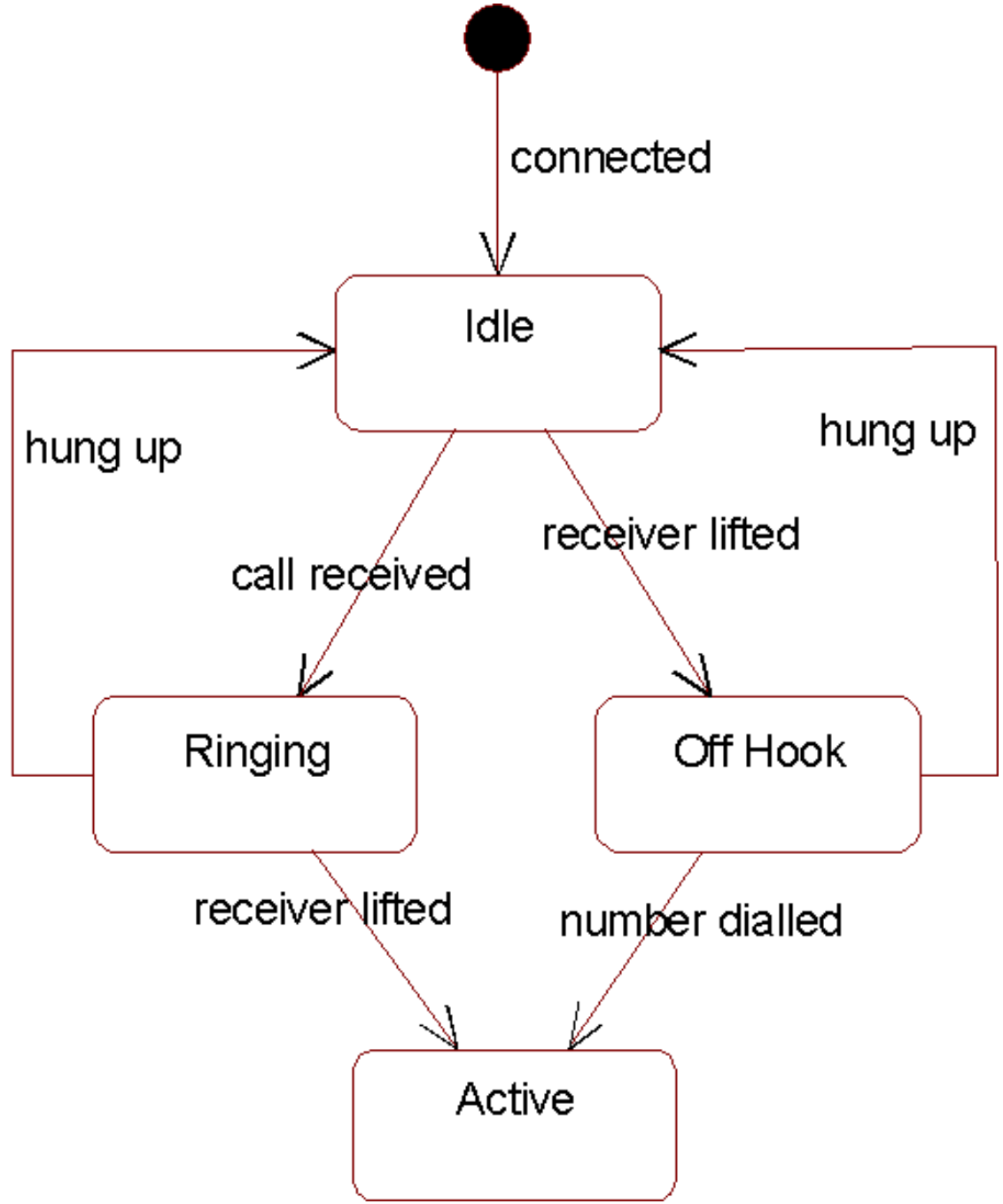
Lecture 11
Modelling States-State Diagrams

- After taking a break to consider Inheritance and System Architecture, we are now going to return to the design stage of the construction phase and consider state modelling.
- State Diagrams allow us to model the possible states that an object can be in.
- The model allows us to capture the significant events that can act on the object, and also the effect of the events.
- These models have many applications, but perhaps the strongest application is to ensure that odd, illegal events cannot happen in our system.

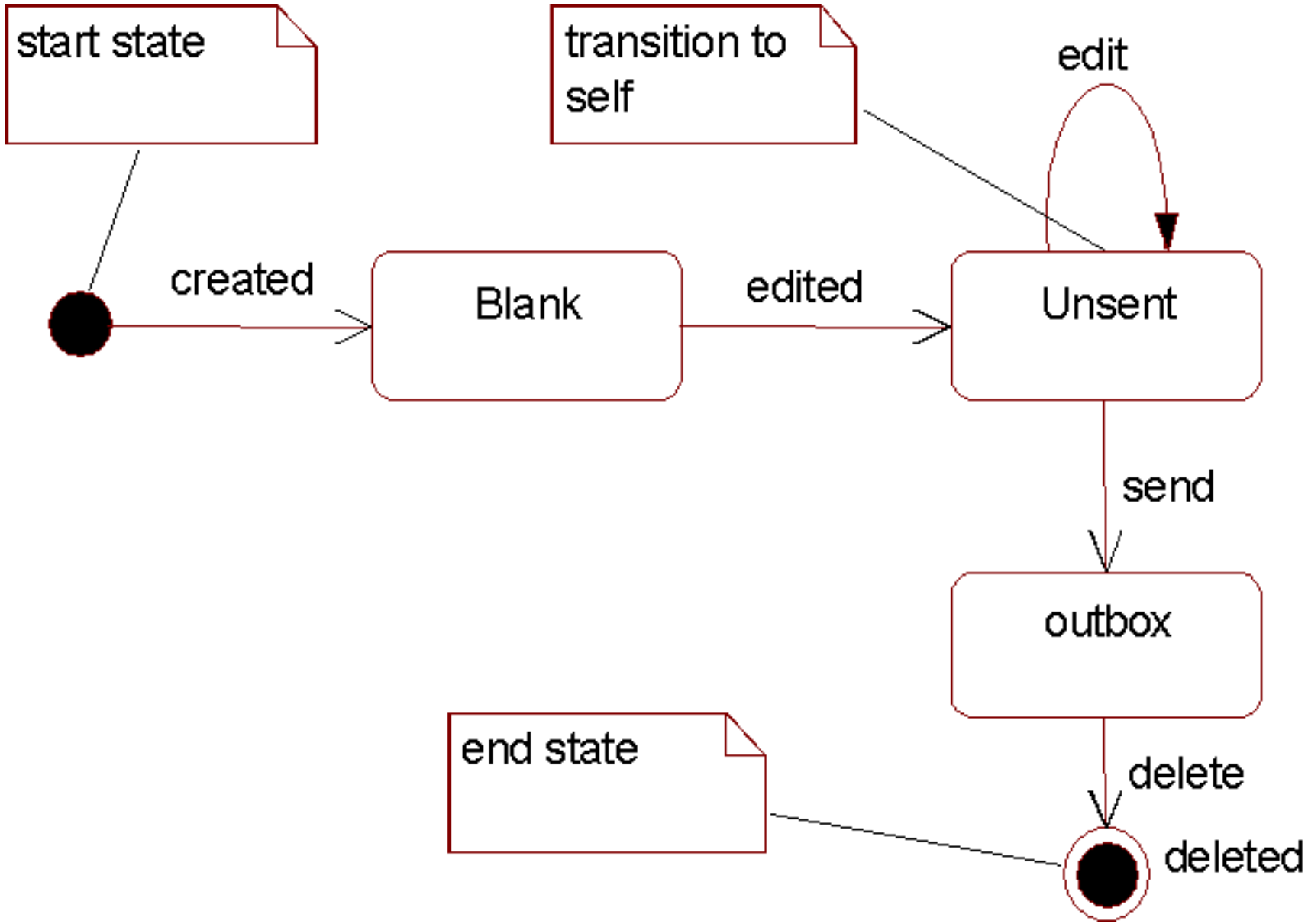
- The example that was given before talks about a situation that seems to happen an awful lot, if local newspapers are anything to go by -a gas bill is sent to a customer who died five years ago!
- Carefully written state diagrams should prevent these kind of erroneous events occurring.

Example Statechart-Telephone



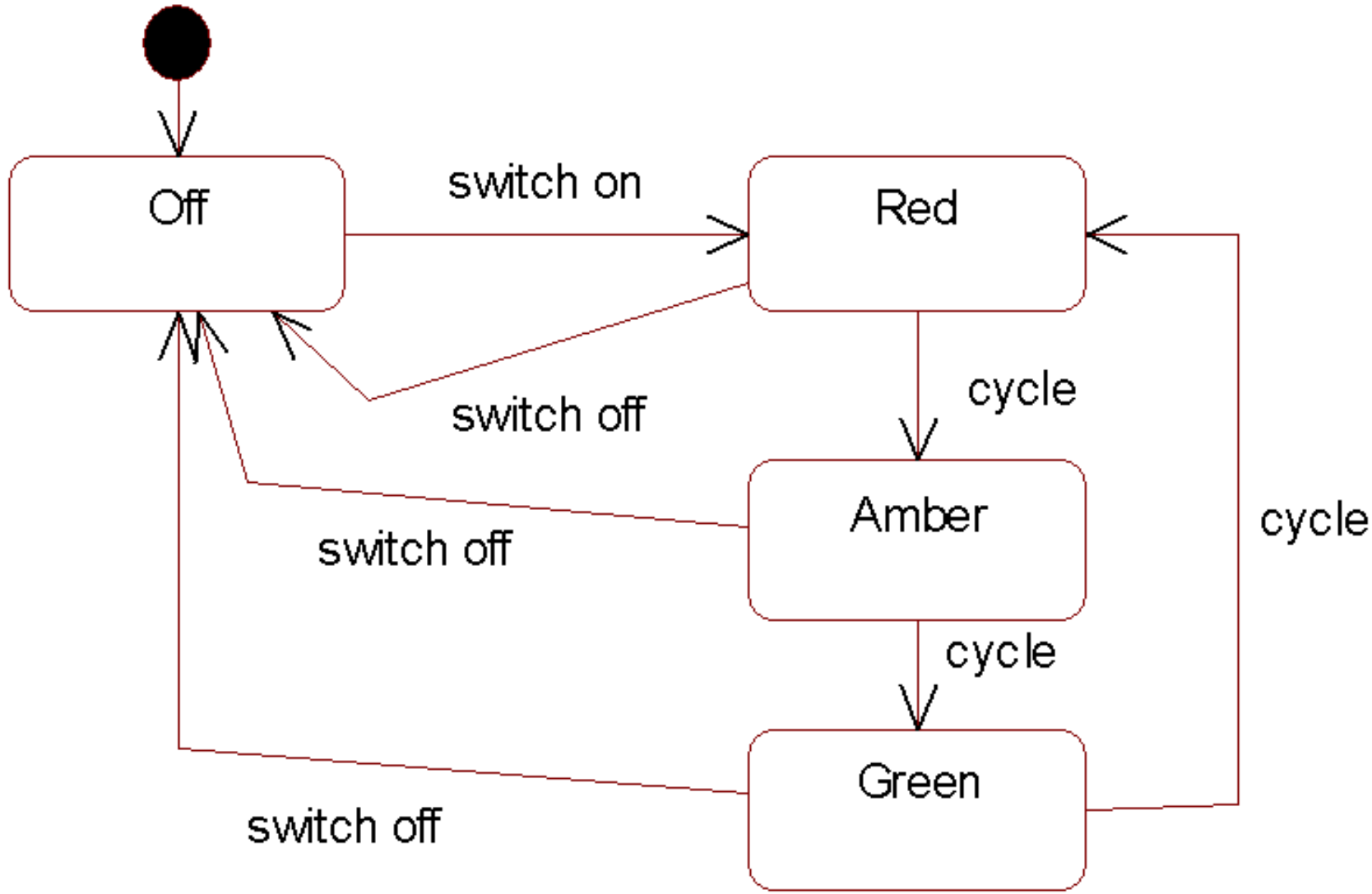
- We will look at the syntax of this diagram in detail shortly, but the basics of the diagram should be obvious.
- The sequence of events that can occur to the telephone are shown, and the states that the telephone can be in are also shown.
- For example, from being idle, the telephone can either go to being "Off the Hook" (if the receiver is lifted), or the telephone can go to "Ringing" (if a call is received).

State Diagram Syntax - an E-Mail example

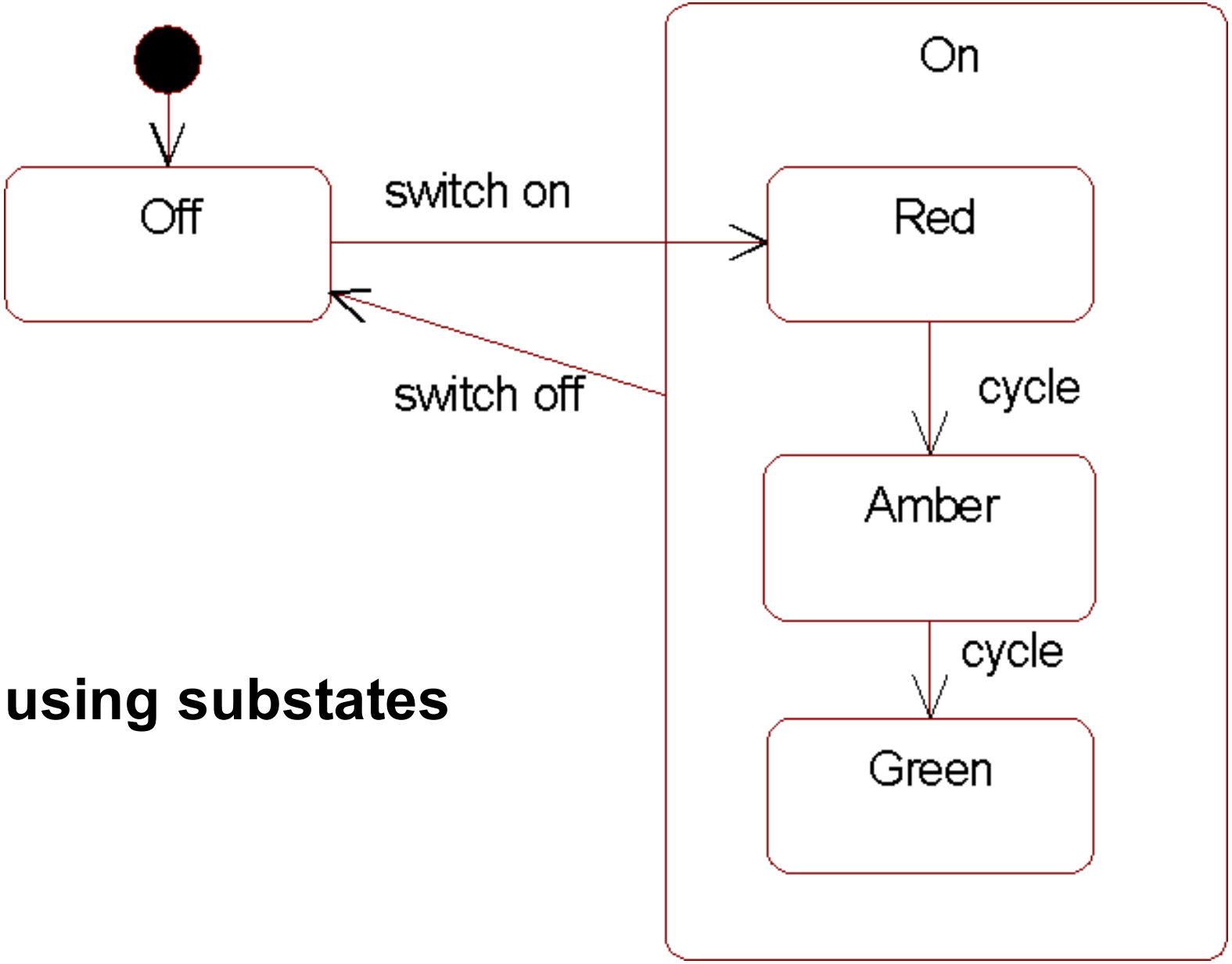


- The diagram above shows most of the state diagram syntax. The object will have a start state (the filled circle), describing the state of the object at the point of creation.
- Most objects have an end state (the "bullseye"), describing the event that happens to destory the object.
- Some events cause a state transition that causes the object to remain in the same state.
- In the example above, the e-mail can receive an "edit" event only if the status of the object is "unsent". But the event does not cause a state change.
- This is a useful syntax to illustrate that the "edit" event can not happen in any of the other states.

Substates


Off
switch on
Red
switch off
cycle
Amber
switch off
cycle
cycle
Green
switch off

- Sometimes, we require a model that describes states within states.
- The above statechart is perfectly valid (describing a traffic light object’s states), but it is hardly elegant.
- Essentially, it can be switched off at any time, and it is this set of events that is causing the mess.
- There is a "superstate" present in this model. The traffic light can be either "On" or "Off".
- When it is in the "On" state, it can be in a series of substates of "Red", "Amber" or "Green".
- The UML provides for this by allowing "nesting" of states:

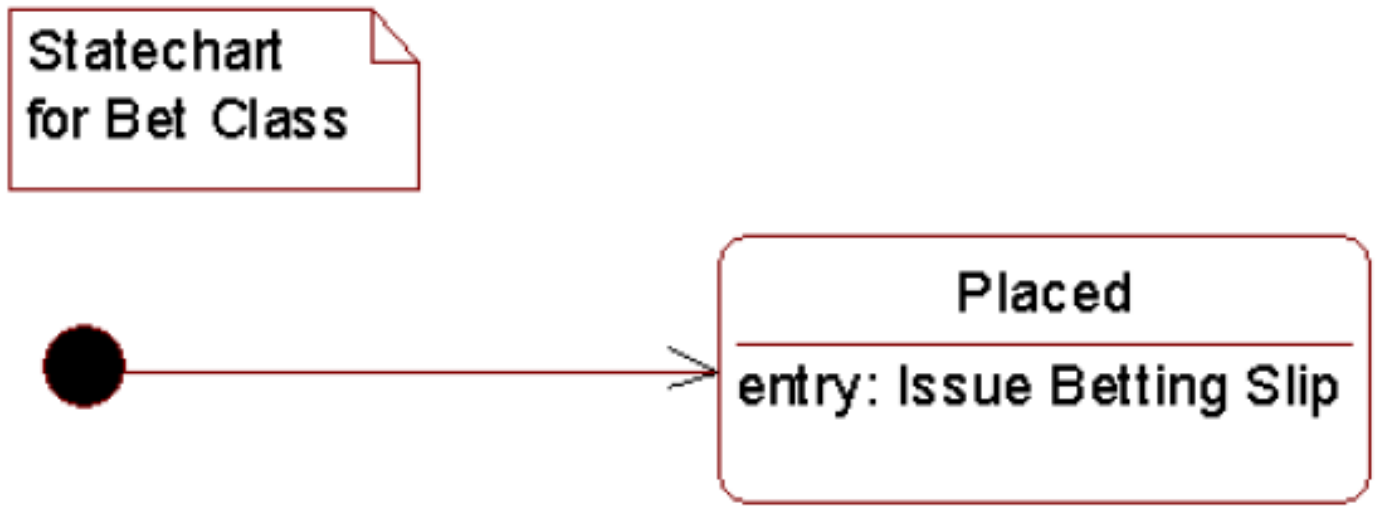


Simpler state model using substates

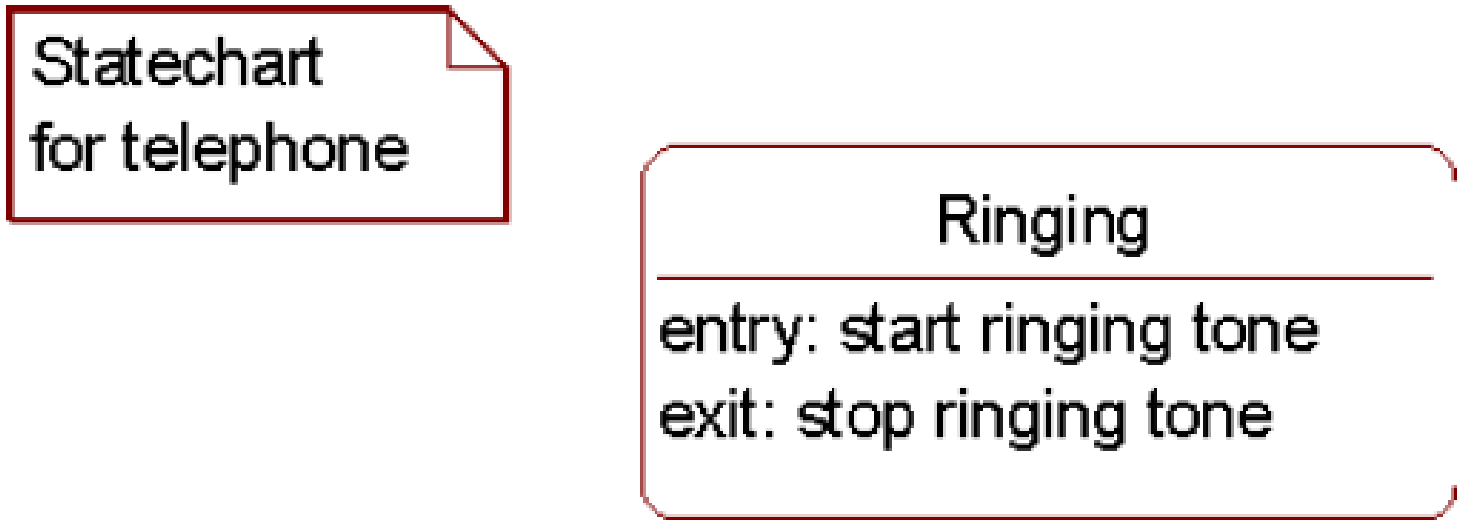
Note that in the diagram above, the small arrow pointing in to the "red" state indicates that this is the default state - on commencement of the "on" state, the light will be set to "Red".

Entry/Exit Events

Sometimes it is useful to capture any actions that need to take place when a state transition takes place. The following notation allows for this:



Here, we need to issue a betting slip when the state change occurs

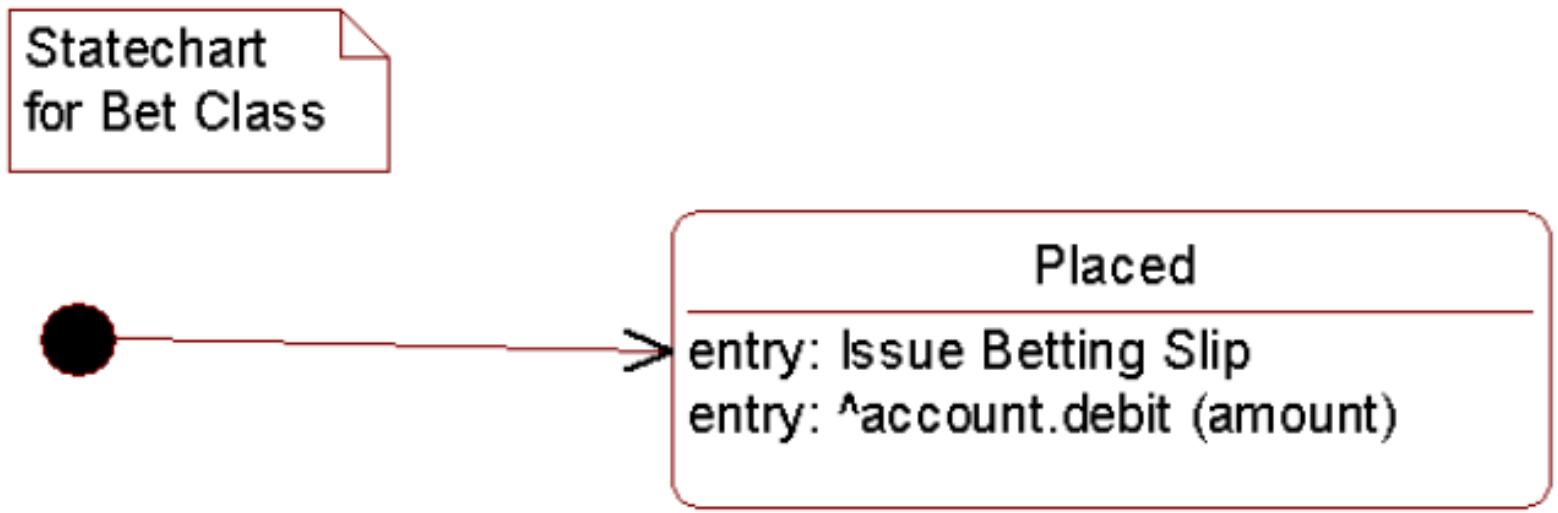


Here, the ring tone starts on entry to the state - the ring tone stops on exit

Send Events

The above notation is useful when you need to comment that a particular action needs to take place. Slightly more formally, we can tie this approach to the idea of objects and collaboration. If a state transition implies that a message has to be sent to another object, then the following notation is used (alongside the entry or exit box):

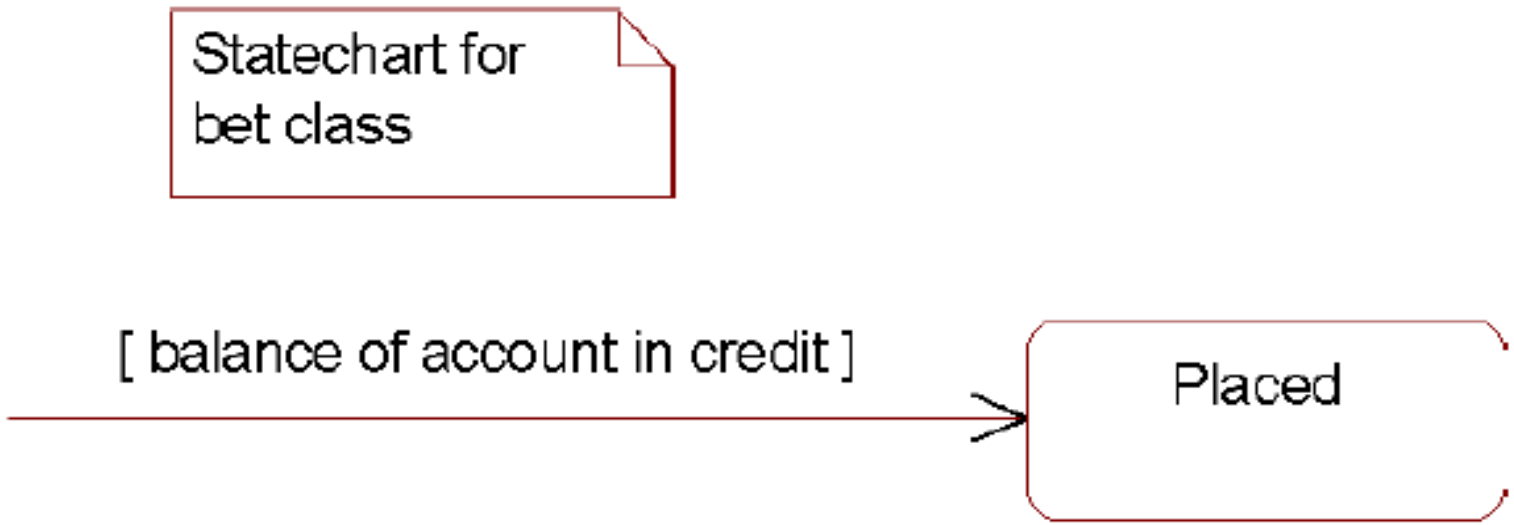
```
^object.method (parameters)
```



formal notation indicating that a message must be sent on the state

Guards

Sometimes we need to insist that a state transition is possible only if a particular condition is true. This is achieved by placing a condition in square brackets as follows:



Here, the transition to the "Placed" state can only occur if the balance of the account is in credit

Other Uses for State Diagrams

Although the most obvious use for these diagrams is to track the state of an object, in fact, statecharts can be used for any state-based element of the system. Use Cases are a clear candidate (for example, a use might only be able to proceed if the user has logged on).

Even the state of the entire system can be modelled using the statechart - this is clearly a valuable model for the "central architecture team" in a large development.

Summary

In this chapter, we looked at State Transition Diagrams.

We saw:

- The syntax of the diagram
- How to use Substates
- Entry and Exit Actions
- Send Events and Guards

Statecharts are quite simple to produce, but often require deep thought processes

Most commonly produced for Classes, but can be used for anything : Use Cases, entire Systems, etc

Transition to Code

This brief section describes some of the issues surrounding the move from the model to code. For the examples, we'll use Java, but the Java is very simple and can be easily applied to any modern Object Oriented language.

Synchronising Artifacts

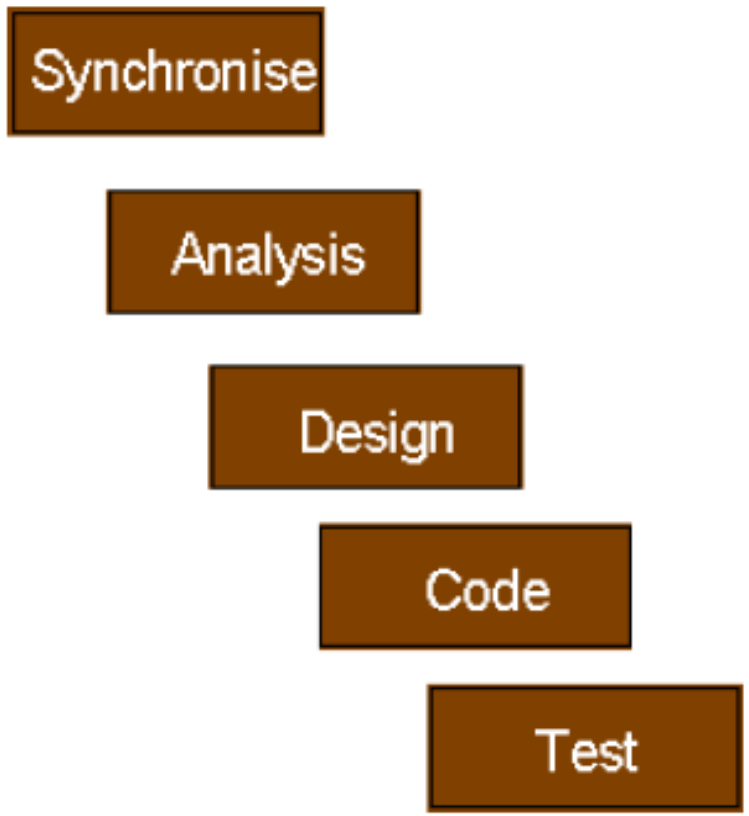
One of the key problems of design and coding is keeping the model in line with the code.

Some projects will want to totally separate design from code. Here, the designs are built to be as complete as possible, and coding is considered a purely mechanical transformation process.

For some projects, the design models will be kept fairly loose, with some design decisions deferred until the coding stage.

Either way, the code is likely to "drift" away from the model to a lesser or greater extent. How do we cope with this?

One approach is to add an extra stage to each iteration - Synchronising artifacts. Here, the models are altered to reflect the design decisions that were made during coding in the previous iteration.



Extra stage in the waterfall - synchronisation

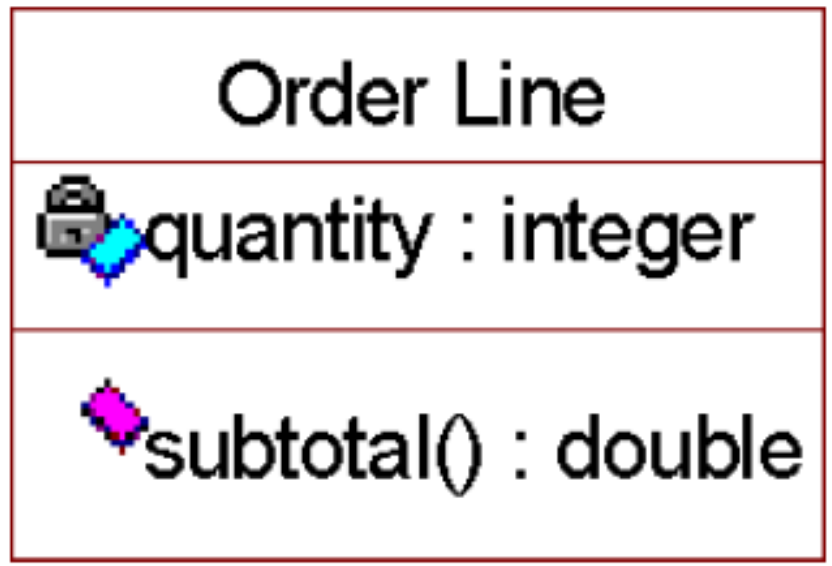
Clearly, this is a far from simple solution, as often, major changes will have been made. However, it is workable as long as the iterations are short and the complexity of each one is manageable. Well, that's what we've been aiming for all along!!

Some CASE tools allow "reverse engineering" - that is, the generation of a model from code. This could be a help with synchronising - at the end of iteration 1, regenerate the model from the code, and then work from this new model for iteration 2 (and repeat the process). Having said that, the technology of reverse engineering is far from advanced, so this may not suit all projects!

Mapping Designs to Code

Your code's class definitions will be derived from the Design Class Diagram. The method definitions will come largely from the Collaboration Diagrams, but extra help will come from the Use Case descriptions (for the extra detail, particularly on exception/alternate flows) and the State Charts (again, for trapping error conditions).

Here's an example class, and what the code might look like:



The Order Line class, with a couple of example members

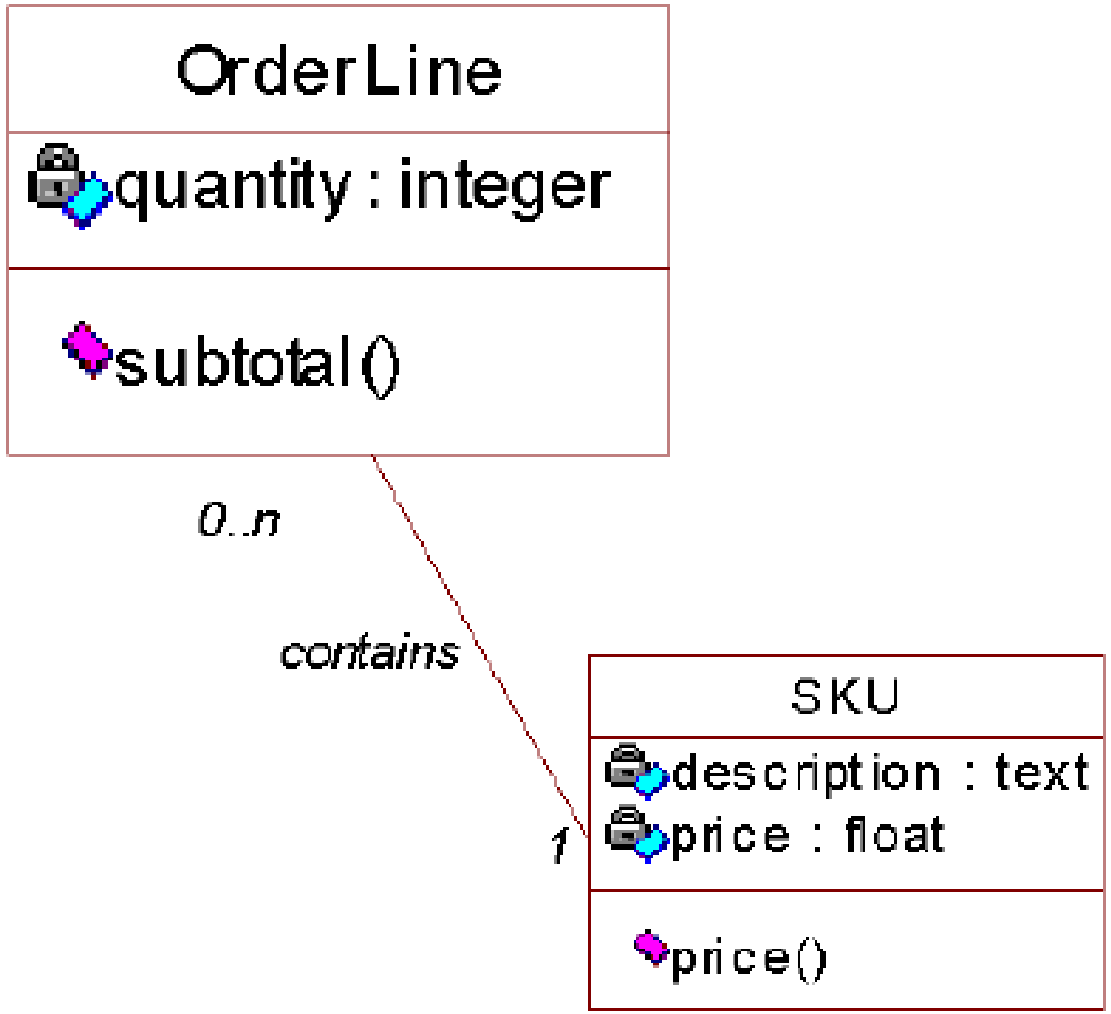
The resulting code would end up looking something like this (following a mechanical conversion process):

```
public class OrderLine
{
    public OrderLine(int qty, SKU product)
    {
        // constructor
    }
    public double subtotal()
    {
        // method definition
    }

    private int quantity;
}
```

Sample Order Line Code

Note that in the code above, I have added a constructor. We omitted the create() methods from the Class Diagram (as it seems to be a convention these days), so this needed to be added.



The aggregation of Order Lines and SKU's

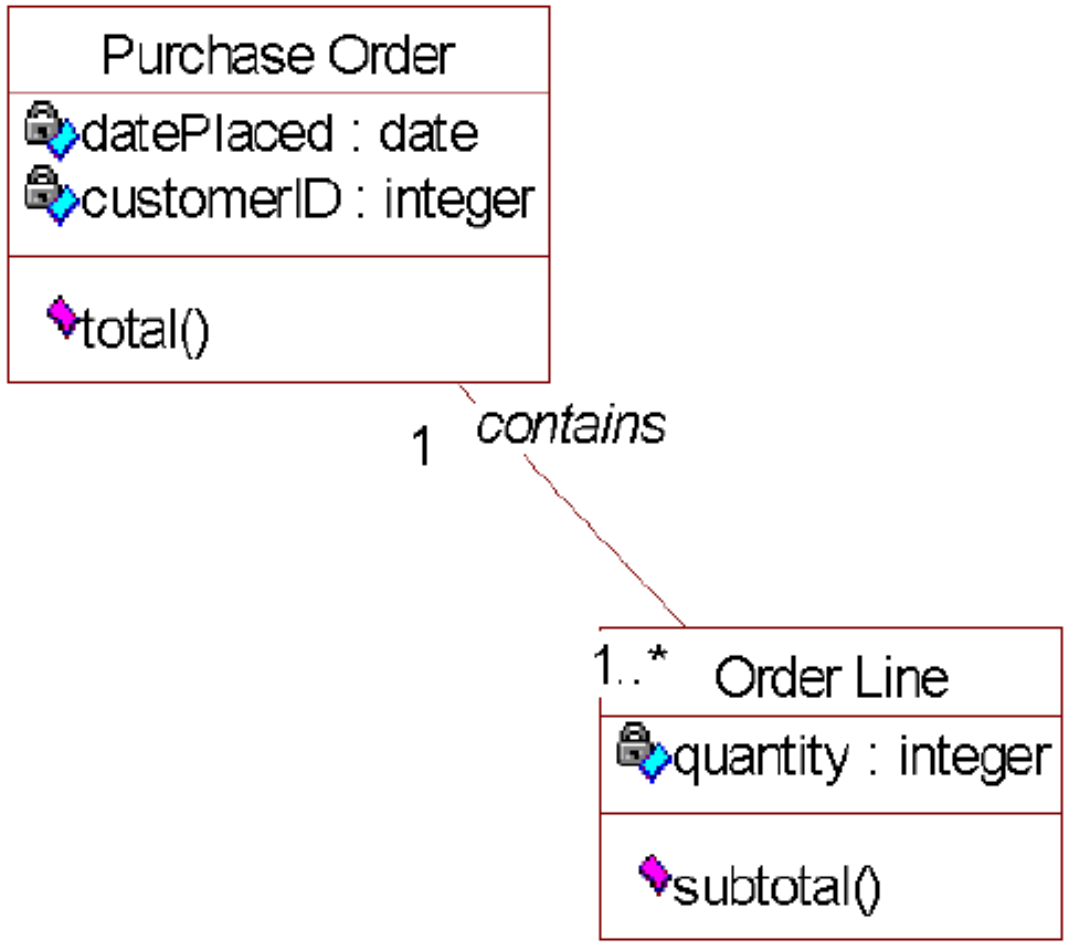
An order line contains a reference to a single SKU, so we also need to add this to the class code:

```
public class OrderLine
{
    public OrderLine(int qty, SKU product);
    public float subtotal();

    private int quantity;
    private SKU SKUOrdered;
}
```

Adding the reference attribute (method blocks omitted for clarity)

What if a class needs to hold a list of references to another class? A good example is the relationship between Purchase Orders and Purchase Order Lines. A Purchase Order "owns" a list of lines, as in the following UML:



A Purchase Order holds a list of Order Lines

The actual implementation of this depends upon the specific requirement (for example, should the list be ordered, is performance an issue, etc), but assuming we need a simple array, the following code will suffice:

```
public class PurchaseOrder
{
   public float total();

   private date datePlaced;
   private int  customerID;
   private Vector OrderLineList;
}
```

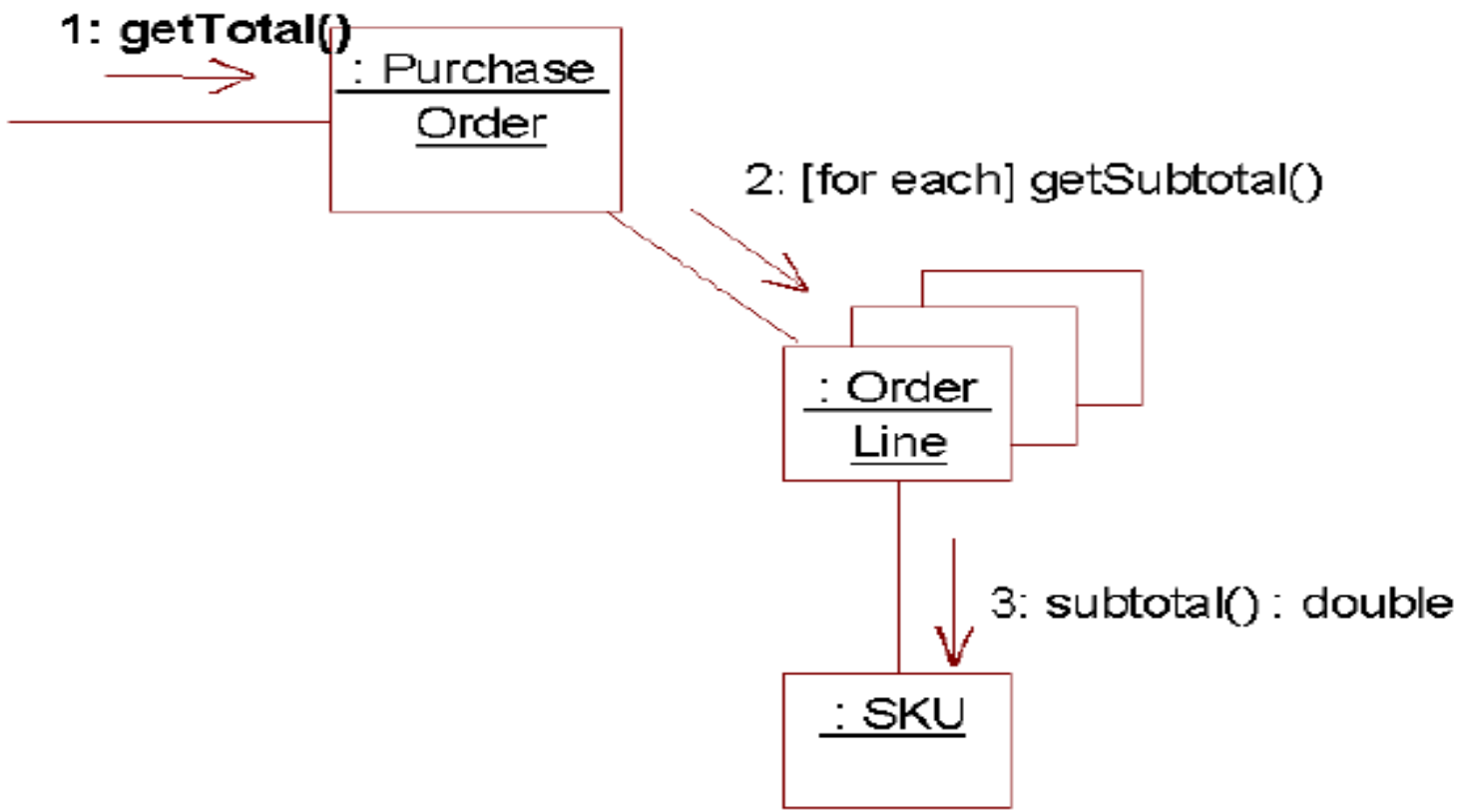
Adding a list of references

Initialising the list would be the job of the constructor. For non Java and C++ coders, a Vector is simply an array that can be dynamically resized. Depending on the requirement, a bog standard array would have worked too.

Defining the Methods

The collaboration diagram is a large input into the method definitions.

The following worked example describes the "get total" method for the Purchase Order. This method returns the total cost of all of the lines in the order:



"Get total" collaboration

Step 1

Clearly, we have a method called "getTotal()" in the purhcase order class:

```
public double getTotal()
{

}
```

method definition in the Purchase Order Class

Step 2

The collaboration says that the purchase order class now polls through each line:

```
public double getTotal()
{
   double total;
    for (int x=0; x<orderLineList.size();x++)
    {
       // extract the OrderLine from the list
       theLine = (OrderLine)orderLineList.get(x);

       total += theLine).getSubtotal();
    }
    return total;
}
```

code for getting the total, by polling all purchase order lines for the order.

Step 3

We have called a method called "getSubtotal()" in the OrderLine class. So this needs to be implemeted:

```
public double getSubtotal()
{
   return quantity * SKUOrdered.getPrice();
}
```

implementation of getSubtotal()

Step 4

We have called a method called "getPrice()" in the SKU Class. This needs implementing and would be a simple method that returns the private data member.

Mapping Packages into Code

We stressed that building packages is an essential aspect of system architecture, but how do we map them into code?

In Java

If you are coding in Java, packages are supported directly. In fact, every single class in Java belongs to a package. The first line of a class declaration should tell Java in which package to place the class (if this is omitted, the class is placed in a "default" package).

So if the SKU class was in a package called "Stock", then the following class header would be valid:

```
package com.mycompany.stock;

class SKU
{  ...
```

Figure 114 - Placing classes in packages

Best of all, Java adds an extra level of visibility on top of the standard private, public and protected. Java includes **package** protection. A class can be declared as being visible only to the classes in the same package - and so can the methods inside a class. This provides excellent support for encapsulation within packages. By making all classes visible only to the packages they are contained in (except the facades), subsystems can truly be developed independently.

In C++

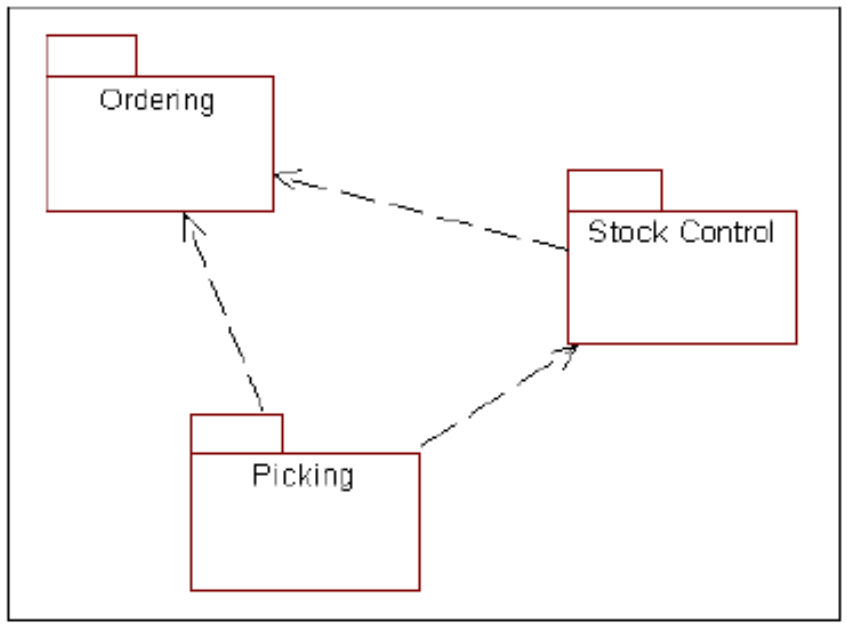
There is no direct support for packages in C++, but recently the concept of a namespace was added to the language. This allows classes to be placed in separate logical partitions, to avoid name clashes between namespaces (so I could create two namespaces, say Stock and Orders, and have a class called SKU in both of them).

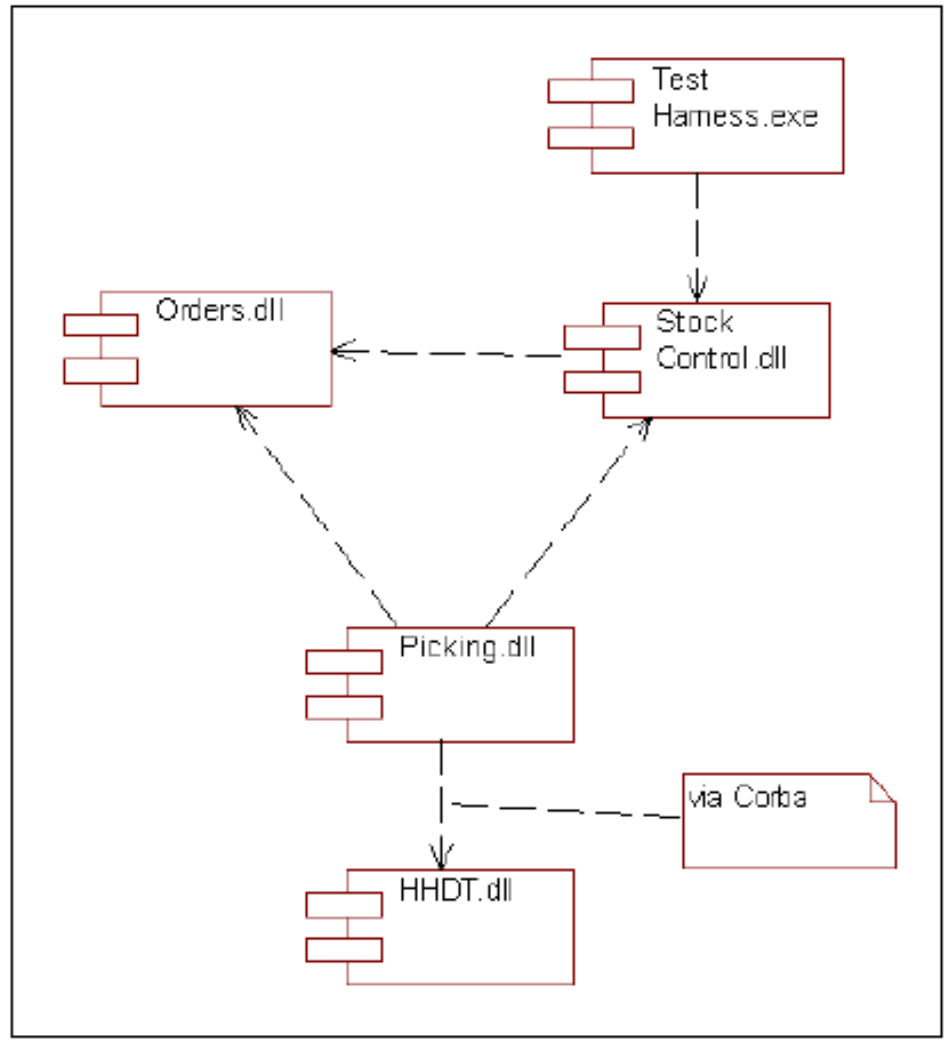
This provides some of the support of packages, but unfortunately it doesn't offer any protection via visibilities. A class in one namespace can access all of the public classes in another namespace.

The UML Component Model

This model shows a map of the physical, "hard", software components (as opposed to the logical view expressed by the package diagram).

Although the model will often be based on the logical package diagram, it can contain physical run time elements that weren't necessary at the design stage. For example, the following diagram shows an example logical model, followed by the eventual software physical model:





the logical compared to the physical view

The Component Model is very simple. It works in the same way as the package diagram, showing elements and the dependencies between them. However, this time, the symbol is different, and each component can be any physical software entity (an executable file, a dynamic link library, an object file, a source file, or whatever).

Note that the Component Model is based heavily on the package diagram, but has added a .dll to handle the Terminal Input/Output, and has added a test harness executable.

Summary

This chapter has described, in rough terms, the general process of converting the models into real code. We looked briefly at the issue of keeping the model synchronised with the code, and a couple of ideas on how to get around the problem.

We saw the component model. The model is not heavily used at present, but it is helpful in mapping the physical, real life software code and the dependencies between them.

The UML Applied Course CD shows how the Case Study followed on the course can be transformed into Java code - please feel free to explore it for more details.