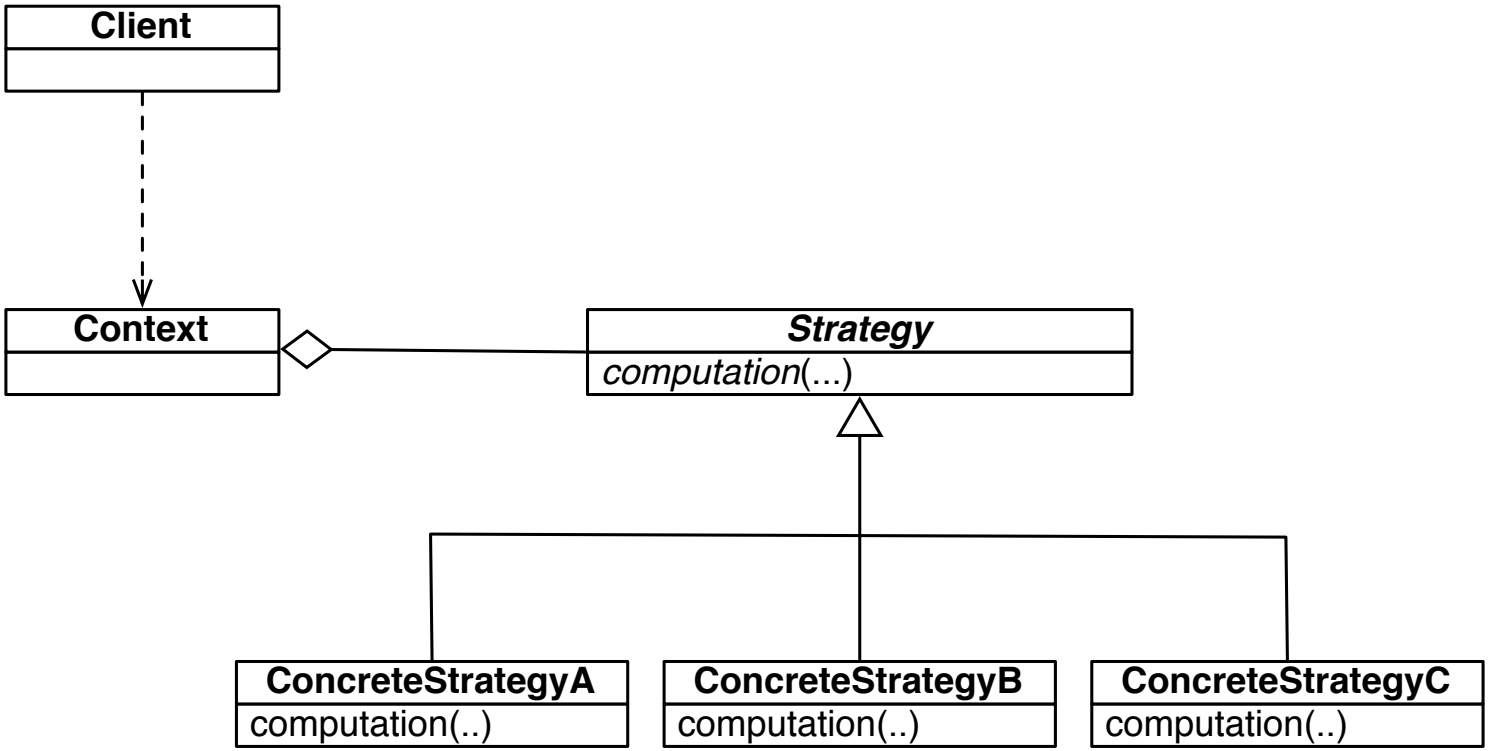
MSO
The Abstract Factory

Hans Philippi

September 20, 2018

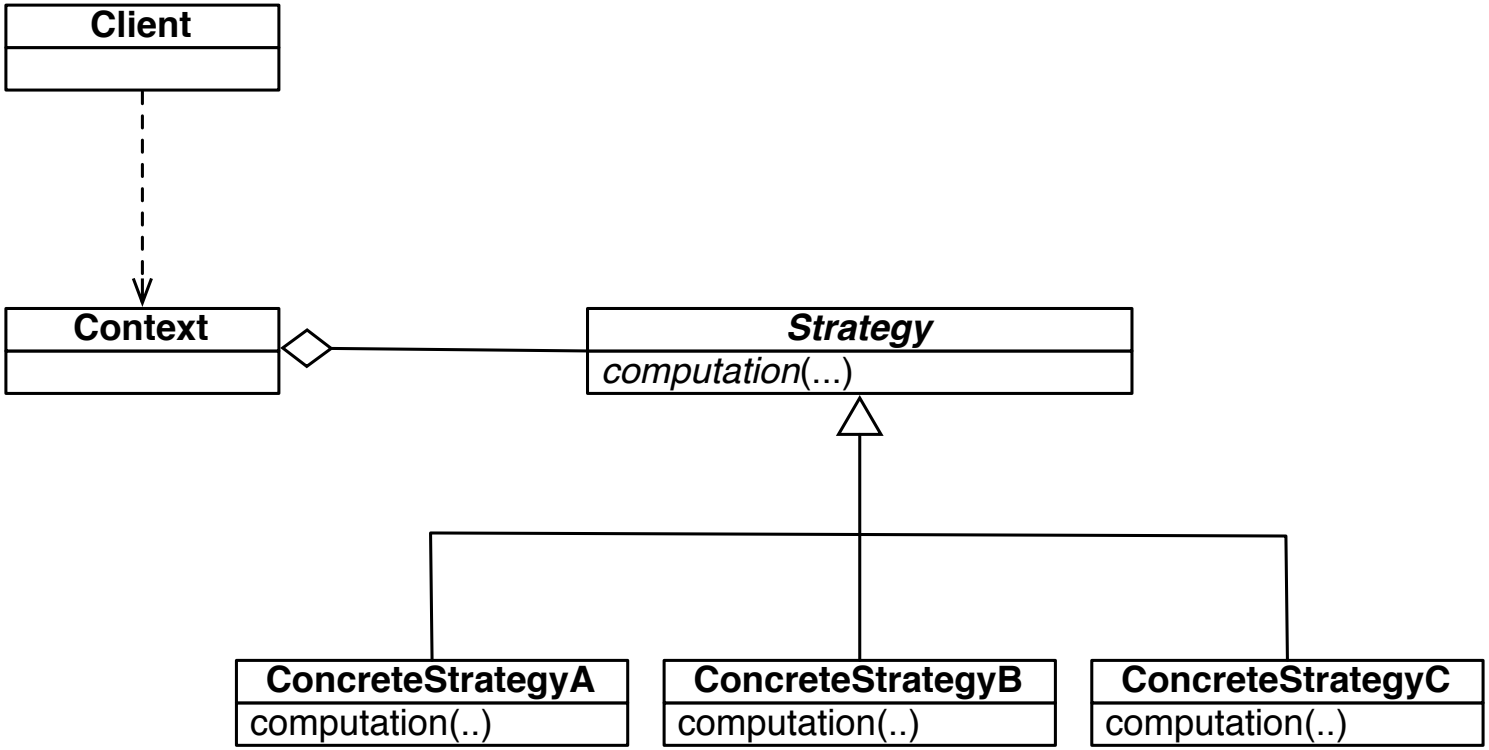
Back to Strategy



Running example: E-commerce; Context = SalesOrder

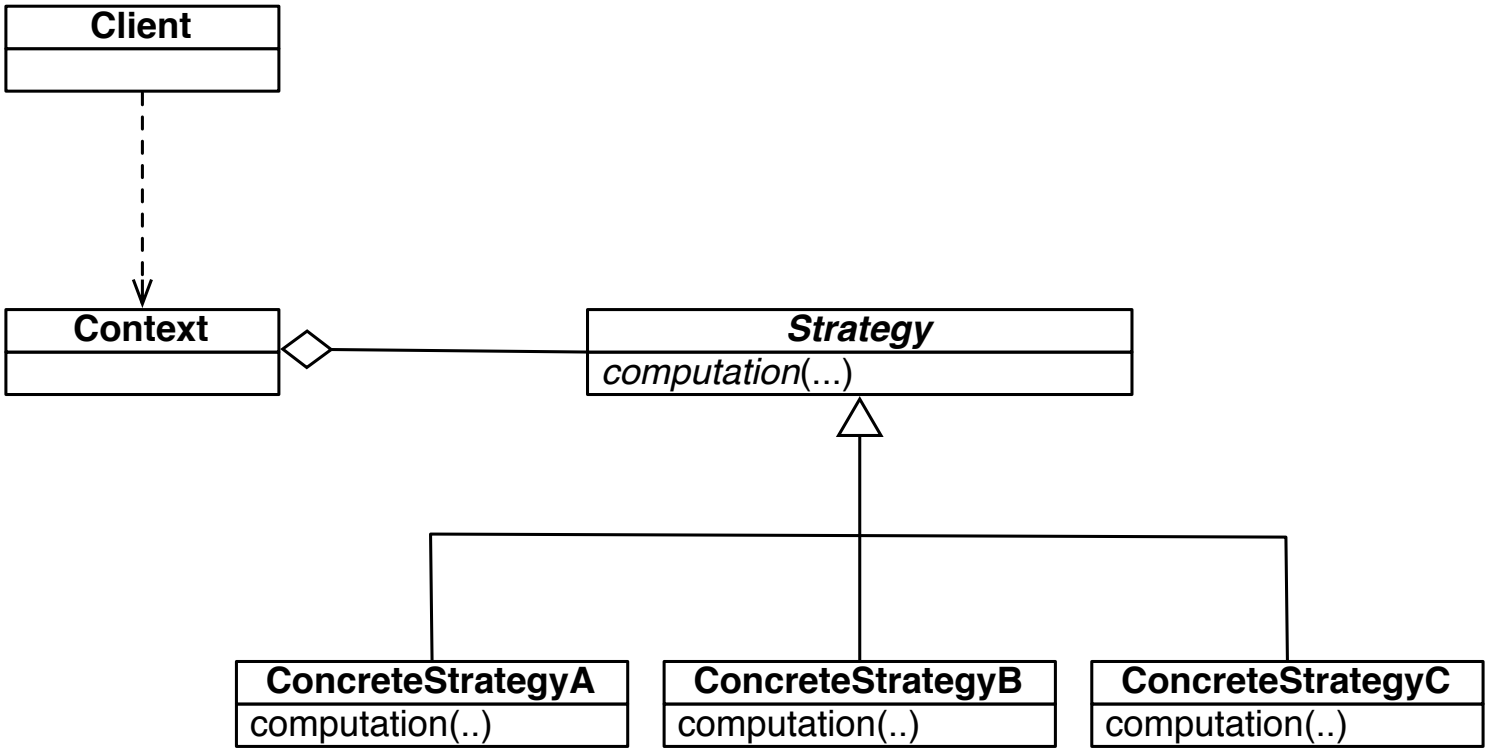
Strategies: taxes, shipping, currency, ...

Strategy



Where is the choice for the specific concrete strategy made?
Where is the `switch()`?

Strategy



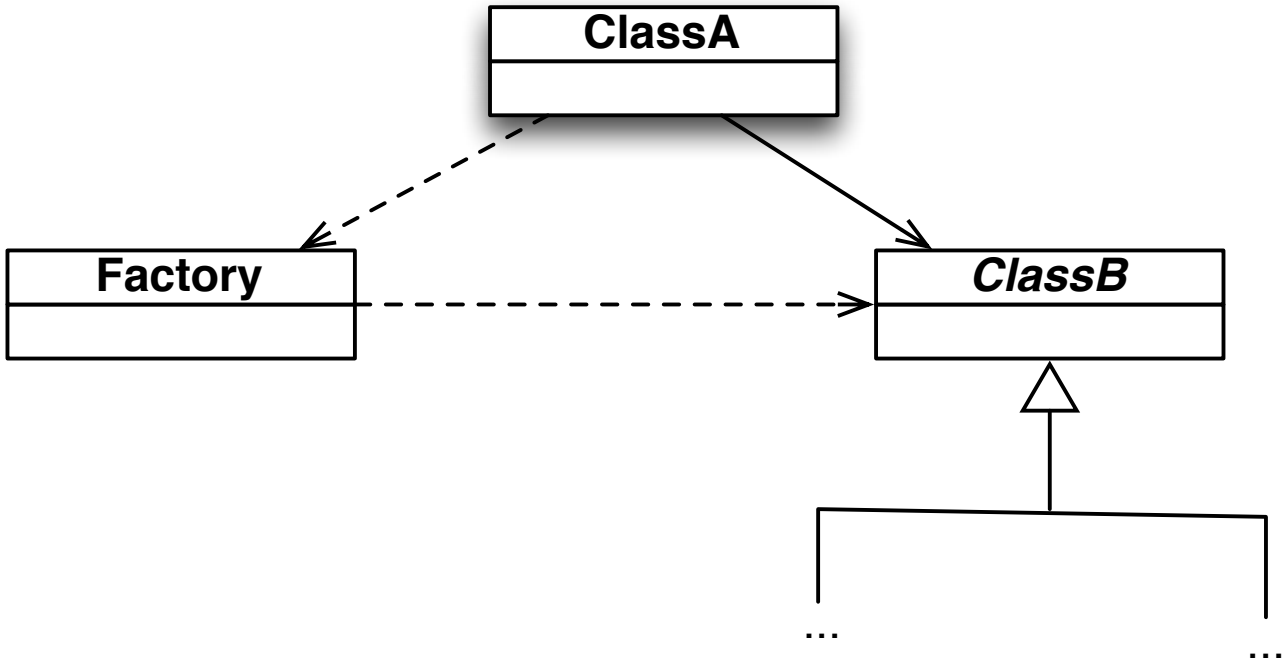
Separate the selection of algorithm from the implementation of the algorithm. This allows for the selection to be made in context.

Pattern: Factories

- At some point we will need to create concrete instances of the abstract class
- To create concrete instances, we need to know about concrete subclasses of the abstract class
- To avoid breaking abstraction, we want to separate the *creation* of objects from their *use*

Factories

Factories are a popular design pattern that separate *creation* from *usage*



Factories – example

In our E-commerce example all classes have a clear responsibility

- The SalesOrder is aware of the country, switches on it, and creates the required concrete Strategy objects
- Each abstract Strategy class defines an interface for SalesOrder to use
- Each concrete Strategy class implements this interface in its own way

Why factories?

Adding or removing concrete Strategy classes changes to two pieces of code to be updated:

- SalesOrder, because it creates the concrete Strategy objects
- the concrete Strategy classes themselves

So far, so good, but SalesOrder has **two** responsibilities: *creating* the concrete Strategy classes and *using* them . . .

Why factories

- There are several design patterns related to object creation
- The key idea is to identify two **separate responsibilities**: *object creation* and *object usage*
- These two responsibilies need to be kept distinct

In our E-commerce example:

- A class that creates the relevant concrete Strategies must know about the specific kinds of Tax and Currency classes, but should never use them
- A class that uses Strategies (like SalesOrder), should never know about the variation in concrete Strategy classes ...
- ... but in our Strategy example, it did both: *choosing* and *using*

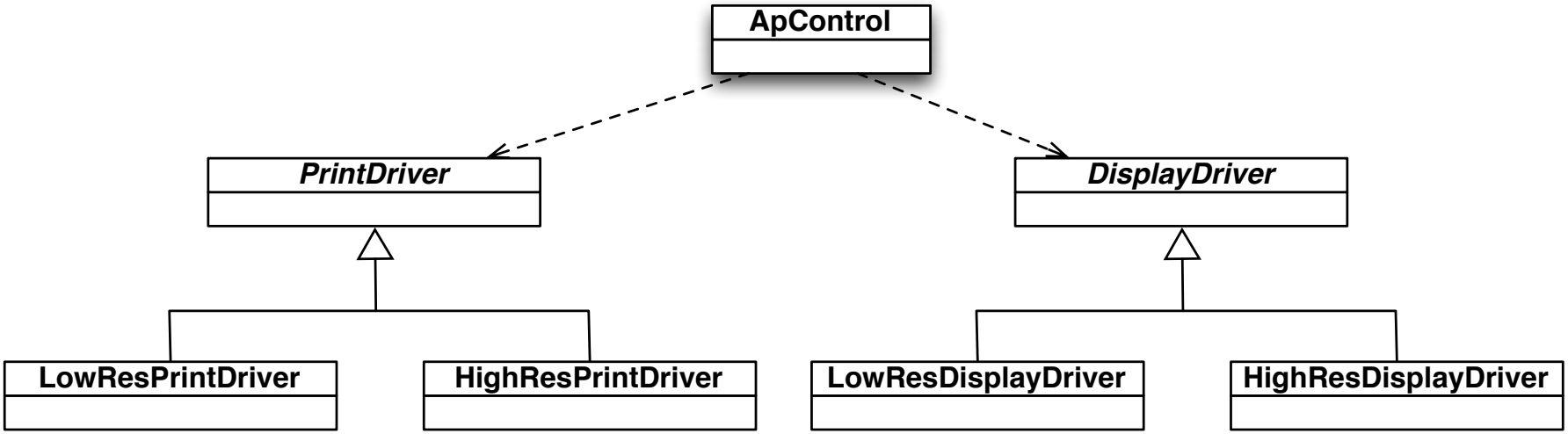
Why factories

- A class that uses Strategies (like SalesOrder), should never know about the variation in concrete Strategy classes ...
- ... but in our example, it did both: *choosing* and *using*
- One of our holy principles is *cohesion*
- But SalesOrders has *two* responsibilities
- It should know about details regarding taxes, currencies and languages in several countries *(Ireland has the euro and English as primary language)*
- It deals with sales orders *(calculate the taxes, print the invoice header)*

Factories: another example

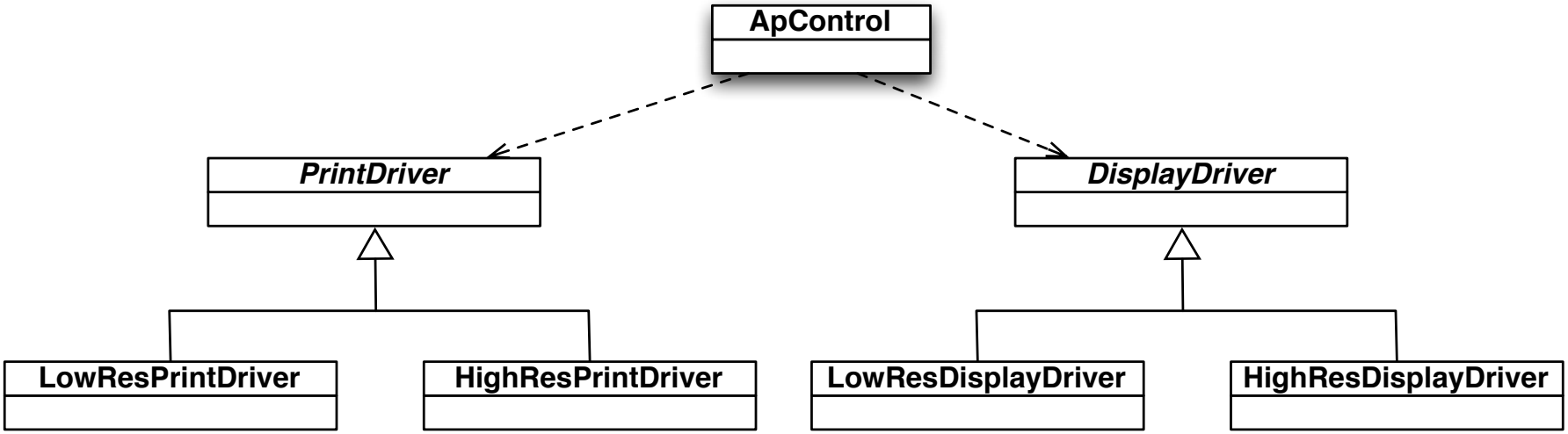
- We will come back to our E-commerce example later
- Suppose have an application that displays data and prints data
- It should be able to run with different resolutions: low and high
- We have to deal with display drivers and print drivers

A Strategy inspired approach



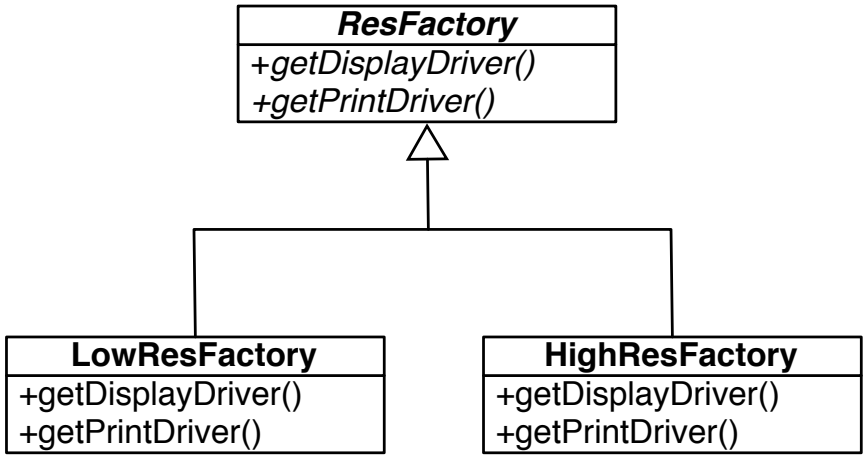
- strong *cohesion*
- easy to add new printer drivers or display drivers

The question



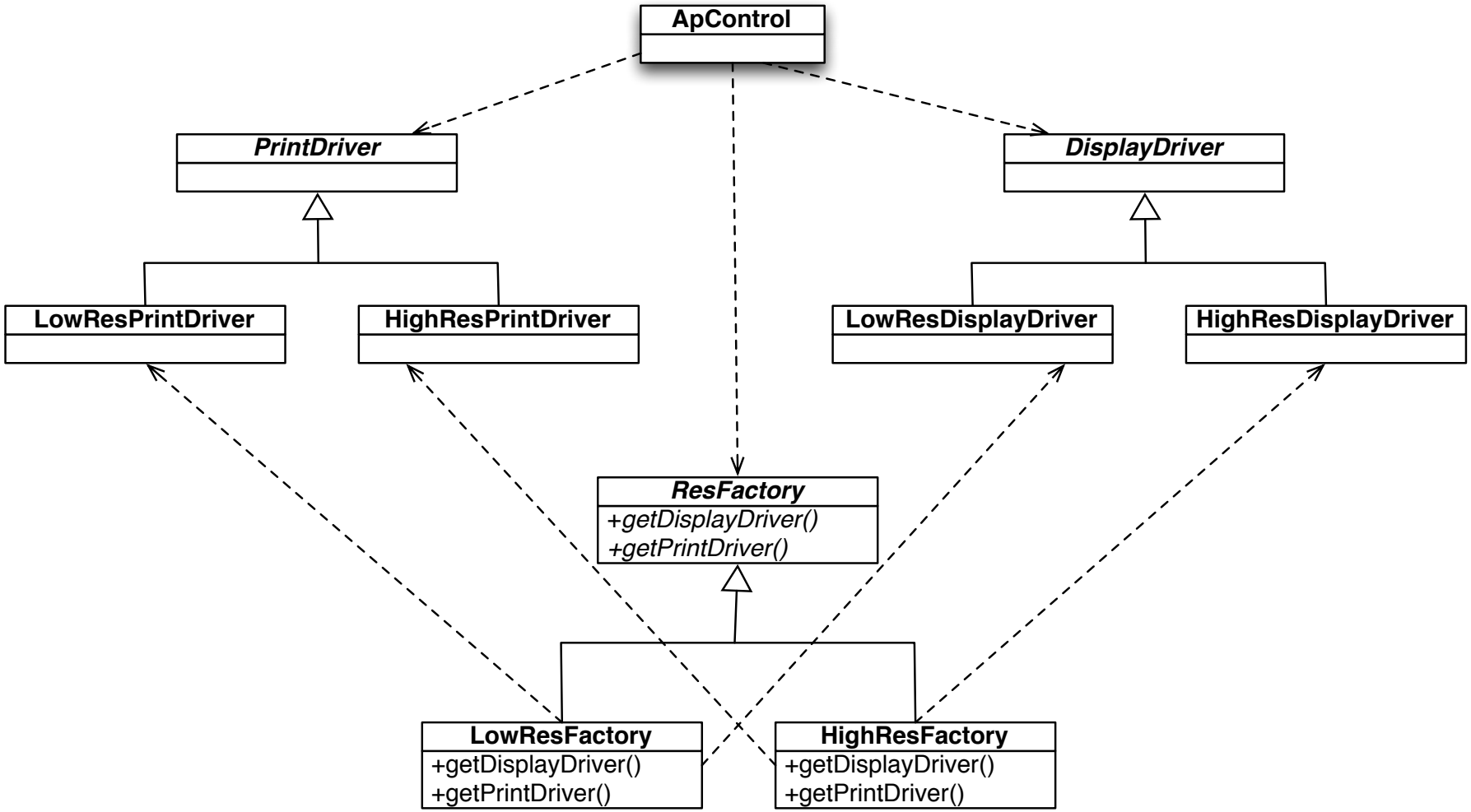
Where does the creation logic go?

Better...



- Here we encapsulate different factories in an abstract class
- The ApControl class only knows about the ResFactory *interface*, not about the individual *implementations*

The big picture



Responsibilities

- The ApControl class only *uses* methods from the PrintDriver and DisplayDriver *abstract* classes
- It creates these drivers using methods from *abstract* class ResFactory
- The concrete LowResFactory and HighResFactory classes create concrete instances of the PrintDriver and DisplayDriver classes
- Let us take a look at some code fragments, related to the previous slide

Resolution example

```
// defining the Factories
abstract class Resfactory
{
    public abstract DisplayDriver
        getDisplayDriver();
    public abstract PrintDriver
        getPrintDriver();
}
```

Resolution example

```
// defining the Factories
class LowResfactory : ResFactory
{
    public override DisplayDriver
        getDisplayDriver();
    {
        return(new LowResDisplayDriver);
    }
    ...
}
```

Resolution example

```
// defining the Factories
class HighResfactory : ResFactory
{
    public override DisplayDriver
        getDisplayDriver();
    {
        return(new HighResDisplayDriver);
    }
    ...
}
```

Resolution example

```
// we also may want to combine  ...
// a high res display with a low res printer
// exercise: fill in the dots

class HiLoResfactory : ResFactory
{
    public override DisplayDriver
        getDisplayDriver();
    {...}


    public override PrintDriver
        getPrintDriver();
    {...}
}
```

Resolution example

```
// in ApControl
switch (Config.RESOLUTION)
{
  case LOW:
    ResFactory myResFactory = new(LowResFactory);
    break;
  case HIGH:
    ResFactory myResFactory = new(HighResFactory);
    break;
  case HILO:
      // High res screen; low res printer
    ResFactory myResFactory =
        new(HiLoResFactory);
    break;
    ...
```

Resolution example

```
// pseudo code for displaying in ApControl

 ...
   DisplayDriver myDisplayDriver =
     myResFactory.getDisplayDriver();
 ...
   showOnDisplay (x, myDisplayDriver);
 ...

 // at this level,
 // all resolution issues have been encapsulated!
```

Responsabilities

- So the only responsibility for ApControl is to *choose* the right concrete factory
- All details and complexity related to the *choice* of the drivers generated by the factory are hidden for ApControl
- All details and complexity related to *using* concrete drivers are hidden for ApControl

What did we gain?

- It might feel a bit like nitpicking . . .
- . . . so let us go back to our E-commerce example
- In the original version, SalesOrder had to be aware of annoying details like *Ireland has the euro*
- In a factory approach, we will see something like this:

```
switch (country)
    case ''IRL'':
        myCurr = myIRLFactory.getCurrency();
```

References

Shalloway and Trott: chapter 11