UML for Managers

Jason Gorman

Chapter 2

February 11, 2005

Introducing the UML

In this chapter we will look at the core UML diagrams and discuss their potential uses. Readers should bear in mind that this chapter is not an attempt to teach you UML. Rather, it is designed to familiarize you with the diagrams and to appreciate what can be modeled using them.

In later chapters, we will look at applications of UML, and you will see how different diagrams can be applied to different kinds of business and software problems.

Object Diagrams

An object diagram models the objects in a system at some specific point in time. The term *snapshot* is sometimes applied to object diagrams, because they can be thought of as being like photographs that show exactly what things were like when the snapshot was taken.

Programmers can best relate to object diagrams when we compare them to the data in their software when they pause execution of the running code, using what we call "breakpoints".

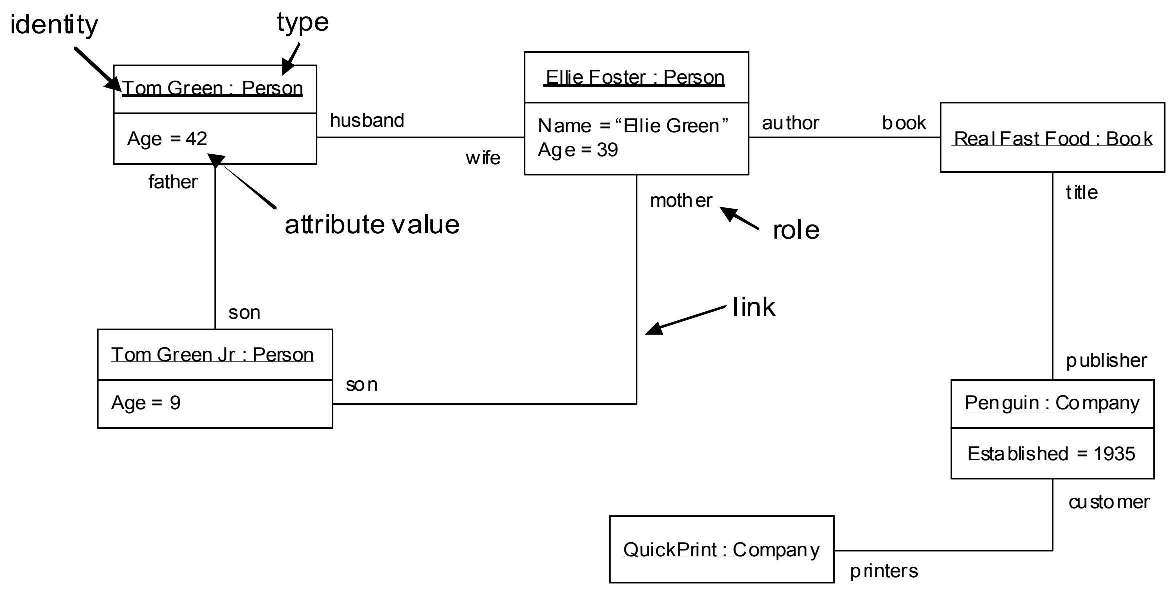

Fig 2.1. An object diagram models objects, their attribute values and the links between them at some specific point in time.

In analysis, we can use sequences of object diagrams to show how actions change the state of the system – by creating new objects, changing their attribute values, inserted them into relationships with other objects and so on. These sequences of snapshots are often referred to as filmstrips, and they are a very helpful way of visualizing what's happening during the execution of some scenario – be it a business scenario or a software usage scenario.

Object diagrams are woefully underutilized in many projects. You will find some very reputable books on UML that just skim over them – a misguided omission, in my humble opinion.

A key thing to note about object diagrams is that the identity and the object type appear in the top rectangle – we will call them compartments from now on – and are underlined. This is to distinguish them from classes, which we will cover next.

Influential authors and practitioners of in object oriented analysis and design methods recommend starting with objects – specific examples or instances – before attempting to create more abstract or general models. Alan Cameron Wills, co-author of the Catalysis methodology, suggests that people are better at solving problems "bottom-up", starting with examples and working towards a theory that explains them. This is a good scientific approach to building models. In other scientific disciplines, we make observations and collect data (our snapshots and filmstrips). Then we formulate a theory – a model – that explains the data. Then we test our model by making predictions and collecting more data. This approach is rarely, if ever, applied to analysis and design. Rather than collecting a sample selection of invoices, for example, and then modeling each example before drawing a class diagram that explains all of them, analysts often jump straight to a class model. They rarely go back and use snapshots to test their class models using real data. It's not uncommon for our system models to fall apart when we try to enter real business information into them.

The important things to remember about object diagrams are:

- They are good for "collecting data" on a business problem
- They are good for understanding how actions effect objects
- They are good for testing our models – they are the UML equivalent of test data

Class Diagrams

Class diagrams model the types of objects in a system, the relationships allowed between types and the types of values their attributes are allowed to have. It is important to understand the relationship between classes and objects. Objects are instances – specific examples – of classes. Every object of a particular type – or class – must obey the rules set out for that type. So, every object diagram must obey the rules set out in the corresponding class diagram(s). If it does not, then either your object diagram is wrong, or your class model is wrong - usually the latter.

Object oriented software is built by programmers writing classes, and then creating objects from those classes that will collaborate with each other to do the work required.

The essence of designing object oriented software is in identifying the classes we will need to write in our code and assigning responsibility for certain actions to each class. We will see how this is done later.

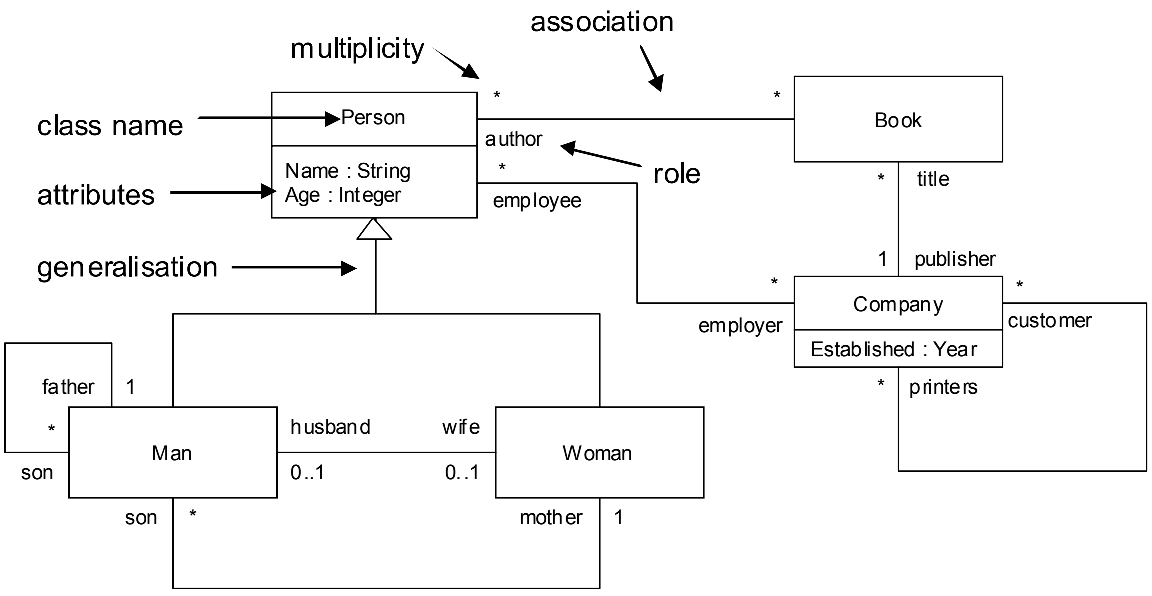

Fig 2.2. A class diagram describes the types of objects in a system, the attribute values allowed for that type, the responsibilities of each type of object, and the relationships allowed between objects of each type.

There are two kinds of relationship between types shown in this example – *associations* and *generalisations*.

An association describes the links that are allowed between objects of different types. At one end of an association, we can specify the role that objects at that end will play, and the multiplicity – the number of objects allowed to play that role at the same time with respect to the same object.

A generalisation relationship tells us that one class is a subset – we call it a *subclass* or *subtype* – of another. For example, Man is a subtype of Person, and so is Woman. Person generalises Man and Woman. Conversely, Man and Woman both specialise Person.

People who are familiar with relational databases might find it useful to think of objects as rows in a table, and classes or types as the table schemas. Attributes relate to columns in the table, and identity relates to a primary key. The analogy is not precise, and readers should be warned that there are marked differences between relational and object technology!

Activity Diagrams

An activity diagram models the flow of actions in some process or workflow. It could be a business process, or it could be the control flow of program code.

An activity diagram shows sequences of *activity states* – "actions" to you and me – where when one action is complete the flow immediately moves on to the next action. This automatic transitioning from one activity state to the next is what distinguishes activity diagrams from their close cousin, state transition diagrams. In state transition diagrams, transitions from one state to another occur as the result of events, and don't happen automatically.

It is perfectly legal in UML to include state transition elements in an activity diagram – for example, to show how, after completing a sequence of actions, a system waits for user input before moving on to the next step in the flow.

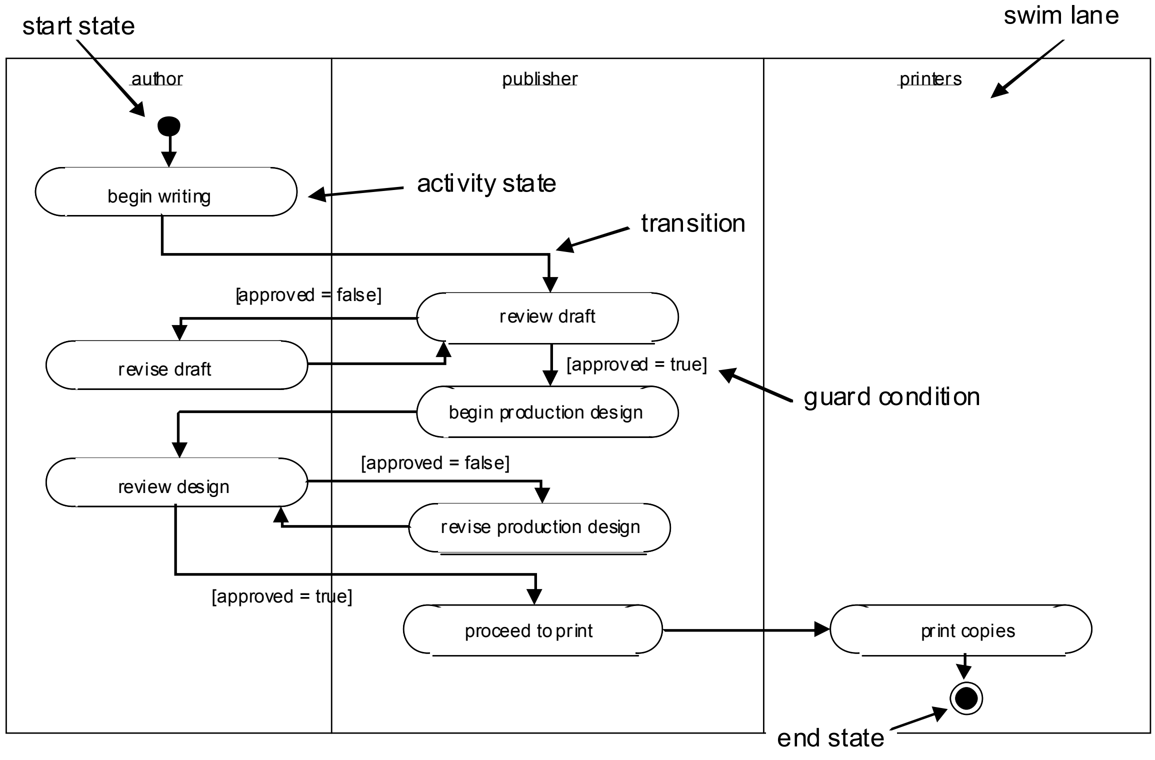

Fig. 2.3. An activity diagram describes the flow of a business process or program code.

Some of the key elements of activity diagrams are:

- Activity states – effectively actions where the completion of one causes a transition automatically to the next step in the process
- Transitions – a move from one state to another
- Guard conditions – rules that tell us under what circumstances a transition will occur
- Start state – the initial state of the system at the beginning of the workflow. Many workflows have pre-conditions – that is, things that must be true before that workflow can begin.
- End state – or "exit point". A workflow may have multiple exit points, each relating to a specific post-condition (things that must be true when the process is complete).
- Swim lanes – optionally we can show how different objects take responsibility for performing certain actions.

State Transition Diagrams

In contrast to activity diagrams, state transition diagrams show how events cause transitions in objects of a specific type from one discrete state – a stage in their lifecycle – to another.

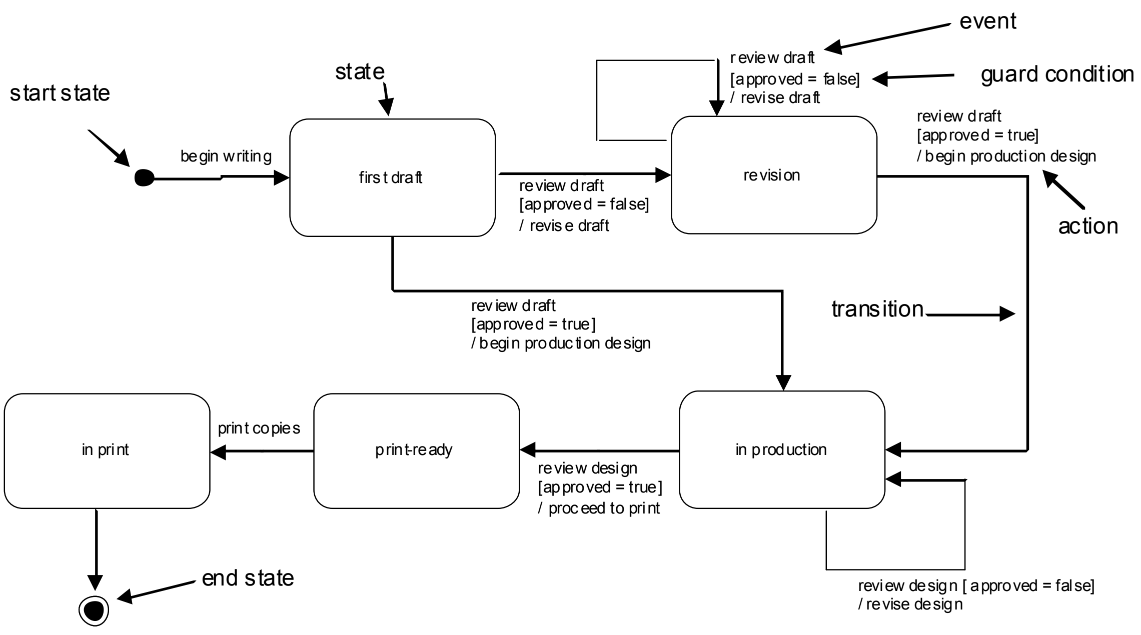
event
guard condition
start state
state
review draft [approved = false] / revise draft
review draft [approved = true] / begin production design
action
begin writing
first draft
revision
review draft [approved = false] / revise draft
review draft [approved = true] / begin production design
transition
in print
print copies
print-ready
in production
review design [approved = true] / proceed to print
end state
review design [approved = false] / revise design

Fig 2.4. State Transition diagrams model object lifecycles and event-driven processes

They are especially useful for describing objects that have a distinct lifecycle, or systems that are especially event-driven – like user interface components. The key elements of state transition diagrams are:

- States – discrete stages in the lifecycle of an object (eg, a bank account can be *in credit* or *overdrawn*)
- Transitions – moves from one state to another
- Events – that cause transitions
- Guard conditions – rules that tell us under what circumstances a specific transition will occur
- Actions – that are executed as a result of a transition (eg, sending out a letter when a bank account goes overdrawn)
- Start & end states – pre and post-conditions for the beginning/end of the life of an object

As it is with object diagrams, many practitioners are unfamiliar with state transition diagrams and avoid their use. This is partly because they are on of the tougher aspects of UML to master. Their use is most prevalent in the specification of real-time software – for example, manufacturing control systems - where event-driven logic is especially important.

It is possible to specify entire systems using only state transition and class diagrams, and this approach has been adopted by a small minority of modeling tool developers to enable the creation of executable UML models – though the use of such techniques is highly specialized at present.

Sequence Diagrams

The essence of object oriented design is to decide how objects will interact with each other, each doing some useful piece of work, to complete a useful task. Objects interact by sending messages to each other through well-defined interfaces that allow one object to request the services of another.

In object oriented analysis and design, after we have identified the objects in the system and used filmstrips to identify the changes that occur to these objects during the execution of some scenario, we assign responsibility for these changes to different objects.

A sequence diagram is therefore very useful for showing which objects are doing what, as well as for defining the public interfaces that the objects will need to "publish" so that other objects can request those services.

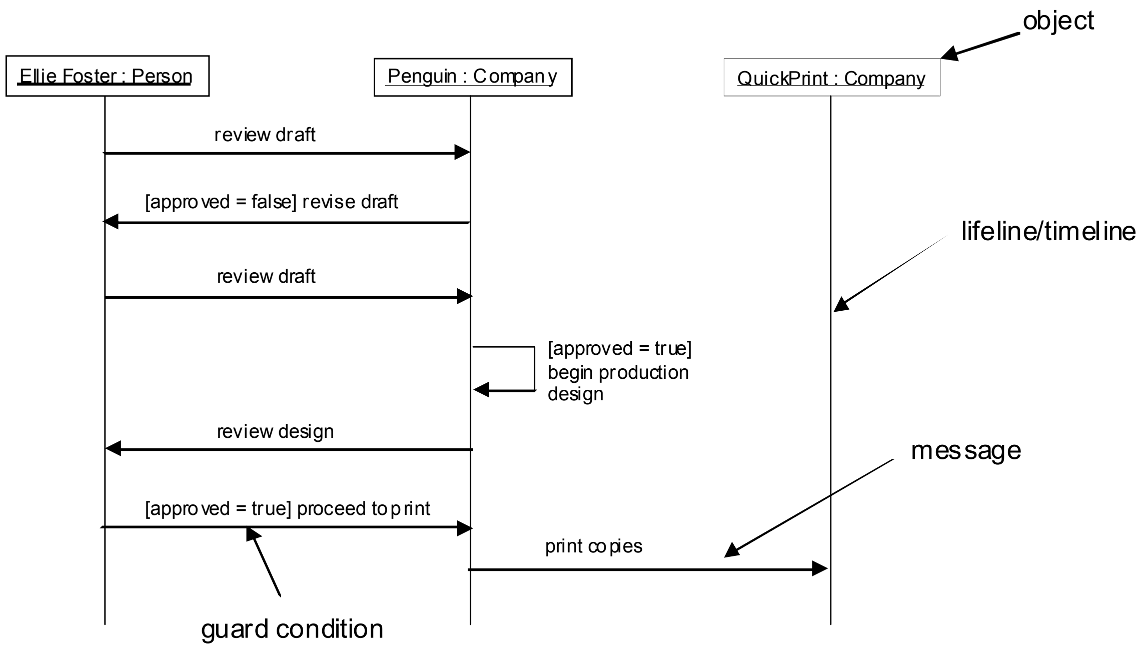

Fig 2.5. Sequence diagrams show how objects interact over time to achieve some goal in a specific scenario

The key elements of sequence diagrams are:

- Objects – draw at the top or the side of the diagram
- Lifelines (or "timelines") – extend downwards or horizontally from the objects to indicate the lifetime of that object
- Messages – arrows going from one object's lifeline to another describe how that object requests the services of the other

- Guard conditions – show that some messages are send only under certain conditions

There is a close relationship between sequence diagrams and object diagrams. It is often a good idea to identify the objects involved in an interaction sequence using an object diagram, which would confirm that objects sending messages to each other are able to do via a link that exists between those objects.

If you were to draw snapshots of the objects system at specific points in the timeline, then you would end up with a filmstrip. It is better to use sequence diagrams and object diagrams and filmstrips together to build a clearer understanding of what is changing, and which objects are performing those changes.

Once we know the objects involved, their attributes, their types, the relationships between them, their responsibilities and the interfaces they must present, then we can draw a class diagram that describes all of this information.

Collaboration Diagrams

A collaborations diagram is another kid of interaction diagram. It shows how messages are sent between objects in the execution of some scenario, but it adds those messages to an object diagram to combine interactions with structure.

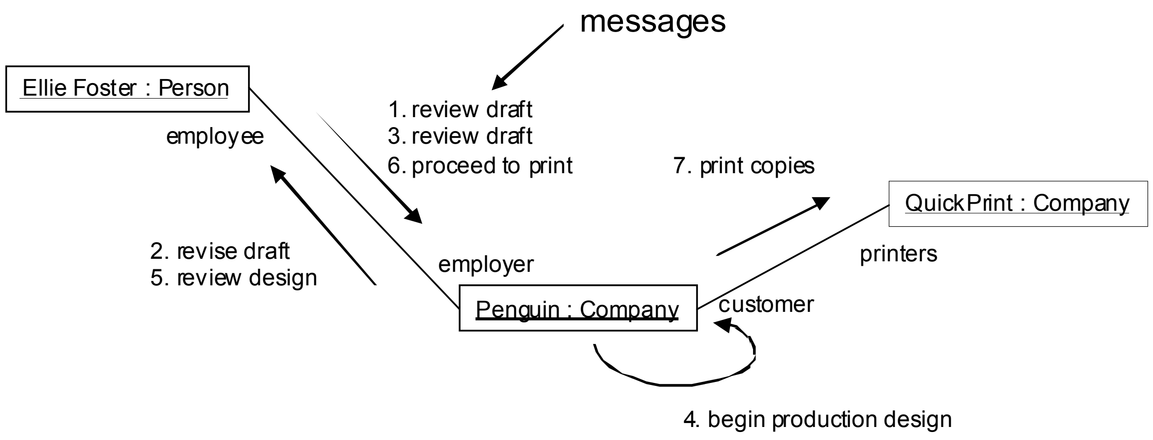

Fig 2.6. A collaboration diagram describing the same interaction as the previous sequence diagram

Because it is harder to read the sequence of interactions in a collaboration diagram, they are far less commonly used in analysis and design. They can be useful, however, for validating that a sequence of interactions is supported by the appropriate links between objects.

Model Constraints & the Object Constraint Language

Often we will want to communicate extra information about the rules of a system that UML diagrams, by themselves, can't describe. A *constraint* is just a rule that applies to some element – a class, for example – in the model. The element to which the constraint applies is said to be the *context* of the constraint. It is important to remember that every constraint/rule must apply to one specific context. Many analysts make the mistake of writing rules in isolation from their models, and making it ambiguous as to which parts of the model the rules apply.

There is no rule in UML about what language we must use to write model constraints. Many people just write them in plain English (or another natural language). The problem with constraints written in natural languages is that they are often ambiguous. We use our considerable brainpower to interpret English, filling in gaps and making assumptions based on a huge cushion of knowledge and experience. Even with our considerable capacity to understand ambiguous sentences, we often make mistakes and draw the wrong conclusion.

If the aim is to write software in a programming language like Java or C#, then there can be no room for such ambiguity. To write unambiguous constraints, we need to write them in a formal language – that is, a language that is precise and that cannot be misinterpreted. The UML comes equipped with a formal language called the Object Constraint Language. OCL allows us to write precise, unambiguous expressions that apply to a UML model. The best way to think of OCL is as an object oriented query language – a bit like an OO version of Structured Query Language (SQL). Just as we can use SQL to write constraints that apply to database tables, we can use OCL to write constraints that apply to types of objects.

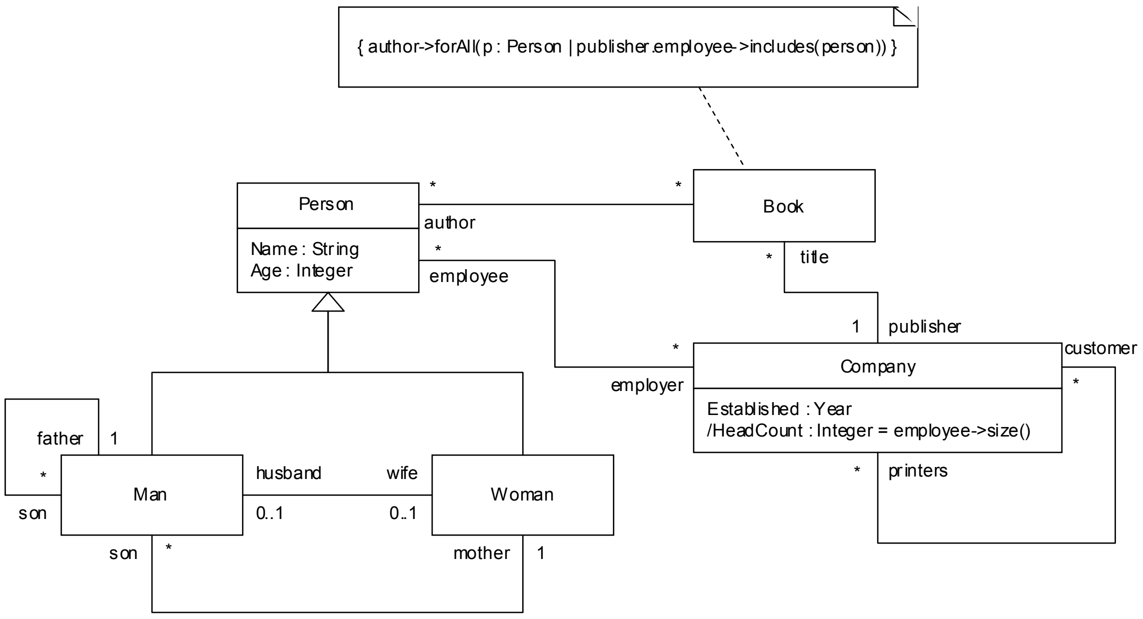

Fig 2.7. A class diagram with an OCL constraint that applies to every object of type Book

OCL has something of a reputation in the software development community. Developers are often hostile to formal specification, claiming – justifiably – that with that level of precision it would be quicker to just write the code. Where OCL is extremely useful is in building in precision when we're modeling at a much higher level of abstraction that executable program code. It's a strange irony that the people who would find it most useful, business and requirements analysts, often find it the hardest to learn. Programmers usually have no problem picking up the language, because they are familiar with OO programming concepts.

OCL figures highly in what's being called Model-driven Architecture. MDA is a process by which executable systems are specified at a higher level of abstraction entirely in UML. The resulting software is generated automatically from this Platform-Independent Model (PIM). We look more at MDA in a later chapter.

The only thing you need to know at this stage is that OCL is more useful the higher the level of abstraction, and that MDA isn't a reality yet, so it has limited uses for actual software development. You should also bear in mind that, even though it is relatively small and easy to learn for programmers, most software developers don't know it.

Component & Deployment Diagrams

Sometimes it's necessary to model the physical architecture of a system – the files and other components that make up the system, the machines on which they are deployed, and the means by which machines and components communicate.

UML has special notations for representing physical components, their interfaces, the dependencies between them, and their deployment architectures. It's worth nothing, though, that at this low level of abstraction the benefits of modeling start to become questionable.

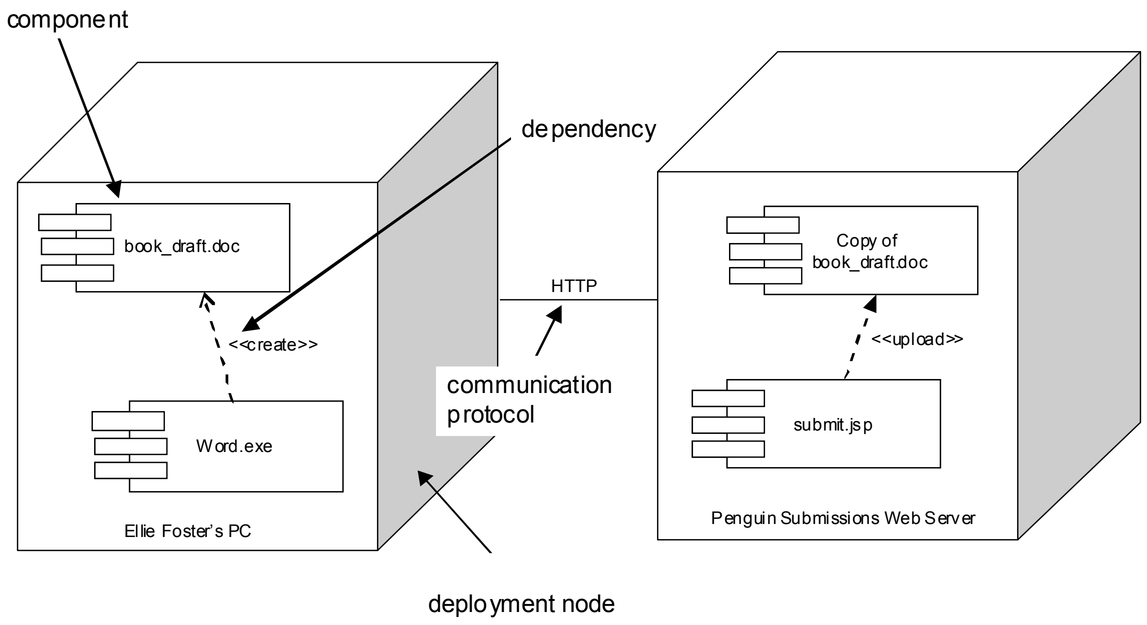

Fig 2.8. Component & deployment diagrams describe the physical architecture of a system

Use Case Diagrams

A *use case* describes a way in which the system will be used to achieve some functional goal. Use case diagrams allow us to model the functional goals of the system and to relate those goals to classes of user – called *actors* – so that we can see who should be able to do what using the system.

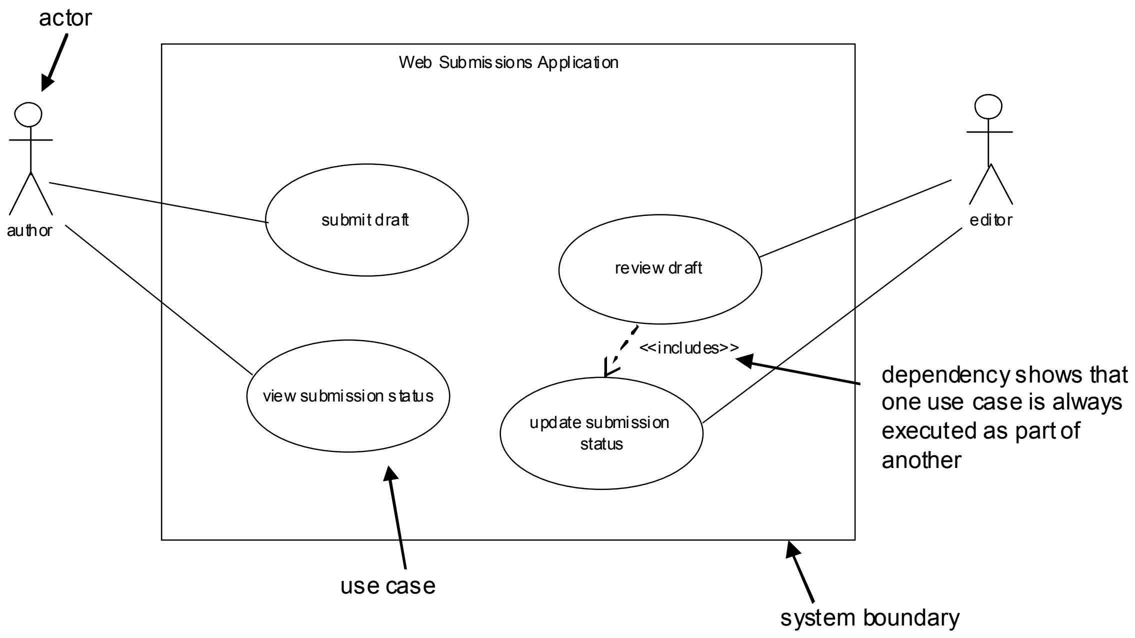

Fig 2.9. Use case diagrams show the different classes of user and the goals they can achieve using the system

Some software development processes are said to be use case-driven, in so much as the development process breaks work down into use cases and use case scenarios (specific paths through the flow of a use case). In the Unified Software Development Process (USDP, or just "Unified Process"), many software development activities are driven by our understanding of the use cases. We will look at the Unified Process in a later chapter.

The key elements of use case diagrams are:

- Use cases – modelling the functional goals of the system
- Actors – classes of user
- Communication – between actors and use cases, showing which use cases are instigated by which actors (often referred to as *functional entitlement*)
- Dependencies – between use cases that describe how the flow of one use case might rely on the flow of another (eg, *proceeding to the checkout* on an E-Commerce web site might require you to *log in* if you're not already)

Use case diagrams are probably the easiest part of the UML to understand for technical and non-technical project stakeholders, and are widely used in requirements analysis.

Packages & Model Management

Just as we can group files on our computers into folders to make them easier to find, we can break large models down into packages that group related model elements together.

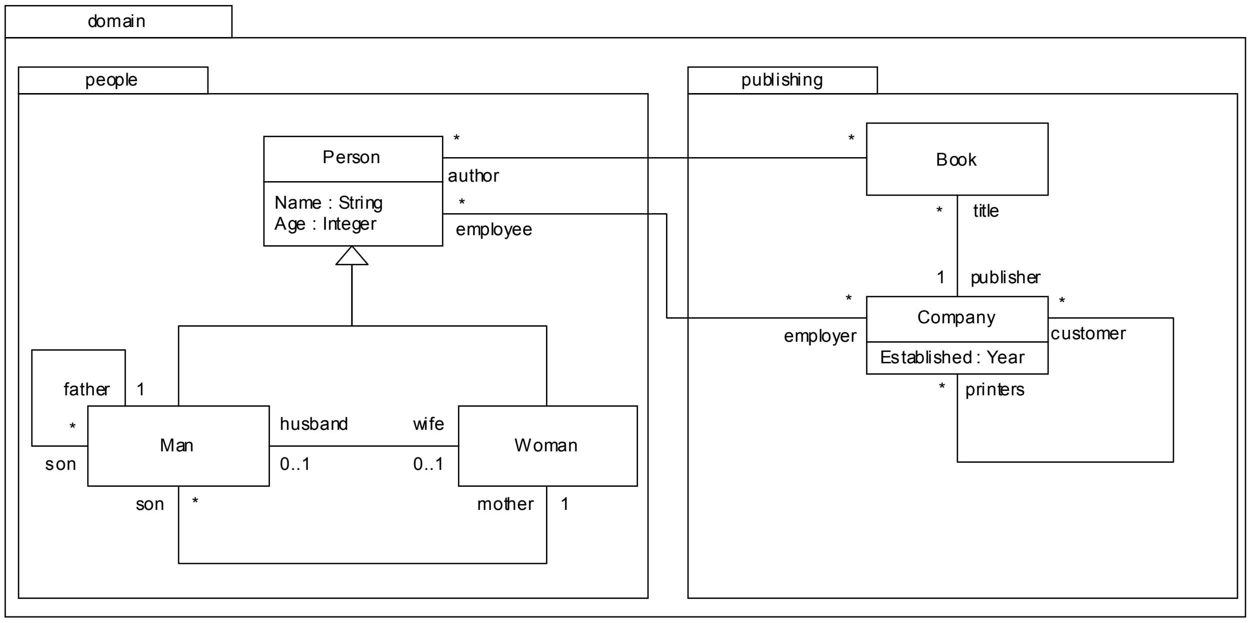

Fig 2.10. Packages can be used to group related model elements together, making larger models easier to work with

Extending the UML

Sometimes it's necessary to convey more information than vanilla UML is able to describe. We have already seen one mechanism for adding extra information to our models – using *constraints*. The UML standard provides two other mechanisms for extending the language: *stereotypes* and *tagged* values.

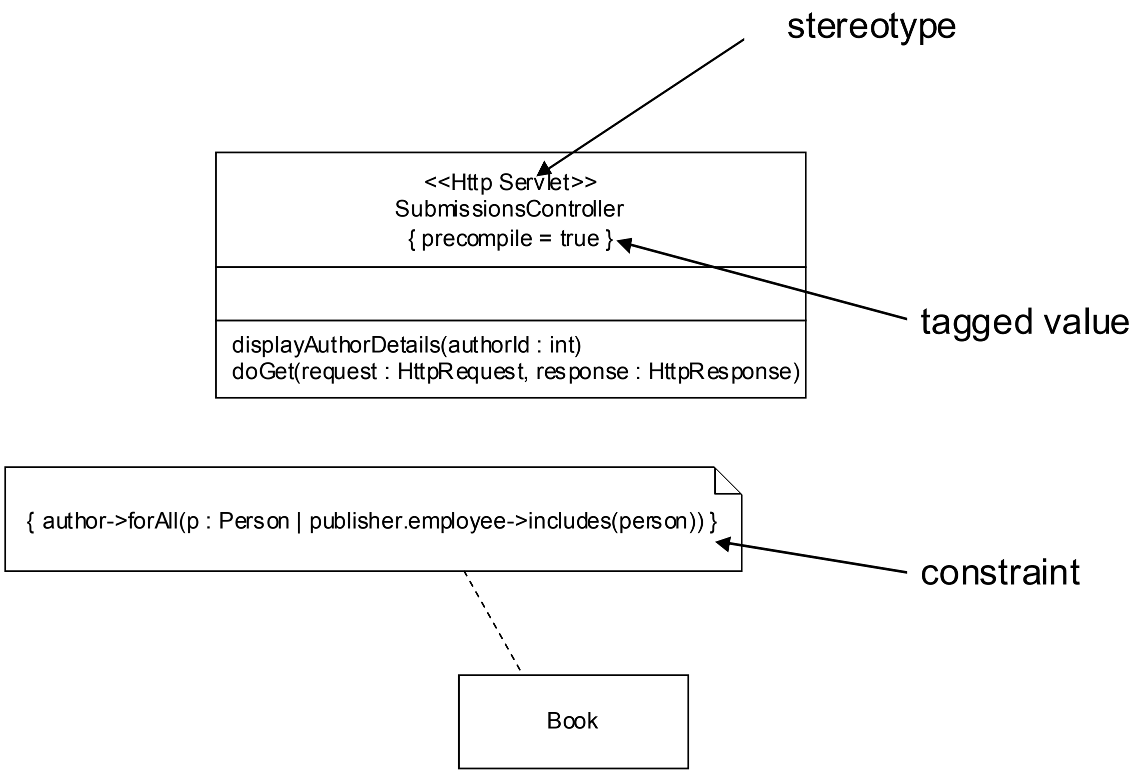
stereotype
<<Http Servlet>>
SubmissionsController
{ precompile = true }
displayAuthorDetails(authorId : int)
doGet(request : HttpRequest, response : HttpResponse)
tagged value
{ author->forAll(p : Person | publisher.employee->includes(person)) }
constraint
Book

Fig 2.11. We can extend our models using stereotypes, tagged values and constraints

A stereotype can be applied to pretty much any model element to tell us what kind of thing it is – over and above what the model already tells us. For example, we might want to show that a class should be implemented as a Java servlet. We can apply the stereotype <<Http Servlet>> to that class.

We can also add extra information using pairs of names and values called "tagged values". For example, we can communicate that our Http Servlet should be precompiled when it is deployed using the tagged value *precompile = true.*

The two examples we've seen come from what is called a UML profile – that is, a related set of stereotypes, tagged values and constraints that apply to them, that allow us to model a specific type of problem or solution.

Optionally, UML profiles can add customized icons that are used in place of the standard UML notations, to make models easier to read. UML profiles exist for modeling Java Enterprise architectures, relational database schemas, and a host of other technologies. Profiles also exist for business modeling.

We will look more closely at the applications of UML profiles in later chapters.