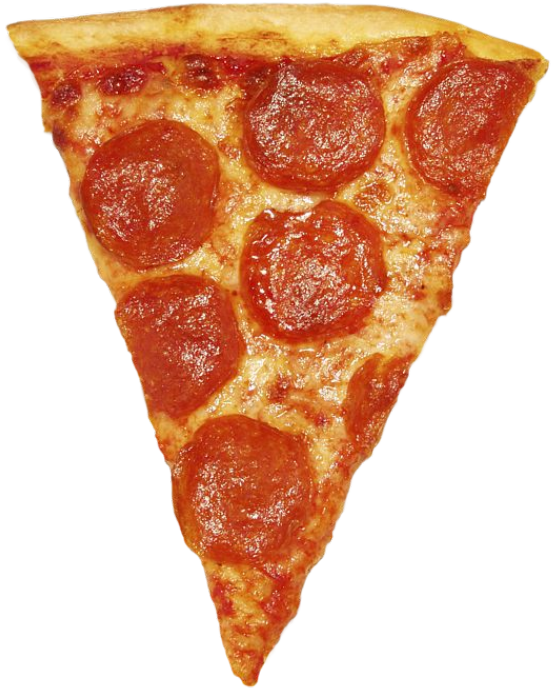
# SWEN 262
## Engineering of Software Subsystems

*Decorator Pattern*

# Pizza POS System

1. The Point of Sale (POS) system for a pizzeria must allow employees to prepare a pizza order.
    a. *The order includes pizza size, crust, & toppings.*
    b. *The order also includes prep and cook instructions, which vary based on the toppings.*
2. The customization options include:
    a. *Available pizza sizes are small ($10), medium ($12), large ($15), and sheet ($21).*
    b. *The available crust types are regular (white) and wheat (extra $1).*
    c. *Available toppings include extra cheese ($1), pepperoni ($2), sausage ($2), bacon ($1.50), hamburger ($1.50), mushrooms ($1), banana peppers ($2), black olives ($1.25), peppers ($1.25), onions ($1), and anchovies ($3).*
    d. *Customers with coupons received a 15% discount on their total order.*
3. Once the order is prepared, the total price is calculated and the employee follows the prep and cook instructions.

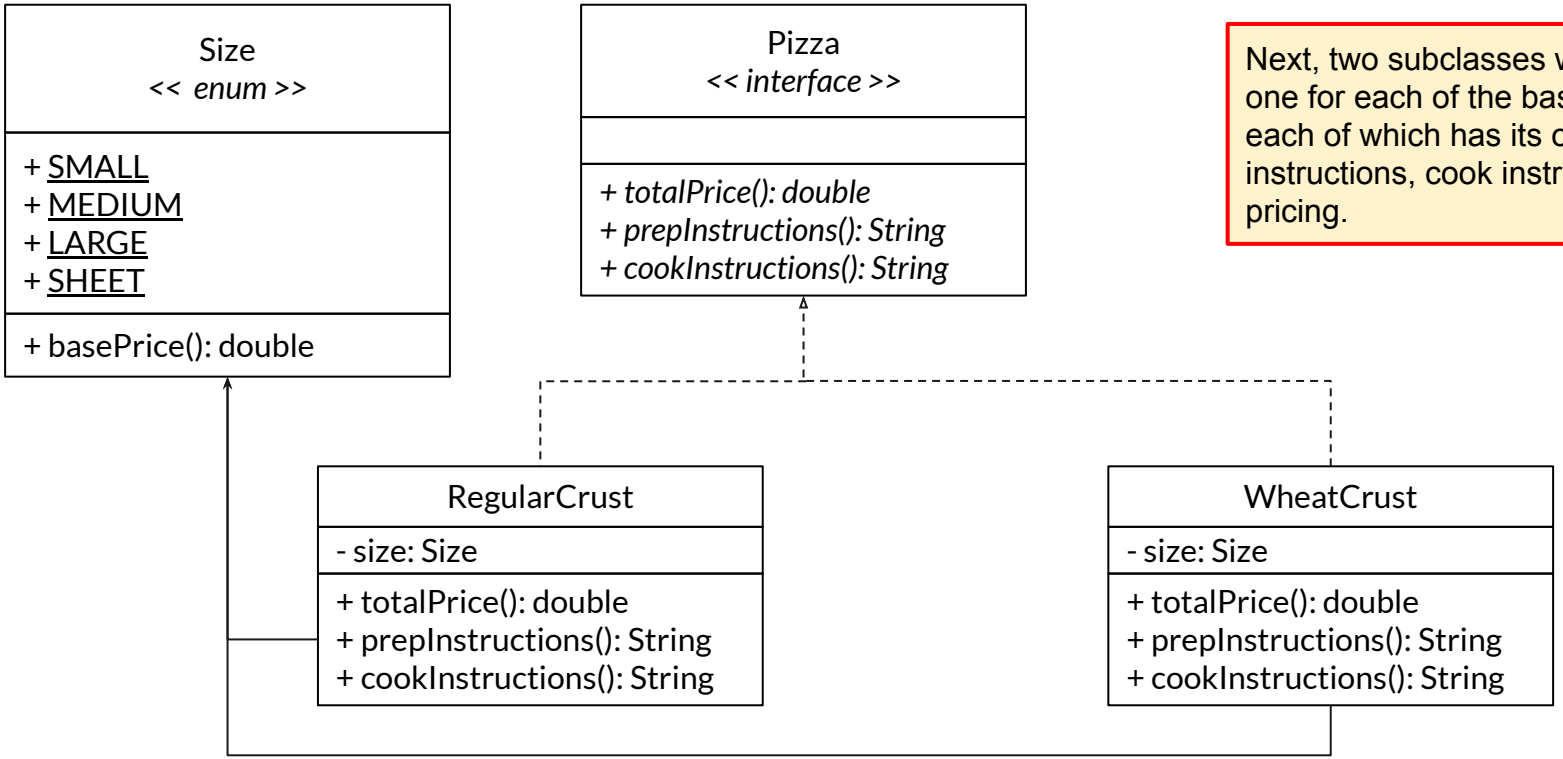


There are a number of different ways to design the system for building a pizza. Let's take a look at some of the options...

# Subclassing

The obvious place to start is a Pizza interface, which defines the behaviors for a pizza order.

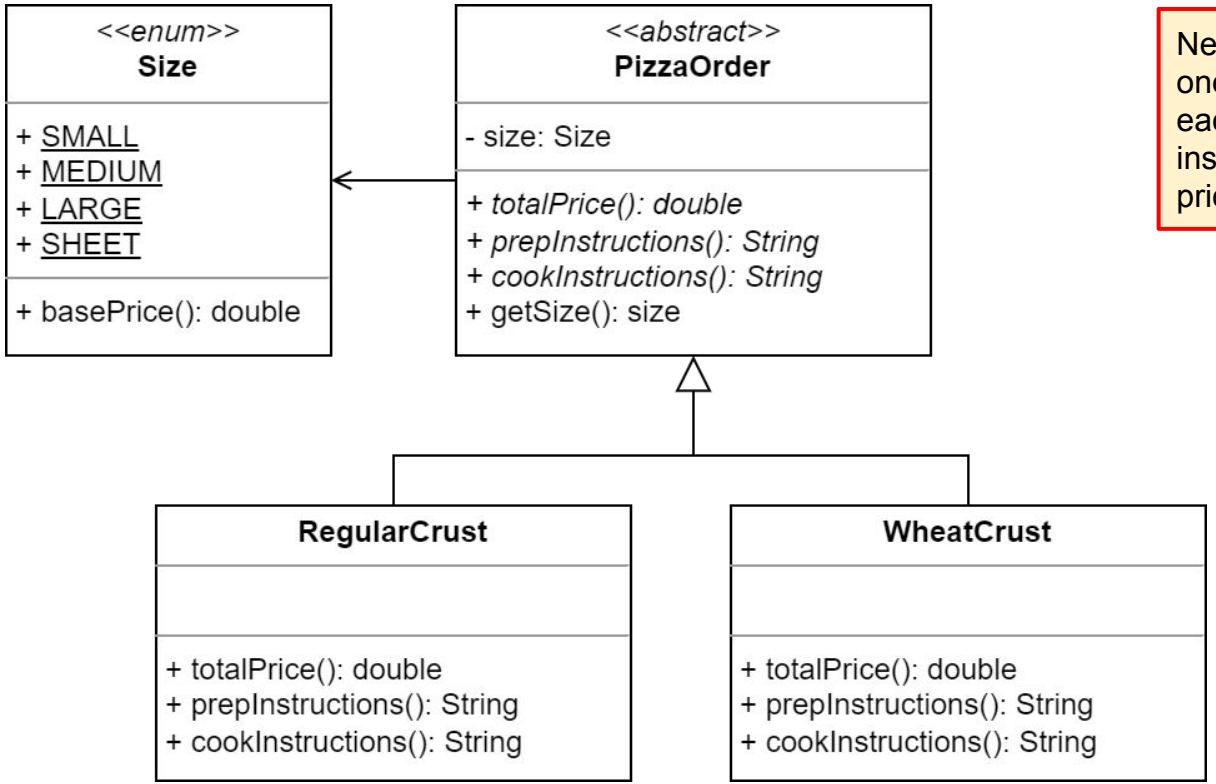
Next, two subclasses will be required: one for each of the basic crust options, each of which has its own prep instructions, cook instructions, and pricing.



# Subclassing

The obvious place to start is a Pizza interface, which defines the behaviors for a pizza order.

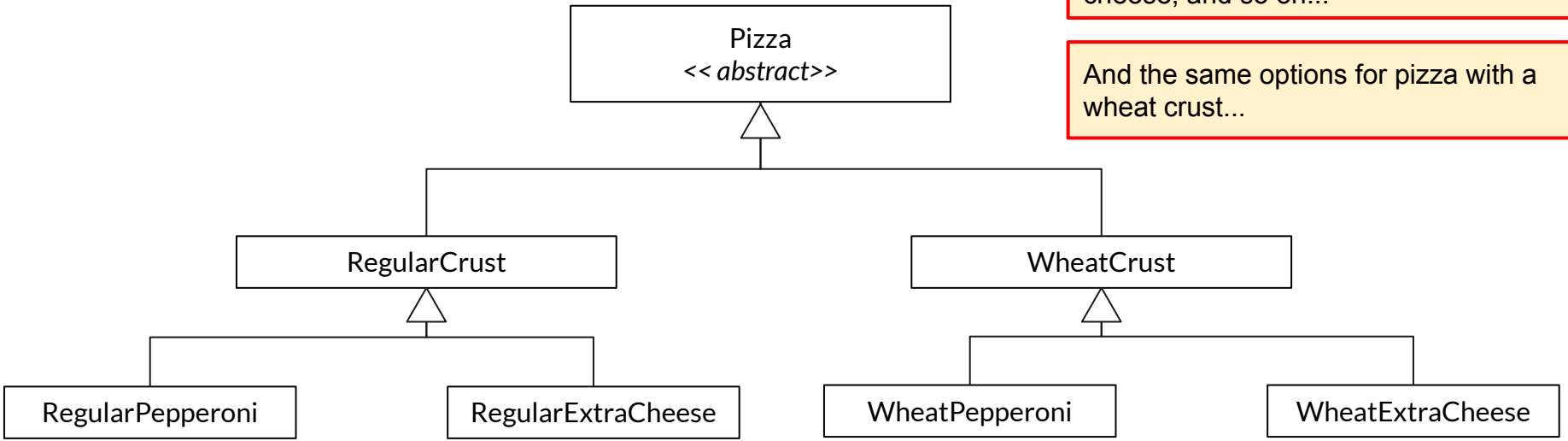
Next, two subclasses will be required: one for each of the basic crust options, each of which has its own prep instructions, cook instructions, and pricing.



# Subclassing

Next, we create subclasses for each topping, e.g. regular crust with pepperoni, wheat crust with extra cheese, and so on...

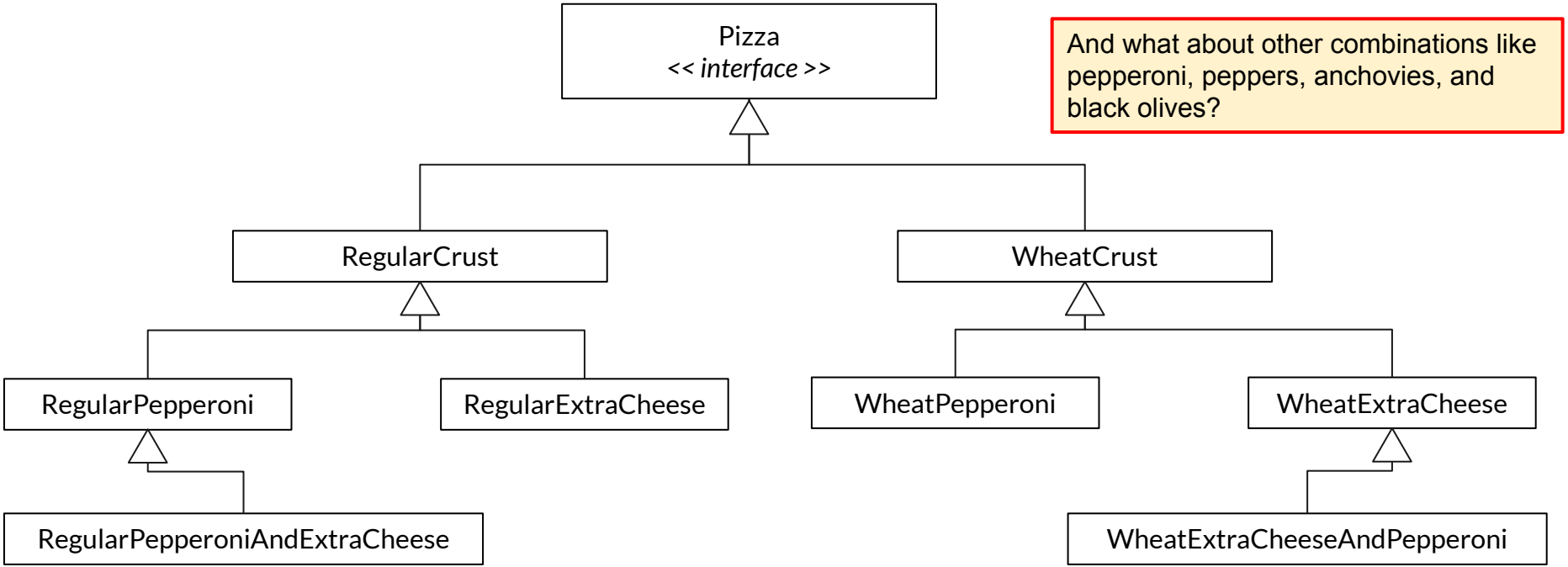
And the same options for pizza with a wheat crust...



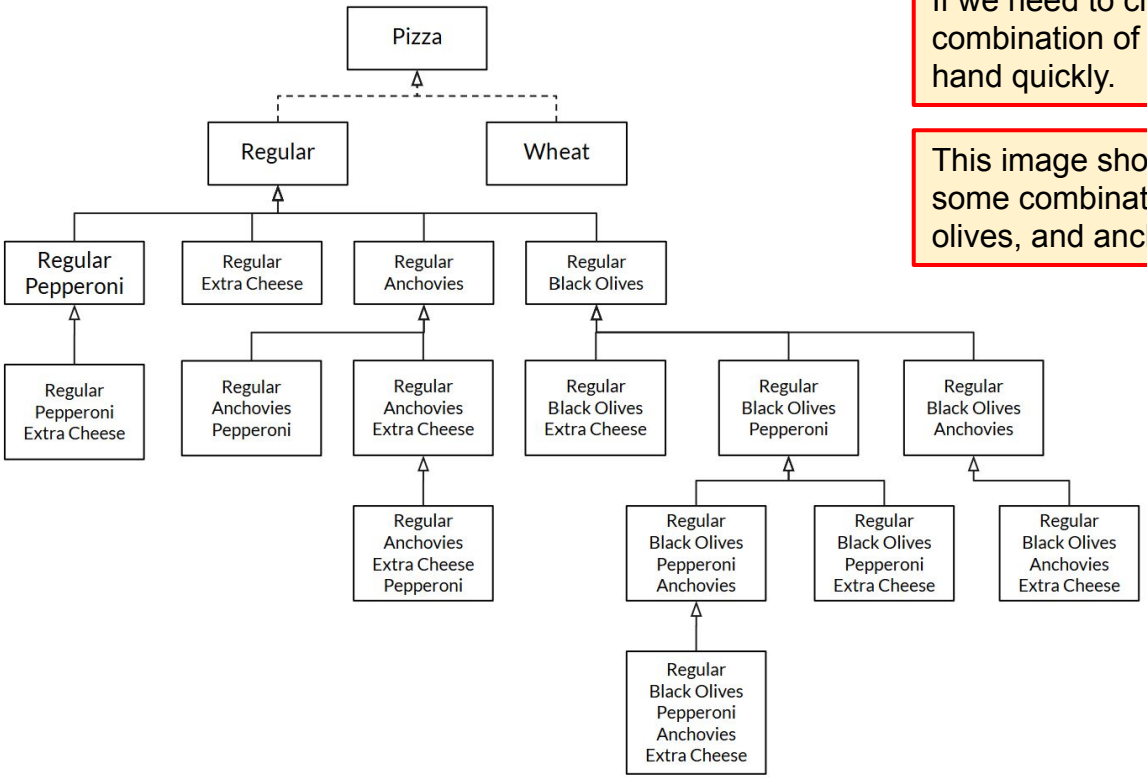
# Subclassing

But what if the customer wants both pepperoni and extra cheese? On either crust?

And what about other combinations like pepperoni, peppers, anchovies, and black olives?



# Combinatorial Class Explosion




If we need to create a different class for every possible combination of toppings and crust, things will get out of hand quickly.

This image shows only the pizzas with *regular* crust and some combination of extra cheese, pepperoni, black olives, and anchovies.

There are ***six*** more toppings to choose from!

Not to mention we'd need all the same options on a wheat crust as well.

And what happens if we add new toppings or crust options?! This solution is obviously not scalable.

# A Component Interface

```
public interface PizzaOrder {
  String prepInstructions();
  String cookInstructions();
  double totalPrice();
}
```

As before, we will start by creating an interface that represents one of the ***components*** that may make up a pizza order.

It should define the behaviors that we expect from any pizza order, i.e. prep instructions, cook instructions, and total price.

# A Concrete Component(s)

Next, we will create one or more ***concrete components*** to implement the most basic kinds of pizza order.

Each will provide an implementation of all of the methods defined in the ***component*** interface.

In this case, a basic wheat crust pizza with red sauce and mozzarella cheese. Remember, wheat crust adds $1 to the base price of the pizza.

We'd need another ***concrete component*** for pizza with regular crust.

```
public class WheatCrust implements PizzaOrder {
  private Size size;

  public String prepInstructions() {
     return "Stretch " + size + " wheat " +
      "dough onto " + size + " pan. " +
      "Add sauce and 1/2\" layer of cheese.";
  }

  public String cookInstructions() {
     return "Place in oven. Bake at 450 " +
       "degrees for 12 minutes.";
  }

  public double totalPrice() {
     return size.basePrice() + 1;
  }
}
```

# A Decorator

Next, we'll need a ***decorator*** class that also implements the ***component*** interface.

But the decorator also *wraps* another instance of our component interface. By default, each of the methods *delegates* to the same method on the wrapped component.

For this reason, sometimes decorators are also called ***wrappers***.

Because the ***decorator*** and ***concrete component*** share the same interface, external clients do not need to distinguish between them.

```
public abstract class OrderOption
                    implements PizzaOrder {
  protected PizzaOrder order;

  public OrderOption(PizzaOrder order) {
    this.order = order;
  }

  public String prepInstructions() {
     return order.prepInstructions();
  }

  public String cookInstructions() {
     return order.cookInstructions();
  }

  public double totalPrice() {
     return order.totalPrice();
  }
}
```

# A Decorator

Finally, we will create a ***concrete decorator*** for each of the different order options that the customer may choose for their pizza.

Each extends the ***decorator*** and *overrides* any of the methods that should change behavior based on the option..

We refer to these methods that add, modify, or replace behavior as ***decorations***.

Each ***concrete decorator*** may alter some or all of the behaviors in its wrapped component.

```
public class ExtraCheese
                    extends OrderOption {

  public ExtraCheese(PizzaOrder order) {
    super(order);
  }

  public String prepInstructions() {
     return super.order.prepInstructions() +
       “Add an additional 1/2\" layer of ” +
       “ extra cheese.”;
  }

  public double totalPrice() {
     return super.order.totalPrice() + 1;
  }
}
```

# Different Decorators

There are three basic ways that a ***concrete decorator*** may implement each of the methods in the ***component*** interface.

It may simply ***pass through*** and use the wrapped component's implementation of the method (e.g. prep instructions and cook instructions in these examples).

It may ***modify*** the behavior in its wrapped component, usually by doing something *before* or *after* calling the method on the wrapped component, e.g. by discounting the price by 15%.

It may ***completely replace*** the behavior, e.g. a buy one get one free offer that reduces the cost of the pizza to $0.

```
public class Coupon extends OrderOption {
  public Coupon(PizzaOrder order) {
    super(order);
  }

  public double totalPrice() {
     return super.order.totalPrice() * 0.85;
  }
}
```

```
public class BOGO extends OrderDecorator {
  public BOGO(PizzaOrder order) {
    super(order);
  }

  public double totalPrice() {
     return 0;
  }
}
```

# Building a Pizza Order

```
PizzaOrder order = new WheatCrust(Size.LARGE);

order = new ExtraCheese(order);
order = new Pepperoni(order);
order = new Coupon(order);

PizzaOrder second = new RegularCrust(Size.SMALL);
second = new Anchovies(order);
second = new BOGO(second);
```

Building a new pizza order starts with choosing which of the ***concrete components*** to instantiate.

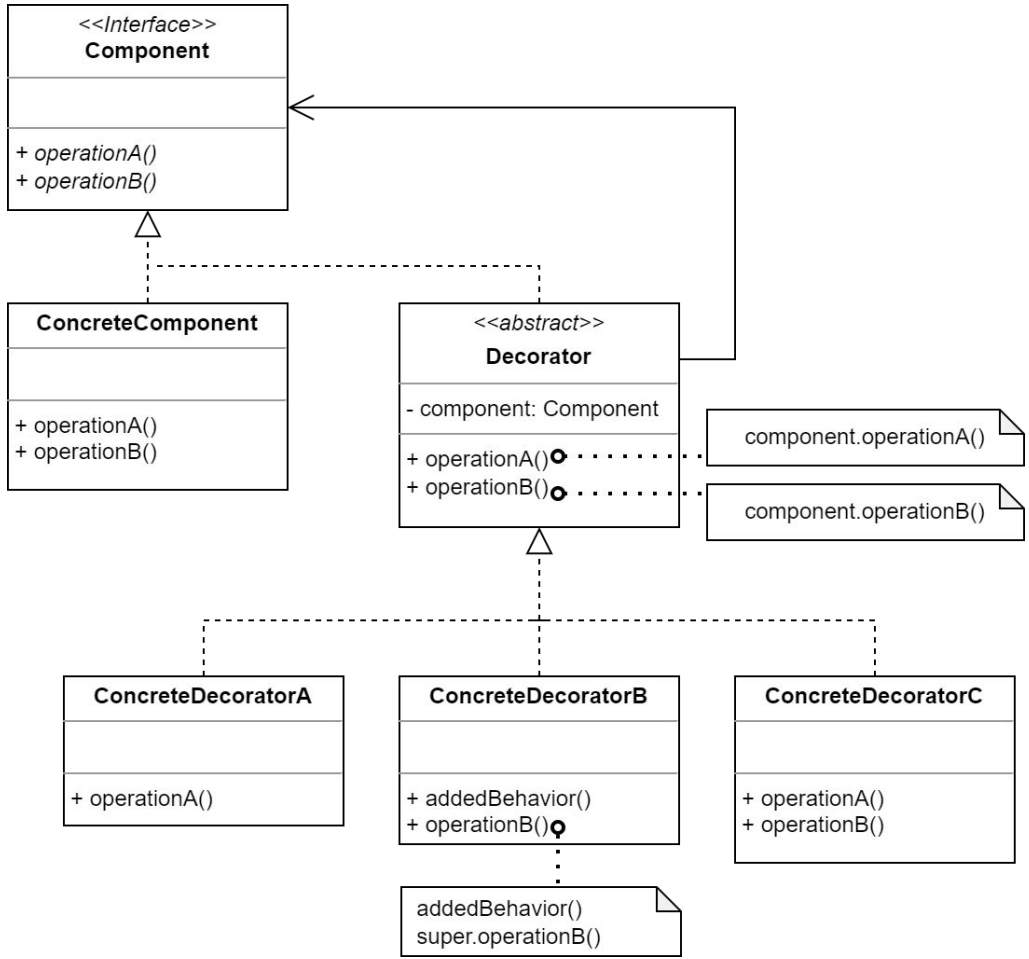
New options are added to the order by creating the appropriate ***concrete decorators*** and passing a ***component*** into the constructor.

Note that the ***component*** passed into the constructor may be a ***concrete component***...

...or another ***concrete decorator***! Each decorator adds its unique behavior to the order, for example, getting the prep instructions on our first order might return...

```
Stretch LARGE wheat dough onto LARGE
pan. Add sauce and 1/2" layer of cheese.
Add an additional 1/2" layer of extra
cheese. Add a single layer of pepperoni
covering 50% of the surface area.
```

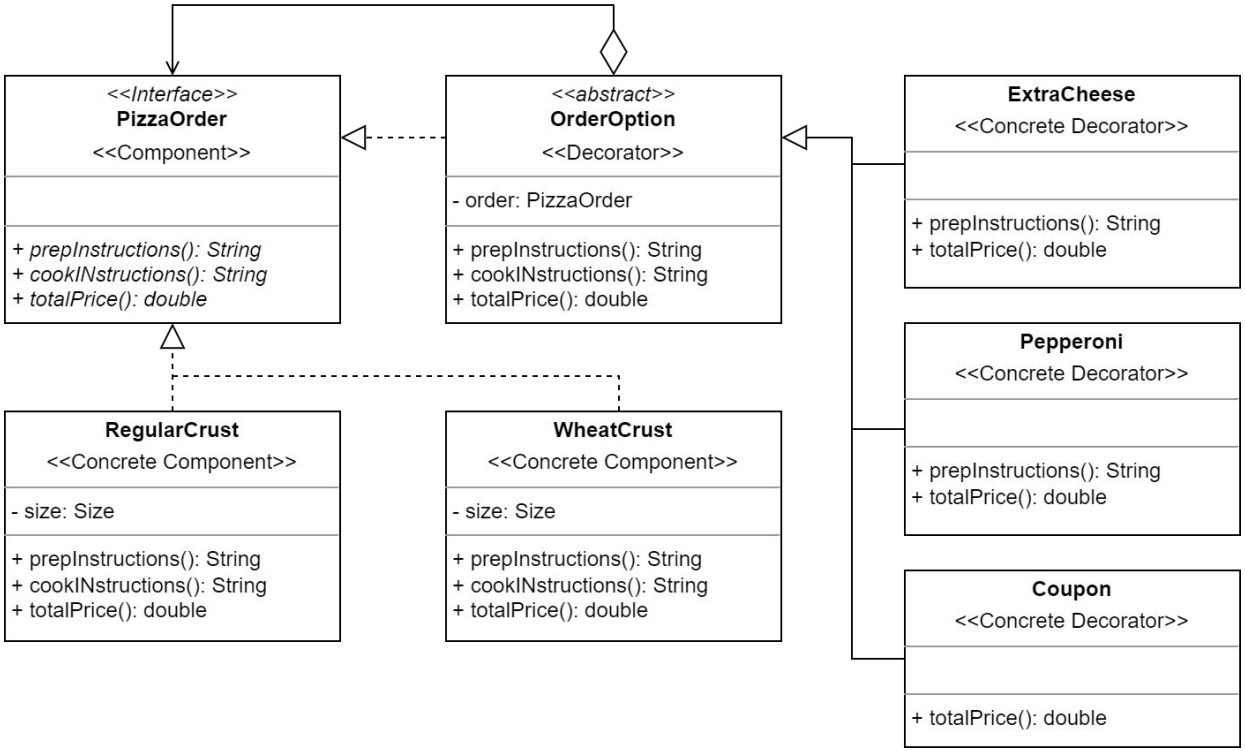
# GoF Decorator Structure




*Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.*

*(Structural)*

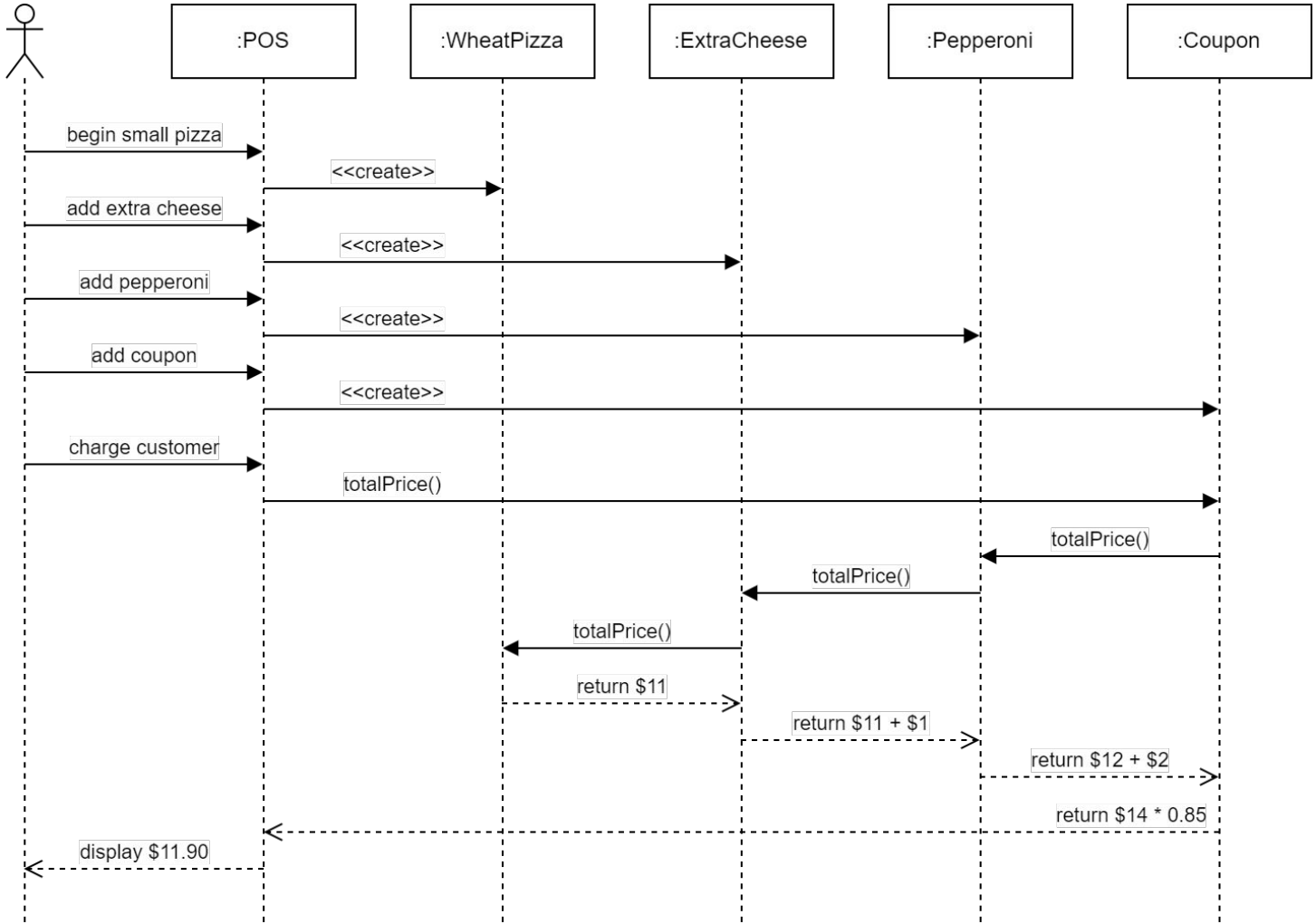
# Pizza POS System Design




As usual, each class has a context specific name...

...but its role in the pattern is indicated in << *guillemets* >>.

Note that there are many more concrete decorators than can be depicted on this slide.


:POS
:WheatPizza
:ExtraCheese
:Pepperoni
:Coupon
begin small pizza
<<create>>
add extra cheese
<<create>>
add pepperoni
<<create>>
add coupon
<<create>>
charge customer
totalPrice()
totalPrice()
totalPrice()
totalPrice()
return $11
return $11 + $1
return $12 + $2
return $14 * 0.85
display $11.90

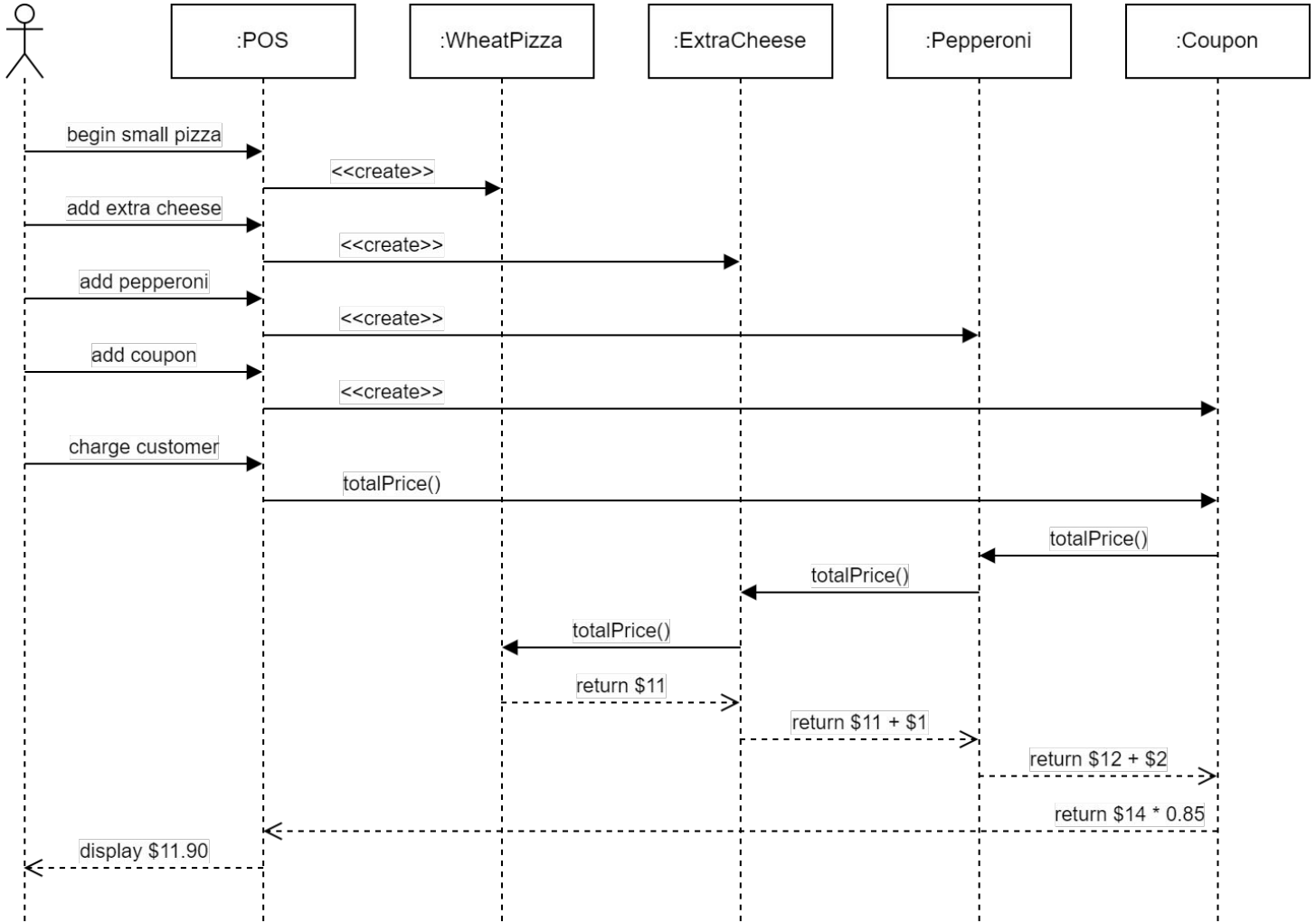
This diagram shows a cashier creating an order for a small, wheat crust pizza with extra cheese and pepperoni.

As each topping is added, a new decorator is created.

Each topping "decorates" or "wraps" the previous order.

The coupon must be applied last (this is a deviation from the standard pattern).

When computing the total price, each decorator must get the price from its wrapped component, then add/modify it accordingly.



# GoF Pattern Card

There are *way* too many concrete decorators to fit on one slide, even with 8 pt font.

Thankfully, this is only an abbreviated example. Any decorator GoF cards that ***you*** submit should include a row for every participant.

Remember that it is OK for a GoF card to span multiple pages in a design document.

| Name: *Pizza POS System* | | GoF Pattern: *Decorator* |
|---|---|---|
| **Participants** | | |
| Class | Role in Pattern | Participant's Contribution in the context of the application |
| *PizzaOrder* | *Component* | *Defines the essential behavior for a pizza order, namely: that every order has prep instructions, cook instructions, and a total price.* |
| *RegularCrust* | *Concrete Component* | *A basic cheese pizza with a regular (white) crust, red sauce, and no additional toppings. Total price is determined by the size of the pizza.* |
| *WheatCrust* | *Concrete Component* | *A basic cheese pizza with a wheat crust, red sauce, and no additional toppings. There is a $1 markup in the price for wheat crust.* |
| *OrderOption* | *Decorator* | *Base class for options that may modify an order including toppings and coupons. Contains a wrapped PizzaOrder. By defualt, passes all method calls through to the wrapped component. Abstract to prevent instantiation.* |
| *ExtraCheese* | *Concrete Decorator* | *An order option that modifies prep instructions to add extra cheese. Also adds $1 to the total price of the wrapped order.* |
| *Coupon* | *Concrete Decorator* | *An order that discounts the total price of the wrapped order by 15%. Must be applied LAST.* |
| Deviations from the standard pattern: *There are two concrete components. Coupon and BOGO decorators must be applied last.* | | |
| Requirements being covered: *1. Pizza orders, 2. Order customization, 3. Calculate total price.* | | |

# Decorator

There are several *consequences* to implementing the decorator pattern:

- *Decorators provide more flexibility than static inheritance.*
- *Avoids feature-laden classes high up in the class hierarchy.*
- *A Decorator and its Component implement the same interface but are not identical.*
- *Lots of little objects.*

Things to Consider

1. How does Decorator help to alleviate class explosion?
2. How does Decorator handle coupling and cohesion in the system?
3. OCP?
4. Given the similar nature of Composite and Decorator, how would you decide which one to use?