UML for Managers

Jason Gorman

Chapter 3

February 24, 2005

Applying UML

UML is, at its most fundamental, an *information systems* modeling language. An information system doesn't necessarily have to be implemented as computer software, since many computing concepts have their origins in real world abstractions. A filing system for patient records in a hospital is an information system, even if those files are actually physical pieces of card and paper. The rules for storing, retrieving and updating patient records remain the same whether we keep them in filing cabinets or store them in an Oracle database.

In the next two chapters, we'll look at practical uses for UML in both software development and business architecture. Hopefully, you will learn that UML has wider applications than some people give it credit for.

Object Oriented Analysis & Design

The most common use of UML is in the analysis and design of software solutions. Many variations exist on the OO analysis and design process (OOA/D), but they all have the same goals in common:

1. Identify key *usage scenarios* for the system (usually from the perspective of different kinds of user – ie, who will be using the software, and what will they be using it to do?
2. Describe the *interactions* between the user and the system in the execution of a usage scenario – ie, what actions does the user perform and how does the system respond to those actions (*outcomes*)?
3. Assign *responsibility* for the outcomes to the most appropriate objects in the system

It really is as simple as that. Once you know what the objects involved are, and what each object is doing in the execution of that scenario, you have enough information to start implementing the code needed to satisfy the scenario.

Many project teams make the mistake of trying to the analysis and design for all the system usage scenarios before and coding begins. This if often referred to as "big design up front" or BDUF. The problem with BDUF is that in any product design process there is a chance you will get it wrong. Without meaningful testing and feedback, your ultimate design is likely to be very flawed. One myth about UML and OOA/D is that you need a complete design in order to write code. You don't. You just need enough of the design to get you started. So, designing and implementing software one usage scenario at a time, so that users can provide feedback on the end product, is a smart way of fleshing out a better design.

So, how do we use UML in our simple analysis and design process? Well, first of all we need to model the types of users and the things they will be using the software to do.

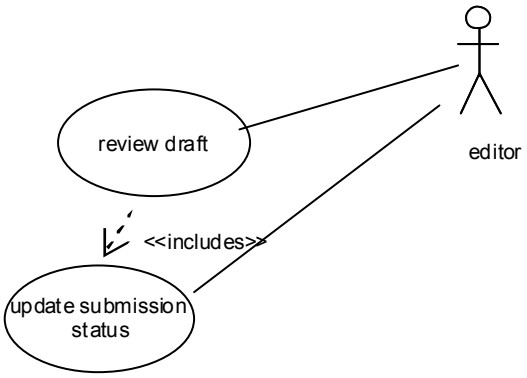

Fig 3.1. A use case diagram can be used to show the types of users and their functional goals for the system

In software projects, there is always a risk that there won't be time or money to finish every feature before the deadline. It's important that the "customer" on that project effectively prioritises usage scenarios – which we will call *use case scenarios* from now on – so that the most important functionality is tackled earlier in the schedule. That way, if there's only time to complete 80% of the functionality, at least you will have completed the most important 80%.

The analysis and design process works through the use case scenarios one at a time, starting with the most important.

To progress, we must now describe how the user and the system interact to achieve the functional goal of that scenario. It's a good idea to avoid committing to a specific kind of user interface this early in the process, as it ties us to a specific solution and can make it difficult to understand the business logic of the interactions.

An *essential use case* scenario describes the interactions in abstract terms, and makes it clear who is doing what in the flow of that scenario.

Use Case: Review Draft (includes Update Submission Status)
Goal: To determine if a book draft is ready for production design or if it needs to be revised.
Actor(s): editor

Scenario: Review and accept draft

editor	system
Select author from author's list	
Request to view author's details	
	Display author's details – including author's first name, surname and a list of submissions from that author
Select a submission – which is a draft manuscript – and request to review it	
	Display selected draft manuscript
Review submission	
Request to accept submission	

	Update submission status as In Production and notify production designer

It's vital to remember that when we describe the flow of a use case scenario, we are actually describing the *interaction design* of the software. Many people make the mistake of treating these descriptions as requirements, and feel un-empowered to change them when necessary, leading to some pretty unusable systems. Use case flows are not requirements. The functional goals of use case scenarios are the actual requirements. How users interact with the software to achieve those goals is the beginnings of the software design.

Another important thing to remember about use cases is that they should relate directly to the business processes in which they will be used. This requires us to think very carefully about the start and end points of use cases, and to ensure that the goals of a use case relate directly to the flow of business processes. All too often we see use case goals like "To log in to the system" or "To view a list of customers". Neither of these has any intrinsic business value. Why do we need to log in to the system? Why do we need to view a list of customers? Analysts commonly make the mistake of identifying the use cases before they've identified the goals. But, since the purpose of a use case is to satisfy a useful goal, this is putting the cart before the horse. In these circumstances, you must question whether the functional goals – and therefore the business case – for the software you're building is well-defined. If you're not satisfied, consider revisiting the initial business analysis yourself and try to get a better feel for who will be using the software and what they will be using it to do.

Once we've decided how the users and the system will interact, we can now model the outcomes of those interactions using a sequence of object diagrams called a *filmstrip*. A filmstrip shows how user actions effect the objects in the system, and allow us to describe all of the outcomes of an interaction in simple object-oriented terms.

First, we need to identify the interactions, which in this case are:

1. Select author
2. Request to view authors details
3. Select submission
4. Request to accept to submission

For each of these interactions, we draw two object diagrams: one showing the state of the objects in the system *before* the interaction, and the other showing the state of the system *after* the interaction. We highlight in bold any changes between the two snapshots.

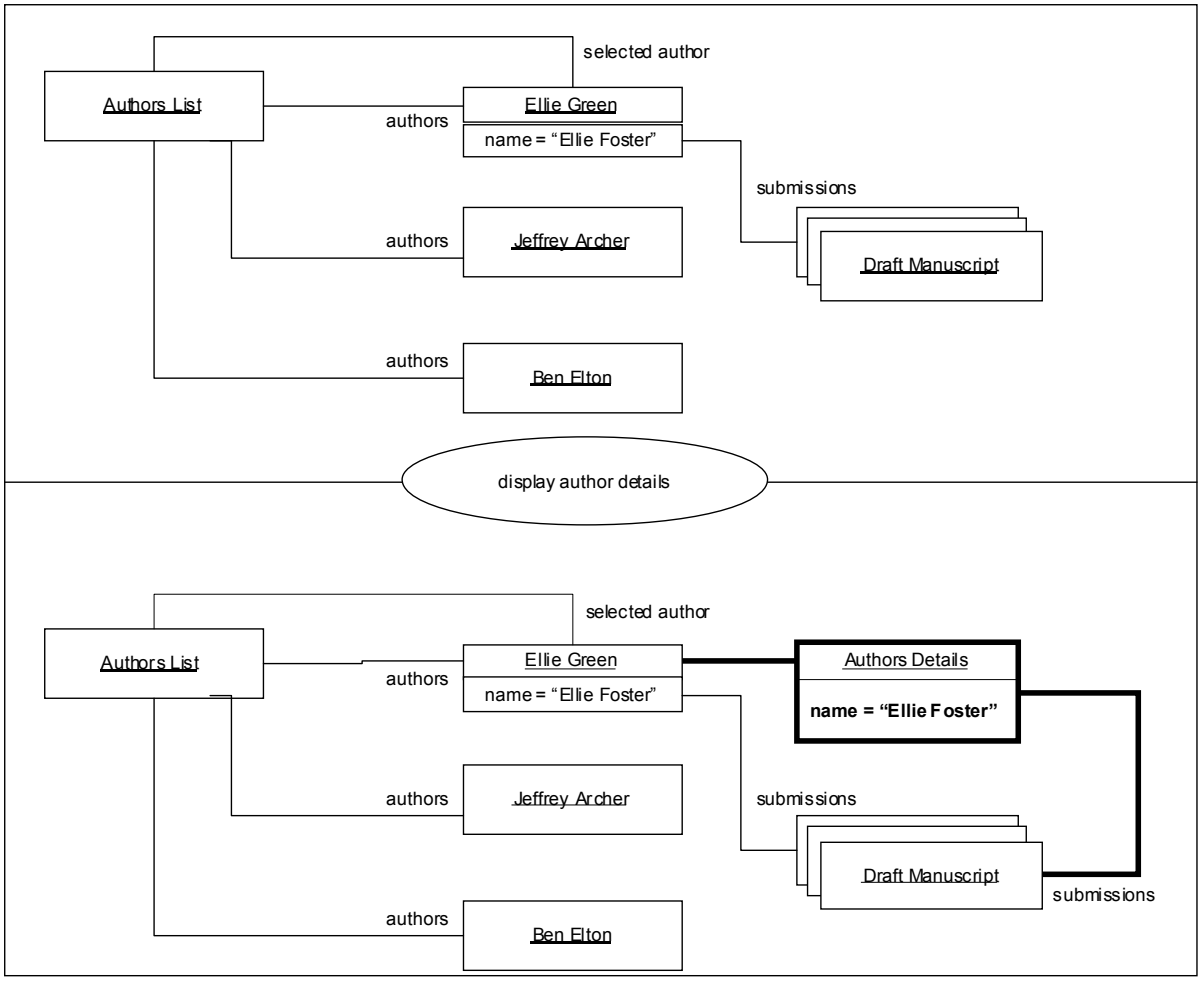

Fig 2.2. Pairs of object diagrams called "filmstrips" can be used to show the changes that happen to objects in the system as a result of user interactions

Anything that happens in an object oriented system is executed by an object. Everything we want our software to achieve must be assigned to an object in order for it to happen at all.

The essence of object oriented design is deciding which objects should be doing which bits of the overall logic of the system.

Our filmstrips identify most of the objects involved. They show two specific kinds of objects – views (or "boundary objects") that the user can interact with, and business domain objects (or "entities") that represent the business information in the system.

In high-level object oriented design, we traditionally concern ourselves with *three* kinds of objects to model the overall logic of a system. The third kind of object is a controller (or just "control"). A controller has the responsibility for responding to user actions – that is, user interactions with views.

In modern Windows software, these controllers are sometimes referred to as "event handlers", because they have operations that are executed in direct response to user input events.

A common *design pattern* in object oriented software such as Windows applications is *Model-View-Controller*. We split our application logic into these three distinct areas of responsibility to make it easier to change one without affecting the others. For example, we may wish to change the way a stock price is displayed on a view without changing the logic for retrieving the latest price and without changing the shape of the underlying business data. By cleanly separating display logic, control logic and business logic we can better ensure that our software will be easier to change – a major goal of good OO design.

So, in order to fully explain the logic of our interactions, we need to identify the views, the entities *and* the controllers that take part. We also need to verify that these objects will be able to 'speak" to each other, which will require them to have a relationship along which messages can be sent.

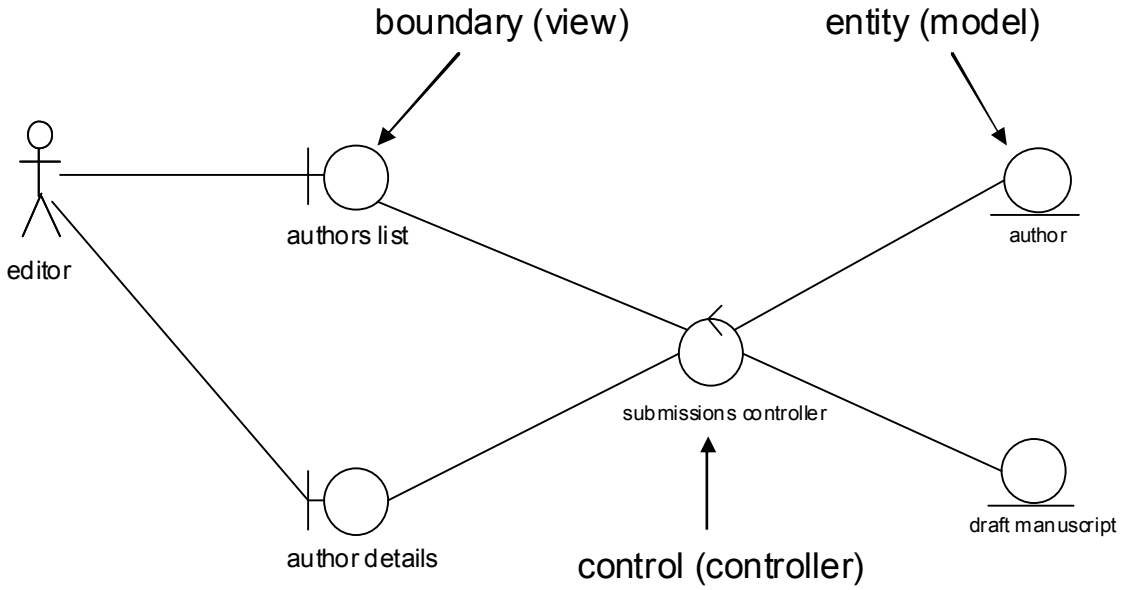

Fig. 3.4. A robustness diagram helps us to validate that we have identified all of the objects playing the roles of model, view and controller

Robustness analysis allows us to validate that we have all of the objects required to realize a use case scenario (or scenarios), that these objects have the necessary relationships to interact with each other, and that we haven't broken the rules of the model-view-controller design pattern, namely:

- Actors can interact with views
- Views can interact with actors
- Views can interact with controllers
- Controllers can interact with views and the model
- Actors cannot interact with controllers or the model
- Views cannot interact with the model
- The model cannot interact with views or actors
- Controllers cannot interacts with actors

If we have followed these rules, then our design is said to be *robust* – that is, it will be more open to change in the future.

In practice, we do see these rules being broken, but only in such a way that it does not create direct dependencies between objects that aren't supposed to know about each other.

Objects in the model can send messages to views by firing events – signals that are transmitted to objects that are *listening* for changes to the model. The model doesn't know who is listening; it only knows that every listener has an interface through which it can send the signal informing the listener(s) of a change in its data. Views listening for model changes can then update themselves as soon as they receive the signal.

This style of design is usually referred to as *implicit invocation*, because the object sending the message doesn't know explicitly which objects will be receiving it.

In this example, we are assuming *explicit invocation* – the views know which controllers they are talking to, and the same views are updated explicitly by the controller.

Once we have identified the objects involved in the scenario and have validated their relationships, we next need to assign responsibility for the changes we want to see happen to the right objects.

A sequence diagram is useful for modelling how objects interact with each other over time to achieve some functional goal. When one object sends a message to another object, the receiver has the responsibility for satisfying that request. So, when a message arrow is pointing towards an object's timeline that means that object has that particular responsibility.

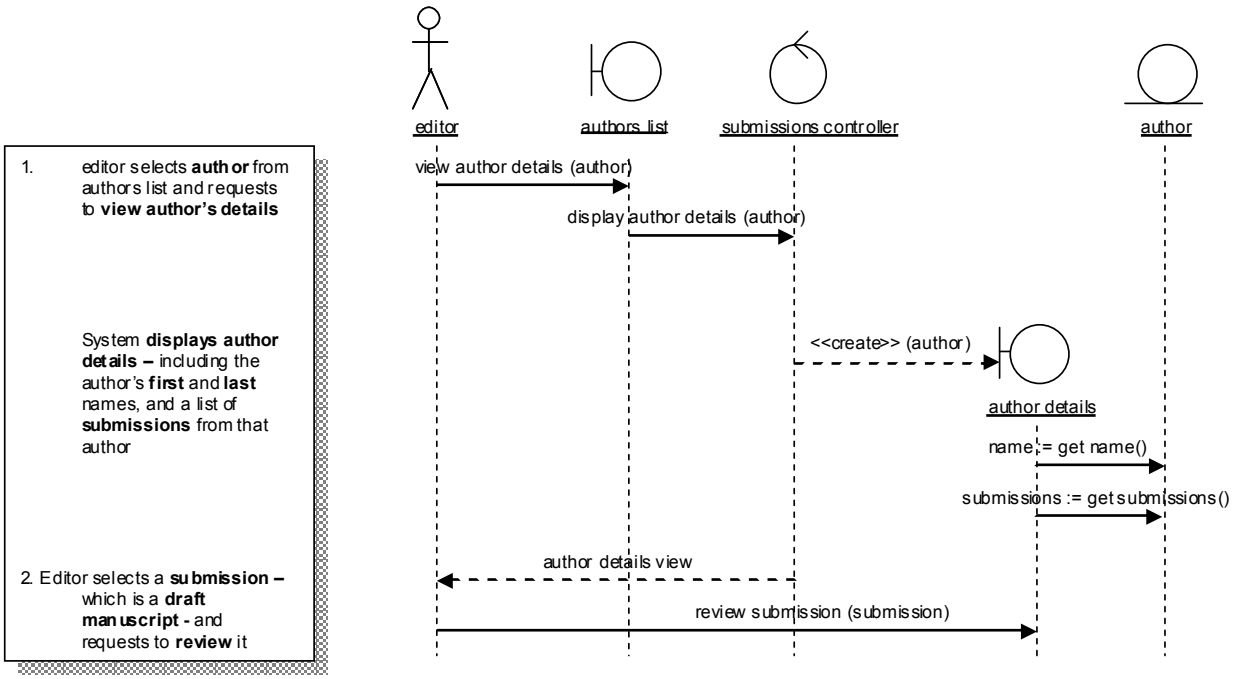

Fig 3.5. Sequence diagrams are useful for deciding which objects are taking responsibility for different parts of the overall work in a scenario

Assigning responsibility is a key part of the OO design process. There's no real science to it, but as a general rule you should aim to put the work as close to the data (attributes and relationships) as possible. A good OO design should produce code that is highly modular, and those modules – in the form of classes in object oriented software – need to be *highly cohesive*.

If objects were expected to change each others' data, that would require each object to know more about the other. The more objects know about each other, the harder is becomes to change one object's design without impacting the others that depend on that design.

You should aim to effectively *encapsulate* as much knowledge about an object inside that object – ie, behind a public interface that allows other objects to use its services without knowing how it does what they're asking it to do.

This is why it is a cardinal rule of good OO design that an object's attributes (its internal data) should never be publicly exposed. They should remain hidden behind public interfaces through which other objects can request to get or to change that data.

At this point in our simple analysis and design process, we know enough to begin thinking about the design of the code we're going to have to write to execute the scenario.

Object oriented programs are written using classes. We know enough about the objects involved, their attributes and relationships and the public interfaces they must support to draw a logical class diagram.

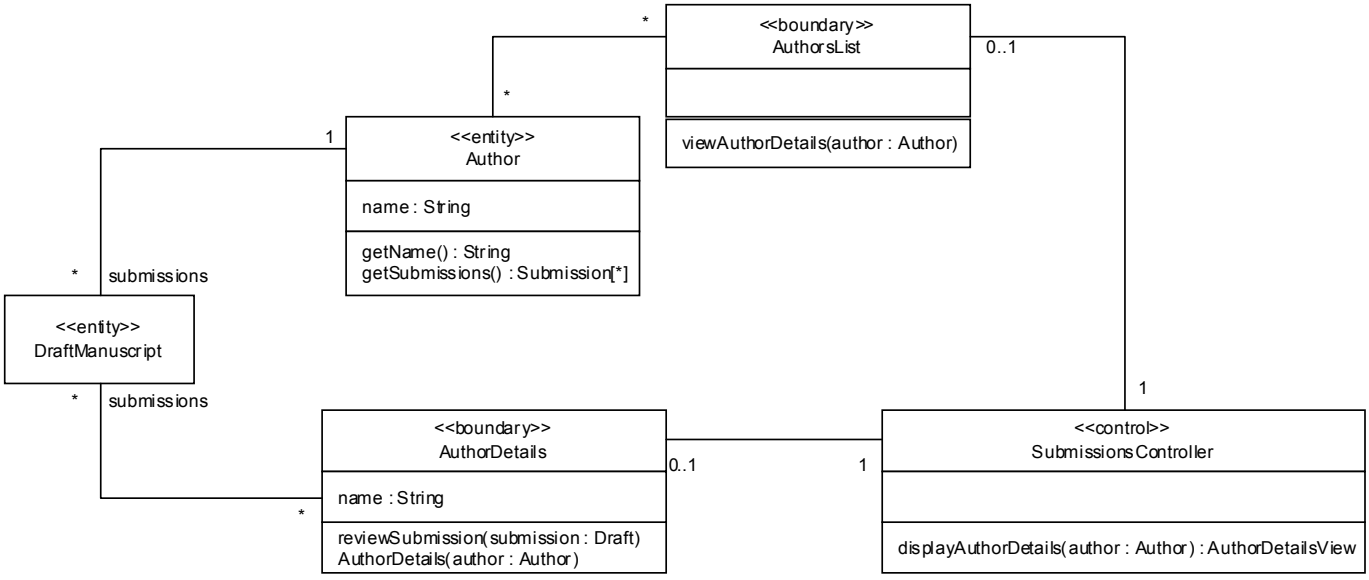

Fig 3.6. A class diagram helps us visualise the logical structure of the code we'll need to write

At this stage, we probably know enough to move on to writing the code. There is one piece of the puzzle missing, however, and it can be a good idea in the beginning of a project to use models to address it – *technical architecture*.

Each programming platform has its own frameworks for implementing the logical model-view-controller design we've created.

For example, in a Java Enterprise (J2EE) application, we need to decide what kind of Java components to use for views, controllers and the underlying data model. We might decide to use Java Server Faces as our views, Servlets as our controllers and EJB Entity Beans for our model. Or we might choose to use EJB Session Beans for the controllers and map our model onto a relational database using a tool like Hibernate.

What architecture you ultimately decide to use will depend on a variety of factors: the technology available, the performance your application needs to achieve, the skills and experience (and, frankly, the tastes) of your development team.

So we need to decide how our logical design will be translated into a platform-specific implementation design. Once those choices are agreed and understood by the team, there's probably no need to keep on doing this next step in the design process. They can just move straight on to writing code. But it can be a valuable exercise at the start of a project to communicate, validate and document your architecture choices.

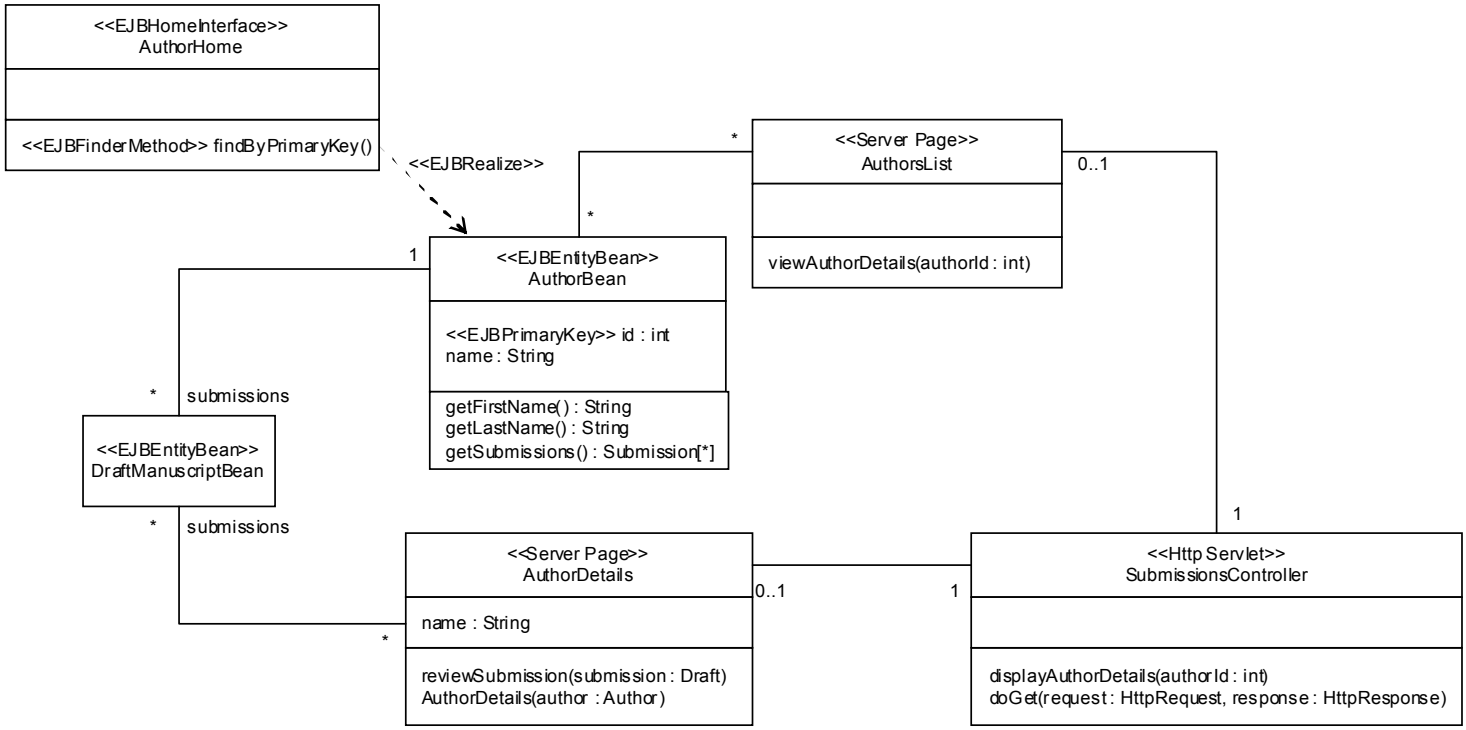

Fig 3.7. An implementation class diagram is useful at the start of a project to visualise the mapping of your designs to a specific technology

In this example, we have used a *UML profile* for J2EE applications to extend our model and translate the logical design into something that could readily be implemented on the Java Enterprise platform.

Another aspect of technical architecture you may find it useful to model at the start of a project is the physical design of the application. What files will be created and how

will they be deployed? What servers will you be using and how will they communicate? Component and deployment diagrams can be valuable in helping your developers to come to an effective physical architecture – but, again, the exercise of component and deployment modeling gets far less useful as the project progresses.

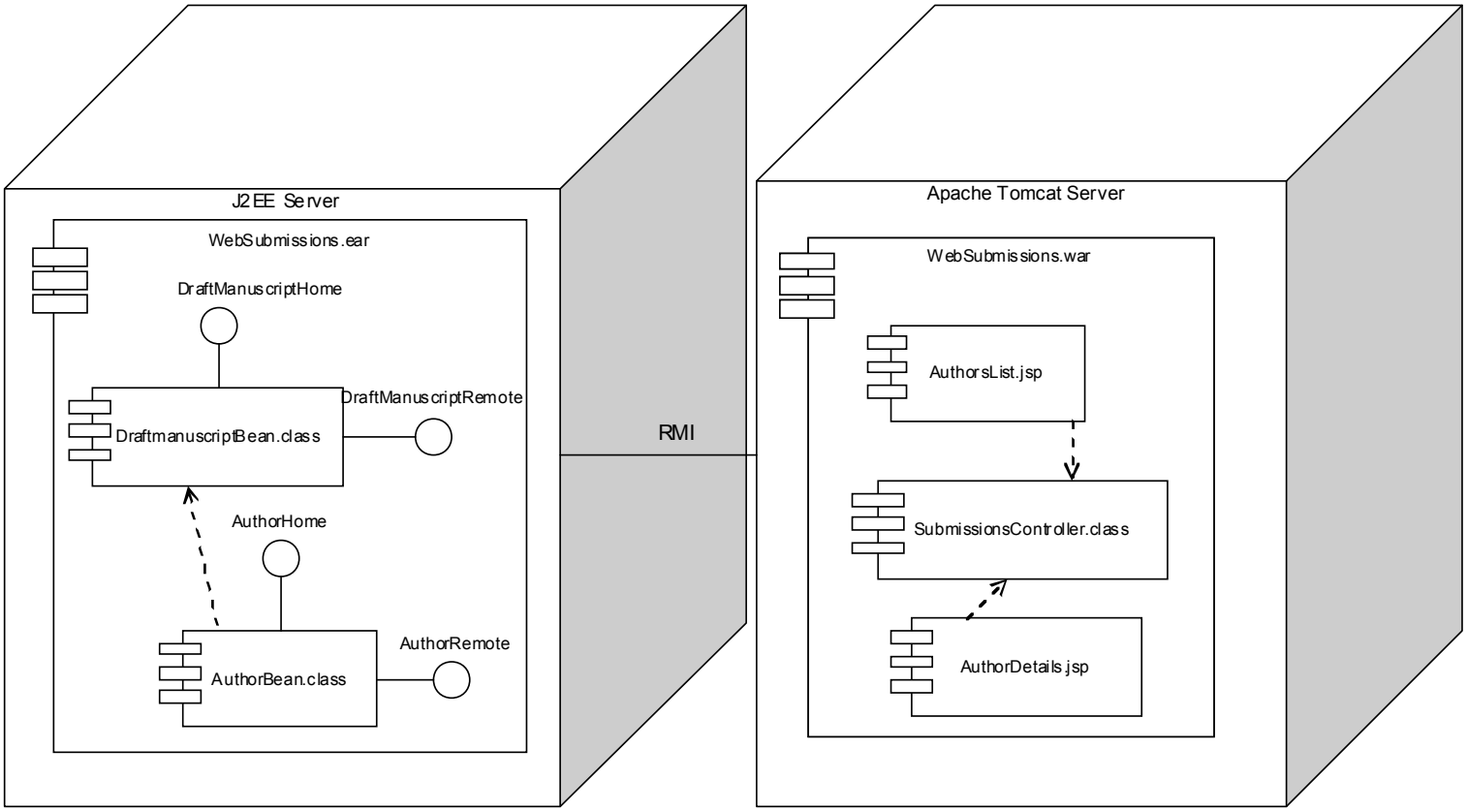

Fig 3.8. Component and deployment diagrams help us visualise the physical design of our application at the start of the project

Summary

The object oriented analysis and design process is, once you get the hang of it, relatively simple and straightforward. Just remember the key elements:

- Identify the types of user and the usage scenarios
- Identify the interactions between the users and the system in each scenario
- Identify the objects involved and show how each interaction changes them
- Validate your model-view-controller design
- Assign responsibility for changes to objects during the execution of a scenario to the most appropriate objects in your MVC design
- Model the logical classes you'll need in the code
- If you're thinking about technical architecture:
  - Do a platform-specific translation of your logical design
  - Model the physical architecture of the system

- Write the code and get user feedback ASAP!

Of course – you guessed it – there's more to it than that. But practice, experience and a good mentor will help your developers get to grips with the many "but what if…?" scenarios that OOA/D inevitably throws up. 90% of the time, however, this simple approach - and the many variations on it - does the trick.