MSO
Decorator

Wouter Swierstra, Hans Philippi

October 17, 2018

This lecture

- Recursion
- Decorator pattern

Recursion

Recursion

```
static void Rec1 (int value)
{
    if (value > 0)
    {
        Console.Write(value);
        Rec1 (value - 1);
    }
}
```

- Note that method `Rec1` uses method `Rec1`
- This phenomenon is called *recursion*
- Suppose we call this method with parameter 4
- What will the output look like?

Recursion

```
static void Rec1 (int value)
{
    if (value > 0)
    {
        Console.Write(value);
        Rec1 (value - 1);
    }
}
```

- Suppose we call this method with parameter 4
- The output will be: 4 3 2 1

Recursion

```
Rec1(4);
->  write(4); Rec1(3);
    ->  write(3); Rec1(2);
        ->  write(2); Rec1(1);
            ->  write(1); Rec1(0);
                ->
                <-
            <-
        <-
    <-
<-
```

- Suppose we call this method with parameter 4
- The output will be: 4 3 2 1

Recursion, another one

```
static void Rec2 (int value)
{
    if (value > 0)
    {
        Rec2 (value - 1);
        Console.Write(value);
    }
}
```

- Suppose we call this method with parameter 4
- The output will be: ?

Recursion, another one

```
Rec2(4);
->  Rec2(3);
    ->  Rec2(2);
        ->  Rec2(1);
            ->  Rec2(0);
                ->          <-
            write(1); <-
        write(2); <-
    write(3); <-
write(4); <-
```

- Suppose we call this method with parameter 4
- The output will be: 1 2 3 4

Recursion

- Is a way to express iteration
- The code before the recursive method call will be executed in the order of recursion (4 3 2 1)
- The code after the recursive method call will be executed in reversed order (1 2 3 4)

Furthermore, recursion . . .

- Is omnipresent in functional programming, but . . .
- . . . keep an eye on termination
- Is essential for the Decorator pattern

Towards the Decorator pattern

Menu

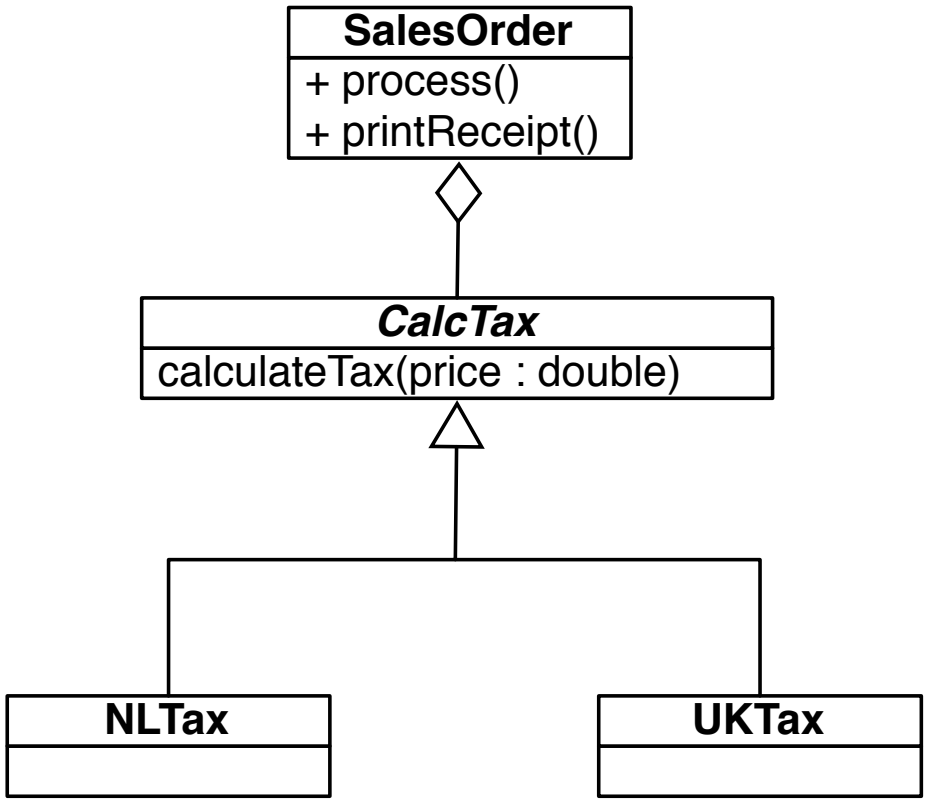
pannenkoek	stroop
pannenkoek	bosbessenjam
pannenkoek	spek
pannenkoek	kaas
pannenkoek	ui

Innovation & changing requirements

pannenkoek	stroop
pannenkoek	bosbessenjam
pannenkoek	spek
pannenkoek	kaas
pannenkoek	ui
pannenkoek	champignons
pannenkoek	spek & kaas
pannenkoek	spek & stroop
pannenkoek	ui & champignons
pannenkoek	spek & kaas & stroop
speltpannenkoek	stroop
speltpannenkoek	bosbessenjam
...	...

Towards the Decorator pattern

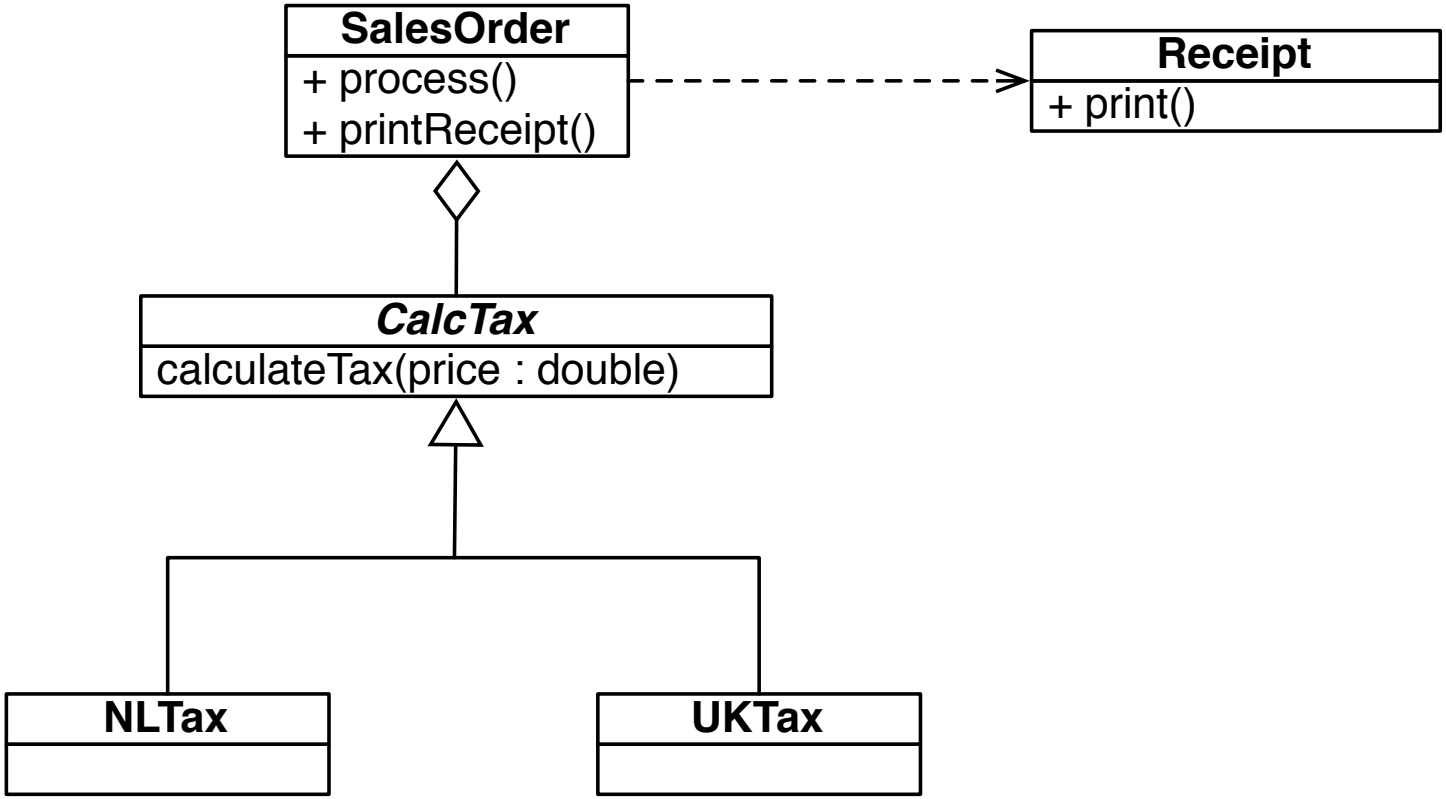
Suppose we have our SalesOrder class, capable of handling different tax calculations



(You should recognize the Strategy pattern)

Changing requirements . . .

But now suppose that we need to print receipts as well

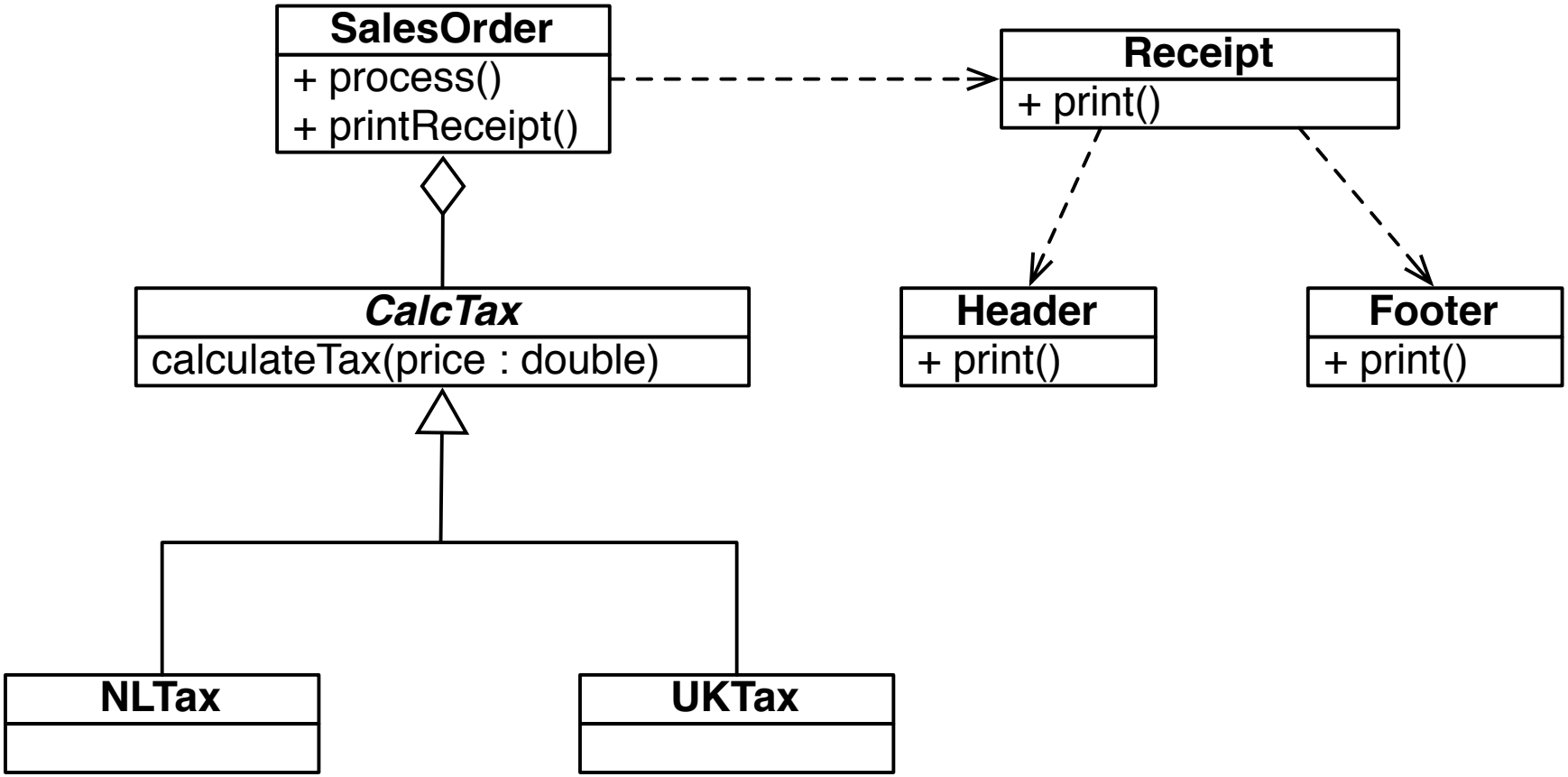


Receipt printing

Each sales receipt may be *decorated* with additional information in the header and footer

- address and tax information
- customer loyalty points, if applicable
- import/export fees, if applicable
- discount code for your next order, if applicable
- ...

First solution



Critique

This will work but:

- you need to include a lot of switches in the Header and Footer classes to decide what to print (bad code smell)
- you will mix the code that prints the headers with the control code, deciding which headers to print (weak cohesion)

Can we do better?

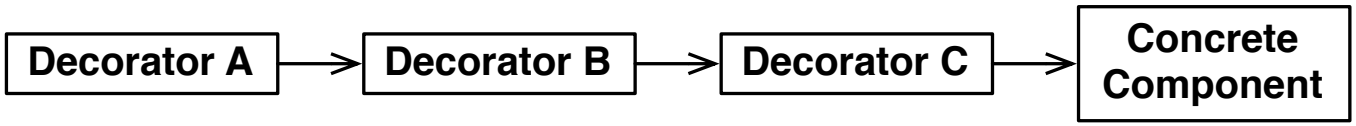
The Decorator Pattern

The Gang of Four describes the intent of the decorator pattern as:

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

A chain of decorators

Intuitively, you can think of decorators as a chain of Decorator objects that all add some new functionality to concrete component.

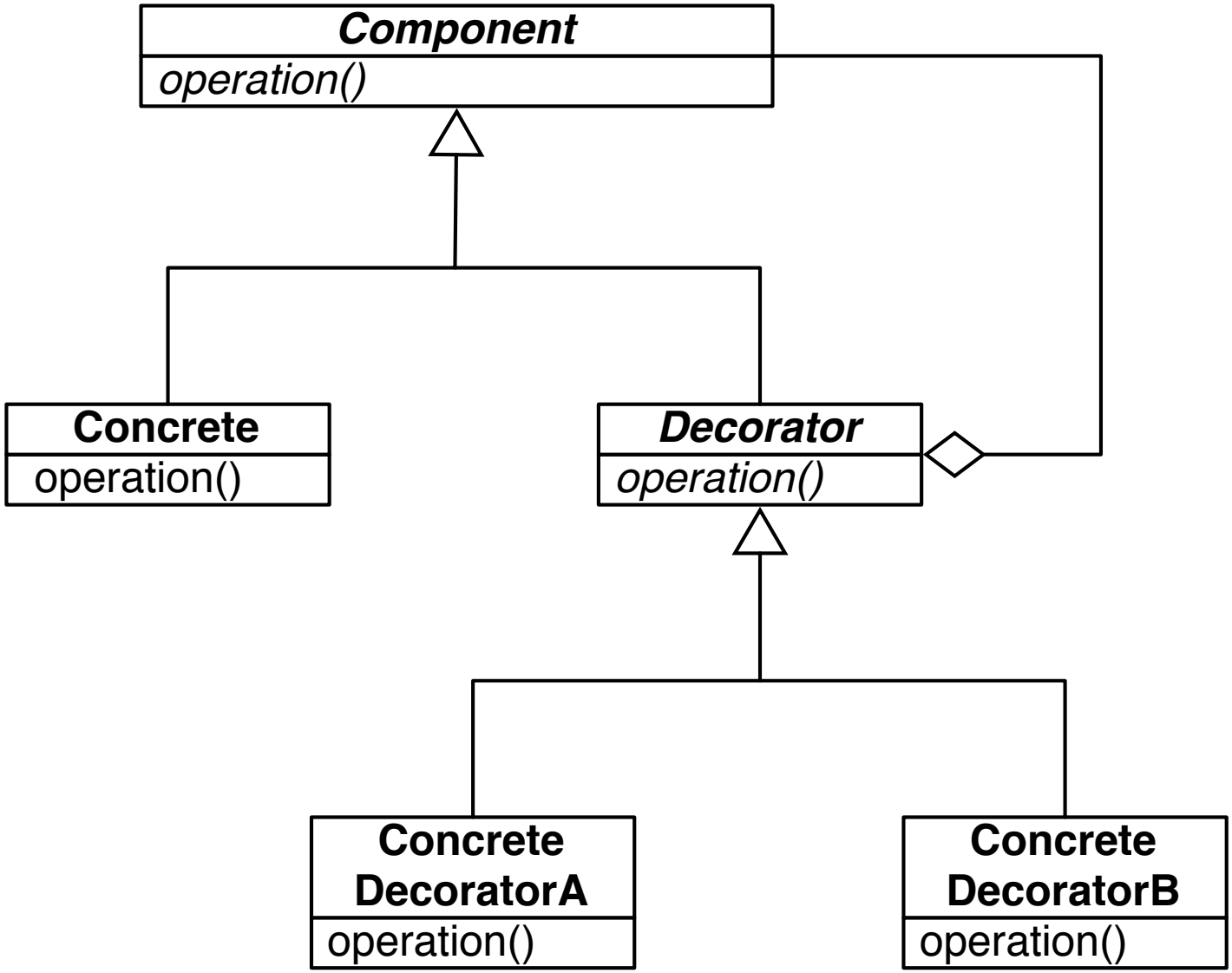


Example:

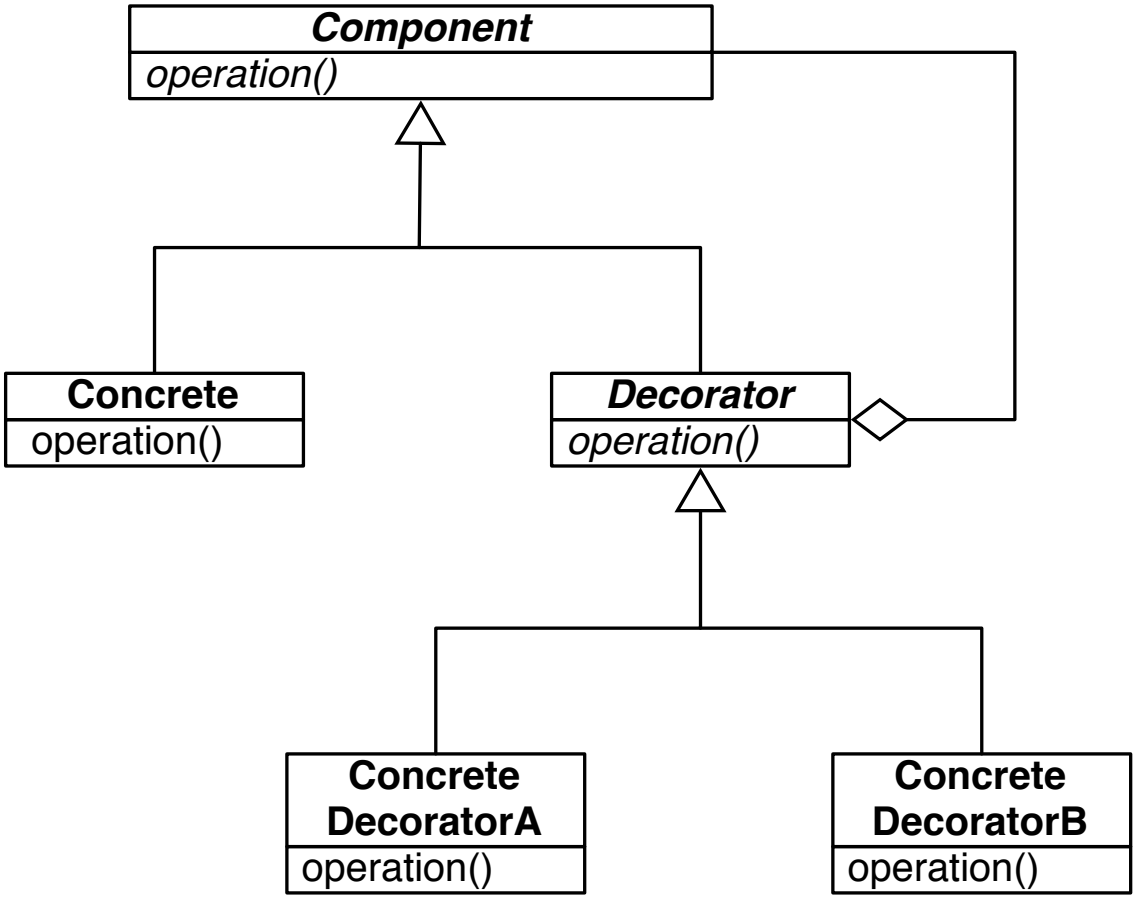
stroop $\rightarrow$ **kaas** $\rightarrow$ **spek** $\rightarrow$ **speltpannenkoek**

How can we organize this in a design?

The Decorator pattern



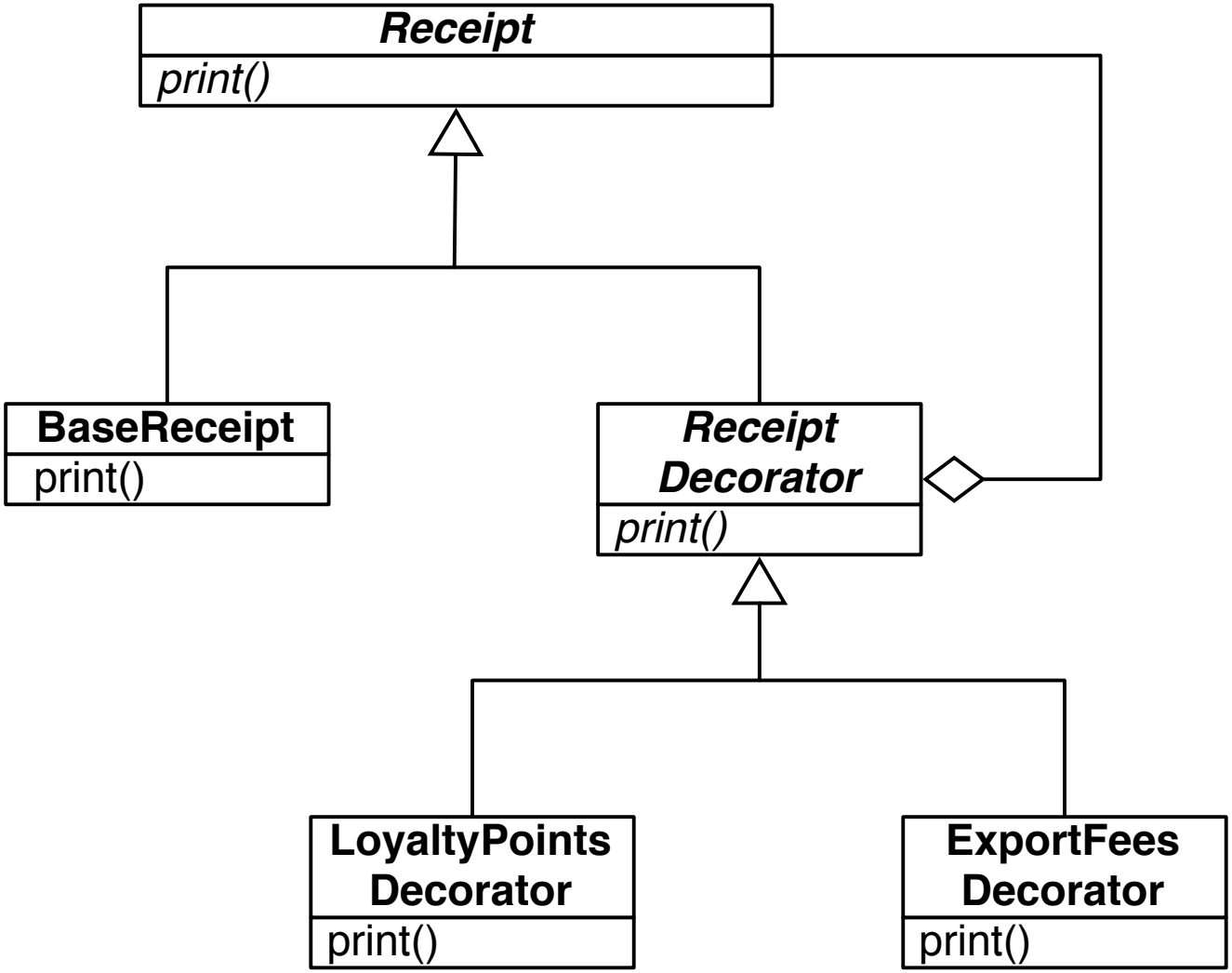
The Decorator pattern



- This Decorator class has a *recursive* component
- It uses a two-level deep inheritance hierarchy – for good reason!

Applying the Decorator pattern

The sales order example:



Keep order!

Note that you can call the next receipt(decorator) *before* or *after* executing the component.

```
class LoyaltyPointsDecorator : ReceiptDecorator
{
  public void print()
  {
    Console.WriteLine("10 points earned");
    receipt.print();
  }
}
```

The order we choose determines whether we get headers or footers.
Question: is LPD a header or a footer?

Ordering example - I

To produce:

```
Header1
Base receipt
Footer1
```

We assemble the following receipt:

```
new Header1(new Footer1(new BaseReceipt()));
```

Or equivalently:

```
new Footer1(new Header1(new BaseReceipt()));
```

Ordering example - II

To produce:

```
Header1
Header2
Base receipt
Footer1
```

We assemble the following receipt:

```
new Header1
  (new Header2
    (new Footer1
      (new BaseReceipt())));
```

Order matters!

Ordering example - II

```
new Header1
  (new Header2
    (new Footer1
      (new BaseReceipt())));
```

Order matters!
But beware:

- the creation order is important for the processing order of the headers w.r.t. each other
- the creation order is important for the processing order of the footers w.r.t. each other
- the order of the recursive method call determines whether it is a header or a footer

Decorators in OO languages

I/O in C# and Java uses lots of decorators

- C# defines an abstract Stream class with methods like BeginRead and Close
- There are concrete instances for writing to a file, network socket, etc.
- Using decorators, you can define your own instance that
  - decrypts the data
  - zips the data
  - buffers data
  - converts the data
  - ...

Decorator – review

- **Intent:** Attach additional responsibilities to an object dynamically.
- **Problem:** The object you want to use does the basic functions you require. However, you may need to add some additional functionality to the the object, occurring before or after the object's basic functionality.
- **Solution:** Allows for extending the functionality without resorting to subclassing.
- **Participants:** The ConcreteComponent is the base class, that is extended with Decorators. Their common abstract superclass, Component, defines the interface for both the ConcreteComponent and Decorator classes.
- **Consequences:** The functionality that needs to be added should be done in (small) Decorator objects. These decorator objects can then be dynamically wrapped around the ConcreteComponent as necessary.

Material covered

- Design Patterns explained: chapter 16