DePaul University

Design Patterns:

Builder

Object-oriented Software Development
SE 350– Spring 2021

Vahid Alizadeh

Week 9.2
May 27, 2021

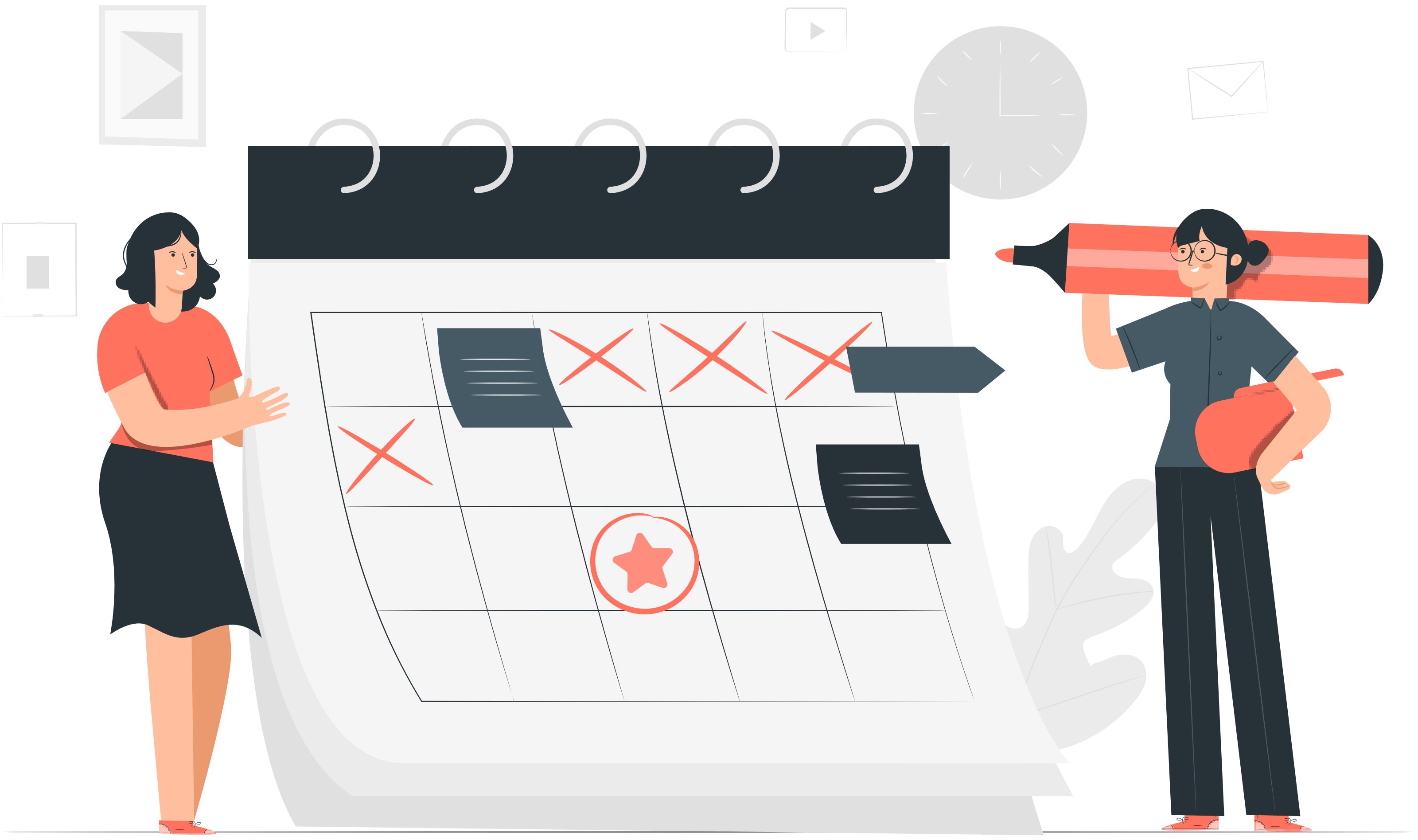
Announcements

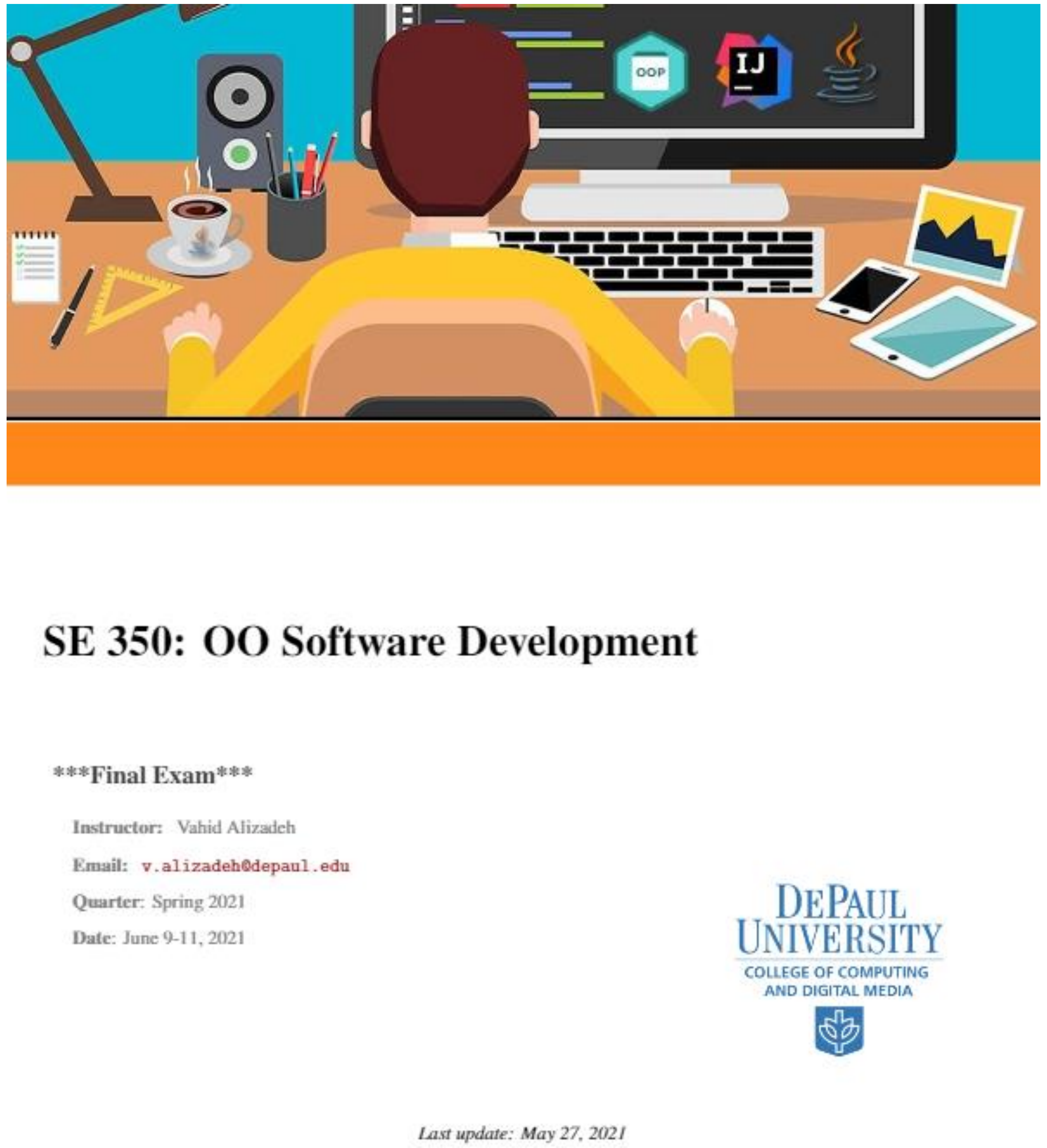
Future Schedule

Final Exam

Week 11: June 9-11 (Wed-Fri)

- ~~Assignment 1~~
- ~~Assignment 2~~
- ~~Mid Term Exam~~
- ~~Assignment 3:~~
  - ~~Release: Week 7~~
  - ~~Due: Week 8~~
- **Assignment 4:**
  - ~~Release: Week 8~~
  - Due: Week 9
- **Bonus Research Project:**
  - Presentation Due: Week 10
  - Report Due: Week 11
- **Final Exam:**
  - Week 11





SE 350: OO Software Development

Final Exam

Instructor: Vahid Alizadeh

Email: v.alizadeh@depaul.edu

Quarter: Spring 2021

Date: June 9-11, 2021

DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

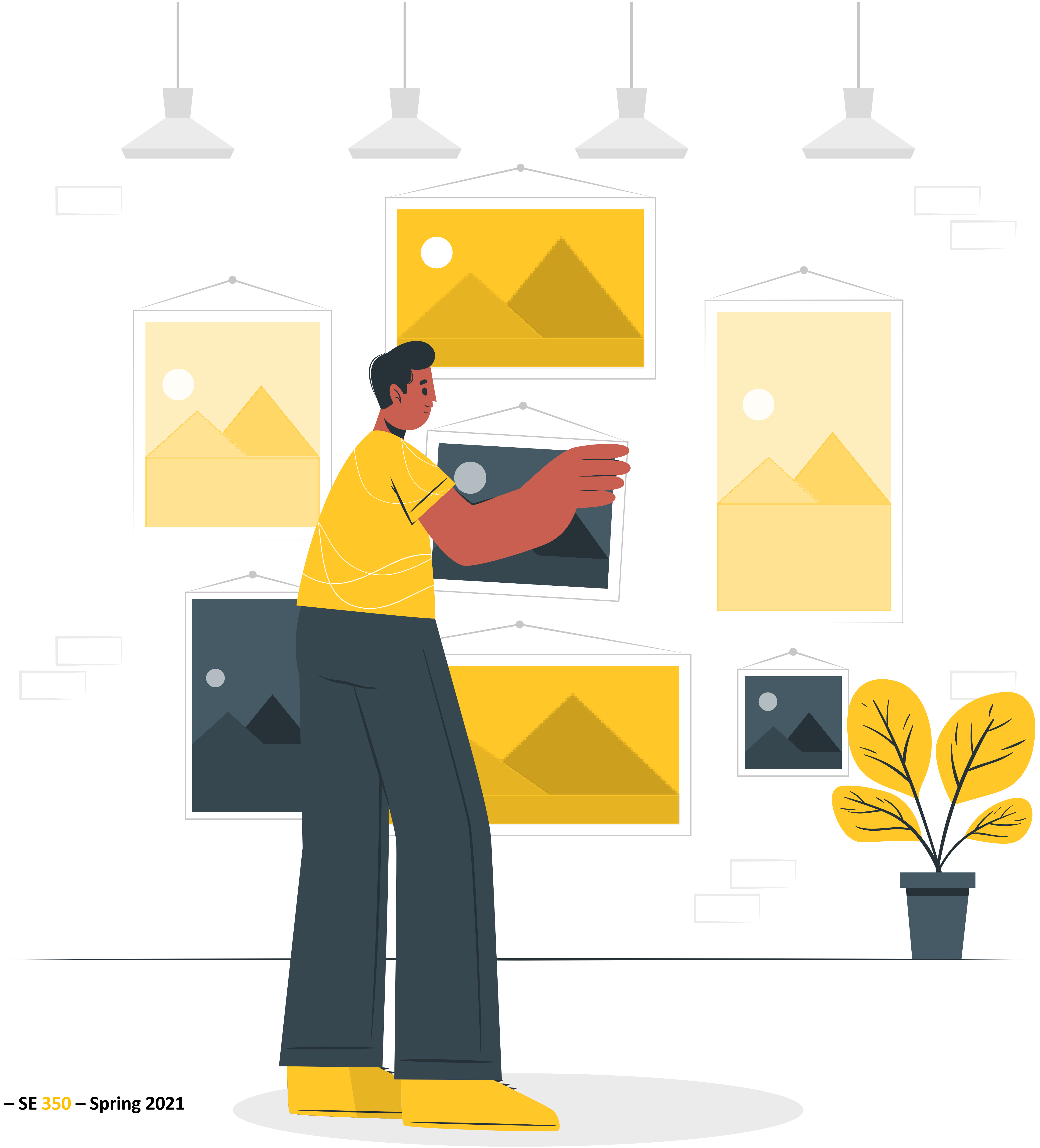
Last update: May 27, 2021

Decorator Pattern

STRUCTURAL

Decorator Pattern Introduction

Decorator is a structural design pattern that allows for an object's behavior to be extended dynamically at run time.



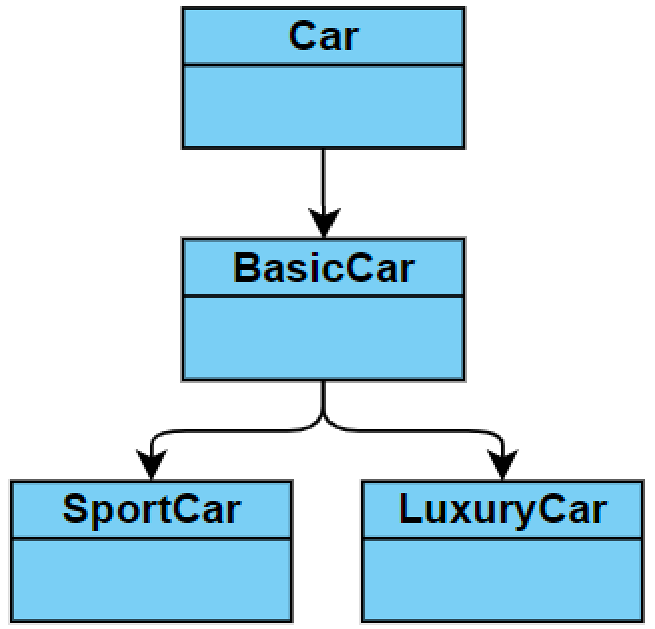
Decorator Design Pattern

INTENT

- Add additional responsibilities to individual objects dynamically.
- withdraw responsibilities from an object.

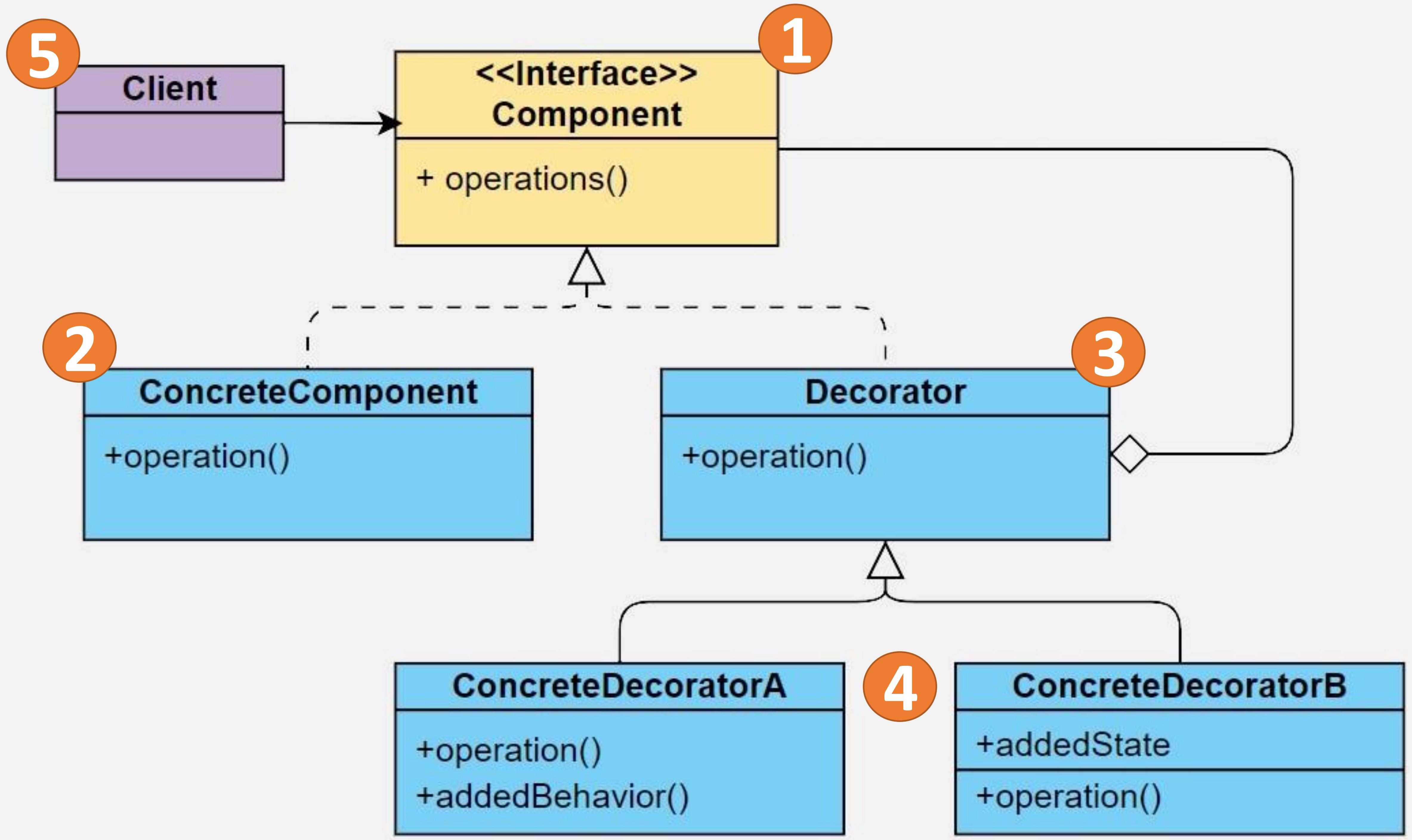
PROBLEM

- Adding a behavior to an object at run-time in not possible by inheritance.
- Car example



STRUCTURE

- 1- Component Interface
- 2- Concrete Component
- 3- Decorator
- 4- Concrete Decorators



Decorator Pattern Pros & Cons

Pros

- ✓ Runtime modification for flexibility and easier maintenance.
- ✓ Combining multiple decorators in any orders.
- ✓ Extending functionality without touching other objects.
- ✓ Single Responsibility Principle

Cons

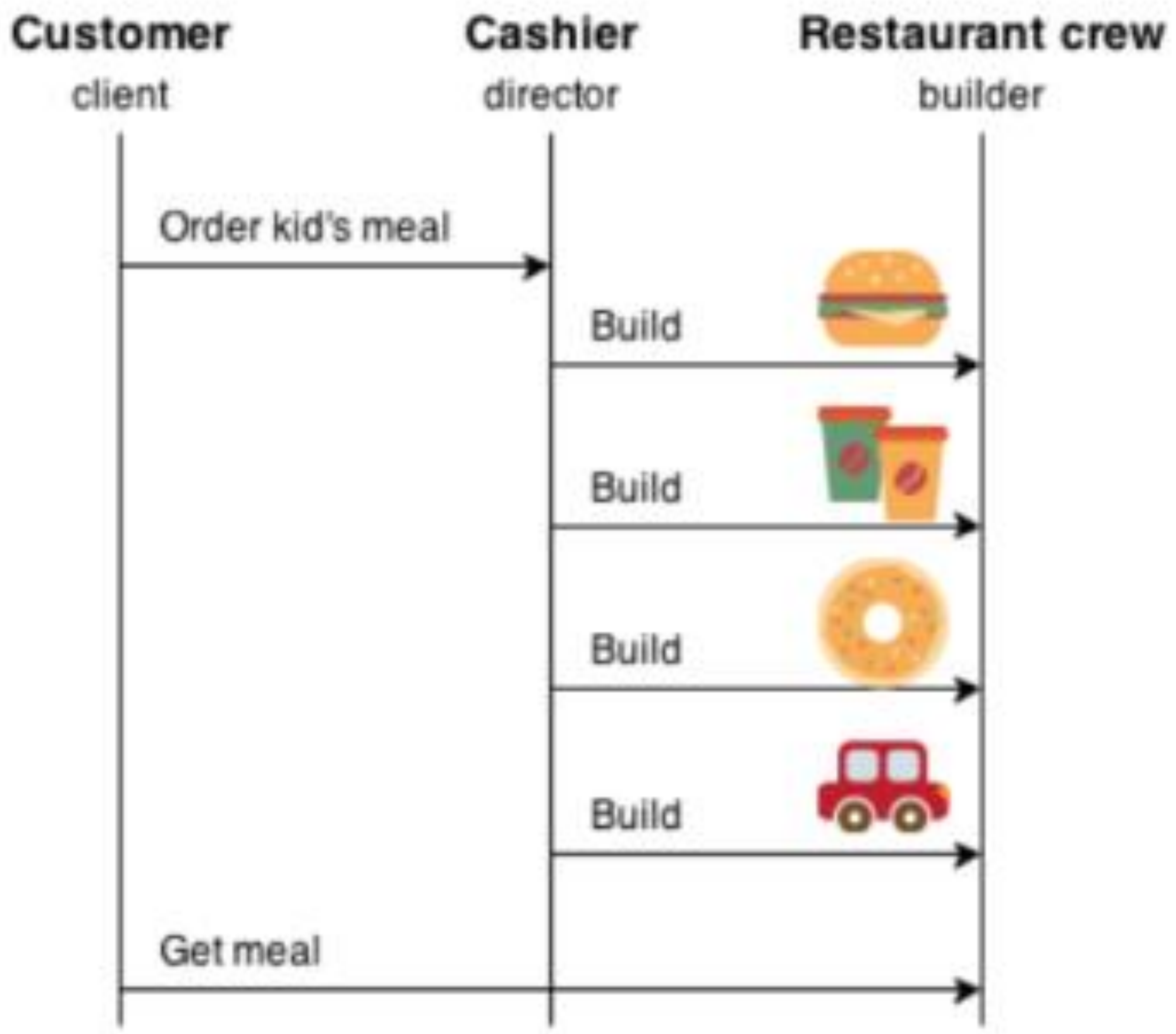
- ✗ Using many similar objects (decorators).
- ✗ Reduce the readability of the code.

Builder Pattern

CREATIONAL

Builder Pattern Introduction

Builder is a creational design pattern that is used to hide the complexity of an object construction.



Builder Design Pattern

INTENT

- Separate the construction of a complex object from its representation.

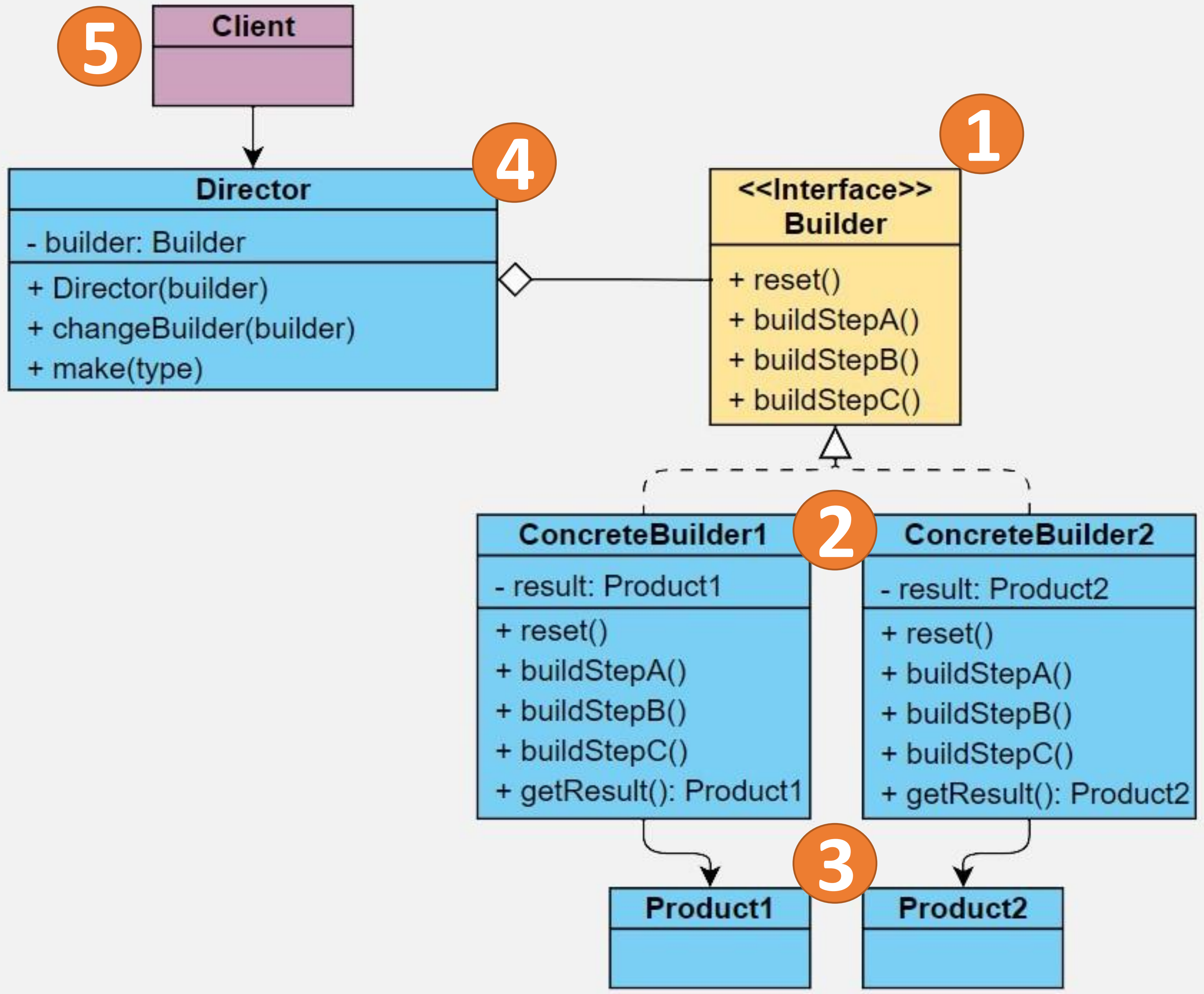
PROBLEM

- How can a class create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

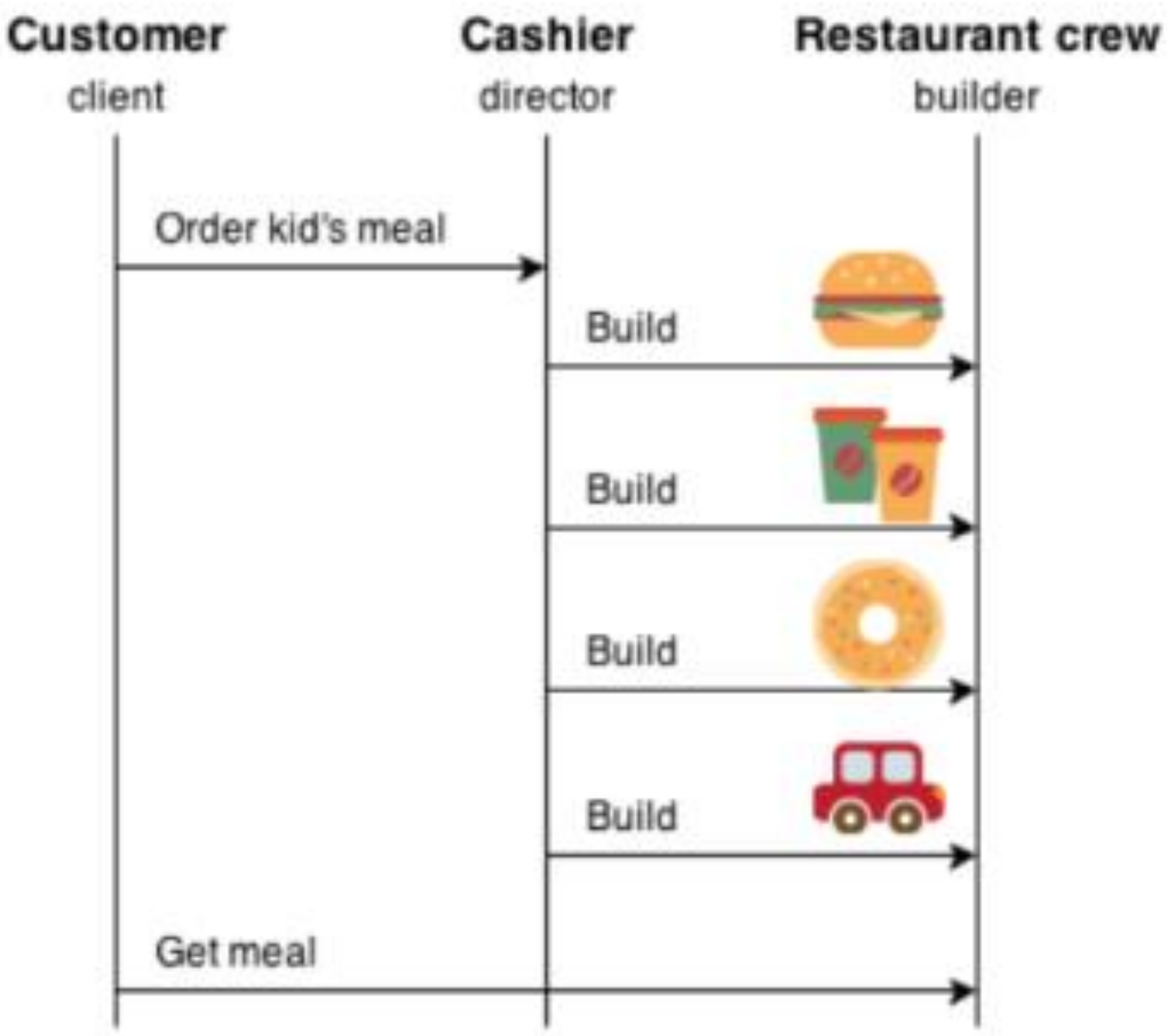
- Building a house example

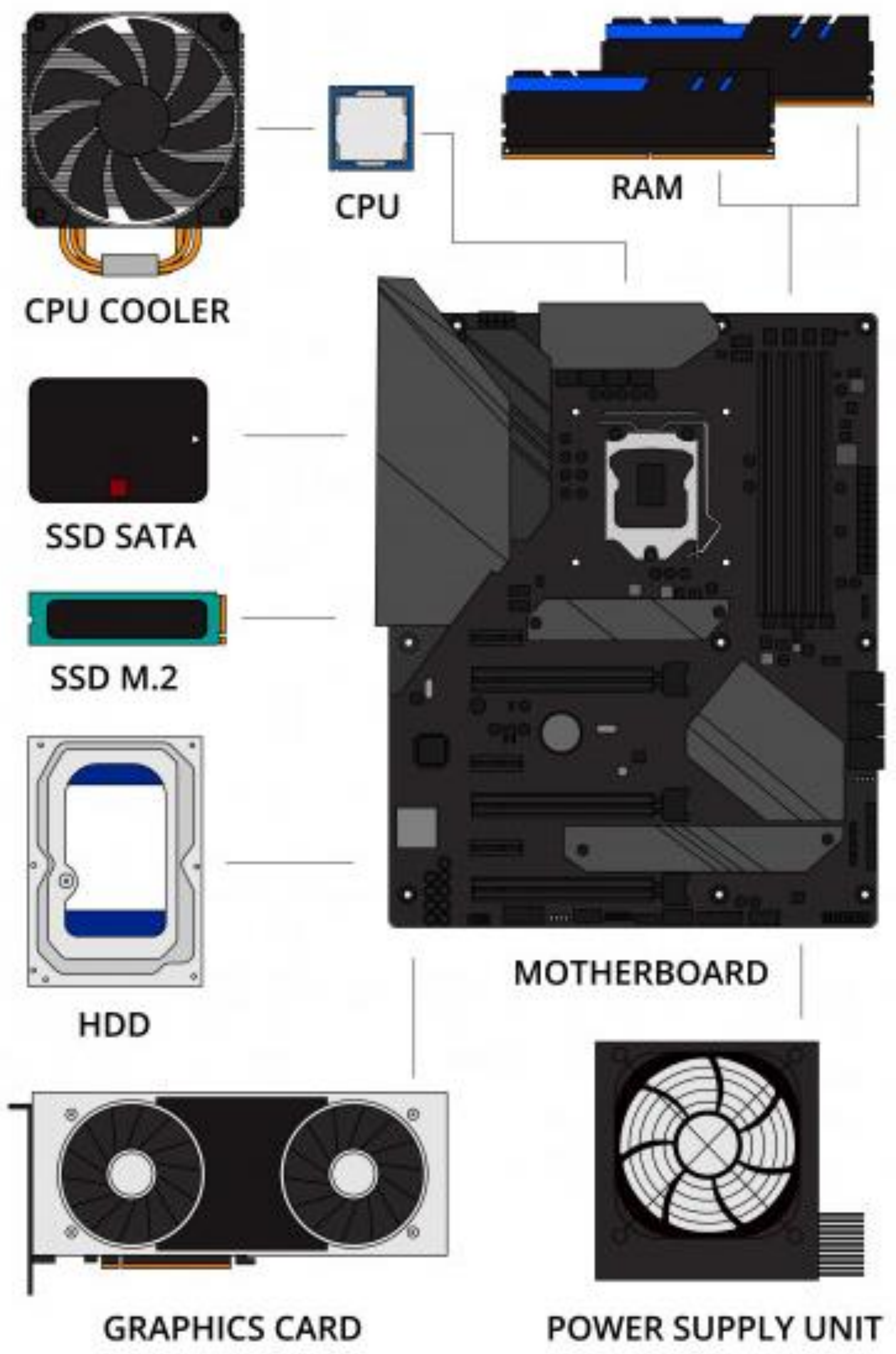
STRUCTURE

- 1- Builder interface
- 2- Concrete Builders
- 3- Products (resulting objects)
- 4- Director
- 5- The Client



Builder Pattern: Real-world Example



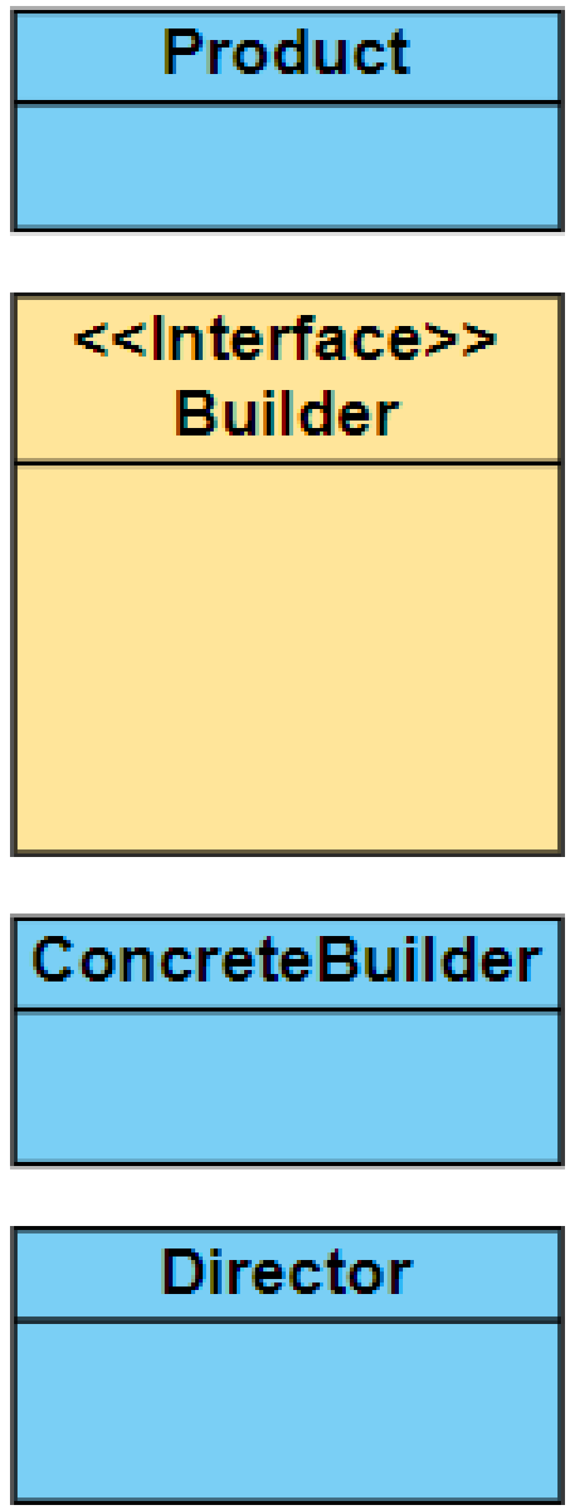


Builder Implementation

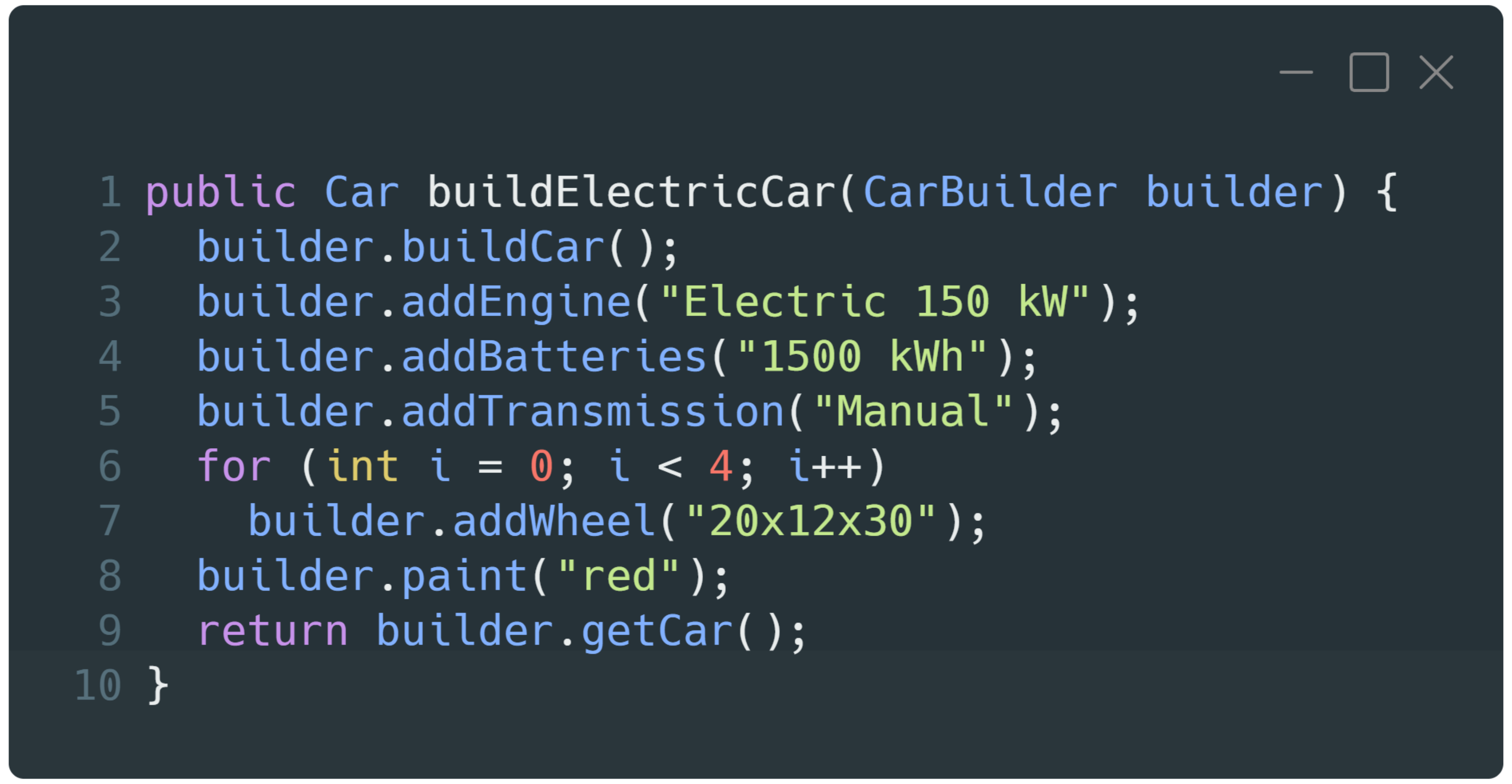
Common Implementation Steps:

- Define common object constructions in the **builder interface**.
- Create **concrete builder** classes for all product variations.
- Create a **director** class containing different approaches to build a product.
- Use Director and Builder in your **Client** code.

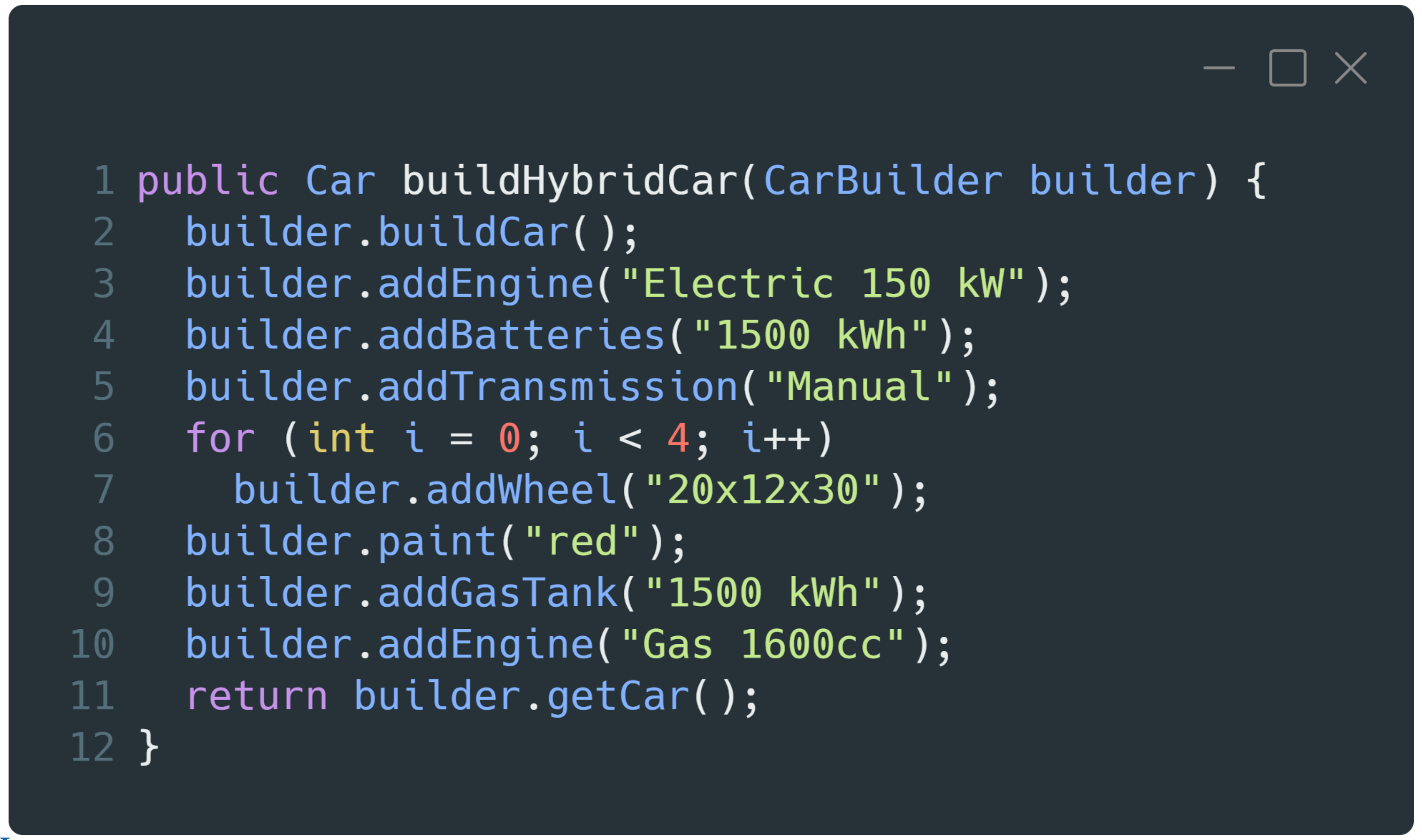
Class Compositions:



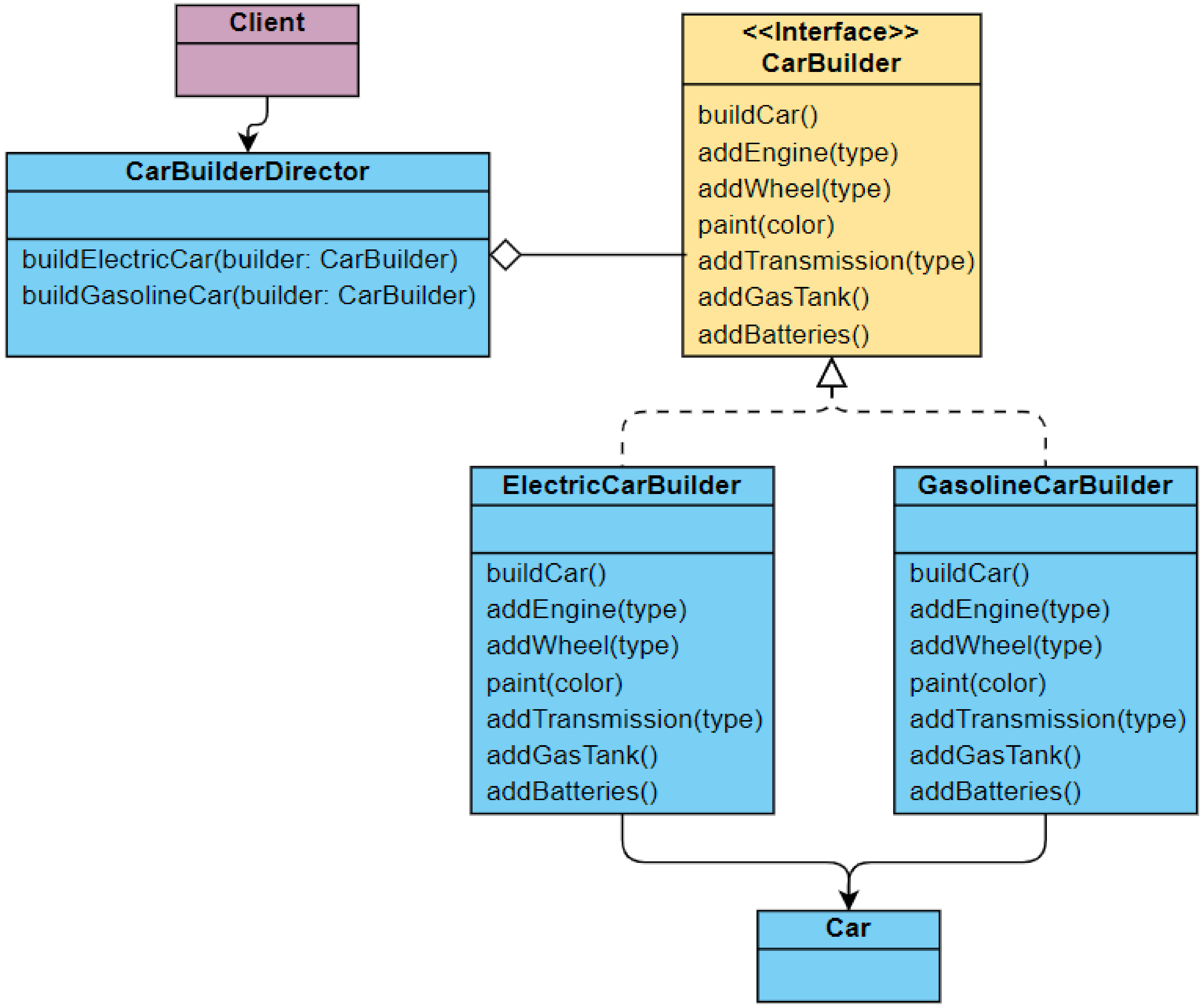
Builder Use Case Example: Car builder



```java
public Car buildElectricCar(CarBuilder builder) {
  builder.buildCar();
  builder.addEngine("Electric 150 kW");
  builder.addBatteries("1500 kWh");
  builder.addTransmission("Manual");
  for (int i = 0; i < 4; i++)
    builder.addWheel("20x12x30");
  builder.paint("red");
  return builder.getCar();
}
```



```java
public Car buildHybridCar(CarBuilder builder) {
  builder.buildCar();
  builder.addEngine("Electric 150 kW");
  builder.addBatteries("1500 kWh");
  builder.addTransmission("Manual");
  for (int i = 0; i < 4; i++)
    builder.addWheel("20x12x30");
  builder.paint("red");
  builder.addGasTank("1500 kWh");
  builder.addEngine("Gas 1600cc");
  return builder.getCar();
}
```



Client

CarBuilderDirector

buildElectricCar(builder: CarBuilder)
buildGasolineCar(builder: CarBuilder)

<<Interface>>
CarBuilder

buildCar()
addEngine(type)
addWheel(type)
paint(color)
addTransmission(type)
addGasTank()
addBatteries()

ElectricCarBuilder

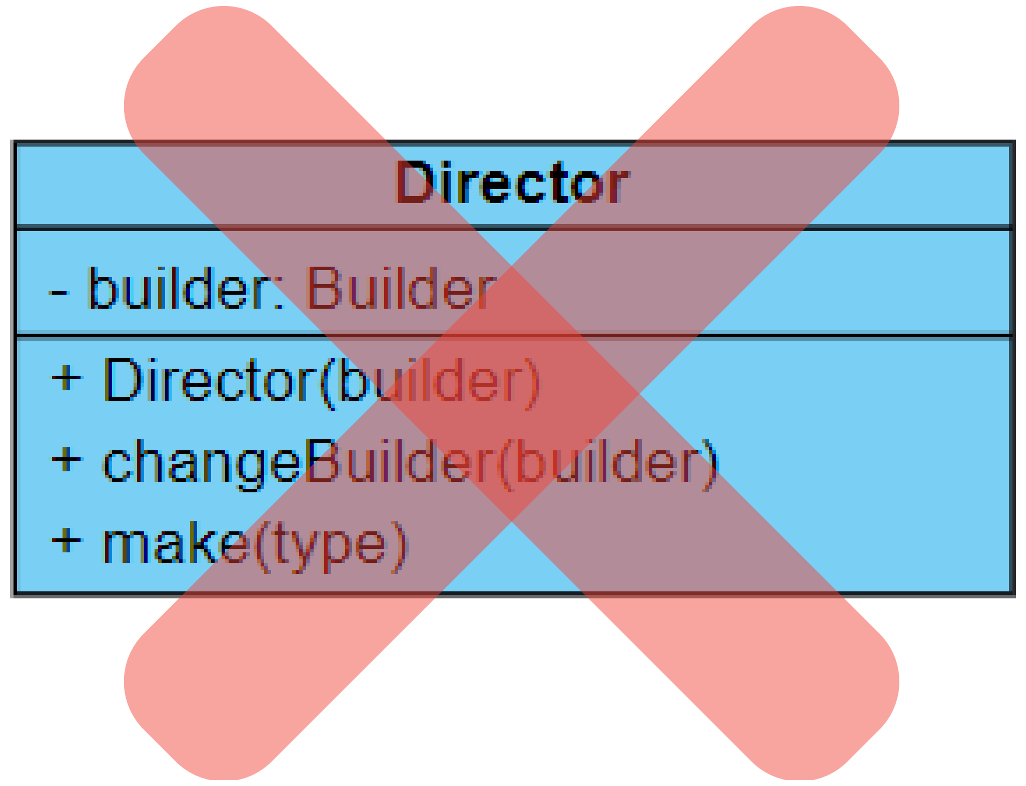
buildCar()
addEngine(type)
addWheel(type)
paint(color)
addTransmission(type)
addGasTank()
addBatteries()

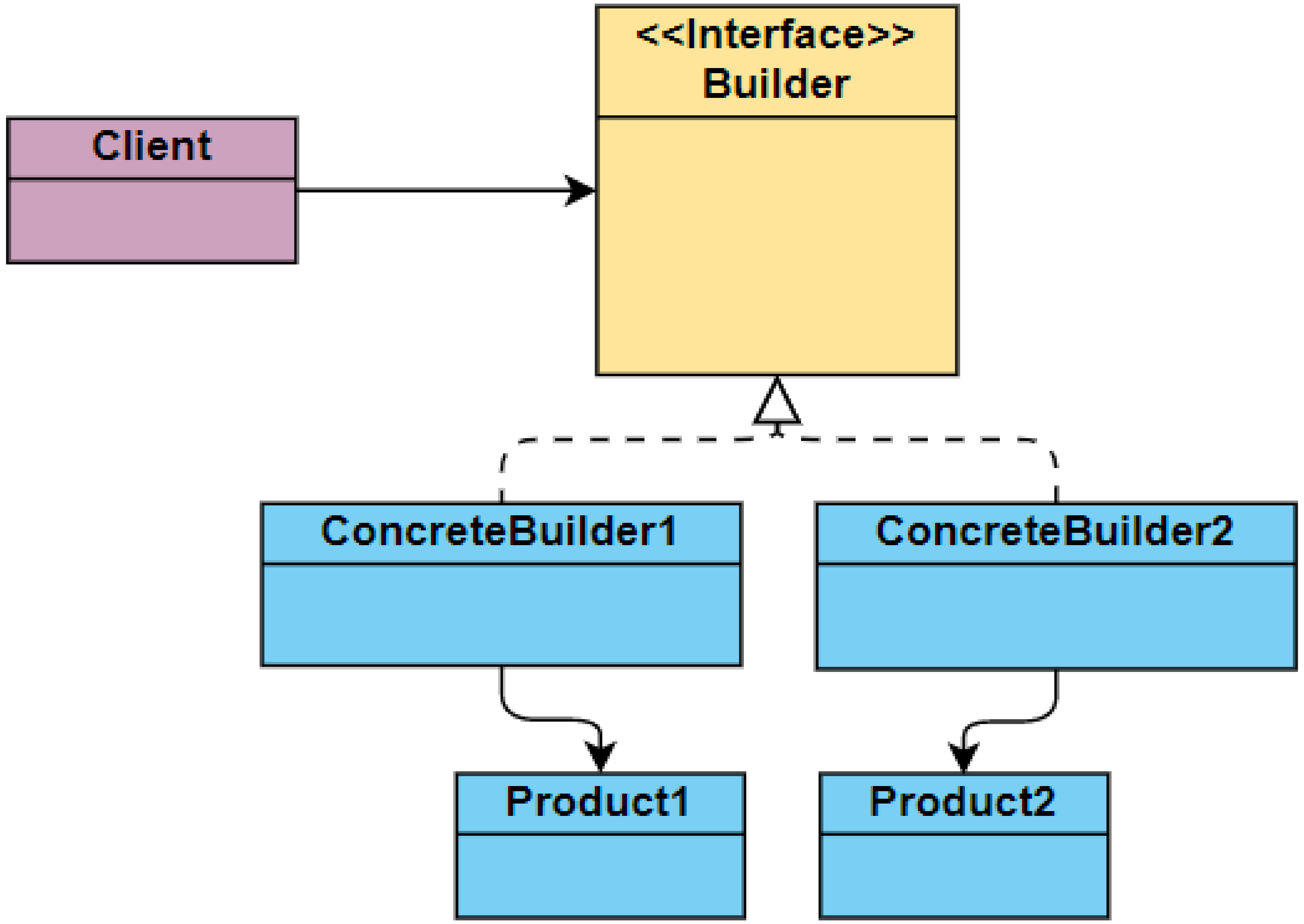
GasolineCarBuilder

buildCar()
addEngine(type)
addWheel(type)
paint(color)
addTransmission(type)
addGasTank()
addBatteries()

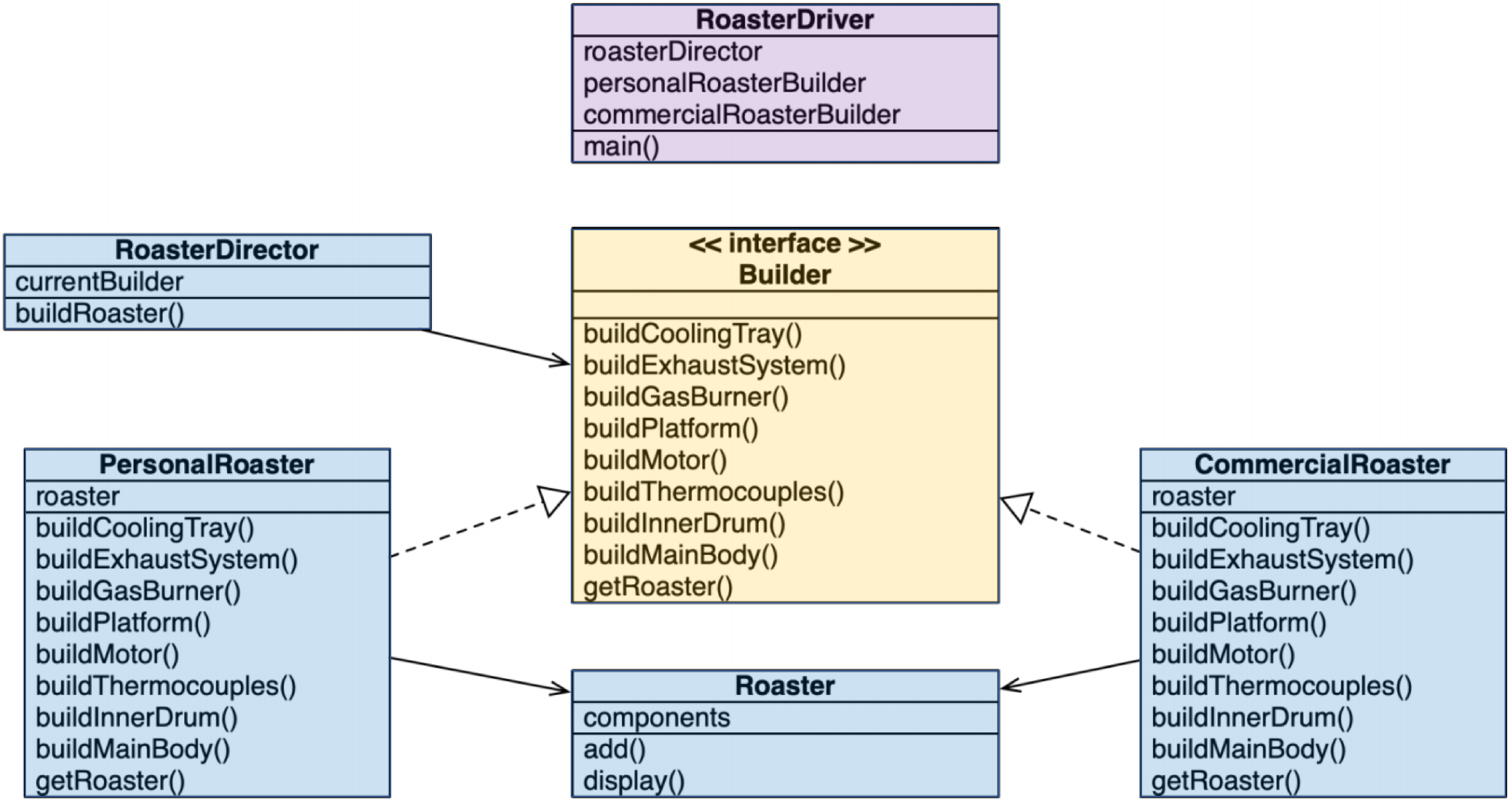
Car

Simplified Builder Pattern





Builder Use Case Example: Coffee Roaster



RoasterDriver
roasterDirector
personalRoasterBuilder
commercialRoasterBuilder
main()

RoasterDirector
currentBuilder
buildRoaster()

<< interface >>
Builder
buildCoolingTray()
buildExhaustSystem()
buildGasBurner()
buildPlatform()
buildMotor()
buildThermocouples()
buildInnerDrum()
buildMainBody()
getRoaster()

PersonalRoaster
roaster
buildCoolingTray()
buildExhaustSystem()
buildGasBurner()
buildPlatform()
buildMotor()
buildThermocouples()
buildInnerDrum()
buildMainBody()
getRoaster()

CommercialRoaster
roaster
buildCoolingTray()
buildExhaustSystem()
buildGasBurner()
buildPlatform()
buildMotor()
buildThermocouples()
buildInnerDrum()
buildMainBody()
getRoaster()

Roaster
components
add()
display()

Component	Personal Roaster	Commercial Roaster
Cooling tray	Model-Specific	Model-Specific
Exhaust system	Model-Specific	Model-Specific
Gas burner	Model-Specific	Model-Specific
Inner drum	Model-Specific	Model-Specific
Main body	Model-Specific	Model-Specific
Motor	Standard	Standard
Platform	Not required	Standard
Thermocouples	Standard	Standard

Find the source codes in the course GitHub repository.

Builder vs. Abstract Factory

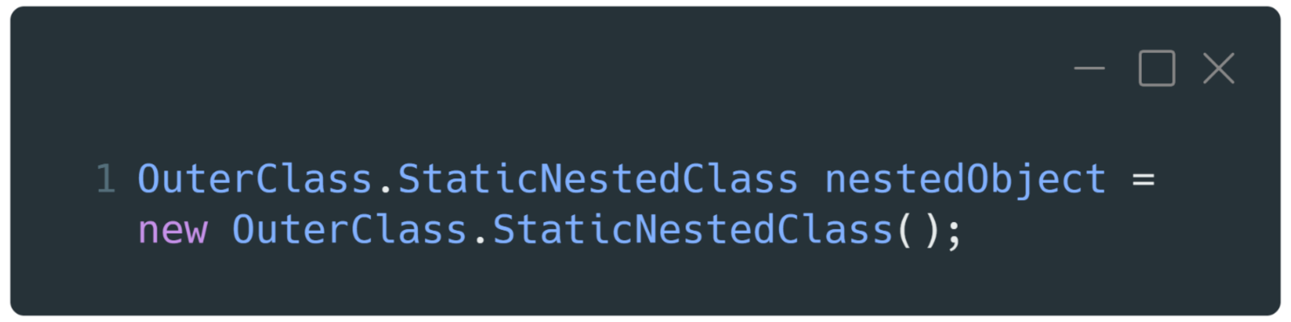
- **Builder pattern solves some of the problems with Factory and Abstract Factory design patterns with objects containing many attributes.**
  - Too Many arguments to pass from client program to the Factory class.
  - Some of the parameters might be optional but in Factory pattern we are forced to send all the parameters.
  - Very complex and confusing Factory class.

Builder:
Step by step object creation
“HOW”
Abstract Factory:
Object creation in one step
“WHAT”

Fluent Builder Using Method Chaining

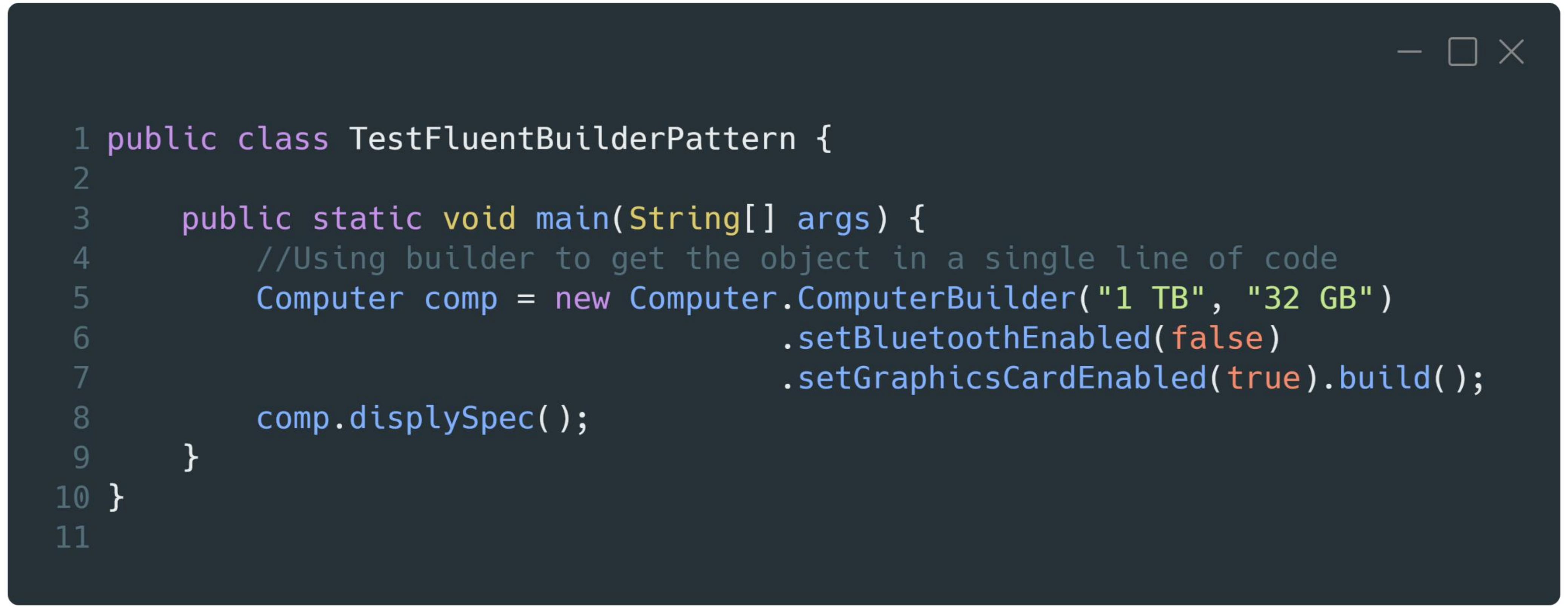
Computer building app use case example

- **Fluent Interface**
- **We create Builder as static nested class.**
  - How to access:



```
OuterClass.StaticNestedClass nestedObject =
new OuterClass.StaticNestedClass();
```

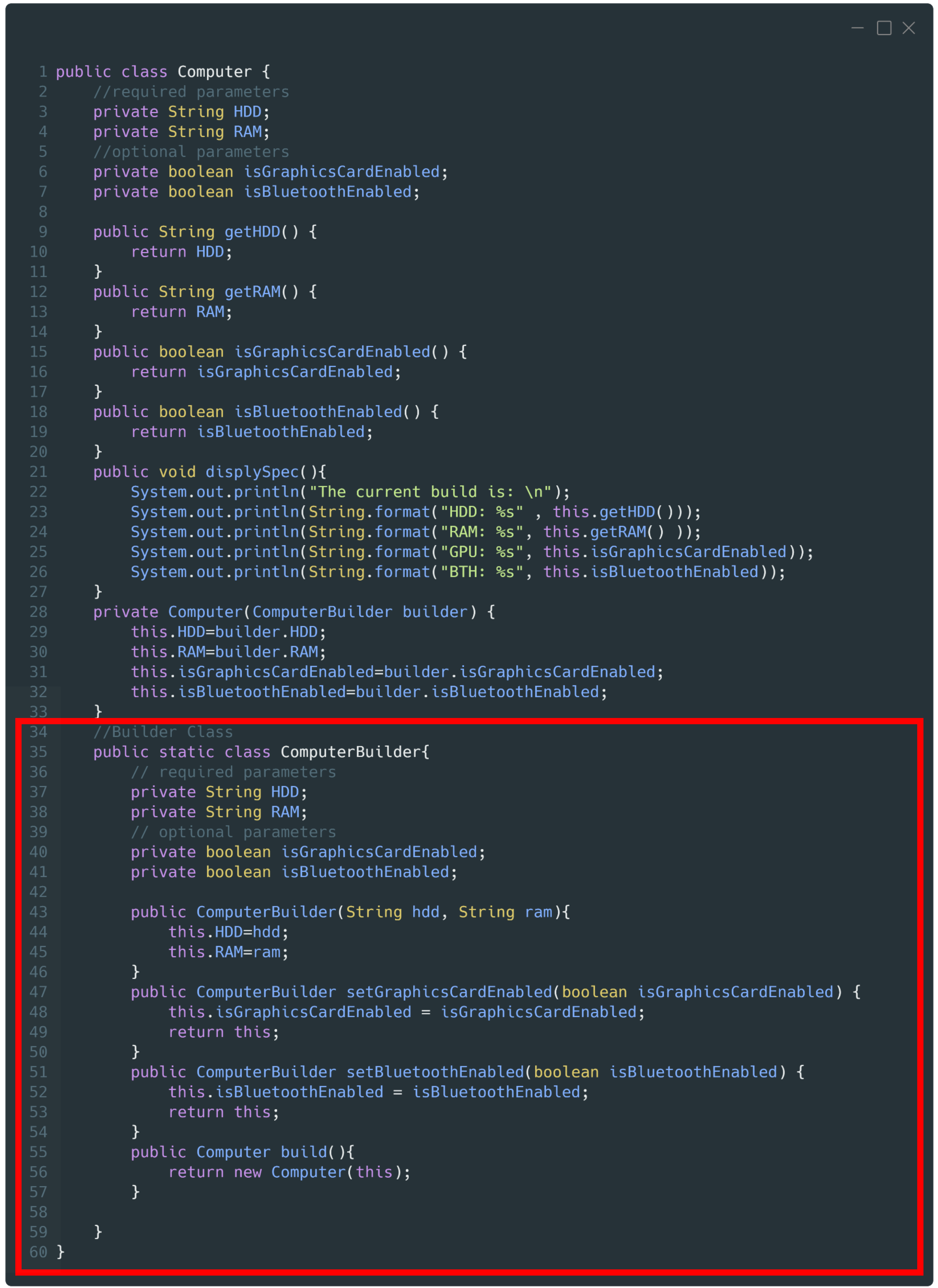
- **Builder has a public constructor with all required attributes.**
- **Builder has methods for optional attributes.**
- **Builder has a Build() method to return the object.**



```
public class TestFluentBuilderPattern {

    public static void main(String[] args) {
        //Using builder to get the object in a single line of code
        Computer comp = new Computer.ComputerBuilder("1 TB", "32 GB")
                                    .setBluetoothEnabled(false)
                                    .setGraphicsCardEnabled(true).build();
        comp.displaySpec();
    }
}

```



```
public class Computer {
    //required parameters
    private String HDD;
    private String RAM;
    //optional parameters
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    public String getHDD() {
        return HDD;
    }
    public String getRAM() {
        return RAM;
    }
    public boolean isGraphicsCardEnabled() {
        return isGraphicsCardEnabled;
    }
    public boolean isBluetoothEnabled() {
        return isBluetoothEnabled;
    }
    public void displaySpec(){
        System.out.println("The current build is: \n");
        System.out.println(String.format("HDD: %s" , this.getHDD()));
        System.out.println(String.format("RAM: %s", this.getRAM() ));
        System.out.println(String.format("GPU: %s", this.isGraphicsCardEnabled));
        System.out.println(String.format("BTH: %s", this.isBluetoothEnabled));
    }
    private Computer(ComputerBuilder builder) {
        this.HDD=builder.HDD;
        this.RAM=builder.RAM;
        this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
        this.isBluetoothEnabled=builder.isBluetoothEnabled;
    }
    //Builder Class
    public static class ComputerBuilder{
        // required parameters
        private String HDD;
        private String RAM;
        // optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        public ComputerBuilder(String hdd, String ram){
            this.HDD=hdd;
            this.RAM=ram;
        }
        public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }
        public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }
        public Computer build(){
            return new Computer(this);
        }

    }
}
```

Find the source codes in the course GitHub repository.

Builder Pattern Pros & Cons

Pros

- ✓ Vary a product's internal representation
- ✓ Encapsulates and reuse code for construction.
- ✓ Control over steps of construction process.
- ✓ Single Responsibility Principle

Cons

- ✗ Increased overall complexity.
- ✗ Requires creating a separate ConcreteBuilder for each product.

Any Question

????????????????

How do you feel about the course?

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Please Send Your Question or Feedback...

Top

New

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**