Chapter 18

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.

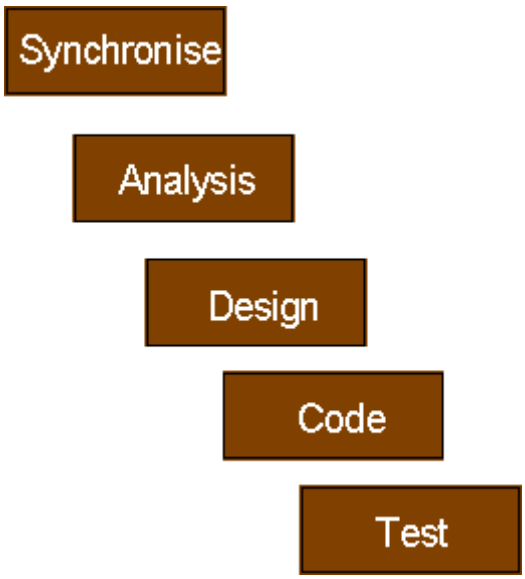

Figure 103 - Extra stage in the waterfall - synchronisation

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:

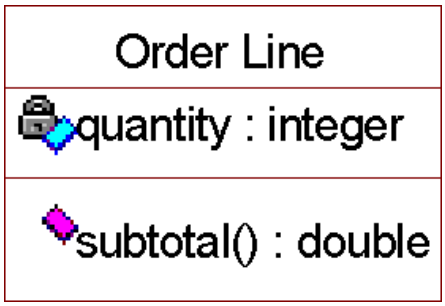

Figure 104 - The Order Line class, with a couple of example members

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;
}
```

Figure 105 - 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.

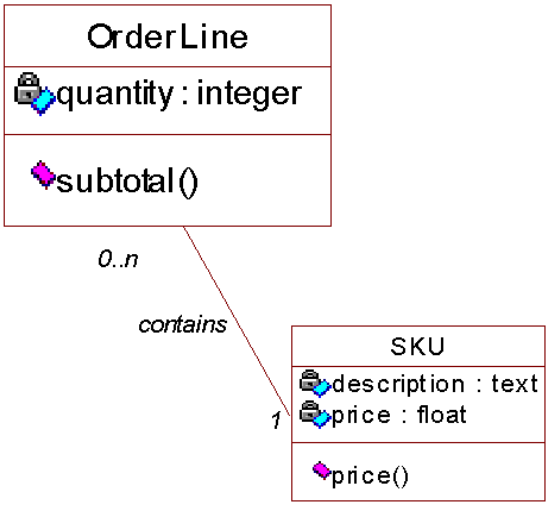

Figure 106 - The aggregation of Order Lines and SKU's

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;
}
```

Figure 107 - 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:

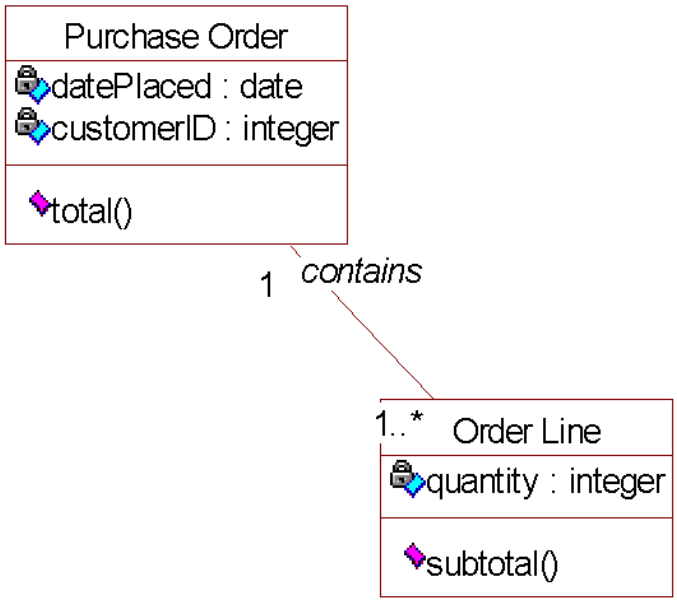

Figure 108 - A Purchase Order holds a list of Order Lines

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;
}
```

Figure 109 - 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:

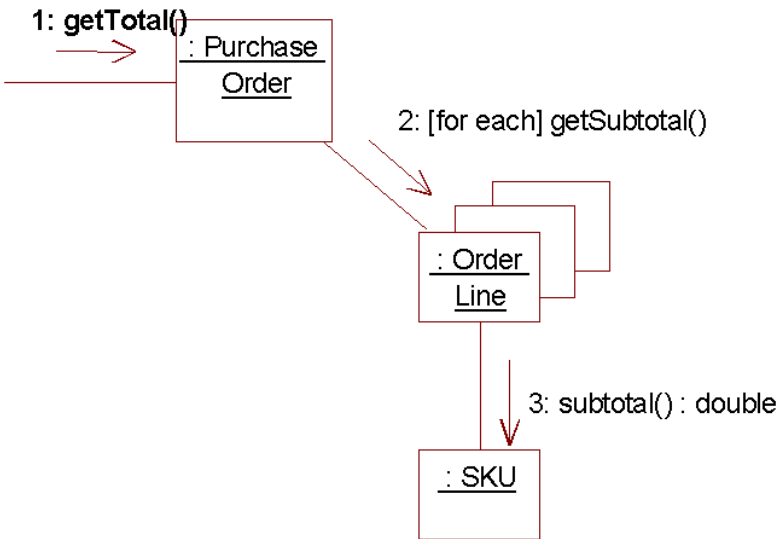

Figure 110 - "Get total" collaboration

Step 1

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

```
public double getTotal()
{

}
```

Figure 111 - 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;
}
```

Figure 112 - 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();
}
```

Figure 113 - 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.

Sadly, the syntax for package protection in Java is rather poor. The notation is to simply declare a class with no **public, protected** or **private** preceding the class definition - exactly as in Figure 114.

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:

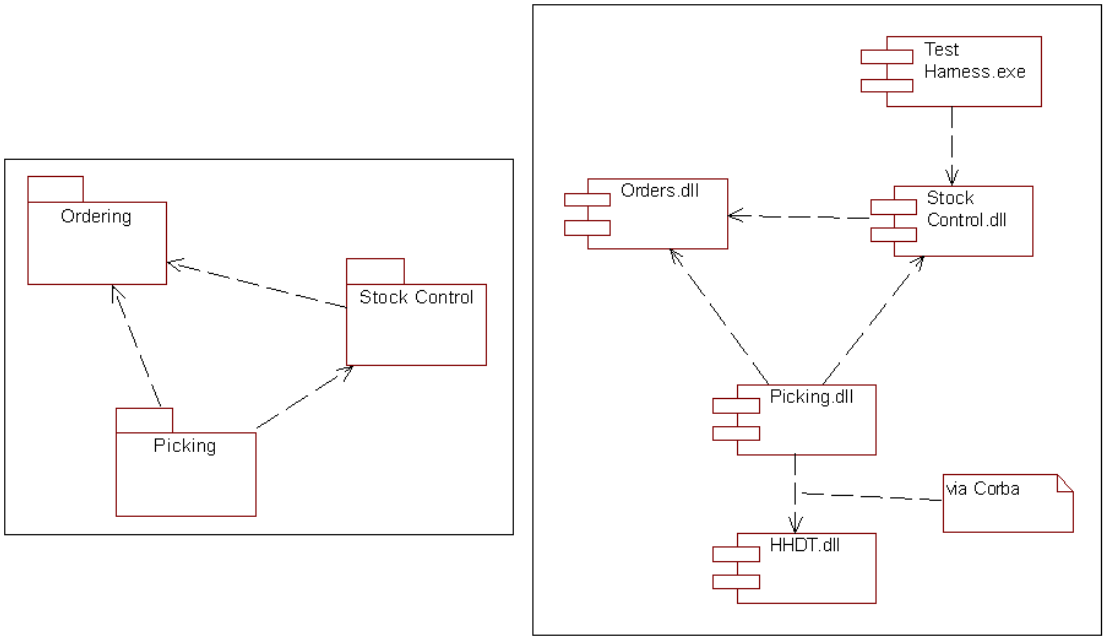

Figure 115 - the logical compared to the physical view

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.

Ada Components

Some extra component icons are available through Rational Rose that seem to be heavily influenced by the Ada language (presumably through the input of Grady Booch). These icons work in exactly the same way as the components above, but notate more specific software components:

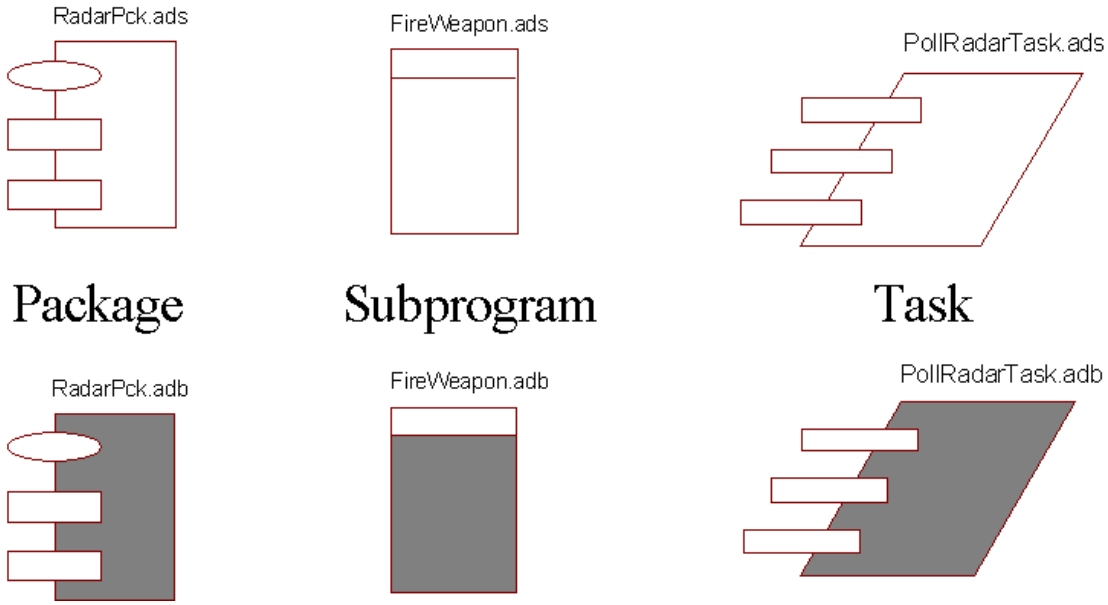

Figure 116 - Extra Components

For readers from a non Ada background, a Package (not to be confused with a UML package) is a collection of related procedures, functions and data (roughly the same as a class), a Subprogram is a procedure or function, and a Task is a subprogram that can run concurrently with other tasks.

These symbols may be of use to you even if you are not working in Ada - in particular the Task symbol is useful to denote that the software element is going to run in parallel with other tasks.

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.

Bibliography

[1] : Krutchten, Philippe. 2000 *The Rational Unified Process An Introduction Second Edition* Addison-Wesley

A brief introduction to the Rational Unified Process, and its relationship with the UML

[2] : Larman, Craig. 1998 *Applying UML and Patterns An Introduction to Object Oriented Analysis and Design* Prentice Hall

An excellent introduction to the UML, applied to real software development. Used as the basis for this course.

[3] : Schmuller, Joseph. 1999 *Teach Yourself UML in 24 Hours* Sams

A surprisingly comprehensive introduction to UML, including details of the metamodel. The first half concentrates on UML syntax, and the second half applies the UML (using a RUP-style process called GRAPPLE)

[4] : Collins, Tony. 1998 *Crash : Learning from the World's Worst Computer Disasters* Simon&Schuster

An entertaining collection of case studies exploring why so many software development projects fail

[5] : Kruchten, Phillipe 2000 *From Waterfall to Iterative Lifecycle - a tough transition for project managers* Rational Software Whitepaper – www.rational.com

An excellent, and short, description of the problems project managers will face on an iterative project

[6] : Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995 *Design Patterns : Elements of Reusable Object Oriented Software* Addison-Wesley

The classic "Gang of Four" catalogue of several design patterns

[7] : Riel, Arthur 1996 *Object Oriented Design Heuristics* Addison-Wesley

Rules of Thumb for Object Oriented Designers

[8] : UML Distilled

Martin Fowler's pragmatic approach to applying UML on real software developments

[9] : Kulak, D., Guiney, E. 2000 *Use Cases : Requirements in Context* Addison-Wesley

An in depth treatment of requirement engineering, driven by Use Cases