Chapter 4

Design Patterns

4.1 Introduction

Coding anything more complicated than a toy program usually benefits from forethought. After you've coded a few medium-sized pieces of object-oriented software, you'll start to notice the same general problems coming up over and over. And you'll start to automatically use the same solution each time. We need to make sure that set of default solutions is a good one!

In his 1991 PhD thesis, Erich Gamma compared this to the field of architecture, where recurrent problems are tackled by using known good solutions. The follow-on book (**Design Patterns: Elements of Reusable Object-Oriented Software, 1994**) identified a series of commonly encountered problems in object-oriented software design and 23 solutions that were deemed elegant or good in some way. Each solution is known as a *Design Pattern*:

A Design Pattern is a general reusable solution to a commonly occurring problem in software design.

The modern list of design patterns is ever-expanding and there is no shortage of literature on them. In this course we will be looking at a few key patterns and how they are used.

4.1.1 So Design Patterns are like coding recipes?

No. Creating software by stitching together a series of Design Patterns is like painting by numbers — it's easy and it probably works, but it doesn't produce a Picasso! Design Patterns are about intelligent solutions to a series of generalised problems that you *may* be able to identify in your software. You might that find they don't apply to your problem, or that they need adaptation. You simply can't afford to disengage your brain (sorry!)

4.1.2 Why Bother Studying Them?

Design patterns are useful for a number of things, not least:

1. They encourage us to identify the fundamental aims of given pieces of code
2. They save us time and give us confidence that our solution is sensible
3. They demonstrate the power of object-oriented programming
4. They demonstrate that naïve solutions are bad
5. They give us a common vocabulary to describe our code

The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments[1] with a single word, the

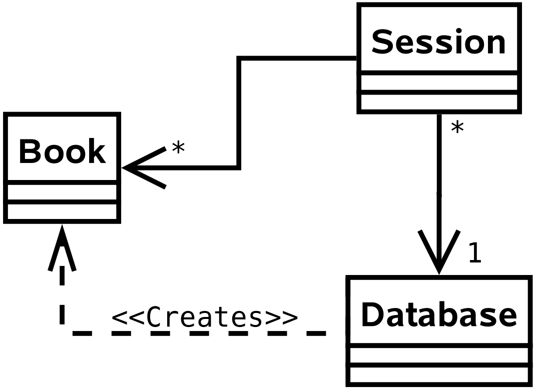
[1] You are commenting your code liberally, aren't you?

code becomes more readable and maintainable. Furthermore, you can insert the word into the class name itself, making the class self-describing.

4.2 Design Patterns By Example

We're going to develop a simple example to look at a series of design patterns. Our example is a new online venture selling books. We will be interested in the underlying ("back-end") code—this isn't going to be a web design course!

We start with a very simple set of classes. For brevity we won't be annotating the classes with all their members and functions. You'll need to use common sense to figure out what each element supports.



Session. This class holds everything about a current browser session (originating IP, user, shopping basket, etc).

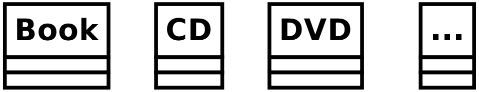
Database. This class wraps around our database, hiding away the query syntax (i.e. the SQL statements or similar).

Book. This class holds all the information about a particular book.

4.3 Supporting Multiple Products

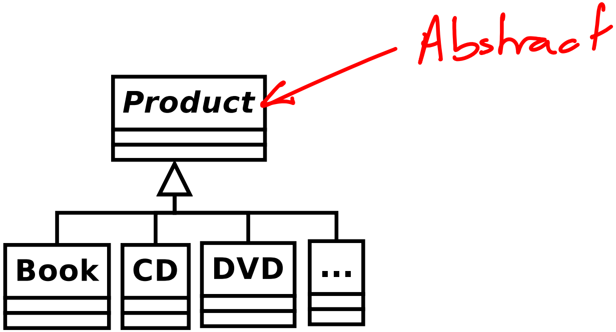
Problem: Selling books is not enough. We need to sell CDs and DVDs too. And maybe electronics. Oh, and sports equipment. And...

Solution 1: Create a new class for every type of item.



- ✔ It works.
- ✗ We end up duplicating a lot of code (all the products have prices, sizes, stock levels, etc).
- ✗ This is difficult to maintain (imagine changing how the VAT is computed...).

Solution 2: Derive from an abstract base class that holds all the common code.



- ✔ “Obvious” object oriented solution
- ✔ If we are smart we would use polymorphism to avoid constantly checking what type a given Product object is in order to get product-specific behaviour.

4.3.1 Generalisation

This isn’t really an ‘official’ pattern, because it’s a rather fundamental thing to do in object-oriented programming. However, it’s important to understand the power inheritance gives us under these circumstances.

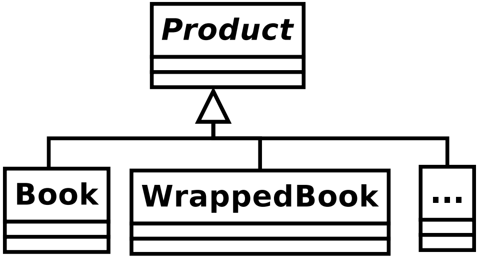
4.4 The Decorator Pattern

Problem: You need to support gift wrapping of products.

Solution 1: Add variables to the Product class that describe whether or not the product is to be wrapped and how.

- ✔ It works. In fact, it's a good solution if all we need is a binary flag for wrapped/not wrapped.
- ✗ As soon as we add different wrapping options and prices for different product types, we quickly clutter up Product.
- ✗ Clutter makes it harder to maintain.
- ✗ Clutter wastes storage space.

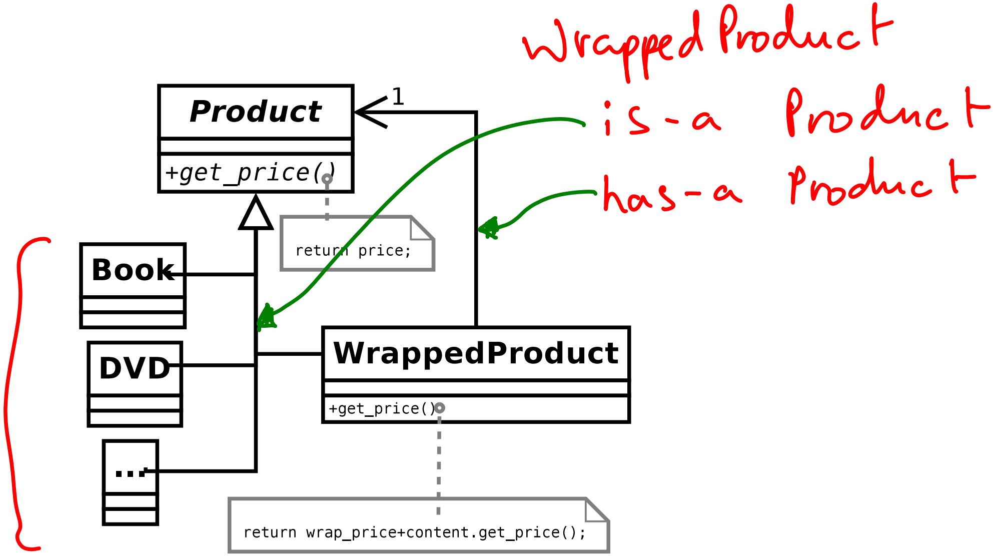
Solution 2: Add WrappedBook (etc.) as subclasses of Product as shown.



Don't. Do. This. Ever.

- ✔ We are efficient in storage terms (we only allocate space for wrapping information if it is a wrapped entity).
- ✗ We instantly double the number of classes in our code.
- ✗ If we change Book we have to remember to mirror the changes in WrappedBook.
- ✗ If we add a new type we must create a wrapped version. This is bad because we can forget to do so.
- ✗ We can't convert from a Book to a WrappedBook without copying lots of data.

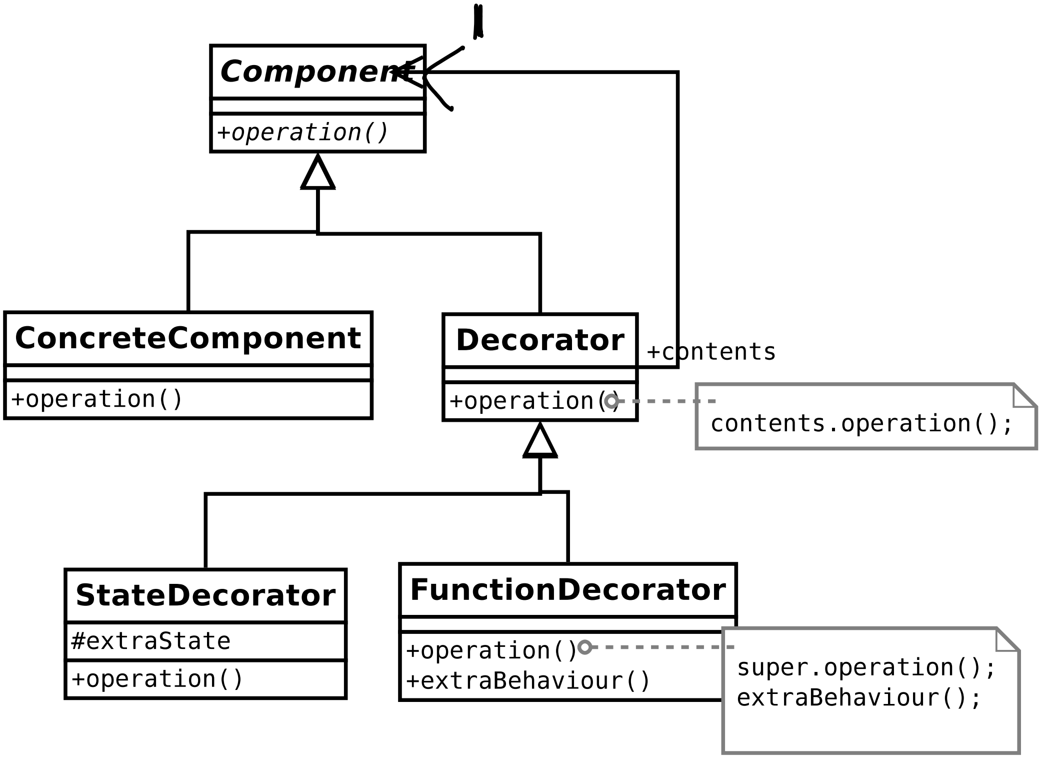
Solution 3: Create a general WrappedProduct class that is both a subclass of Product and references an instance of one of its siblings. Any state or functionality required of a WrappedProduct is 'passed on' to its internal sibling, unless it relates to wrapping.



- ✔ We can add new product types and they will be automatically wrappable.
- ✔ We can dynamically convert an established product object into a wrapped product and back again without copying overheads.
- ✗ We can wrap a wrapped product!
- ✗ We could, in principle, end up with lots of chains of little objects in the system

4.4.1 Generalisation

This is the **Decorator** pattern and it allows us to add to an object *dynamically*. By that I mean we can take an object in the system and effectively give it extra state or functionality. I say ‘effectively’ because the actual object in memory is untouched. Rather, we create a new, small object that ‘wraps around’ the original. To remove the wrapper we simply discard the wrapping object. Real world example: humans can be ‘decorated’ with contact lenses to improve their vision.



Note that we can use the pattern to add state (variables) or functionality (methods), or both if we want. In the diagram above, I have explicitly allowed for both options by deriving StateDecorator and FunctionDecorator.

This is usually unnecessary — in our book seller example we only want to decorate one thing so we might as well just put the code into Decorator.

4.5 State Pattern

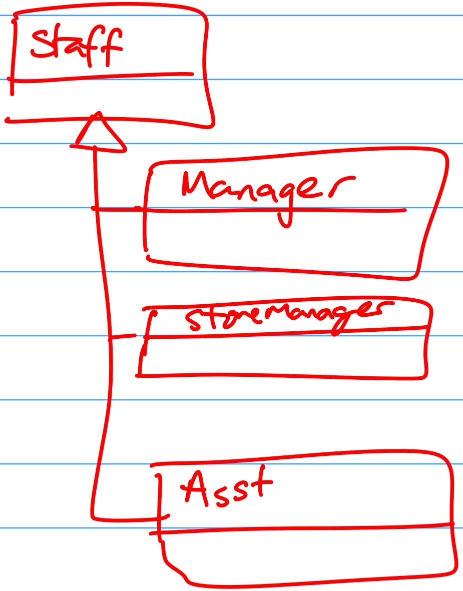
Problem: We need to handle a lot of gift options that the customer may switch between at will (different wrapping papers, bows, gift tags, gift boxes, gift bags, ...).

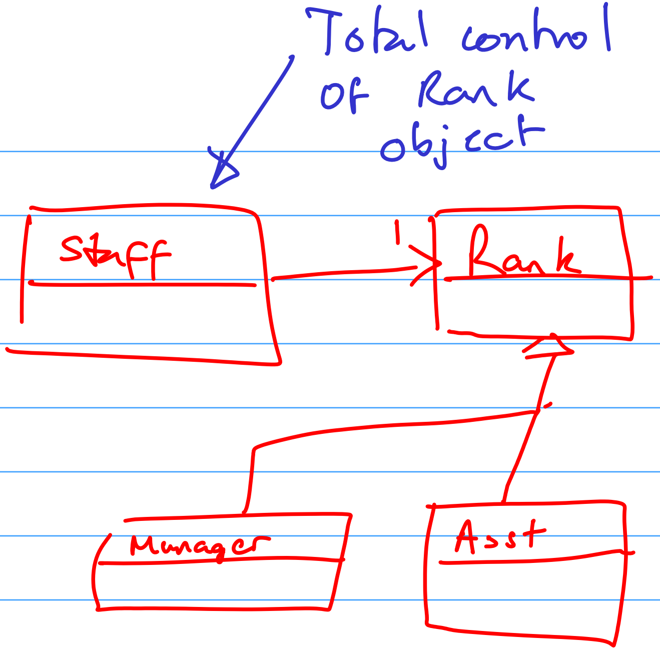
Solution 1: Take our WrappedProduct class and add a lot of if/then statements to the function that does the wrapping — let's call it initiate_wrapping().

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

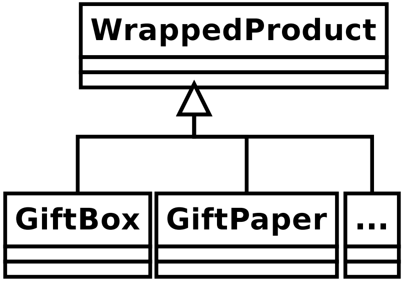
Bad for lots of option

✔ It works.
✗ The code is far less readable.
✗ Adding a new wrapping option is ugly.


Staff
Manager
StoreManager
Asst

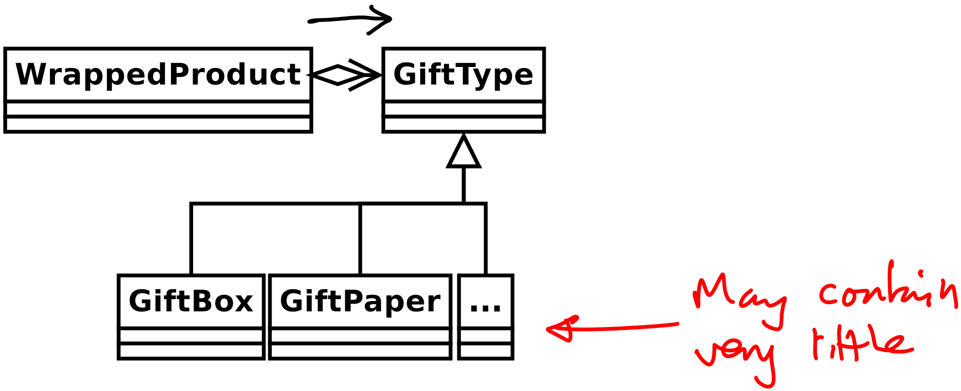

Total control of Rank object
Staff
1
Rank
Manager
Asst

Solution 2: We basically have type-dependent behaviour, which is code for “use a class hierarchy”.



- ✔ This is easy to extend.
- ✔ The code is neater and more maintainable.
- ✗ What happens if we need to change the type of the wrapping (from, say, a box to a bag)? We have to construct a new GiftBag and copy across all the information from a GiftBox. Then we have to make sure every reference to the old object now points to the new one. This is hard!

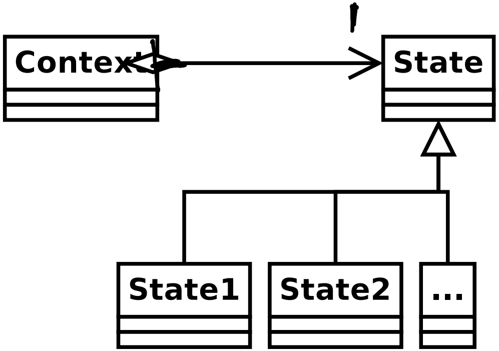
Solution 3: Let’s keep our idea of representing states with a class hierarchy, but use a new abstract class as the parent:



Now, every WrappedProduct *has-a* GiftType. We have retained the advantages of solution 2 but now we can easily change the wrapping type in-situ since we know that only the WrappedObject object references the GiftType object.

4.5.1 Generalisation

This is the **State** pattern and it is used to permit an object to change its behaviour *at run-time*. A real-world example is how your behaviour may change according to your mood. e.g. if you're angry, you're more likely to behave aggressively.


Context
1
State
State1
State2
...

4.6 Strategy Pattern

Problem: Part of the ordering process requires the customer to enter a postcode that is then used to determine the address to post the items to. At the moment the computation of address from postcode is very slow. One of your employees proposes a different way of computing the address that should be more efficient. How can you trial the new algorithm?

Solution 1: Let there be a class AddressFinder with a method getAddress(String pcode). We could add lots of if/then/else statements to the getAddress() function.

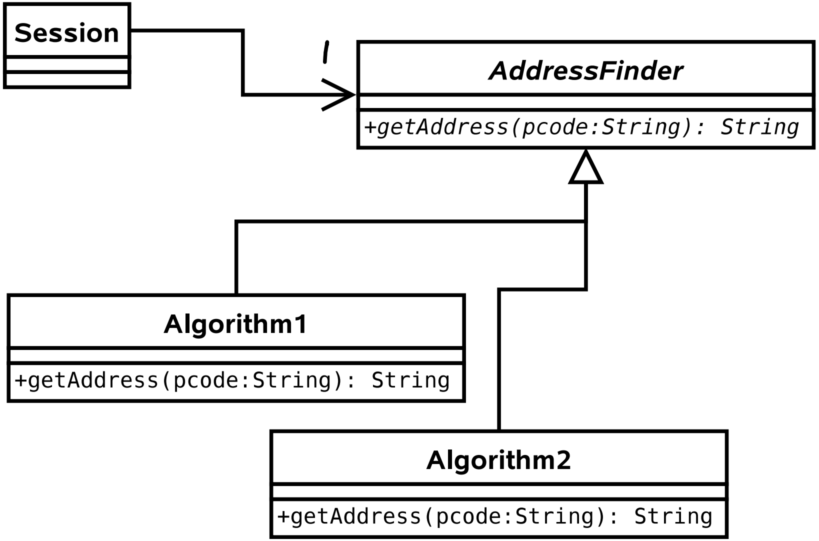
```
String getAddress(String pcode) {
   if (algorithm==0) {
      // Use old approach
      ...
   }
   else if (algorithm==1) {
      // use new approach
      ...
   }
}
```

✗ The getAddress() function will be huge, making it difficult to read and maintain.

✗ Because we must edit AddressFinder to add a new algorithm, we have violated the open/closed principle[2].

[2] This states that a class should be open to extension but closed to modification. So we allow classes to be easily extended to

Solution 2: Make AddressFinder abstract with a single abstract function getAddress(String pcode). Derive a new class for each of our algorithms.


Session
1
AddressFinder
+getAddress(pcode:String): String
Algorithm1
+getAddress(pcode:String): String
Algorithm2
+getAddress(pcode:String): String

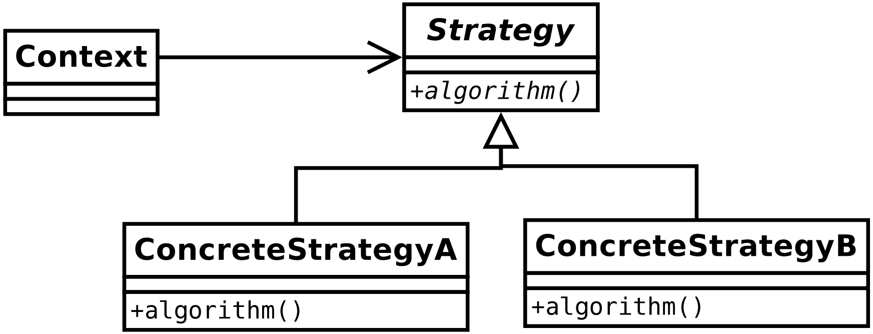
✔ We encapsulate each algorithm in a class.

incorporate new behavior without modifying existing code. This makes our designs resilient to change but flexible enough to take on new functionality to meet changing requirements.

✔ Code is clean and readable.
✗ More classes kicking around

4.6.1 Generalisation

This is the **Strategy** pattern. It is used when we want to support different algorithms that achieve the same goal. Usually the algorithm is fixed when we run the program, and doesn't change. A real life example would be two consultancy companies given the same brief. They will hopefully produce the same result, but do so in different ways. i.e. they will adopt different strategies. From the (external) customer's point of view, the result is the same and he is unaware of how it was achieved. One company may achieve the result faster than the other and so would be considered 'better'.



Note that this is essentially the same UML as the **State** pattern! The *intent* of each of the two patterns is quite different however:

- **State** is about encapsulating behaviour that is linked to specific internal state within a class.
- Different states produce different outputs (externally the class behaves differently).
- **State** assumes that the state will continually change at run-time.
- The usage of the **State** pattern is normally invisible to external classes. i.e. there is no setState(State s) function.

- **Strategy** is about encapsulating behaviour in a class. This behaviour does not depend on internal variables.
- Different concrete Strategys may produce exactly the same output, but do so in a different way. For example, we might have a new algorithm to compute the standard deviation of some variables. Both the old algorithm and the new one will produce the same output (hopefully), but one may be faster than the other. The **Strategy** pattern lets us compare them cleanly.
- **Strategy** in the strict definition usually assumes the class is selected at compile time and not changed during runtime.
- The usage of the **Strategy** pattern is normally visible to external classes. i.e. there will be a setStrategy(Strategy s) function or it will be set in the constructor.

However, the similarities do cause much debate and you will find people who do not differentiate between the two patterns as strongly as I tend to.