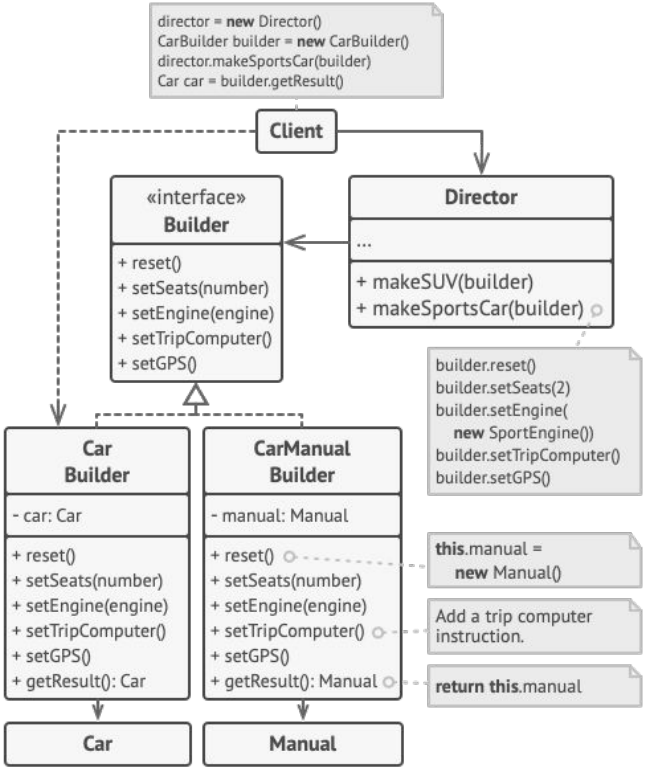
In-depth example of Builder Pattern

- Building a car
- Builder is the main builder
- Car builder and CarManual Builder are the concrete builders that implement the main builder
- Client is the class that calls the director
- Car and Manual are products that are built by each concrete builder



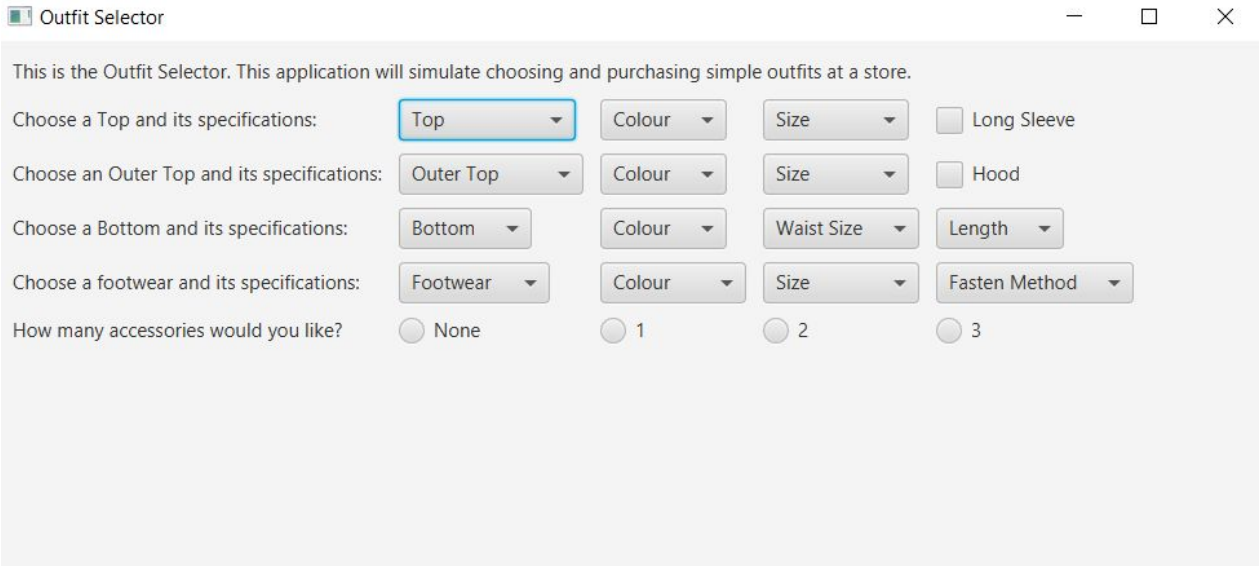
In-depth example of Builder Pattern

- Client calls the director to request a sports car to build
- Director calls the methods of the builder class to build the car
- The client receives the car after it is built
- This shows that the builders can be used to build a variety of complex objects by building them step-by-step
- Builders are hidden from the Client



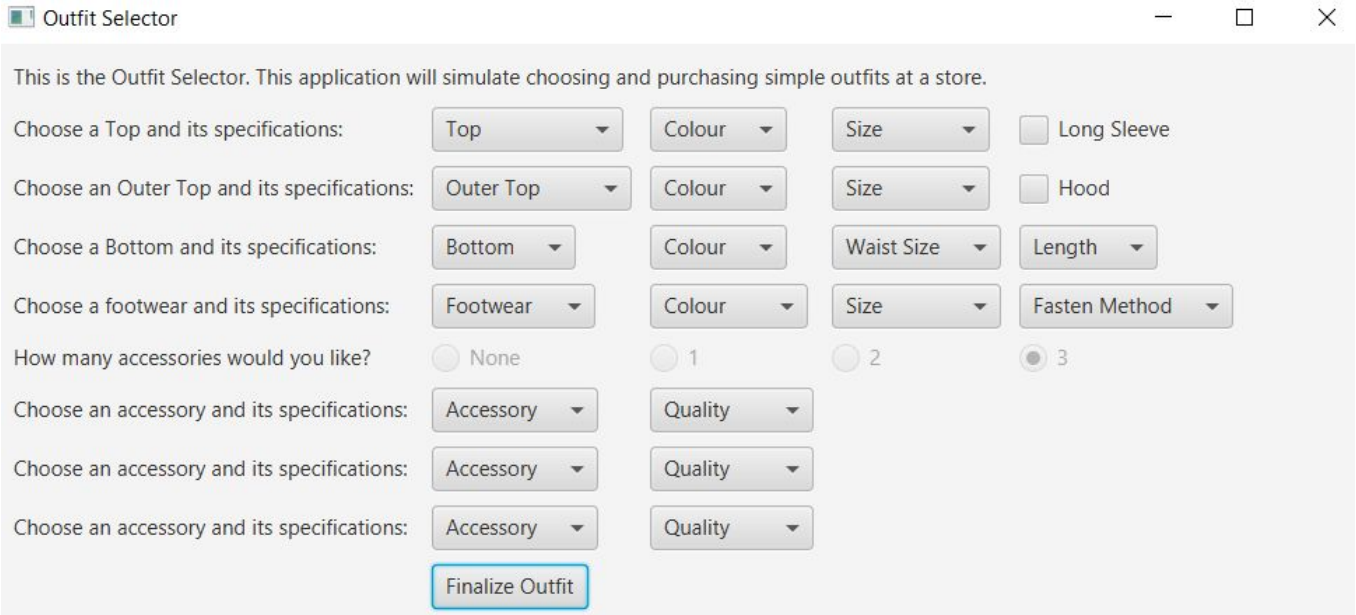
Implementation - Overview

- My implementation of the builder pattern is an outfit builder
- Users will be able to enter the clothes they want
- The outfit will be then built and shown to the user along with the cost of the outfit



Implementation – The Client

- No proper client class since I used a GUI so it serve as the client
- Clicking the “Finalize Outfit” button will call the director



Implementation - The Builder

- This is the main builder class
- They receive the details of the outfit from the engineer and builds the outfit class
- The built outfit can be used to show the details of the outfit to the user

```java
public class OutfitBuilder {
    final private Outfit outfit = new Outfit();

    public void buildOuterTop(String name, String colour, Boolean hasHood, String size)
    {
        if(name.equals("Jacket")) {
            Jacket jacket = new Jacket();
            jacket.setColour(colour);
            jacket.setHood(hasHood);
            jacket.setSize(size);

            this.outfit.addClothes(jacket);
        }
        else
        {
            Windbreaker windbreaker = new Windbreaker();
            windbreaker.setColour(colour);
            windbreaker.setHood(hasHood);
            windbreaker.setSize(size);

            this.outfit.addClothes(windbreaker);
        }

    }
  public void buildBottom(String name, String colour, int length, int waistSize)
  {
      if(name.equals("Shorts"))
      {
          Shorts shorts = new Shorts();
          shorts.setColour(colour);
          shorts.setLength(length);
          shorts.setWaistSize(waistSize);

          this.outfit.addClothes(shorts);
      }
      else
      {
          Jeans jeans = new Jeans();
          jeans.setColour(colour);
          jeans.setLength(length);
          jeans.setWaistSize(waistSize);

          this.outfit.addClothes(jeans);
      }
  }
```

```java
public void buildAccessory(String name, String quality)
{
    int multiplier;

    if(quality.equals(("Cheap")))
    {
        multiplier = 1;
    }
    else if(quality.equals("Regular"))
    {
        multiplier = 3;
    }
    else
    {
        multiplier = 5;
    }

    if(name.equals("Necklace")) {
        Necklace necklace = new Necklace();
        necklace.setQuality(quality);
        necklace.setPrice(multiplier);

        this.outfit.addAccessory(necklace);
    }
    else if (name.equals("Bracelet"))
    {
        Bracelet bracelet = new Bracelet();
        bracelet.setQuality(quality);
        bracelet.setPrice(multiplier);

        this.outfit.addAccessory(bracelet);
    }
    else
    {
        Watch watch = new Watch();
        watch.setQuality(quality);
        watch.setPrice(multiplier);

        this.outfit.addAccessory(watch);
    }

}
```

Only snippets are shown as the rest of the class is the same but for the other pieces of clothing

Implementation - The Concrete Builders

- They build the smaller objects that make up the complex object
- The main builder uses these classes to build said complex object
- In this implementation, they represent pieces of clothing
- Two out of the many concrete builders

```java
public class Boots extends Footwear{
    private String colour;
    private int size;
    private String fastenMethod;

    @Override
    public String getName() {
        return "Boots";
    }

    @Override
    public float getPrice() {
        return 140.0f;
    }

    @Override
    public String getColour() {
        return colour;
    }

    @Override
    public void setColour(String colour) {
        this.colour = colour;
    }

    @Override
    public String getFastenMethod() {
        return fastenMethod;
    }

    @Override
    public void setFastenMethod(String fastenMethod) {
        this.fastenMethod = fastenMethod;
    }

    @Override
    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public int getSize() {
        return size;
    }
}
```

```java
public class ButtonShirt extends Top{
    private String colour;
    private boolean hasLongSleeve;
    private String size;

    @Override
    public String getName() {
        return "Button Shirt";
    }

    @Override
    public float getPrice() {
        return 50.0f;
    }

    @Override
    public String getColour() {
        return colour;
    }

    @Override
    public void setColour(String colour) {
        this.colour = colour;
    }

    @Override
    public void setLongSleeve(boolean hasLongSleeve) {
        this.hasLongSleeve = hasLongSleeve;
    }

    @Override
    public boolean getLongSleeve() {
        return hasLongSleeve;
    }

    @Override
    public void setSize(String size) {
        this.size = size;
    }

    @Override
    public String getSize() {
        return size;
    }
}
```

Implementation - The Director

- This portion will serve as the director
- It will take all the information that the user entered and send it to the main builder
- The main builder will use this information to build the whole outfit

```java
// event handler for when user clicks the submit button
submit.setOnAction(event -> {
    gp.setDisable(true); // disabling the GridPane so user cannot make any more changes

    // calling the builder to build the top clothing
    outfitBuilder.buildTop(
            topBox.getValue(), colourTop.getValue(),
            sizeTop.getValue(),sleeveChk.isSelected()
    );

    // calling the builder to build the outer top clothing
    outfitBuilder.buildOuterTop(
            outerTopBox.getValue(),colourOuterTop.getValue(),
            hoodChk.isSelected(), sizeOuterTop.getValue()
    );

    // calling the builder to build the bottom clothing
    outfitBuilder.buildBottom(
            bottomBox.getValue(),colourBottom.getValue(),
            bottomLenBox.getValue(), bottomWSBox.getValue()
    );

    // calling the builder to create the footwear
    outfitBuilder.buildFootwear(
            footwearBox.getValue(),colourFootwear.getValue(),
            footwearSizeBox.getValue(),fastenBox.getValue()
    );

    // calling builder to create build 1-3 accessories based on user choice
    if(access.getValue() != null)
    {
        outfitBuilder.buildAccessory(access.getValue(),qualityBox.getValue());
    }
    if(access1.getValue() != null)
    {
        outfitBuilder.buildAccessory(access1.getValue(),qualityBox1.getValue());
    }
    if(access2.getValue() != null)
    {
        outfitBuilder.buildAccessory(access2.getValue(),qualityBox2.getValue());
    }
```

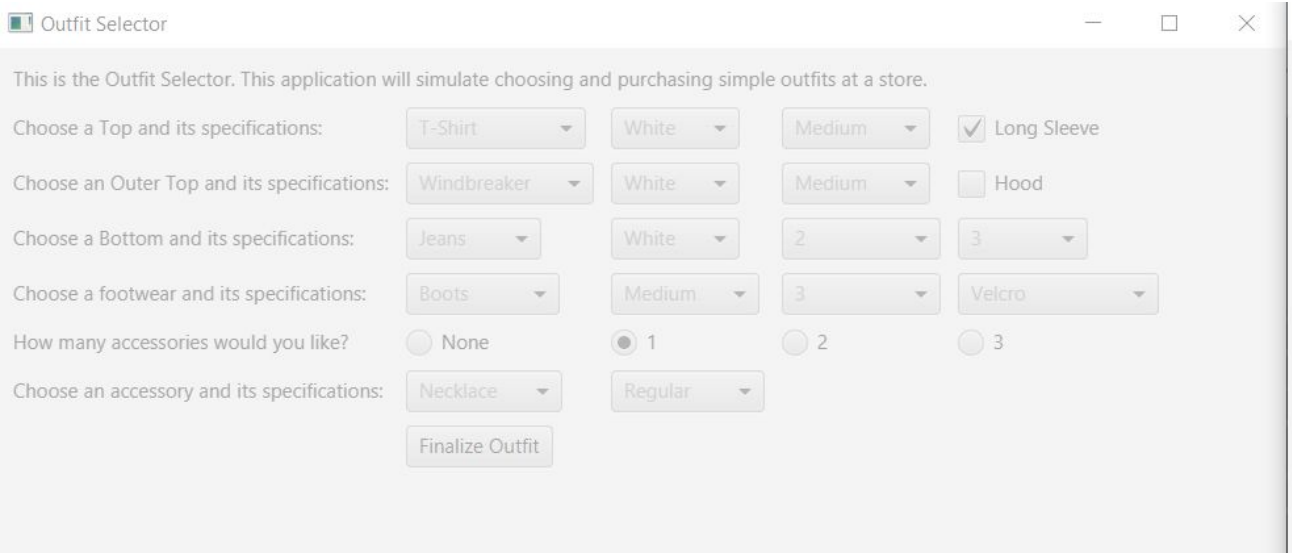
Implementation – The Product

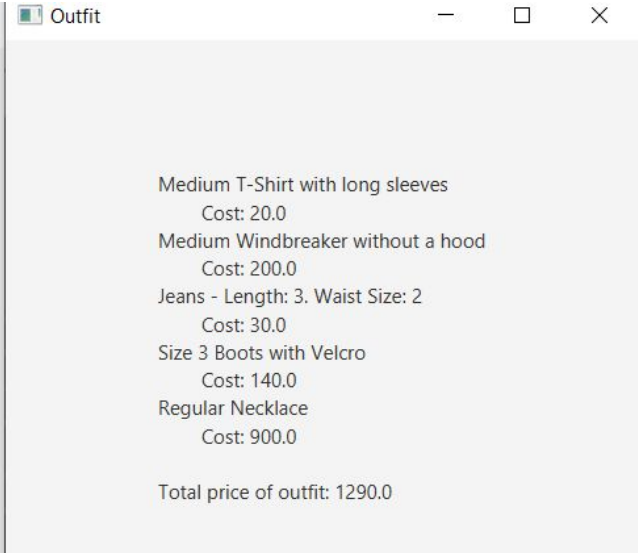
- The product is the final product made by the builder classes
- In this implementation, they are the final outfit based on the user's choices

```java
package com.example.builderpattern;

import java.util.ArrayList;

public class Outfit {
    final private ArrayList<Clothes> outfit = new ArrayList<>();
    final private ArrayList<Accessories> accessories = new ArrayList<>();

    public void addClothes(Clothes clothes)
    {
        outfit.add(clothes);
    }

    public void addAccessory(Accessories accessory)
    {
        accessories.add(accessory);
    }
```

```java
public float priceTotal()
{
    float total = 0.0f;

    for(Clothes clothes : outfit)
    {
        total += clothes.getPrice();
    }

    for(Accessories accessory : accessories)
    {
        total += accessory.getPrice();
    }

    return total;
}

@Override
public String toString() {
    StringBuilder finalOutfit = new StringBuilder();

    for(Clothes cloth : outfit)
    {
        finalOutfit.append(cloth.toString()).append("\n");
    }
    for(Accessories access : accessories)
    {
        finalOutfit.append(access.toString()).append("\n");
    }

    finalOutfit.append("\nTotal price of outfit: ").append(priceTotal());

    return finalOutfit.toString();
}
```

Implementation – The Product Example





References

- *Design Pattern - Overview*. (n.d.). Tutorialspoint. Retrieved March 3, 2022, from https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
- *Design Patterns - Builder Pattern*. (n.d.). Tutorialspoint. Retrieved March 3, 2022, from https://www.tutorialspoint.com/design_pattern/builder_pattern.htm
- *Refactoring.Guru*. (n.d.). Design Patterns. Retrieved March 3, 2022, from https://refactoring.guru/design-patterns
- *SourceMaking*. (n.d.). Builder Design Pattern. Retrieved March 3, 2022, from https://sourcemaking.com/design_patterns/builder