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.

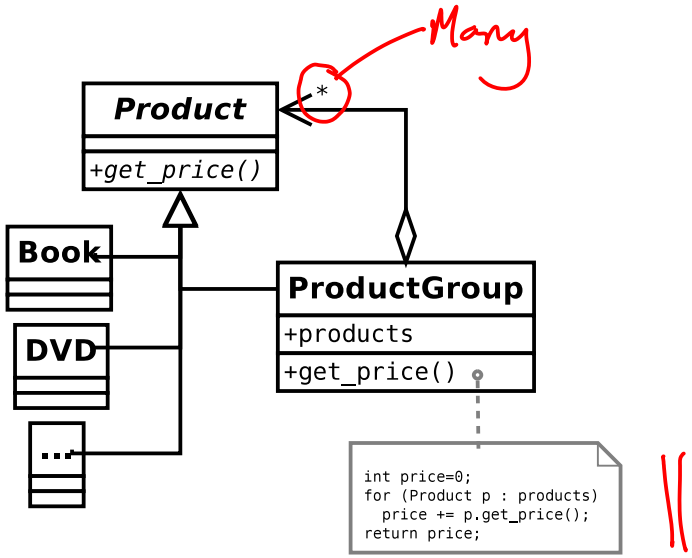
4.7 Composite Pattern

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.

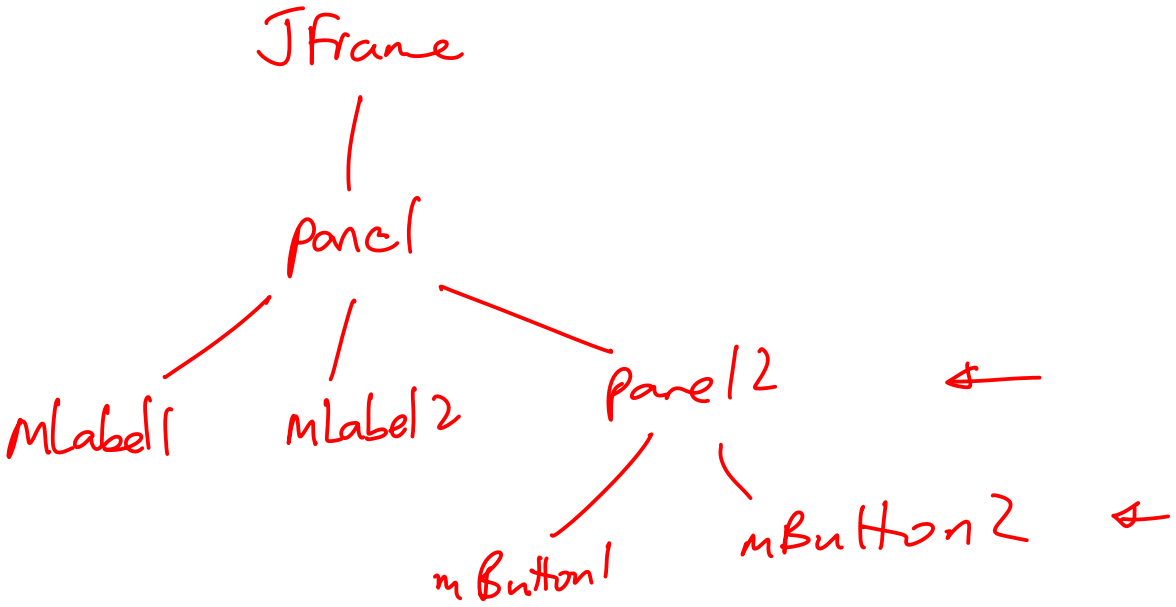
- ✔ Does the basic job.
- ✗ What if a product belongs to no groups (which will be the majority case)? Then we are wasting memory and cluttering up the code.
- ✗ What if a product belongs to multiple groups? How many groups should we allow for?

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



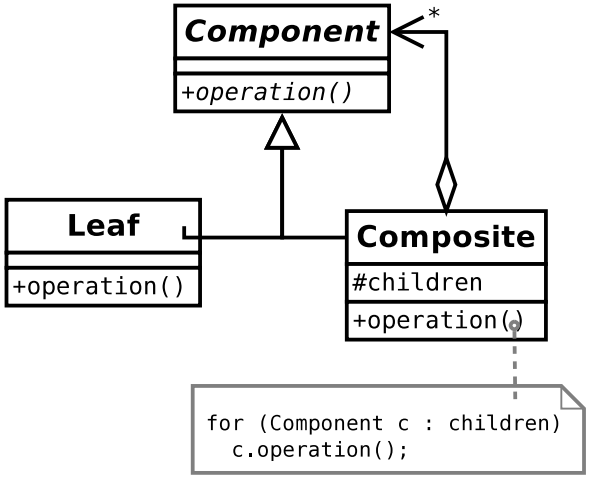
If you're still awake, you may be thinking this is a bit like the **Decorator** pattern, except that the new class supports associations with multiple Products (note the * by the arrowhead). Plus the intent is different – we are not adding new functionality but rather supporting the same functionality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


JFrame
panel
mLabel1
mLabel2
panel2
mButton1
mButton2

4.7.1 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.

4.8 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 that 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;
}
```

Database d2 = new 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.

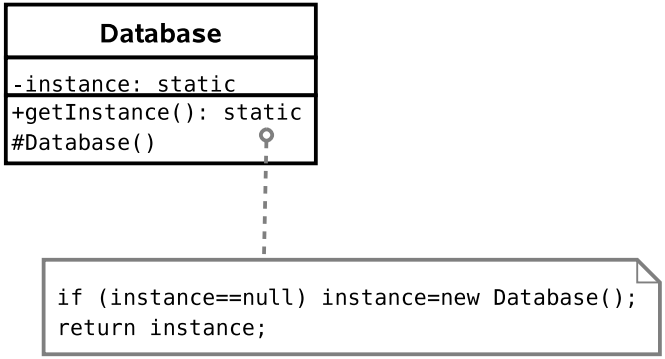
```
public class System {
  public System(Database d) {...}
}

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 instantiated – 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.

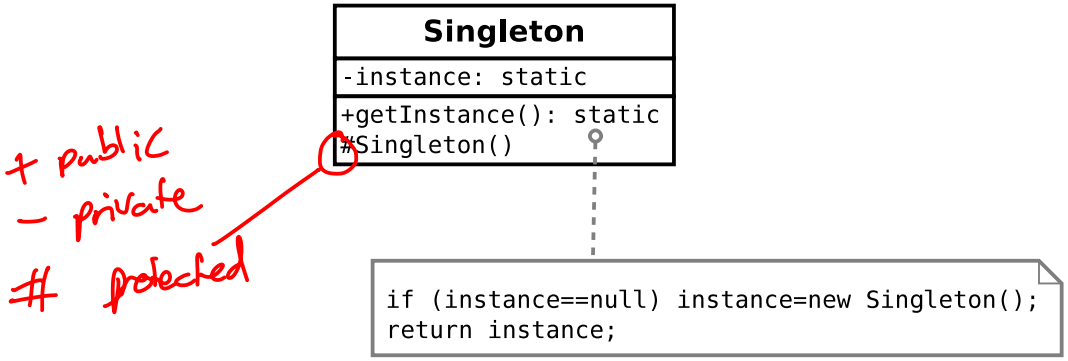
Of course, nothing stops a programmer from ignoring the getInstance() function and just creating a new Database object. So we use a neat trick: we make the constructor *private* or *protected*. This means code like new Database() isn't possible from an arbitrary class.



- ✔ *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!

4.8.1 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.

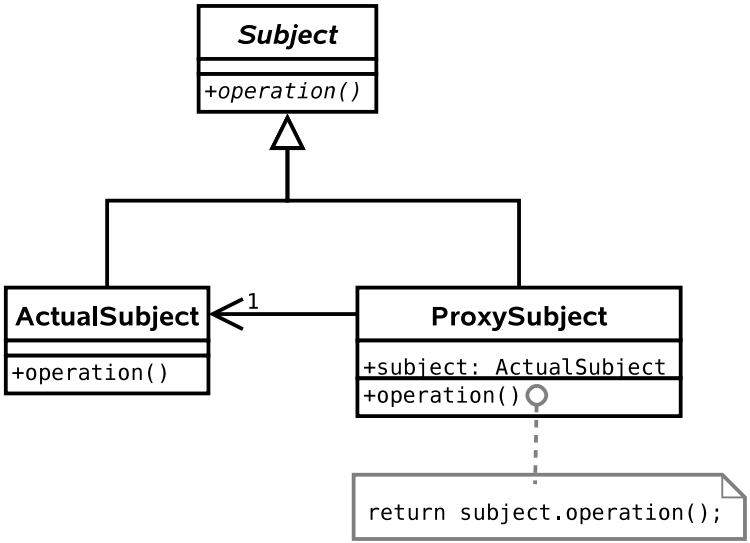
Protected members are accessible to the class, any subclasses, *and all classes in the same package*. Therefore, any class in the same package as your base class will be able to instantiate Singleton objects at will, using the new keyword!

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.

4.9 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.



4.9.1 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 does want to access more information, the associated get() methods in the proxy object automatically retrieve them from the database.

4.9.2 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.

4.9.3 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.

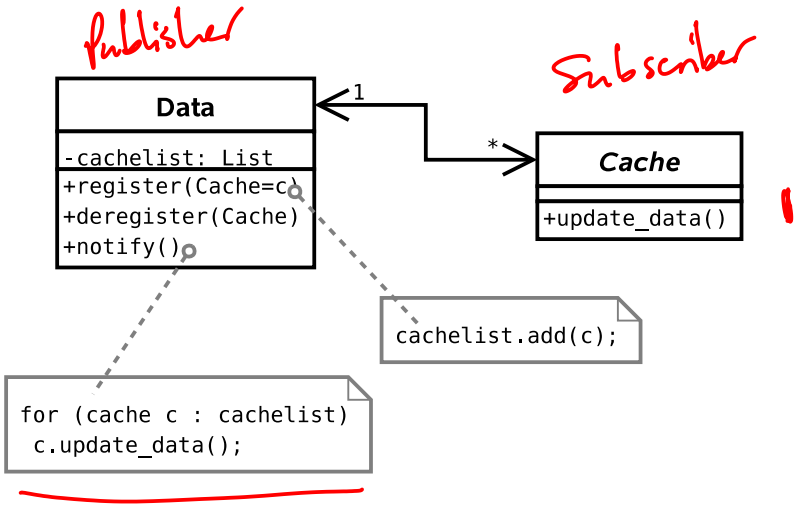
4.10 Observer Pattern

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).

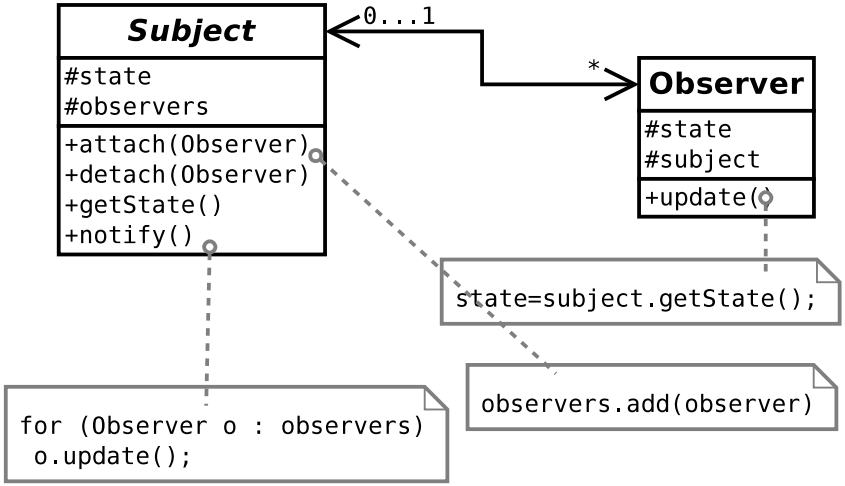
✗ How frequently should we poll? Too quickly and we might as well not have cached at all. Too slow and changes will be slow to propagate.

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.



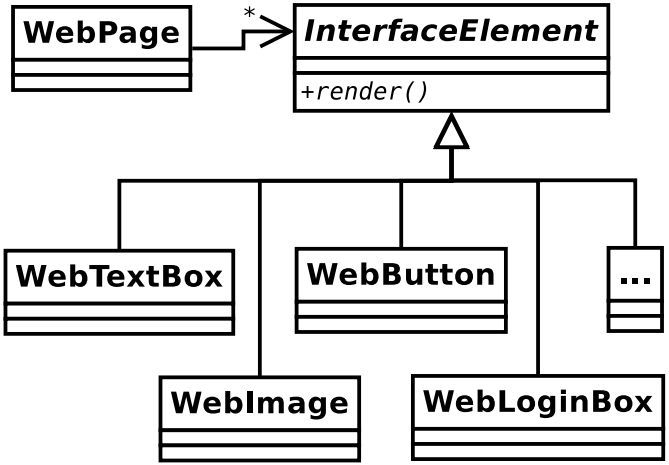
4.10.1 Generalisation

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.


Subject
#state
#observers
+attach(Observer)
+detach(Observer)
+getState()
+notify()
0...1
*
Observer
#state
#subject
+update()
state=subject.getState();
observers.add(observer)
for (Observer o : observers)
o.update();

4.11 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:



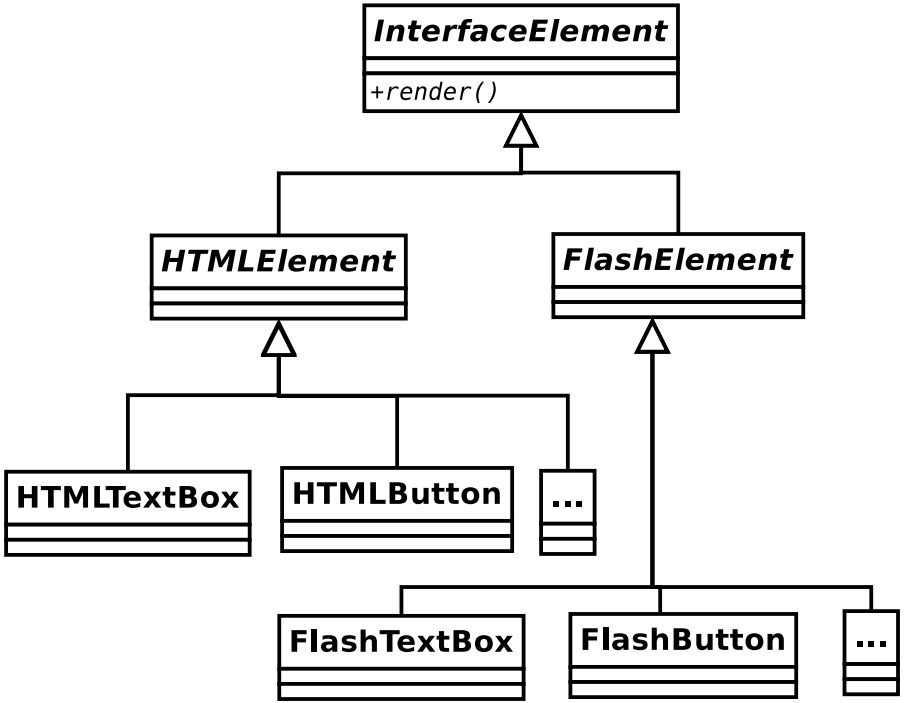
Let's assume that there is a render() method that generates some HTML which can then be sent on to web browsers.

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.

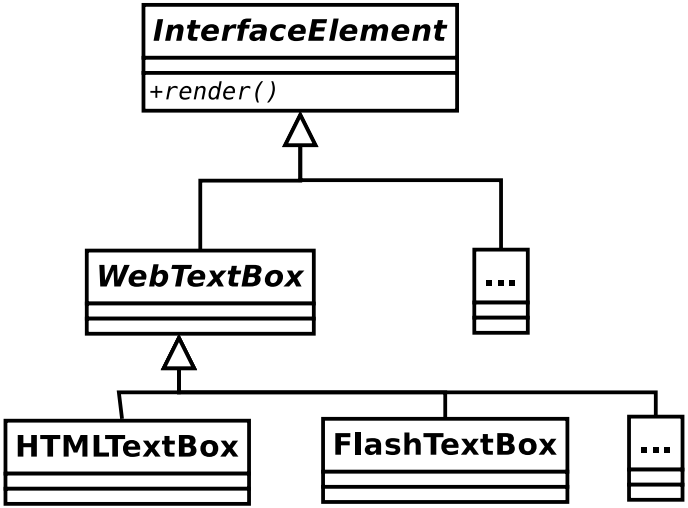
Solution 2: Create specialisations of InterfaceElement:



✗ Lots of code duplication.
✗ Nothing keeps the different TextBoxes in sync as far as the interface goes.

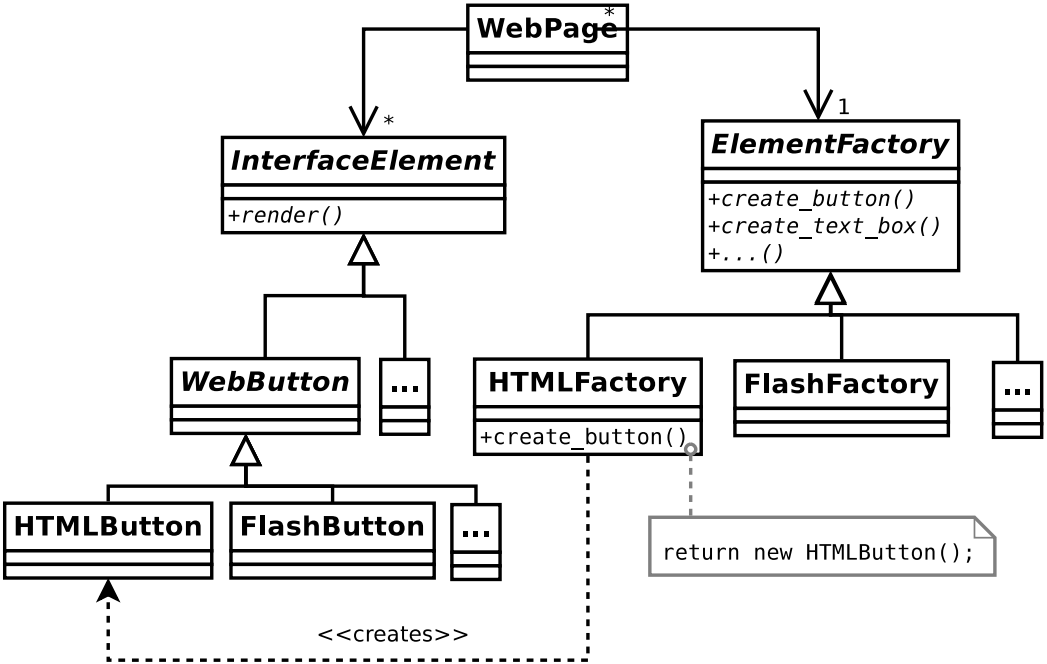
✗ A lot of work to add a new interface component type.
✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

Solution 3: Create specialisations of each InterfaceElement subclass:



✔ Standardised interface to each element type.
✗ Still possible to inadvertently mix element types.

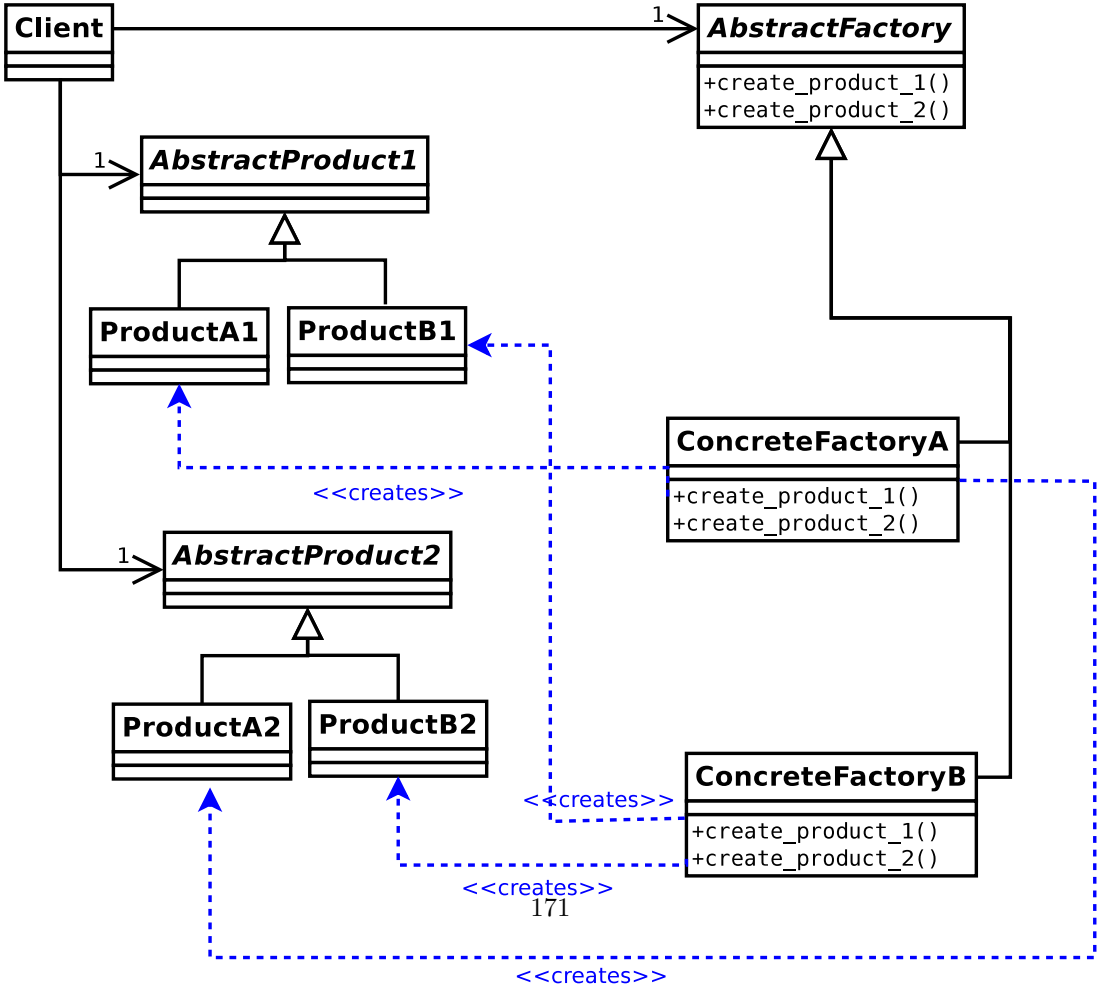
Solution 4: Apply the **Abstract Factory** pattern. Here we associate every WebPage with its own 'factory' — an object that is there just to make other objects. The factory is specialised to one output type. i.e. a FlashFactory outputs a FlashButton when create_button() is called, whilst a HTMLFactory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

4.11.1 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.


Client
1
AbstractFactory
+create_product_1()
+create_product_2()
1
AbstractProduct1
ProductA1
ProductB1
ConcreteFactoryA
+create_product_1()
+create_product_2()
<<creates>>
1
AbstractProduct2
ProductA2
ProductB2
ConcreteFactoryB
+create_product_1()
+create_product_2()
<<creates>>
<<creates>>
<<creates>>

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.

4.12 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.

4.12.1 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**).

4.12.2 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 s 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.

4.12.3 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!].

Once we have compiled our Java source code, we end up with a set of .class files; these contain bytecode. We can then distribute these files without their source code (.java) counterparts.

In addition to javac you will also find a javap program which allows you to poke inside a class file. For example, you can disassemble a class file to see the raw bytecode using javap -c classfile:

Input:

```
public class HelloWorld {
  public static void main(String[] args) {
```

```
    System.out.println("Hello World");
  }
}
```

javap output:

```
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object{
public HelloWorld();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

public static void main(java.lang.String[]);
  Code:
   0: getstatic #2; //Field java/lang/System.out:
                    //Ljava/io/PrintStream;
   3: ldc #3; //String Hello World
   5: invokevirtual #4; //Method java/io/PrintStream.println:
                        //(Ljava/lang/String;)V
   8: return

}
```

This probably won't make a lot of sense to you right now: that's OK. Just be aware that we can view the bytecode and that sometimes this can be a useful way to figure out *exactly* what the JVM will do with a bit of code. You aren't expected to know bytecode.