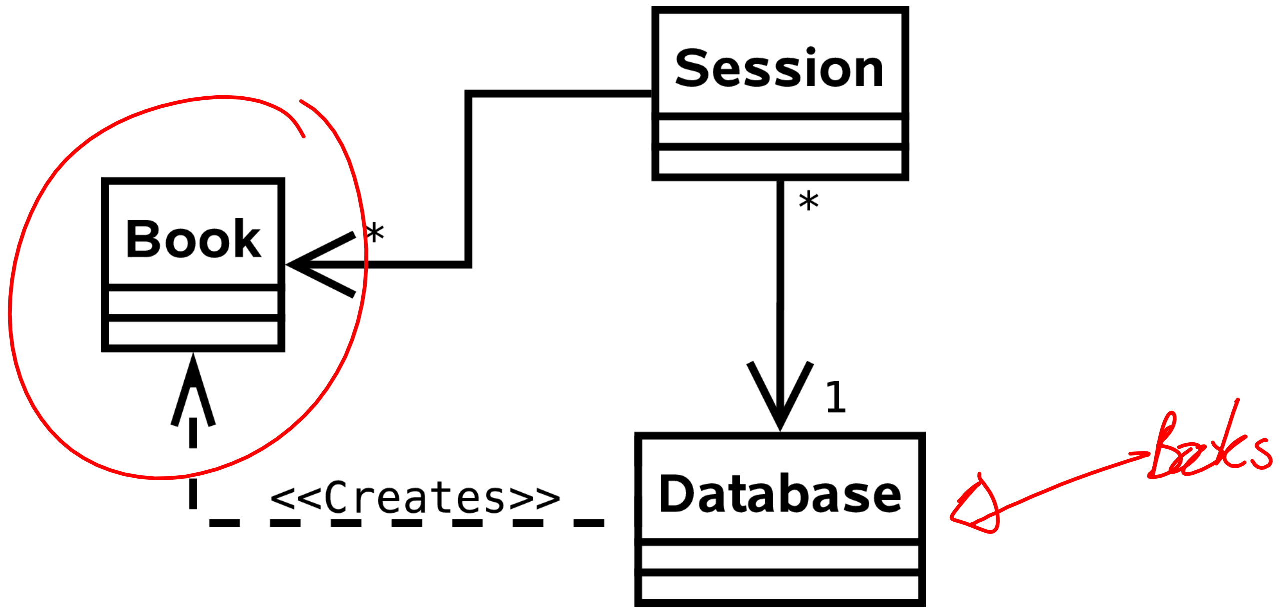
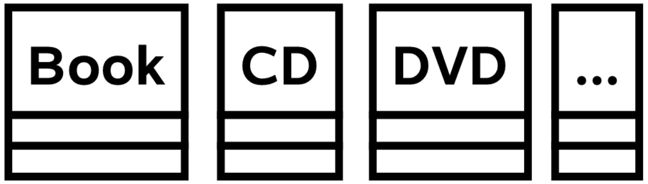
Design Patterns

- Back to generic OOP (not Java)
- Design patterns are generally reusable solutions to commonly occurring problems in software design
- We will spend some time looking at some patterns to:
  - Show you (hopefully) that OOP has some power
  - Demonstrate that naïve solutions may be bad
  - Give you a programmer's vocab.

Scenario

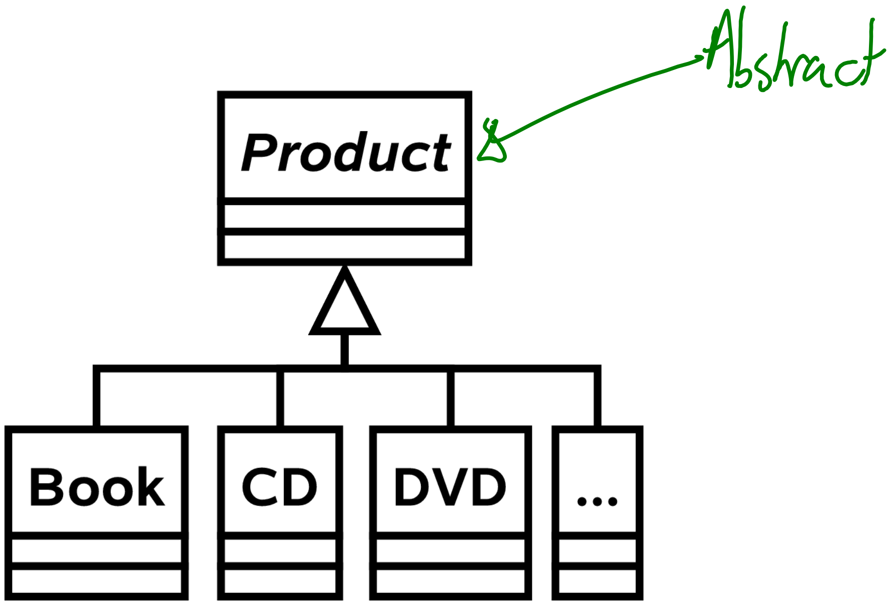


Multiple Products I



Poor extensibility

Multiple Products II



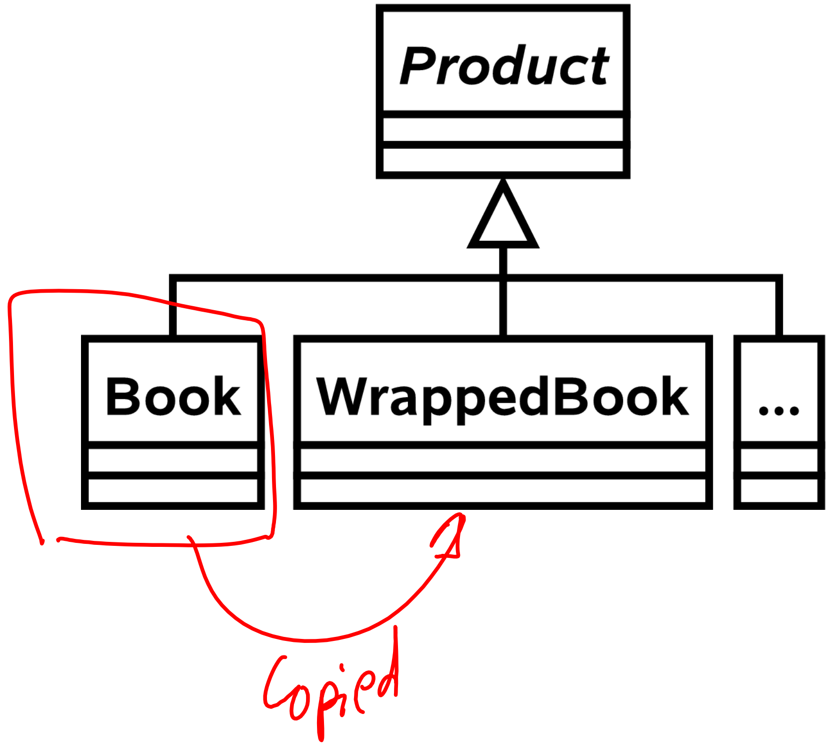
Problem: Want to support gift wrapped products

Decorator I

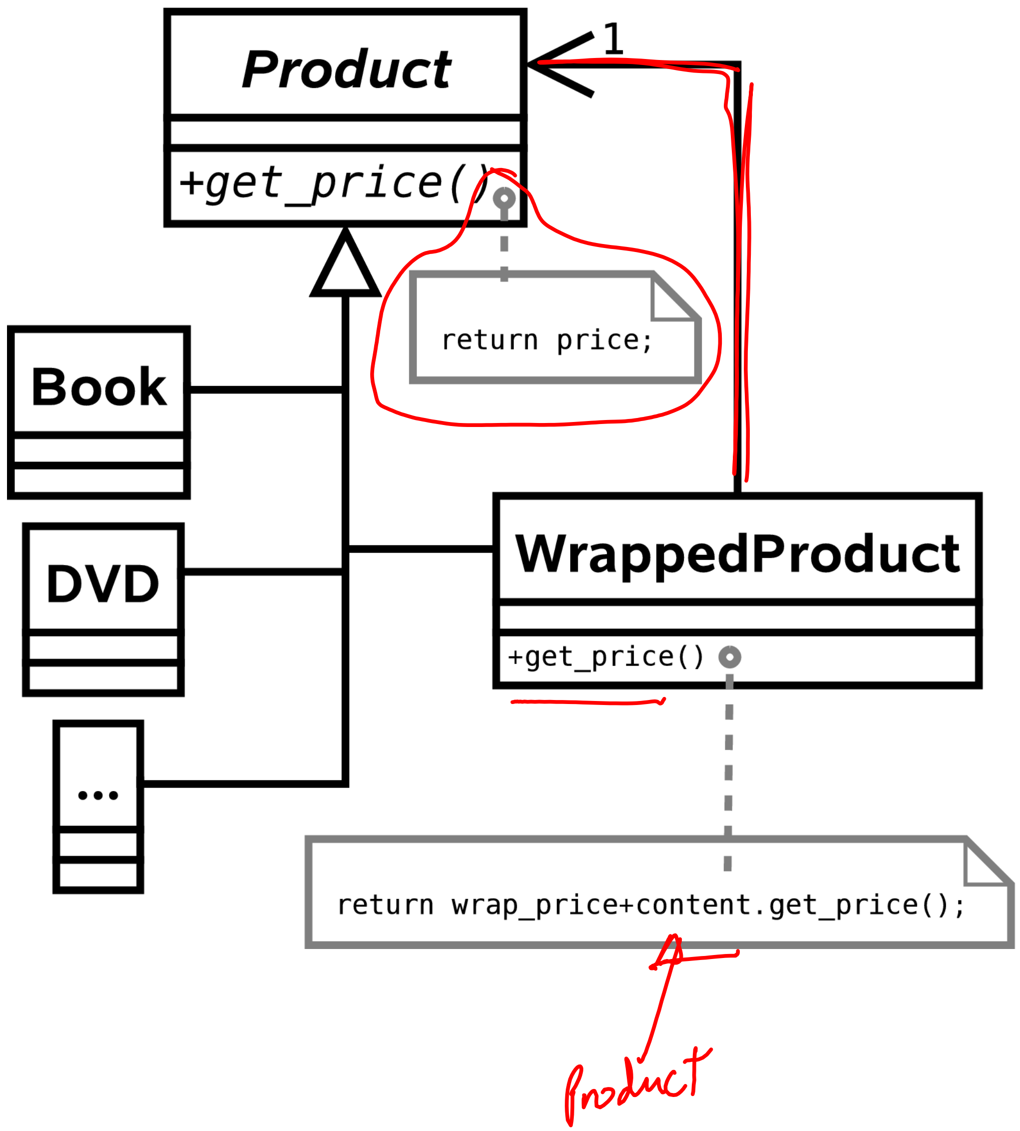
Add state to Product to describe whether or not we should wrap and how

```
public class Product {

      private boolean mWrap = false;
      private int mWrapType = 0;

      ...
}
```

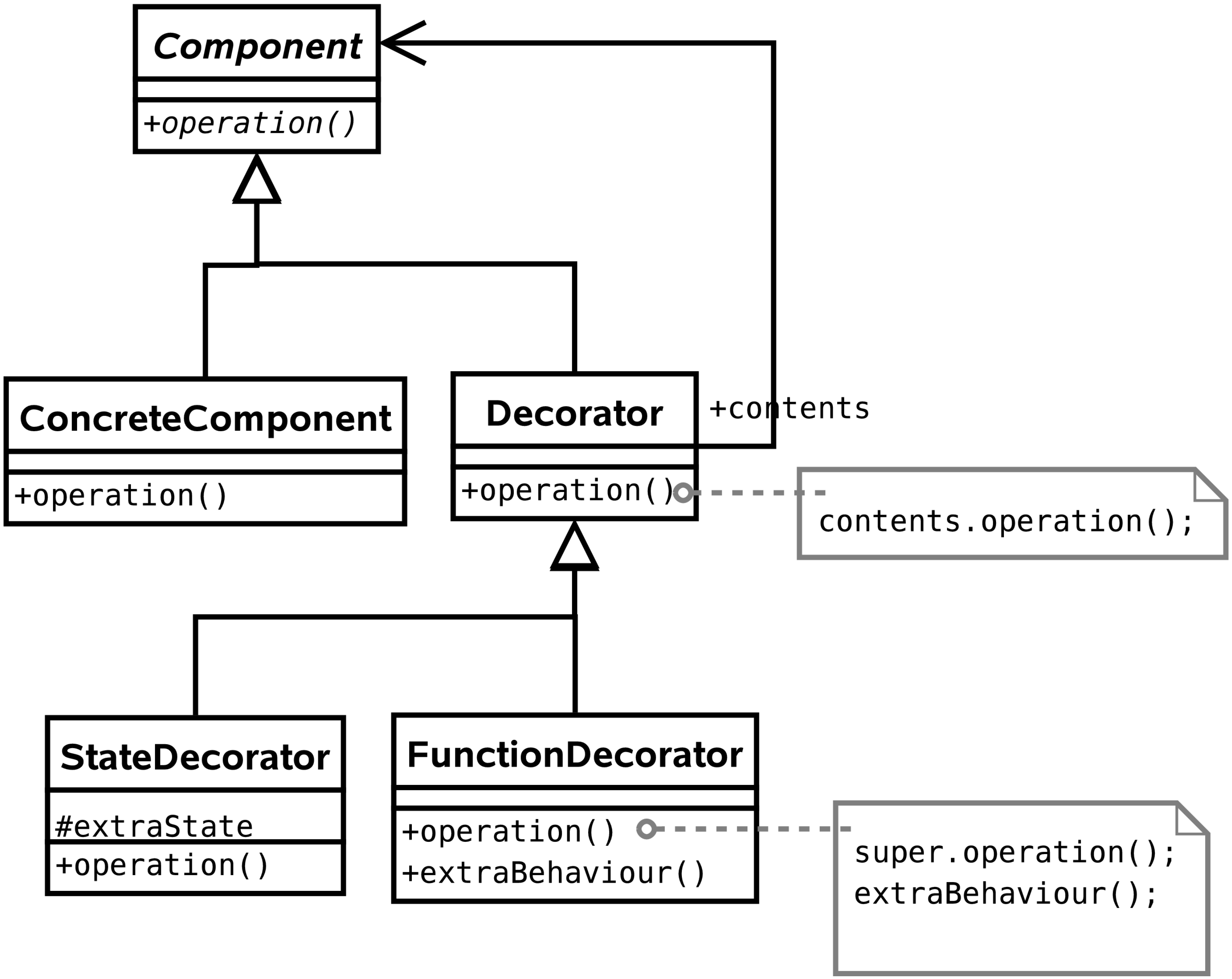
Decorator II



Decorator III



Decorator (General)



Problem: Now need to support gift bags, gift wrapping, gift boxes...

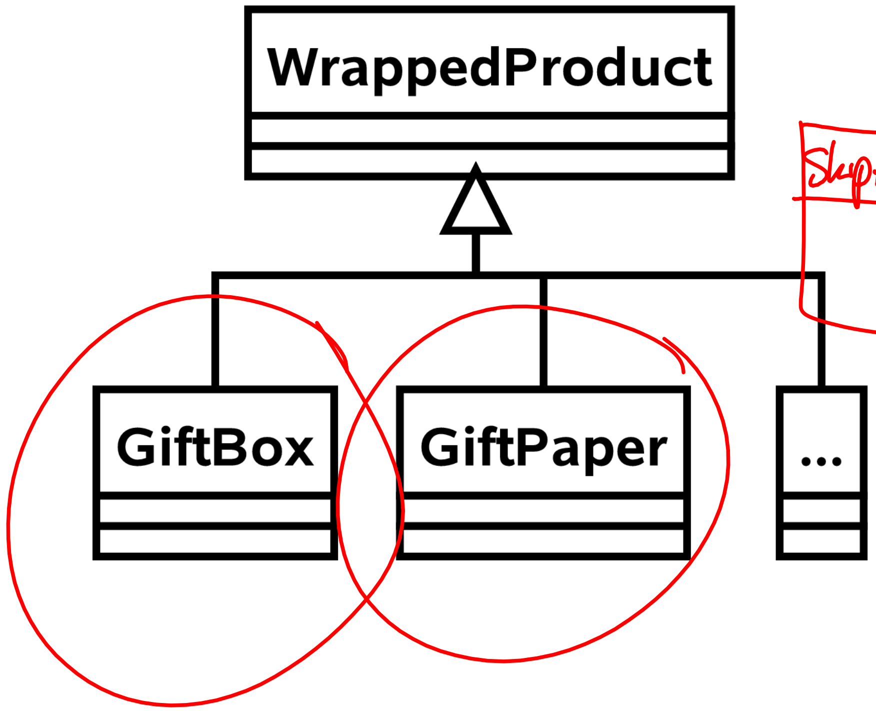
State I

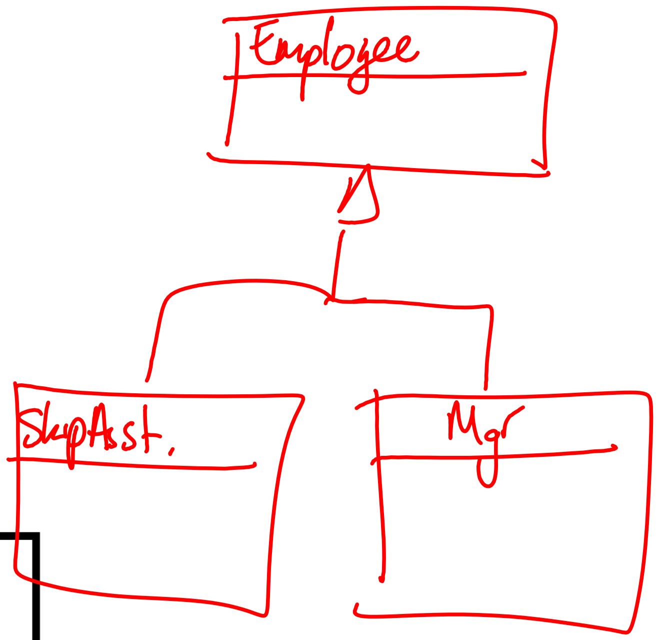
```
void initiate_wrapping() {
   if (wrap.equals("BOX")) {
      ...
   }
   else if (wrap.equals("BAG")) {
      ...
   }
   else ...
```

Huge list

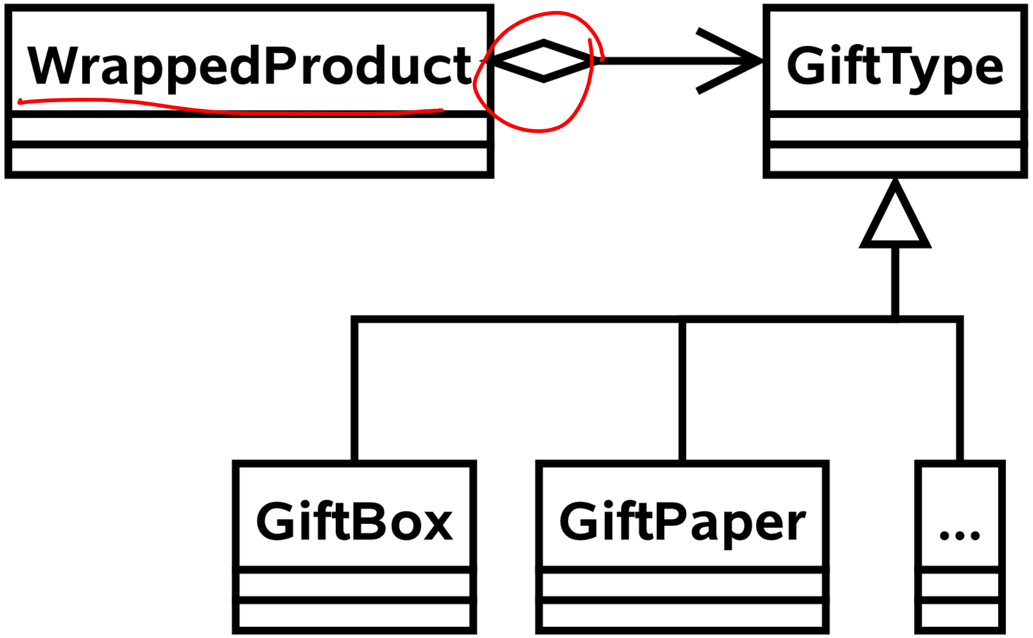
Hard to maintain

State II

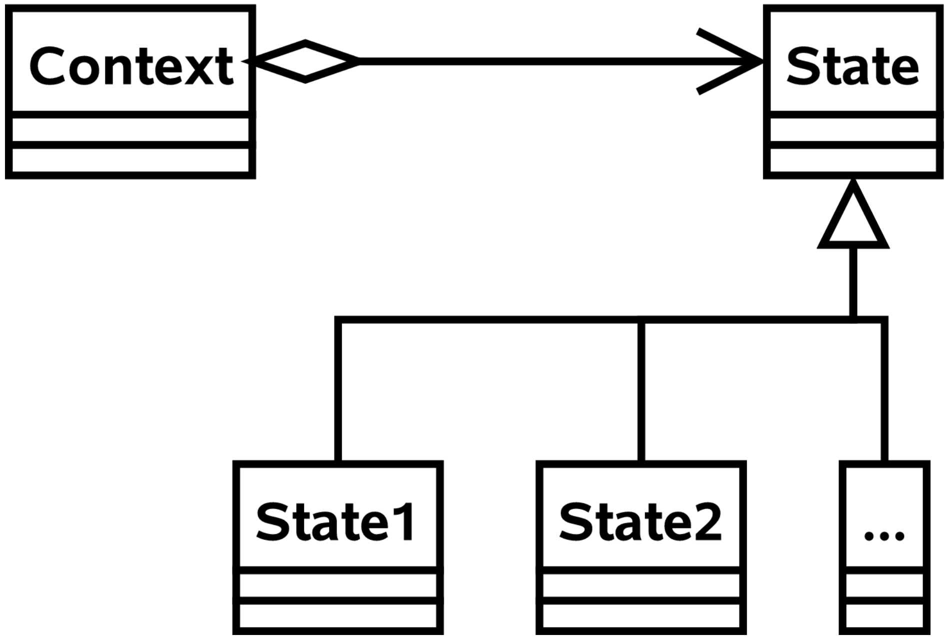




State III



State (General)



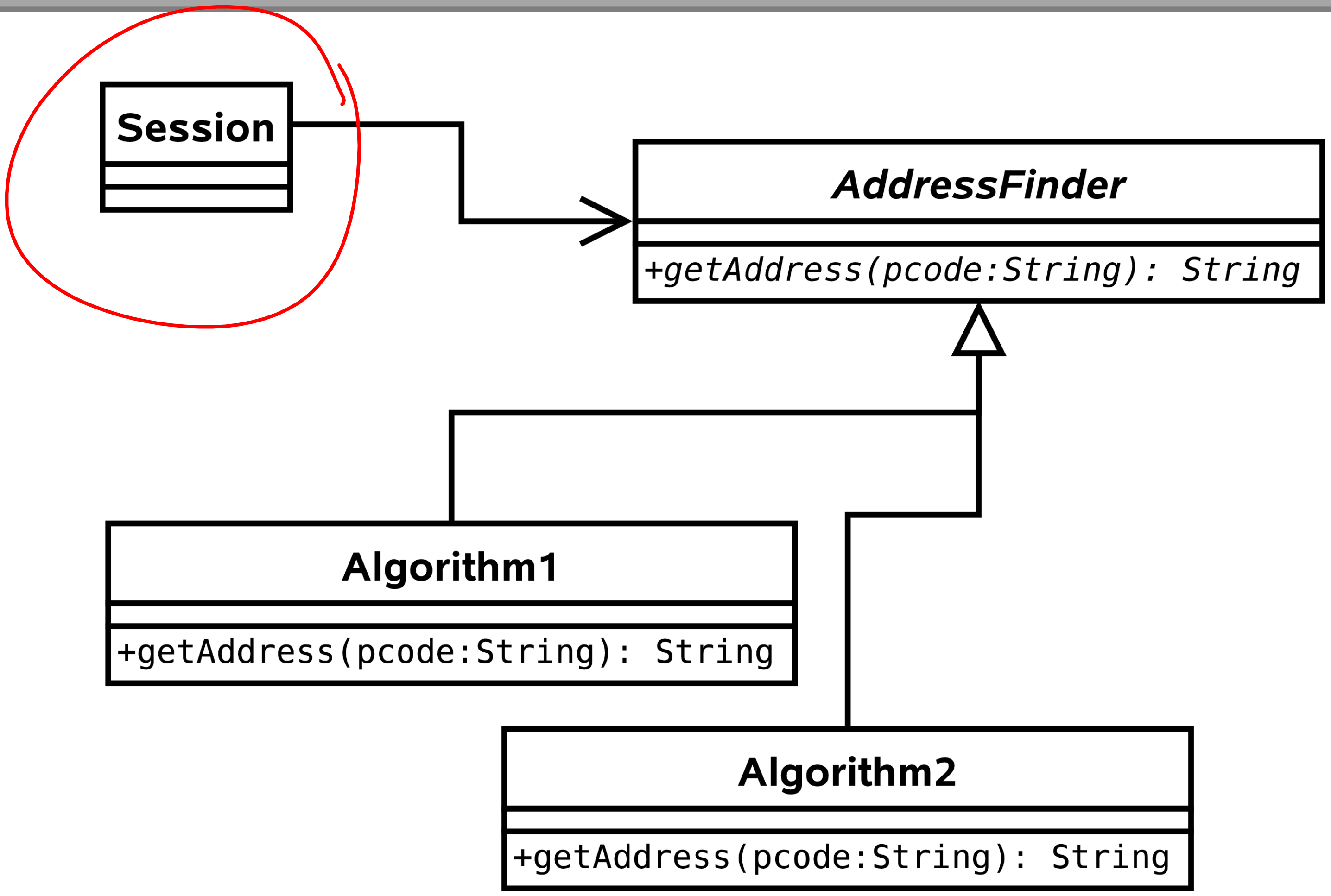
Problem: Want to trial a new lookup algorithm for the postcode->address translation

Strategy I

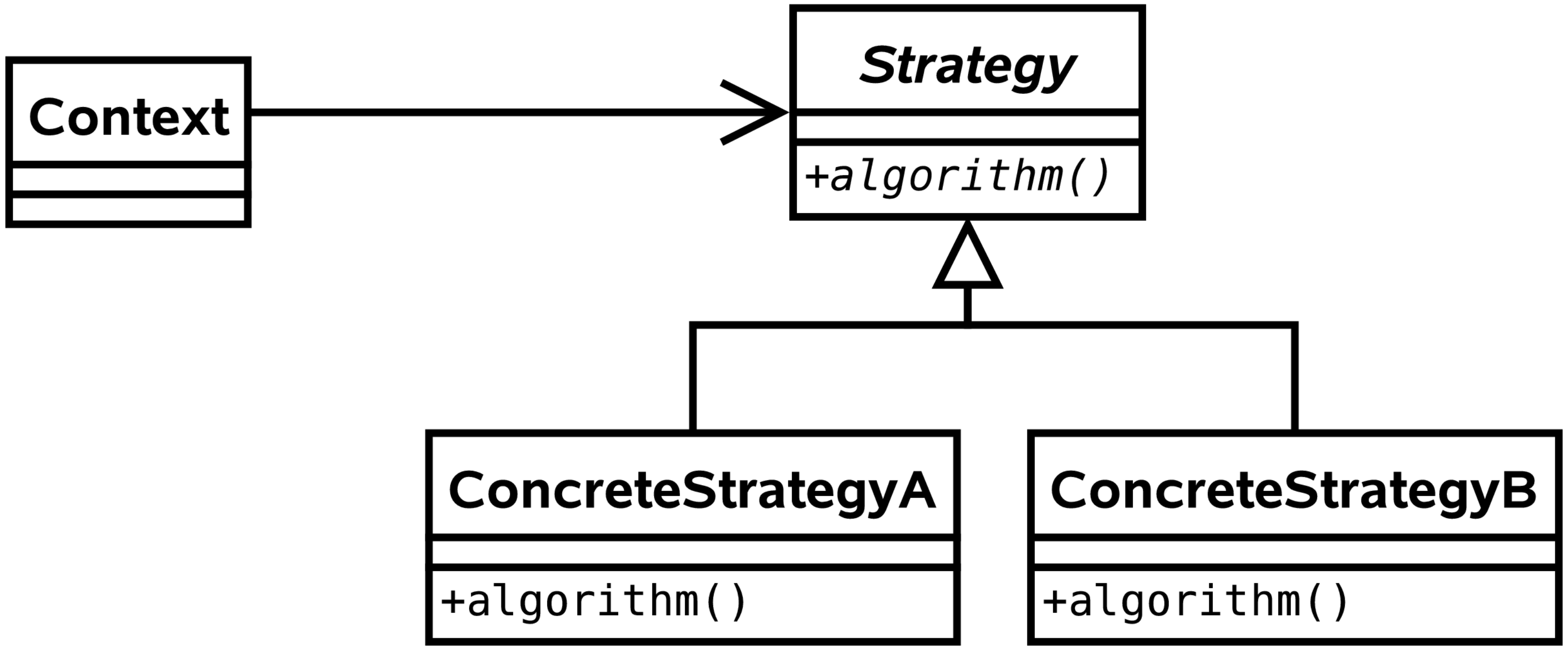
```
String getAddress(String pcode) {
    if (algorithm==0) {
       // Use old approach
       ...
    }
    else if (algorithm==1) {
       // use new approach
       ...
    }
 }
```

Hard to maintain

Strategy II


Session
AddressFinder
+getAddress(pcode:String): String
Algorithm1
+getAddress(pcode:String): String
Algorithm2
+getAddress(pcode:String): String

Strategy (General)



State vs Strategy

- Seems like the same design..?

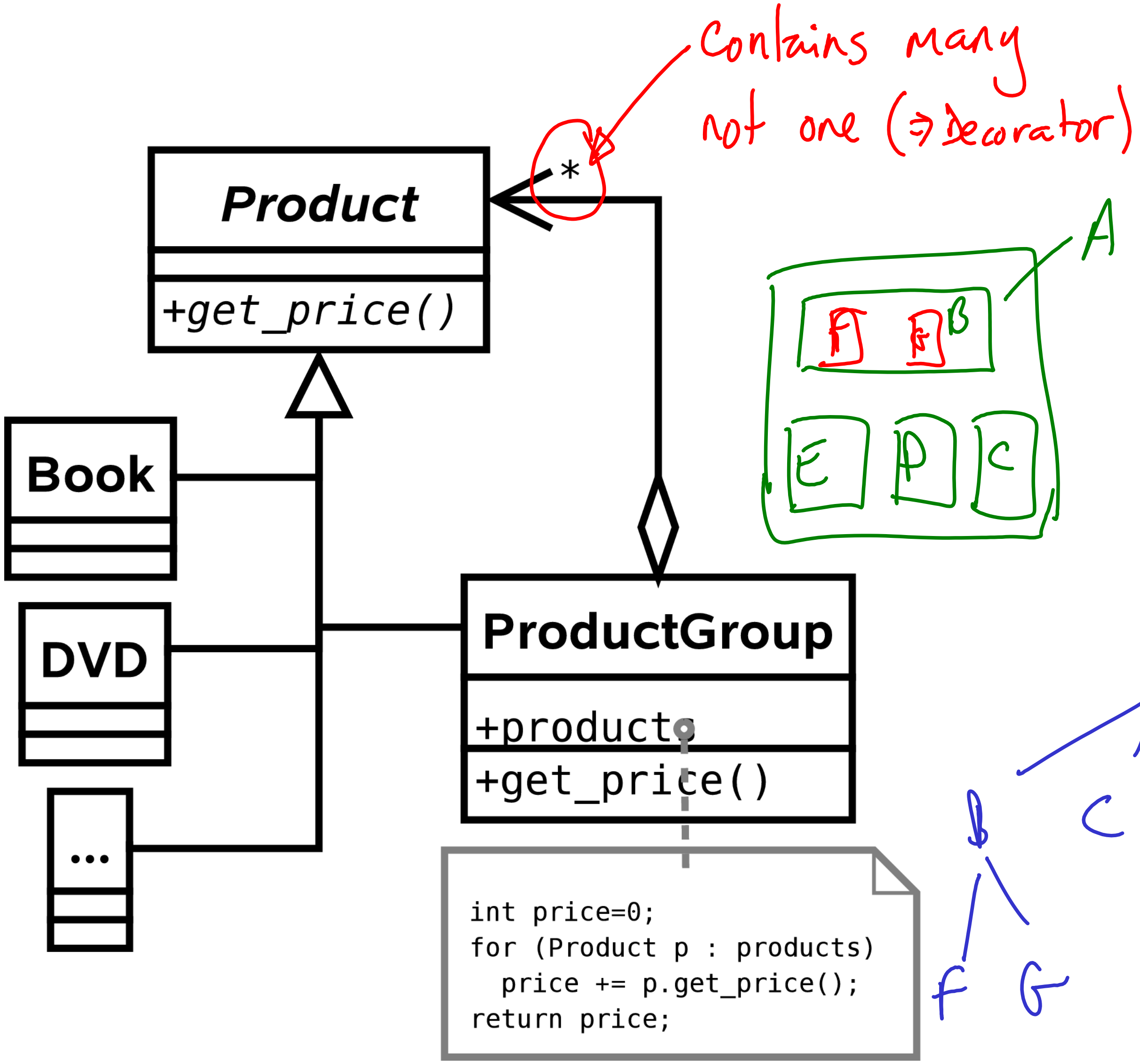
State	Strategy
Different state ⇒ different output	Different algo ⇒ same result
Dynamic - change at runtime	Select one at runtime
Invisible to external classes	Visible.

Problem: Want to support groups of related products

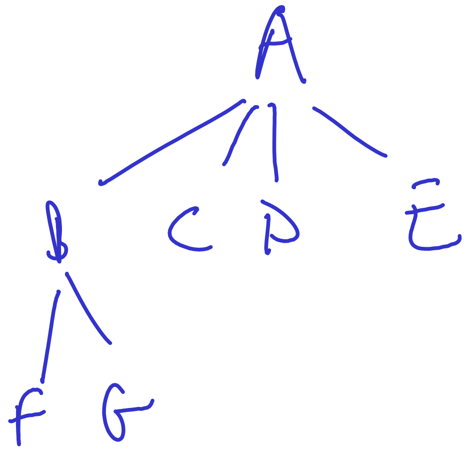
Composite I

```
public class Product {

          private int mGroupID;

}
```

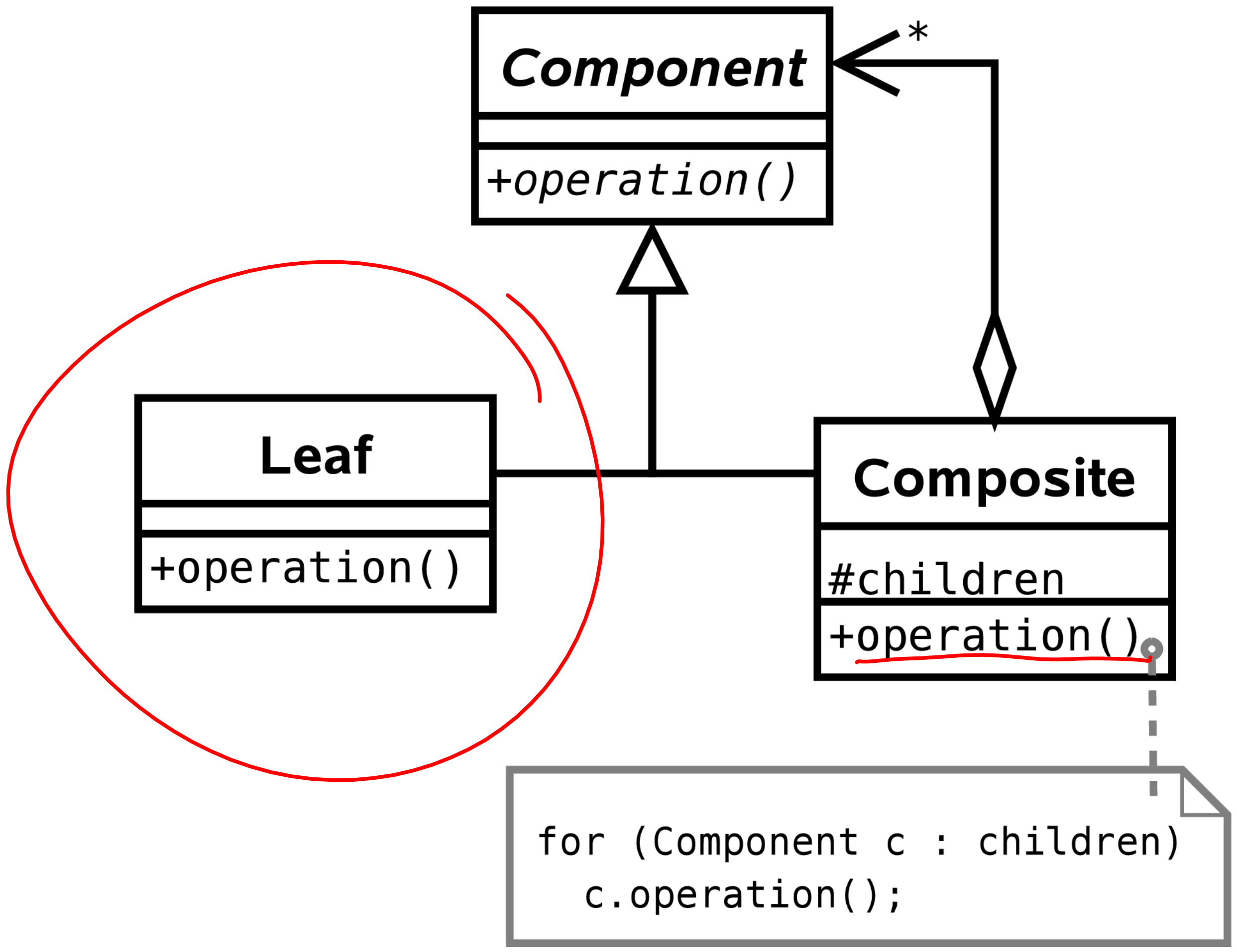
Composite II



```
int price=0;
for (Product p : products)
  price += p.get_price();
return price;
```



Composite (General)



```
for (Component c : children)
  c.operation();
```

Problem: Don't want lots of simultaneous connections to the database

Singleton I/II

- Use a global variable or a public static variable

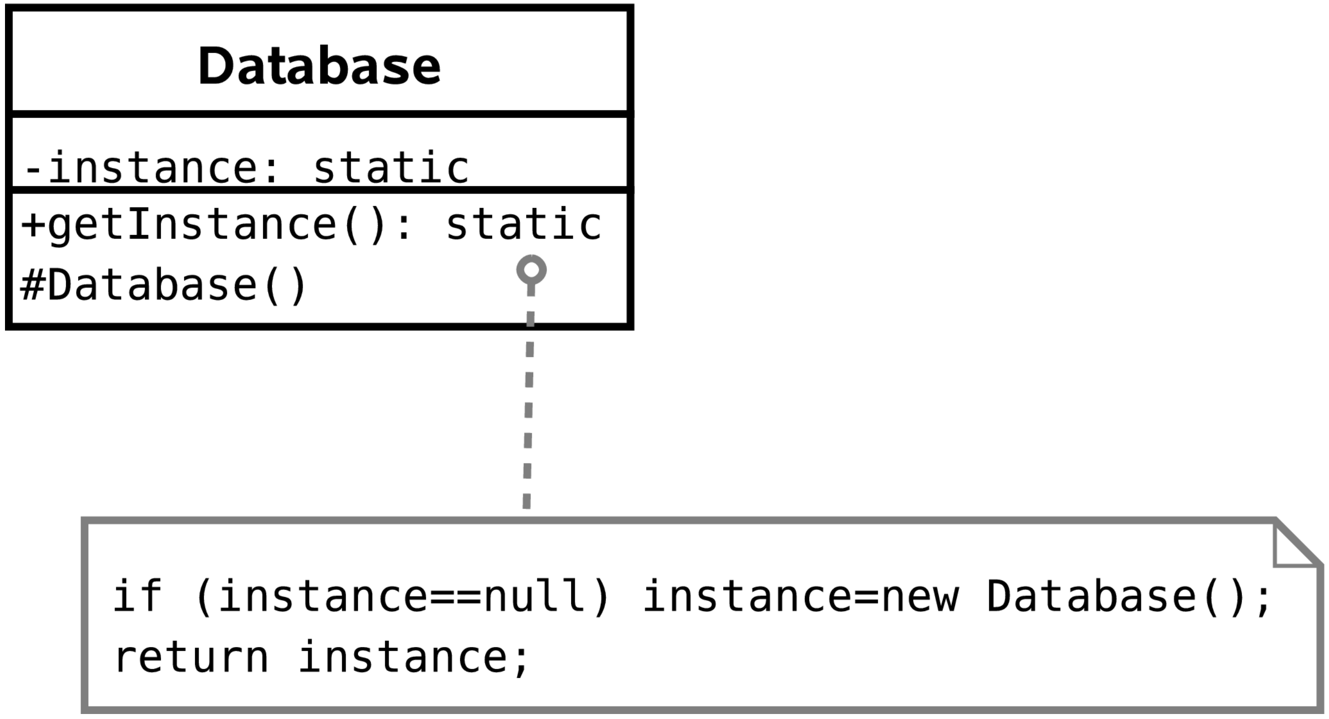
```
 public class GlobalStuff {
         public static Database sDatabase = new Database();
}

...

Database d = ~~GlobalStuff.sDatabase~~;
                  new Database();
```

Singleton III

- Pass in a Database object to everything that might use it

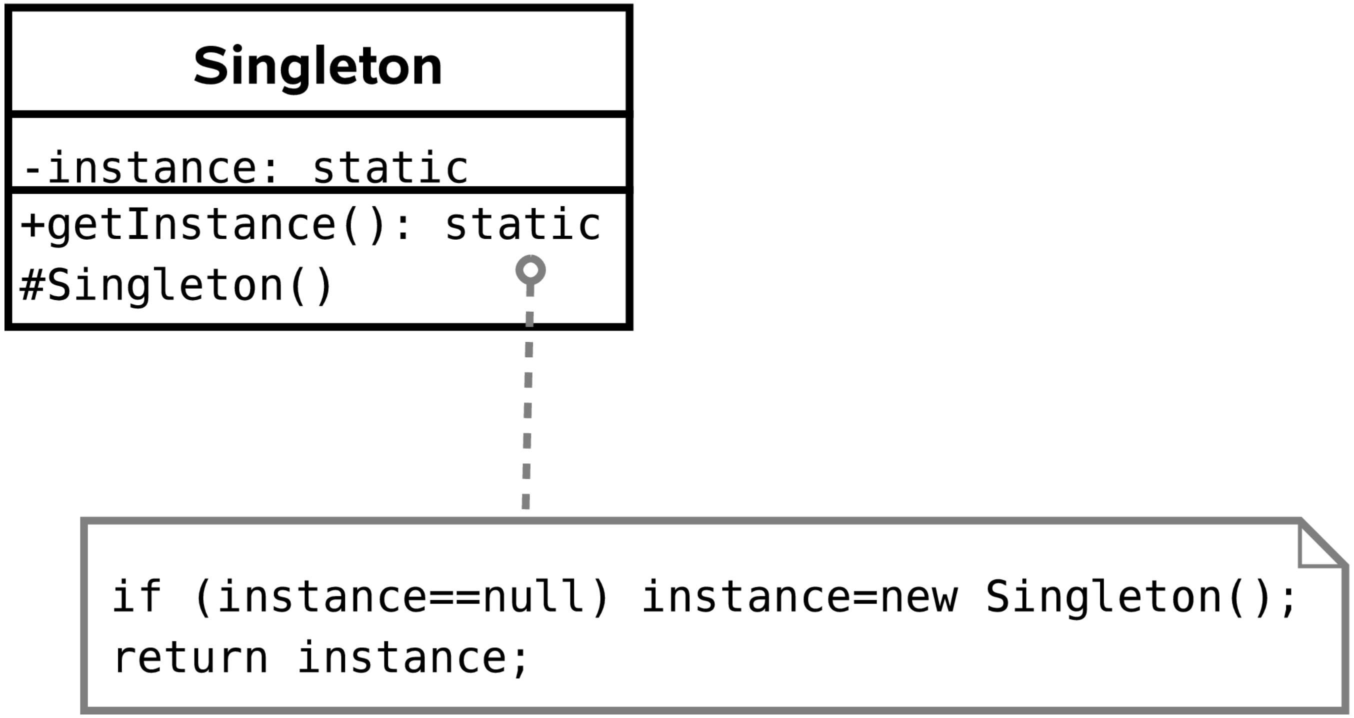
```
 public class System {
         public System (Database d) {...};
}

public class Session {
         public Session(Database d) {...}


...

Database d = new Database(); // Create the one database
System s = new System(d);
Session sesh = new Session(d);
```

Singleton IV



```
if (instance==null) instance=new Database();
return instance;
```

Singleton (General)



```
if (instance==null) instance=new Singleton();
return instance;
```