# Programming Methods Handout 2: Design Patterns

As you've learnt in your Software Design course, 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.

## 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 find they don't apply to your problem, or they need adaptation. You simply can't afford to disengage your brain.

## 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

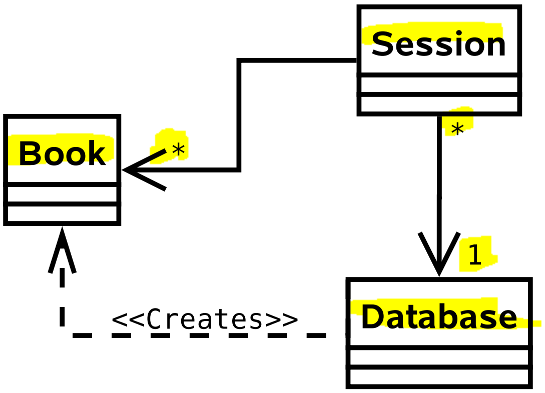
The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments[1] with a single word, the code becomes more readable and maintainable. Furthermore, you can insert the word into the class name itself, making the class self-describing.

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

## 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 setup 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).

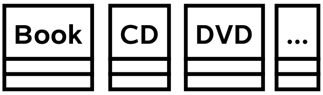
Database. This class wraps around our database, hiding away the query syntax (i.e. the SQL statements or similar).

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

## Supporting Multiple Products

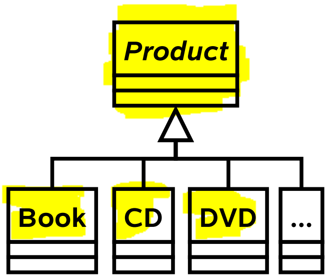
**Problem:** Selling books is not enough. We need to sell CDs and DVDs too. And maybe electronics. Oh, and sports equipment. And...

**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[2] to avoid constantly check-

[2]There are two types of polymorphism. *Ad-hoc* polymorphism (a.k.a. runtime or dynamic polymorphism) is concerned with object inheritance. It is familiar to you from Java, when the computer automatically figures out which version of an inherited method to run. *Parametric* polymorphism (a.k.a. static polymorphism) is where the compiler figures out which version of a type to use *before* the program runs. You are familiar with this in ML, but you also find it in C++ (templates) and Java (look up generics).

ing what type a given Product object is in order to get product-specific behaviour.

## 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.

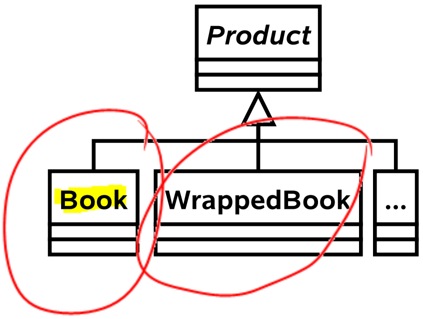
# The Decorator Pattern

**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.

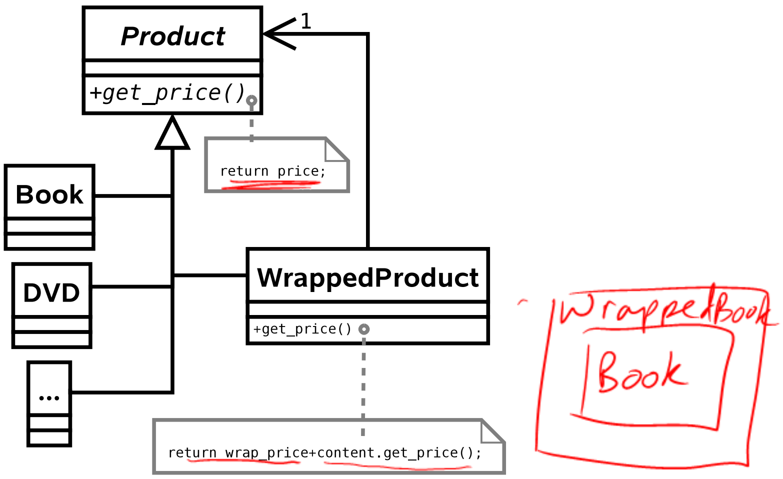


Implementing this solution is a shortcut to the Job centre.

- ✔ 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.

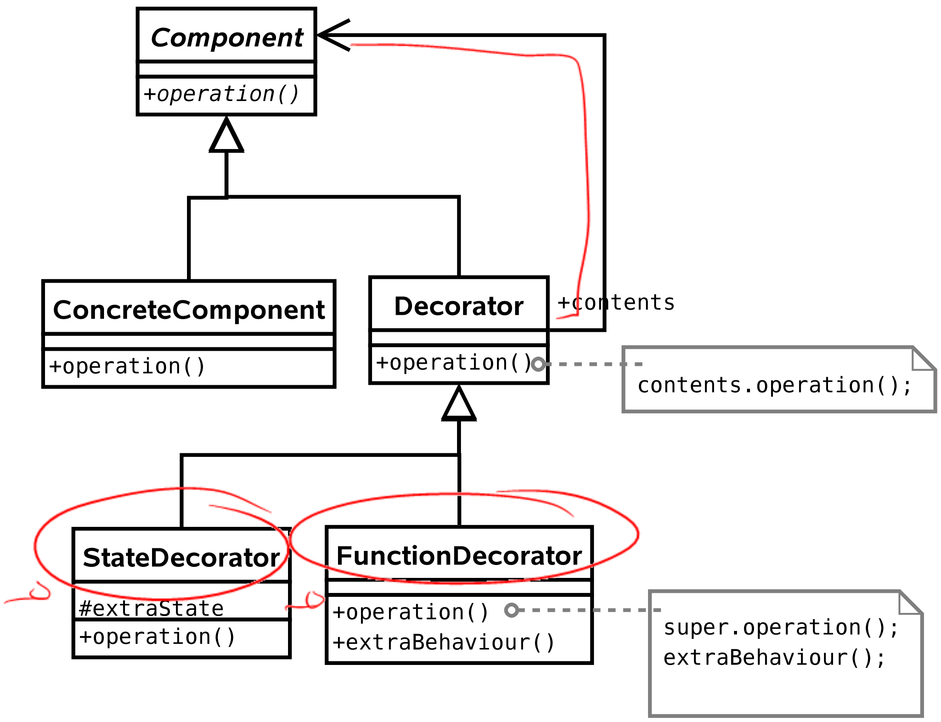
**Solution 3:** Create a general WrappedProduct class that is both a subclass of Product and references an instance of one of its siblings. Any state or functionality required of a WrappedProduct is 'passed on' to its internal sibling, unless it relates to wrapping.



- ✔ 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

## Generalisation

This is the **Decorator** pattern and it allows us to add functionality to a class *dynamically* without changing the base class or having to derive a new subclass. 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.

## State Pattern

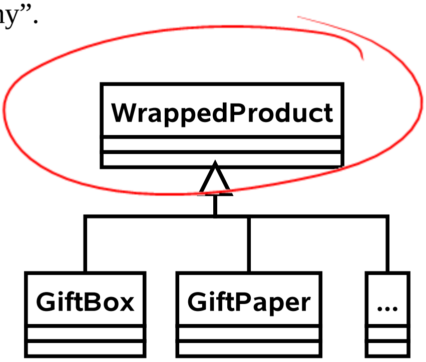
**Problem:** We need to handle a lot of gift options that the customer may switch between at will (different wrapping papers, bows, gift tags, gift boxes, gift bags, ...).

**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 ...
}
```

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

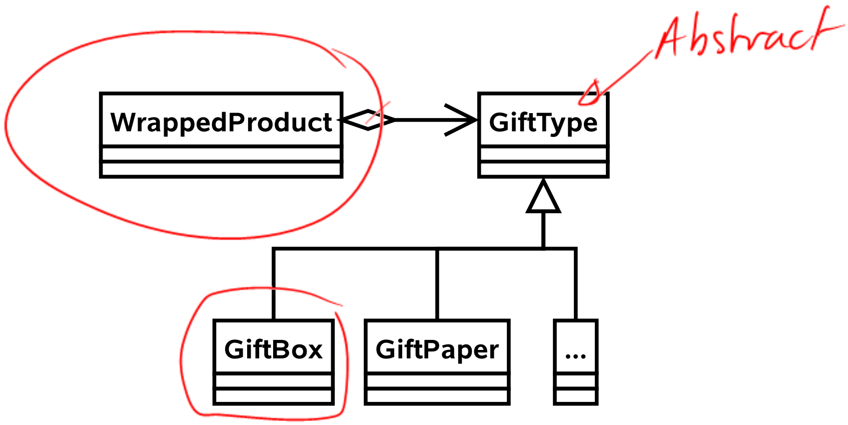
**Solution 2:** We basically have type-dependent behaviour, which is code for "use a class hierarchy".



- ✔ 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!

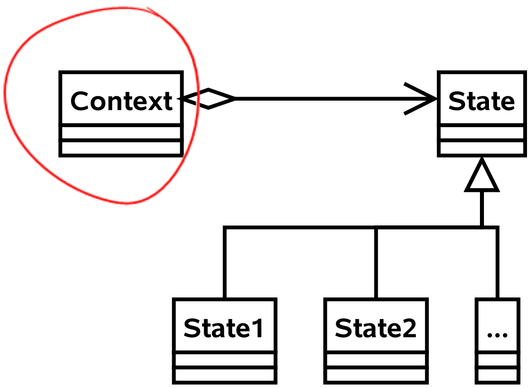
**Solution 3:** Let's keep our idea of representing states with a class hierarchy, but use a new abstract class as the parent:

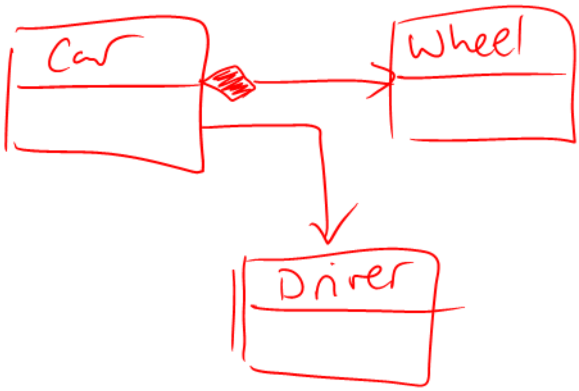


Now, every WrappedProduct *has-a* GiftType. We have retained the advantages of solution 2 but now we can easily change the wrapping type in-situ since we know that only the WrappedObject object references the GiftType object.

## Generalistion

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
State
State1
State2
...


Car
Wheel
Driver

## Strategy Pattern

**Problem:** Part of the ordering process requires the customer to enter a post-code which 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?

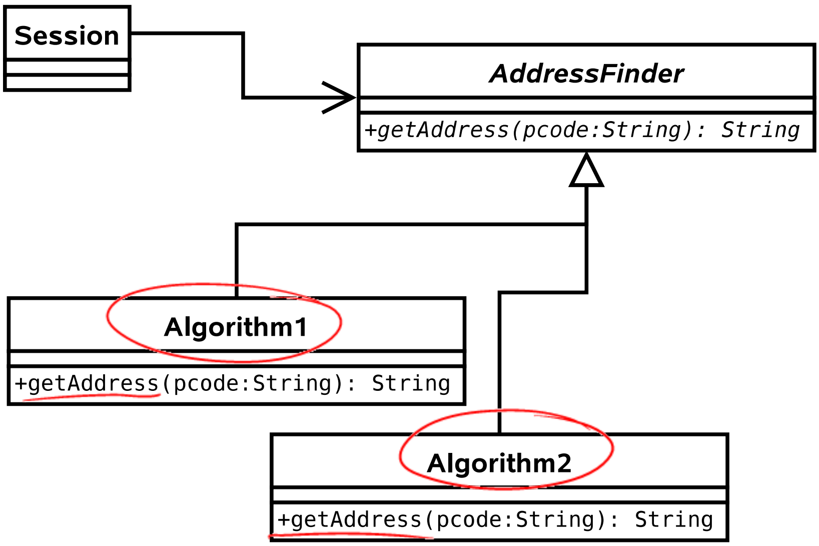
**Solution 1:** Let there be a class AddressFinder with a method getAddress(String pcode). We could add lots of if/then/else statements to the getAddress() function.

```
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[3].

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

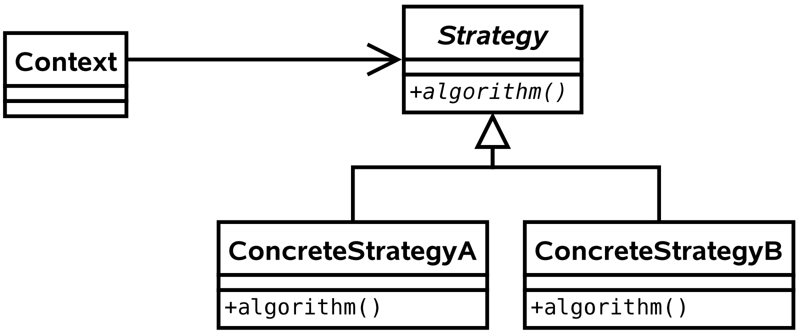
[3]This states that a class should be open to extension but closed to modification. So we allow classes to be easily extended to incorporate new behavior without modifying existing code. This makes our designs resilient to change but flexible enough to take on new functionality to meet changing requirements.


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

- ✔ We encapsulate each algorithm in a class.
- ✔ Code is clean and readable.
- ✗ More classes kicking around

## 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 the two patterns are 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.

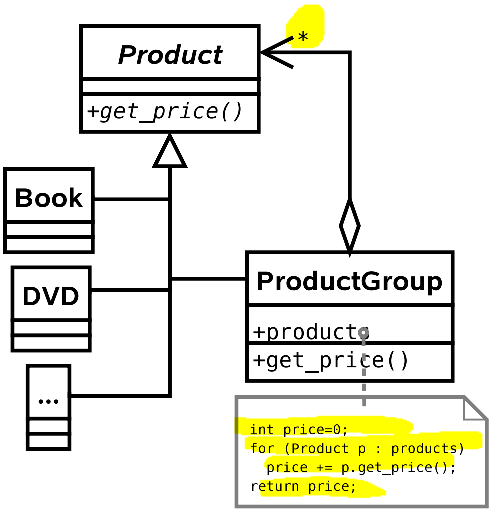
## Composite Pattern

**Problem:** We want to support entire *groups* of products. e.g. The Lord of the Rings gift set might contain all the DVDs (plus a free cyanide capsule).

**Solution 1:** Give every Product a group ID (just an int). If someone wants to buy the entire group, we search through all the Products to find those with the same group ID.

- ✔ Does the basic job.
- ✗ What if a product belongs to no groups (which will be the majority case)? Then we are wasting memory and cluttering up the code.
- ✗ What if a product belongs to multiple groups? How many groups should we allow for?

**Solution 2:** Introduce a new class that encapsulates the notion of groups of products:



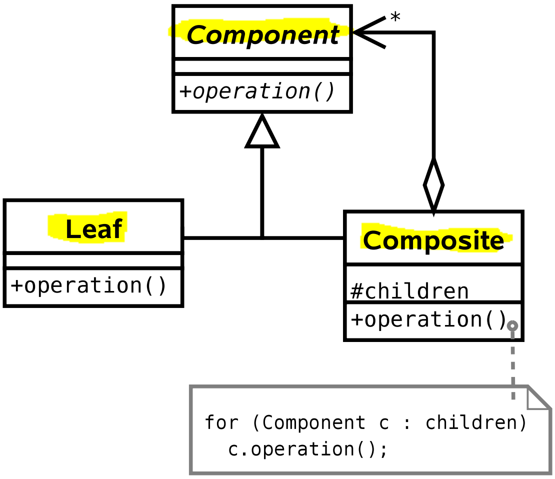
```
int price=0;
for (Product p : products)
  price += p.get_price();
return price;
```

If you're still awake, you may be thinking this is a bit like the **Decorator** pattern, except that the new class supports associations with multiple Products (note the * by the arrowhead). Plus the intent is different – we are not adding new functionality but rather supporting the same functionality for groups of Products.

- ✔ Very powerful pattern.
- ✗ Could make it difficult to get a list of all the individual objects in the group, should we want to.

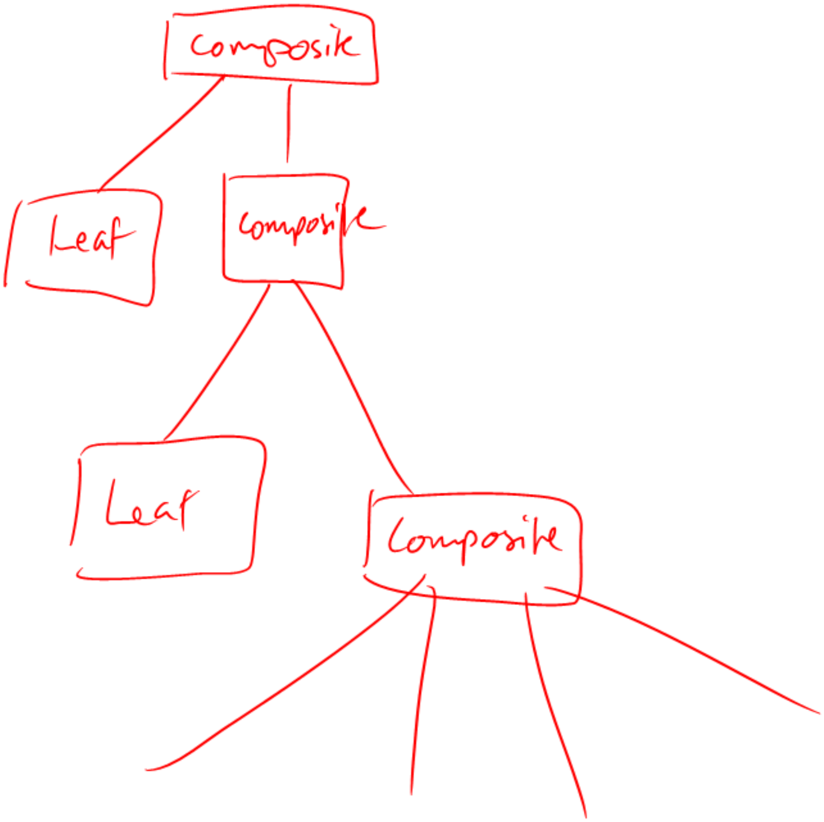
## Generalisation

This is the **Composite** pattern and it is used to allow objects and collections of objects to be treated uniformly. Almost any hierarchy uses the **Composite** pattern. e.g. The CEO asks for a progress report from a manager, who collects progress reports from all those she manages and reports back.



Notice the terminology in the general case: we speak of Leafs because we can use the Composite pattern to build a *tree* structure. Each Composite object will represent a node in the tree, with children that are either Composites or Leafs.

This pattern crops up a lot, and we will see it in other contexts later in this course.



## Singleton Pattern

**Problem:** Somewhere in our system we will need a database and the ability to talk to it. Let us assume there is a Database class that abstracts the difficult stuff away. We end up with lots of simultaneous user Sessions, each wanting to access the database. Each one creates its own Database object and connects to the database over the network. The problem is that we end up with a lot of Database objects (wasting memory) and a lot of open network connections (bogging down the database).

What we want to do here is to ensure that there is only one Database object ever instantiated and every Session object uses it. Then the Database object can decide how many open connections to have and can queue requests to reduce instantaneous loading on our database (until we buy a half decent one).

**Solution 1:** Use a global variable of type Database that everything can access from everywhere.

- ✗ Global variables are less desirable than David Hasselhoff's greatest hits.
- ✗ Can't do it in Java anyway...

**Solution 2:** Use a public static variable which everything uses (this is as close to global as we can get in Java).

```
public class System {
  public static Database database;
}

...

public static void main(String[]) {
  // Always gets the same object
  Database d = System.database;
}
```

- ✗ This is really just global variables by the back door.

- ✗ Nothing fundamentally prevents us from making multiple Database objects!

**Solution 3:** Create an instance of Database at startup, and pass it as a constructor parameter to every Session we create, storing a reference in a member variable for later use.

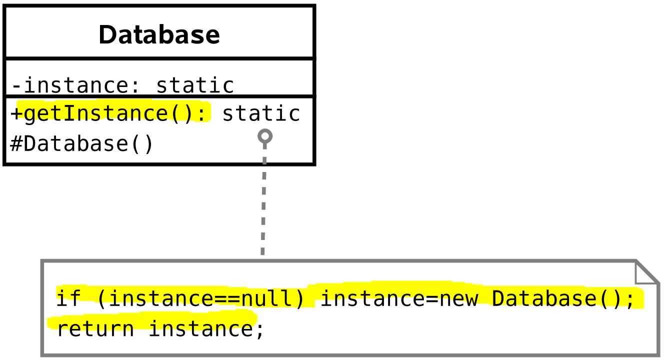
```
public class System {
  public System(Database d) {...}
}

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

...

public static void main(String[]) {
  Database d = new Database();
  System sys = new System(d);
  Session sesh = new Session(d);

}
```

- ✗ This solution could work, but it doesn't *enforce* that only one Database be instatiated – someone could quite easily create a new Database object and pass it around.
- ✗ We start to clutter up our constructors.
- ✗ It's not especially intuitive. We can do better.

**Solution 4:** (**Singleton**) Let's adapt Solution 2 as follows. We *will* have a single static instance. However we will access it through a static member function. This function, getInstance() will either create a new Database object (if it's the first call) or return a reference to the previously instantiated object.

Of course, nothing stops a programmer from ignoring the getInstance() function and just creating a new Database object. So we use a neat trick: we make the constructor *private* or *protected*. This means code like new Database() isn't possible from an arbitrary class.

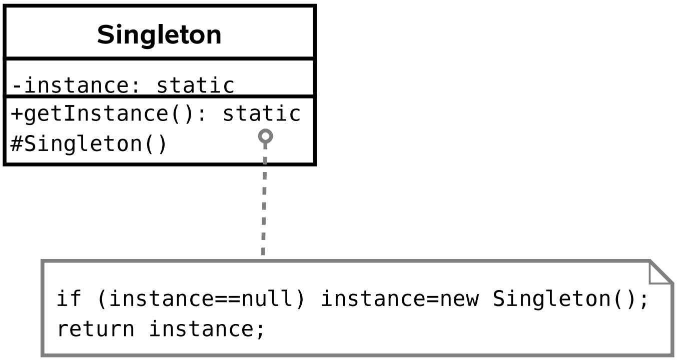
Database d = new Database();



- ✔ *Guarantees* that there will be only one instance.
- ✔ Code to get a Database object is neat and tidy and intuitive to use. e.g. (Database d=Database.getInstance();)
- ✔ Avoids clutter in any of our classes.
- ✗ Must take care in Java. Either use a dedicated package or a private constructor (see below).
- ✗ Must remember to disable clone()-ing!

## Generalisation

This is the **Singleton** pattern. It is used to provide a global point of access to a class that should be instantiated only once.



There is a caveat with Java. If you choose to make the constructor protected (this would be useful if you wanted a singleton base class for multiple applications of the singleton pattern, and is actually the 'official' solution) you have to be careful.

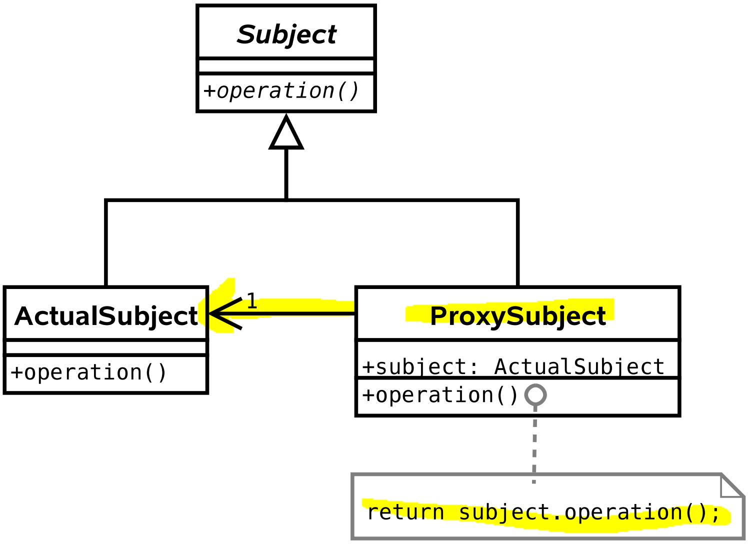
Protected members are accessible to the class, any subclasses, *and all classes in the same package*. Therefore, any class in the same package as your base class will be able to instantiate Singleton objects at will, using the new keyword!

Additionally, we don't want a crafty user to subclass our singleton and implement Cloneable on their version. The examples sheet asks you to address this issue.

## Proxy Pattern(s)

The **Proxy** pattern is a very useful *set* of three patterns: **Virtual Proxy**, **Remote Proxy**, and **Protection Proxy**.

All three are based on the same general idea: we can have a placeholder class that has the same interface as another class, but actually acts as a pass through for some reason.



### Virtual Proxy

**Problem:** Our Product subclasses will contain a lot of information, much of which won't be needed since 90% of the products won't be selected for more detail, just listed as search results.

**Solution :** Here we apply the **Proxy** pattern by only loading part of the full class into the proxy class (e.g. name and price). If someone requests more details, we go and retrieve them from the database.

## Remote Proxy

**Problem:** Our server is getting overloaded.

**Solution :** We want to run a farm of servers and distribute the load across them. Here a particular object resides on server A, say, whilst servers B and C have proxy objects. Whenever the proxy objects get called, they know to contact server A to do the work. i.e. they act as a pass-through.

Note that once server B has bothered going to get something via the proxy, it might as well keep the result locally in case it's used again (saving us another network trip to A). This is *caching* and we'll return to it shortly.

## Protection Proxy

**Problem:** We want to keep everything as secure as possible.

**Solution :** Create a `User` class that encapsulates all the information about a person. Use the **Proxy** pattern to fill a proxy class with public information. Whenever private information is requested of the proxy, it will only return a result if the user has been authenticated.

In this way we avoid having private details in memory unless they have been authorised.

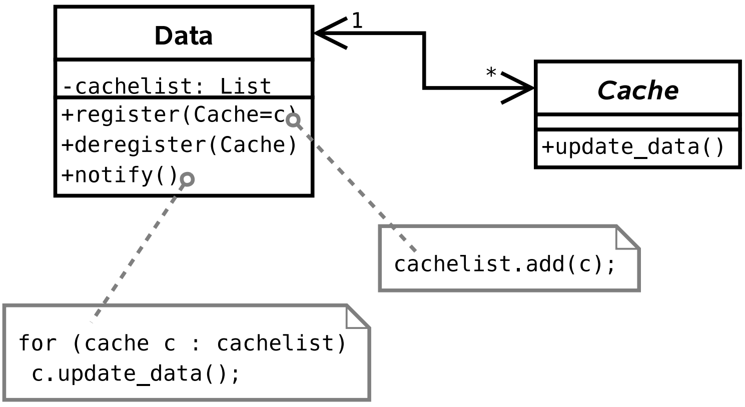
## Observer Pattern

**Problem:** We use the **Remote Proxy** pattern to distribute our load. For efficiency, proxy objects are set to cache information that they retrieve from other servers. However, the originals could easily change (perhaps a price is updated or the exchange rate moves). We will end up with different results on different servers, dependent on how old the cache is!!

**Solution 1:** Once a proxy has some data, it keeps polling the authoritative source to see whether there has been a change (c.f. polled I/O).

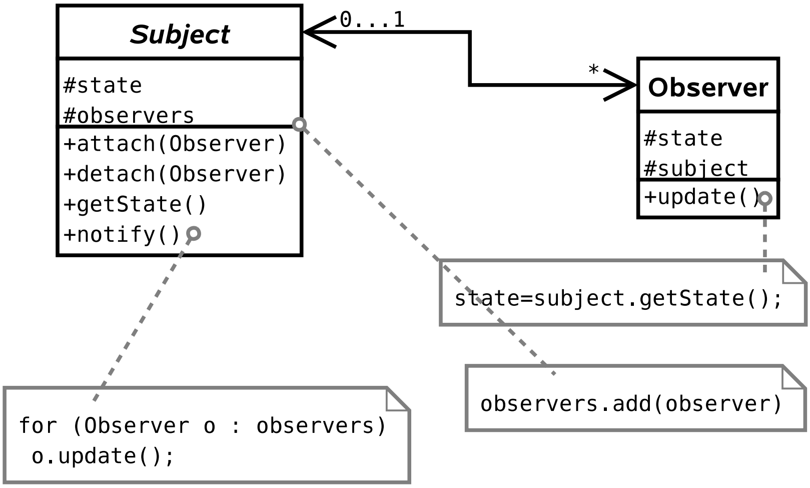
✗ How frequently should we poll? Too quickly and we might as well not have cached at all. Too slow and changes will be slow to propagate.

**Solution 2:** Modify the real object so that the proxy can 'register' with it (i.e. tell it of its existence and the data it is interested in). The proxy then provides a *callback* function that the real object can call when there are any changes.



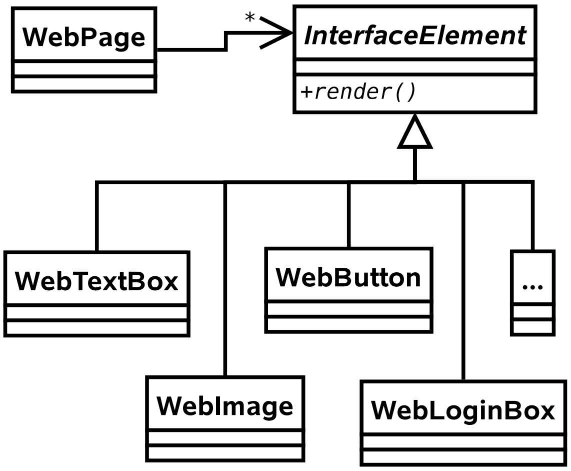
## Generalisation

This is the **Observer** pattern, also referred to as **Publish-Subscribe** when multiple machines are involved. It is useful when changes need to be propagated between objects and we don't want the objects to be tightly coupled. A real life example is a magazine subscription — you register to receive updates (magazine issues) and don't have to keep checking whether a new issue has come out yet. You unsubscribe as soon as you realise that 4GBP for 10 pages of content and 60 pages of advertising isn't good value.



## Abstract Factory

Assume that the front-end part of our system (i.e. the web interface) is represented internally by a set of classes that represent various entities on a web page:



Let's assume that there is a render() method that generates some HTML which can then be sent on to web browsers.

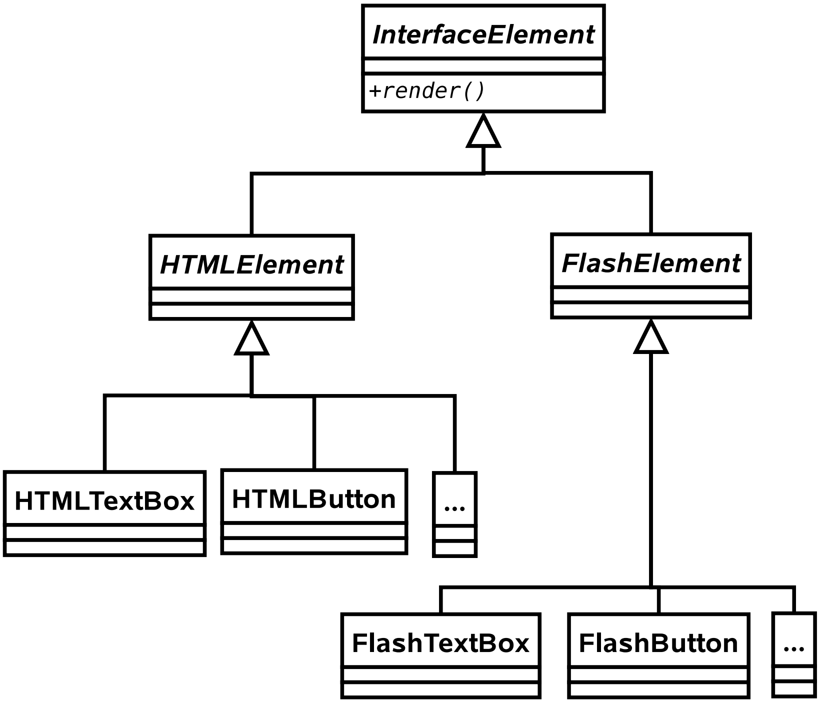
**Problem:** Web technology moves fast. We want to use the latest browsers and plugins to get the best effects, but still have older browsers work. e.g. we might have a Flash site, a SilverLight site, a DHTML site, a low-bandwidth HTML site, etc. How do we handle this?

**Solution 1:** Store a variable ID in the InterfaceElement class, or use the **State** pattern on each of the subclasses.

- ✔ Works.
- ✗ The **State** pattern is designed for a single object that regularly changes state. Here we have a family of objects in the same state (Flash, HTML, etc.) that we choose between at compile time.

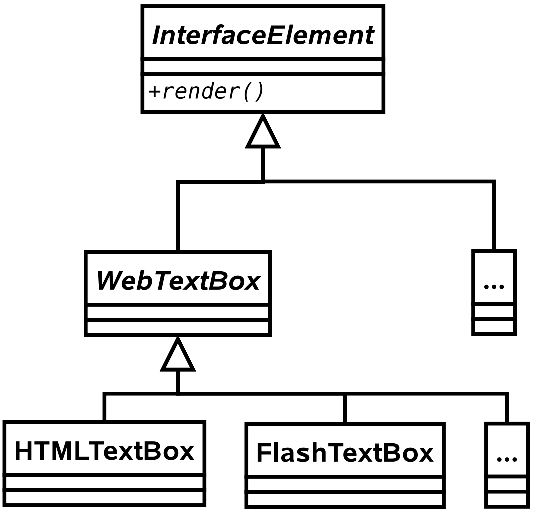
✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

**Solution 2:** Create specialisations of InterfaceElement:



✗ Lots of code duplication.
✗ Nothing keeps the different TextBoxes in sync as far as the interface goes.
✗ A lot of work to add a new interface component type.
✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

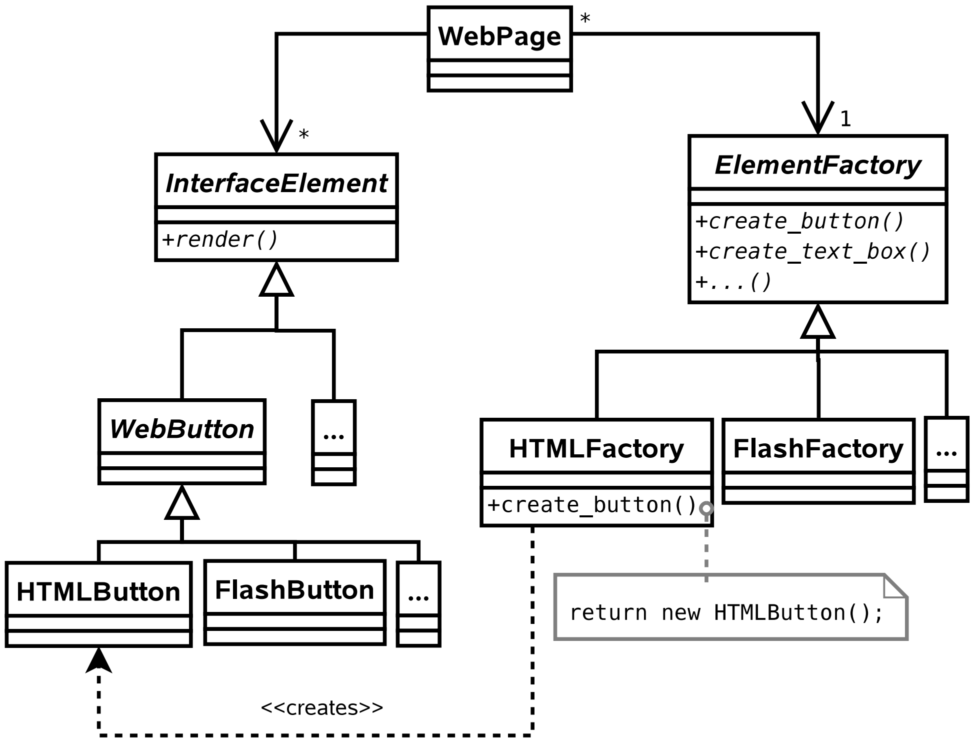
**Solution 3:** Create specialisations of each InterfaceElement subclass:



✔ Standardised interface to each element type.

✗ Still possible to inadvertently mix element types.

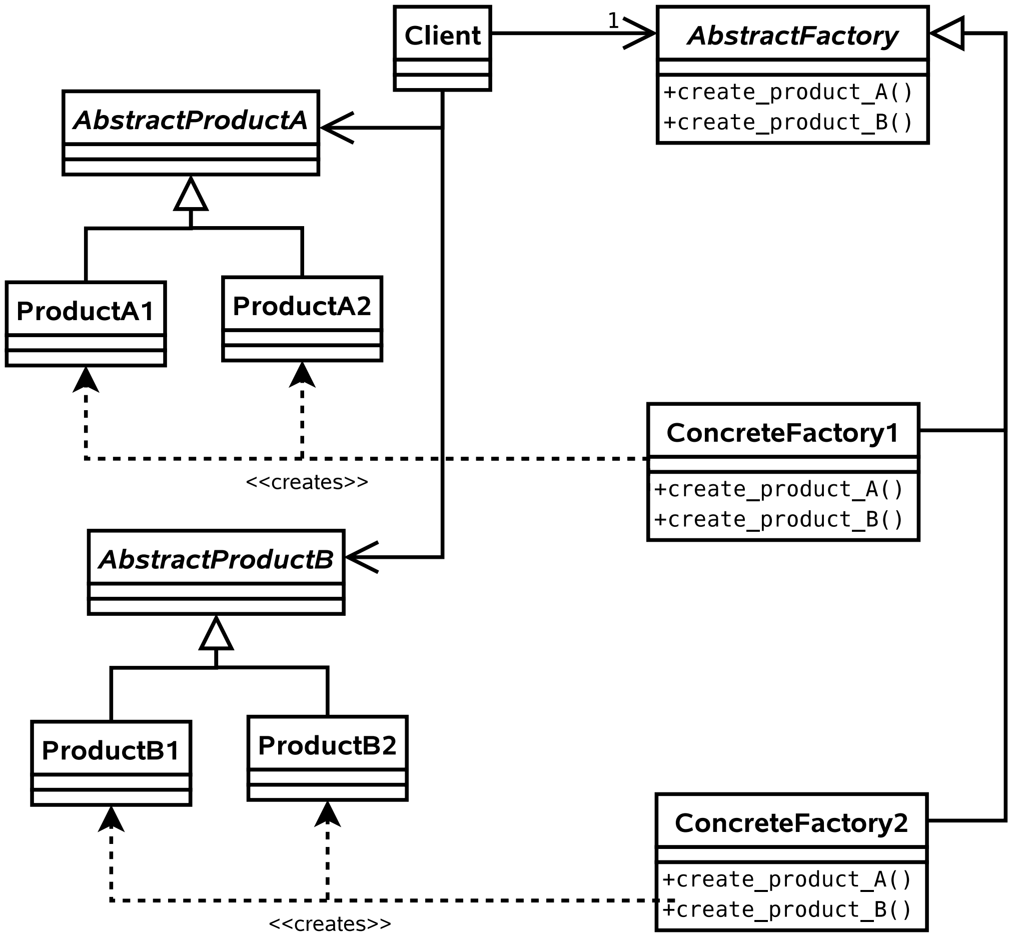
**Solution 4:** Apply the **Abstract Factory** pattern. Here we associate every WebPage with its own 'factory' — an object that is there just to make other objects. The factory is specialised to one output type. i.e. a FlashFactory outputs a FlashButton when create_button() is called, whilst a HTMLFactory will return an HTMLButton() from the same method.



- ✔ Standardised interface to each element type.
- ✔ A given WebPage can only generate elements from a single family.
- ✔ Page is completely decoupled from the family so adding a new family of elements is simple.
- ✗ Adding a new element (e.g. SearchBox) is difficult.
- ✗ Still have to create a lot of classes.

## Generalisation

This is the **Abstract Factory** pattern. It is used when a system must be configured with a specific family of products that must be used together.



Note that usually there is no need to make more than one factory for a given family, so we can use the **Singleton** pattern to save memory and time.

## Summary

From the original Design Patterns book:

**Decorator** Attach additional responsibilities to an object dynamically. Decorators provide flexible alternatives to subclassing for extending functionality.

**State** Allow and object to alter its behaviour when its internal state changes.

**Strategy** Define a family of algorithms, encapsulate each on, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objecta uniformly.

**Singleton** Ensure a class only has one instance, and provide a global point of access to it.

**Proxy** Provide a surrogate or placeholder for another object to control access to it.

**Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated accordingly.

**Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

## Classifying Patterns

Often patterns are classified according to what their intent is or what they achieve. The original book defined three classes:

**Creational Patterns** . Patterns concerned with the creation of objects (e.g. **Singleton**, **Abstract Factory**).

**Structural Patterns** . Patterns concerned with the composition of classes or objects (e.g. **Composite**, **Decorator**, **Proxy**).

**Behavioural Patterns** . Patterns concerned with how classes or objects interact and distribute responsibility (e.g. **Observer**, **State**, **Strategy**).

## Other Patterns

You've now met eight Design Patterns. There are plenty more (23 in the original book), but this course will not cover them. What has been presented here should be sufficient to:

- Demonstrate that object-oriented programming is powerful.
- Provide you with (the beginnings of) a vocabulary to describe your solutions.
- Make you look critically at your code and your software architectures.
- Entice you to read further to improve your programming.

Of course, you probably won't get it right first time (if there even is a 'right'). You'll probably end up *refactoring* your code as new situations arise. However, if a Design Pattern *is* appropriate, you should probably use it.

## Performance

Note that all of the examples here have concentrated on structuring code to be more readable and maintainable, and to incorporate constraints structurally where possible. At no point have we discussed whether the solutions *perform* better. Many of the solutions exploit runtime polymorphic behaviour, for example, and that carries with it certain overheads.

This is another reason why you can't apply Design Patterns blindly. [This is a good thing since, if it wasn't true, programming wouldn't be interesting, and you wouldn't get jobs!].