UML for Managers

Jason Gorman

Chapter 1

February 7, 2005

Visual Models & Why We Need Them

A visual model is an abstract representation of something that contains only the pertinent details we need to help us understand it. For example, to understand how to get from one place to another place, an aerial photograph probably contains too much unnecessary information. This is why we use maps – visual abstractions – to help us get around instead of photographs. A map tells us only what we need to know to plan our route.

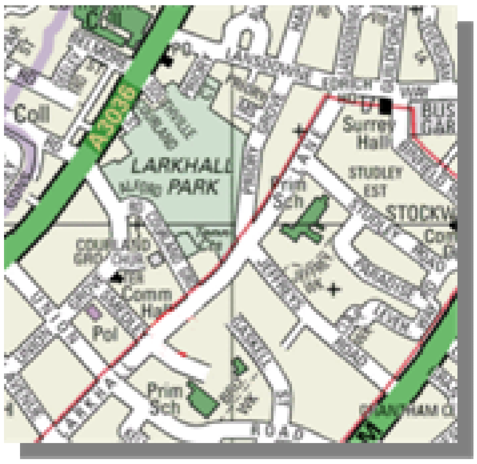


Fig 1.0 A street map is a visual abstraction of the information we might found on an aerial photograph

Visual models are written using **visual languages**. Just as written English is made up of symbols that have some meaning in the context of a sentence, visual languages are made up of graphical symbols that have some meaning in the context of a visual model. English has a grammar; that is, it has a set of rules that must be applied to the construction of meaningful sentences. "This are an exception rule the to" is not a valid English sentence because it does not conform to the rules of English grammar. Visual languages, too, have rules about what symbols can be used and where in the construction of valid visual models.

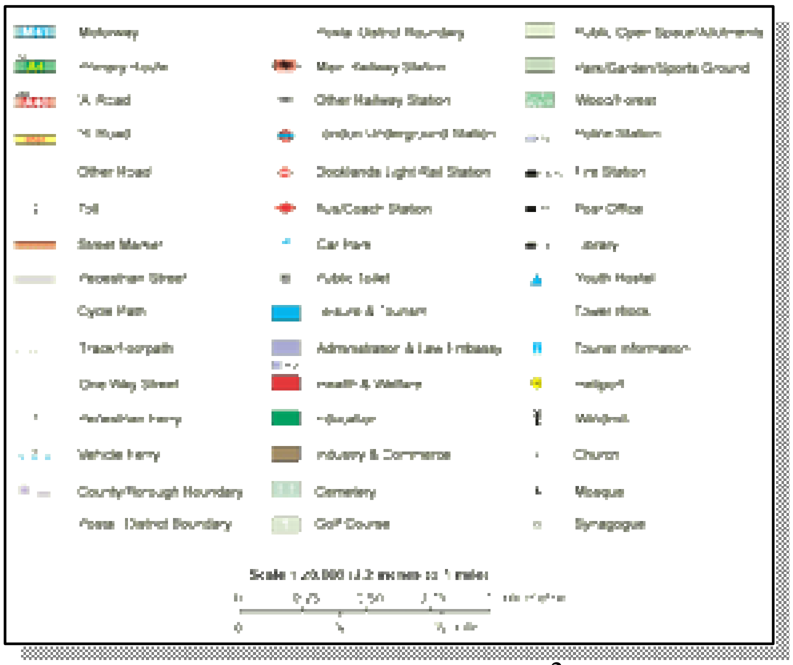
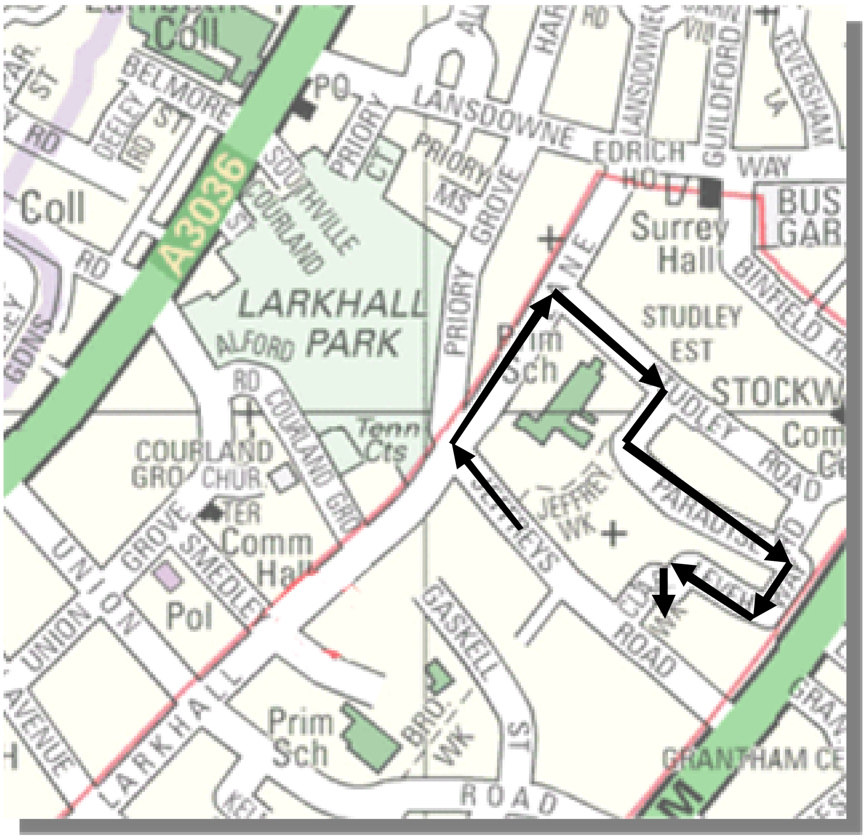


Fig 1.1. A visual language is a set of symbols that have some meaning in the context of a visual model. Visual languages have rules, just like written languages.

The value of visual models is in their ability to succinctly communicate large amounts of complex information. They say that "a picture speaks a thousand words", and this is why visual models are so important for many kinds of problem. Read the directions below, and see if you can spot the mistake:

> "Head towards Larkhall Park, and turn right out of Jeffries Road, then take the next right into Studley Road. Paradise Road is about half way down. Follow that all the way round into Levehane Way. Keep following that and go straight through to Clarence Walk. No 5 is on your right."

It's much more obvious when we look at a map:



The route takes us right around the block and almost back to where we started! Without the map, we may not have noticed and wasted time taking the long way round. The power of visual models is to present rich, complex information in a way that can be easily digested.

There are all sorts of problems that are best explained using visual models. Almost every discipline on the modern world exploits visual models in some form another – from electronic engineering, to weather forecasting to the design of computer networks. Visual models play a big part in the problem solving process.

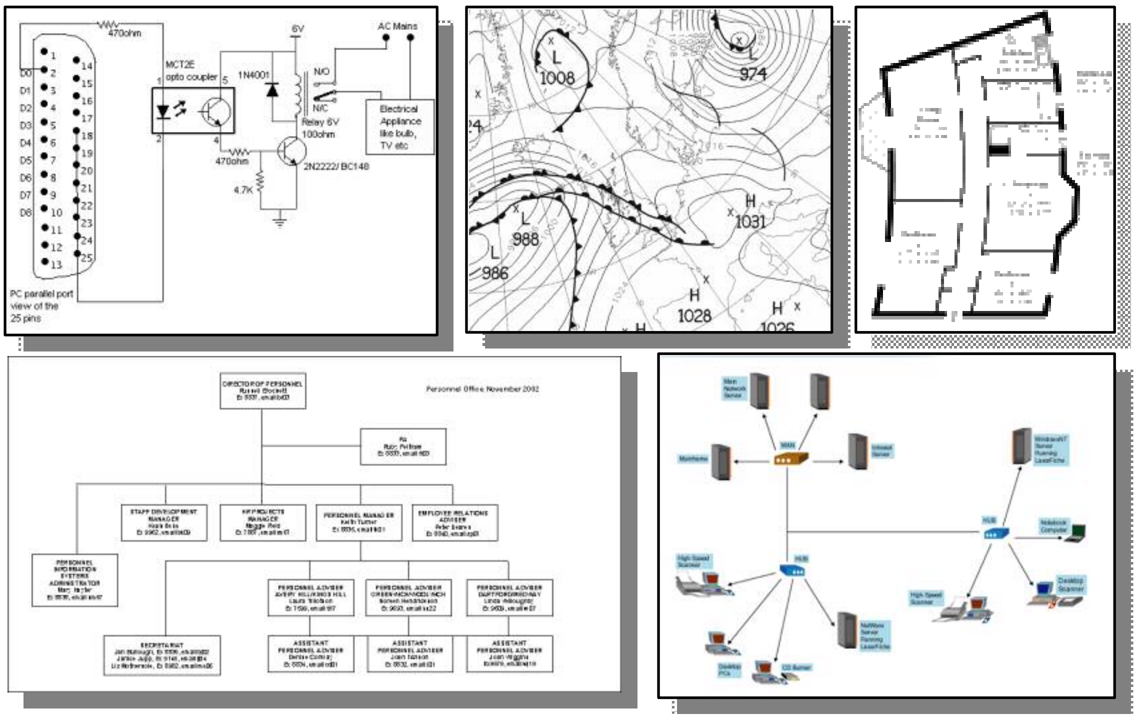
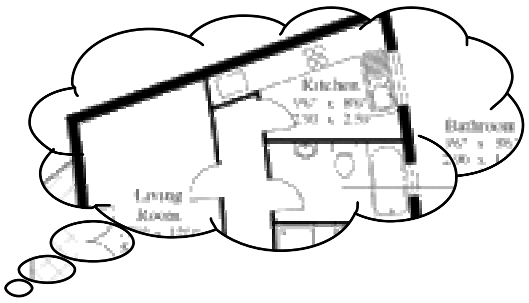
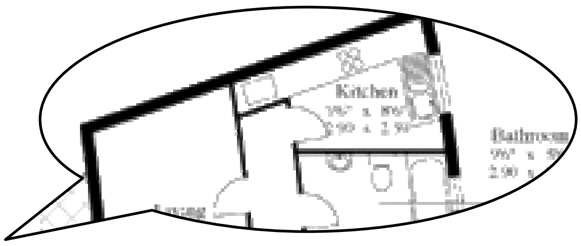


Fig 1.2 Visual models are used in many disciplines

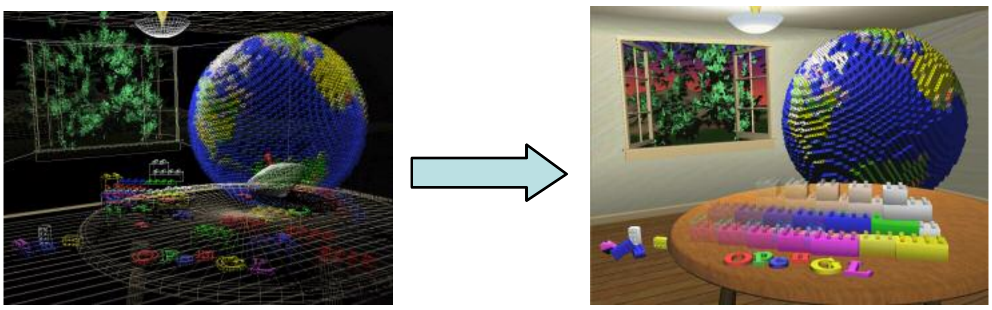
Visual models have three uses: they help us to *understand*, they help us to *communicate* with other people who speak the same visual language, and they can be used to *synthesize* new objects or models.



Visualization is the key to understanding all sorts of problems. Imagine how difficult it would be to understand the layout of a building without a set of visual plans.



Communication relies on shared languages to represent information so that it can be transferred from one person to another. Visual languages are the key to communicating complex information quickly and with less chance of misinterpretation. Effective communication is critical in software projects.



We can also transform models into other useful objects. For example, a 3D wire-frame model can be transformed into a photorealistic image by applying a set of rules about optics to the original model. In mechanical engineering, 3D models can be used to generate physical prototypes automatically to enable engineers to get early feedback on their designs. Synthesis is widely used in many engineering and product design disciplines, but is only beginning to be exploited in software development. We will discuss the synthesis of executable software from models later in this book.

What Can We Model Using UML?

UML is designed to help us design effective object oriented software. Unsurprisingly, the basic currency of object orientation is *objects*. We can use UML to describe objects, their attributes – facts that we can know about them - and the relationships between objects at some point in time.

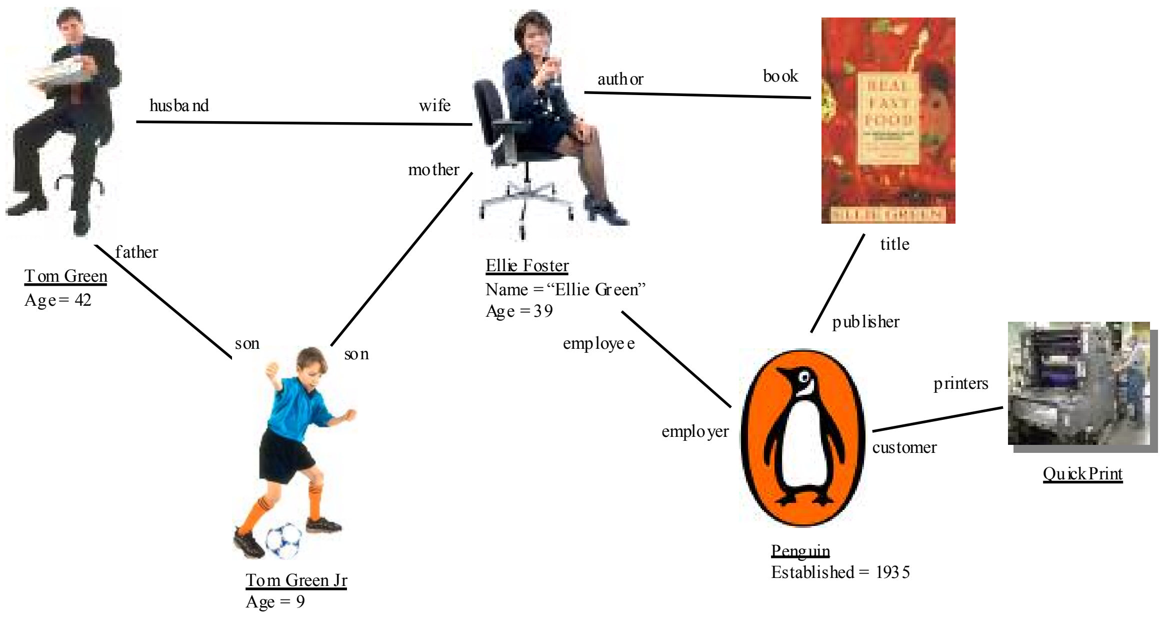


Fig 1.3. A model of objects, their attribute values and links between them

It is important to note that objects have a *unique identity* that remains unchanged throughout their lifetimes. Ellie Foster is still the same person, even after she has changed her name to Ellie Green.

Another important thing to note about objects is that they can play *roles* in respect of each other. They can play more than one role at the same time. For example, Ellie Foster is a wife to Tom Green, a mother to Tom Green Jr and the author of Real Fast Food.

As well as describing objects, we can use UML to model *types* of objects – that is, sets of similar objects that share the same characteristics. Types – or *classes*, as they are more commonly known in object oriented software development – tell us what attribute values any instance of that type is allowed to have, what roles objects of that type are allowed to play, and how many objects are allowed to play the same role with respect to the same object.

We use the term *multiplicity* to refer to the number of objects that are allowed to play the same role at the same time with respect to another object. For example, in the relationship mother->son, the role of son can be played by *zero or more* objects of type Man at the same time with respect to the same Woman, so the multiplicity of the role son is zero or more (or 0..*, or just *, in UML). In the reverse, only one Woman can play the role of mother with respect to the same Man, so the multiplicity of the role mother is exactly one (or simply 1 in UML).

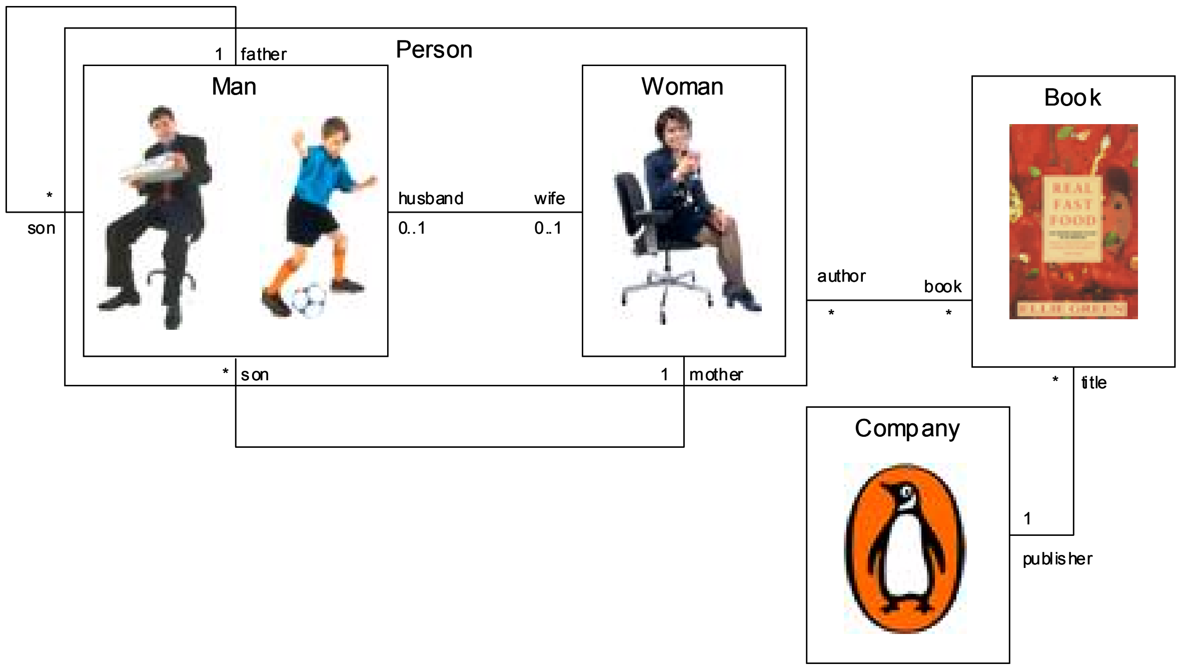


Fig 1.4. A model of objects types and the relationships allowed between objects of each type

A very important thing to understand is the relationship between objects and types (or classes). An object is said to be an *instance* of a type – a specific example of that type. Every instance of a type has the characteristics defined by that type, and must obey any rules that apply to it. If our type model tells us that every Man has exactly one mother, for example, then an object of type Man with no mother, or with two mothers, does not *conform* to its type. It is a strict rule of object oriented software that objects must always conform to their type – but, as we'll see, that's not always easy to achieve!

So a type – or class – model tells us the rules about what instances of objects, their attributes and the relationships between them are allowed. But quite often they do not tell us all of the rules. Sometimes rules about types can be more complex and subtle than "a Man must have exactly one mother".

For example, how can we model the fact that a Woman cannot have a son who is also the father of any of her other sons? We can use object models to illustrate scenarios that might break these subtle rules.

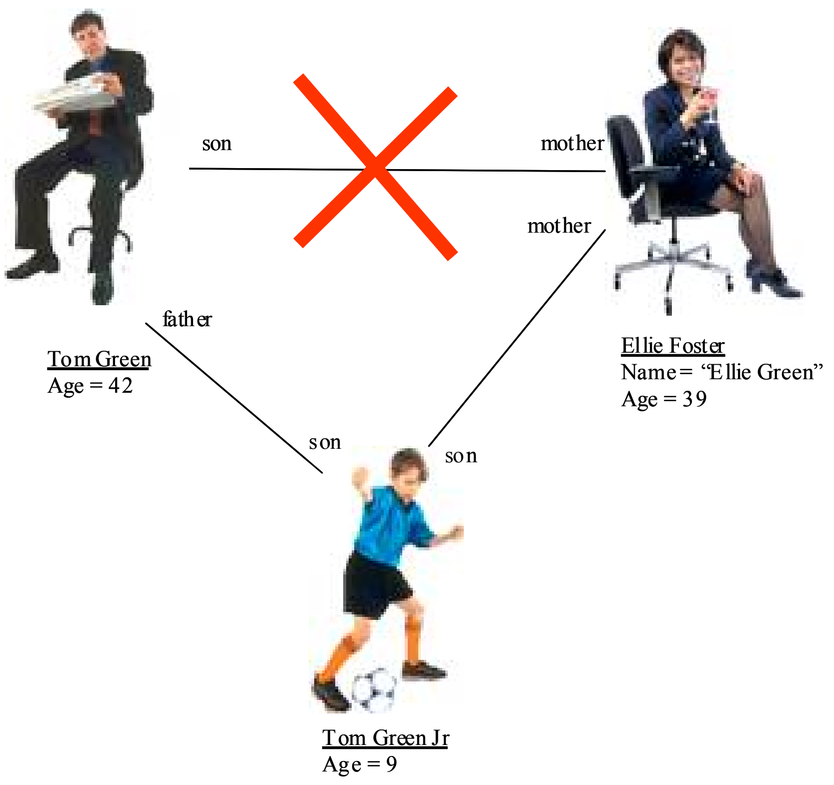


Fig 1.5. Ellie Foster cannot be the mother of Tom Green because he is the father of her other son, Tom Green Jr.

We can model complex and subtle *constraints* that apply to types of objects by adding *rules* to our type models. These rules apply to every instance of that type every bit as much as the rules about attribute values, the types of objects that can play certain roles in relationships, and the multiplicity of objects that can play those roles.

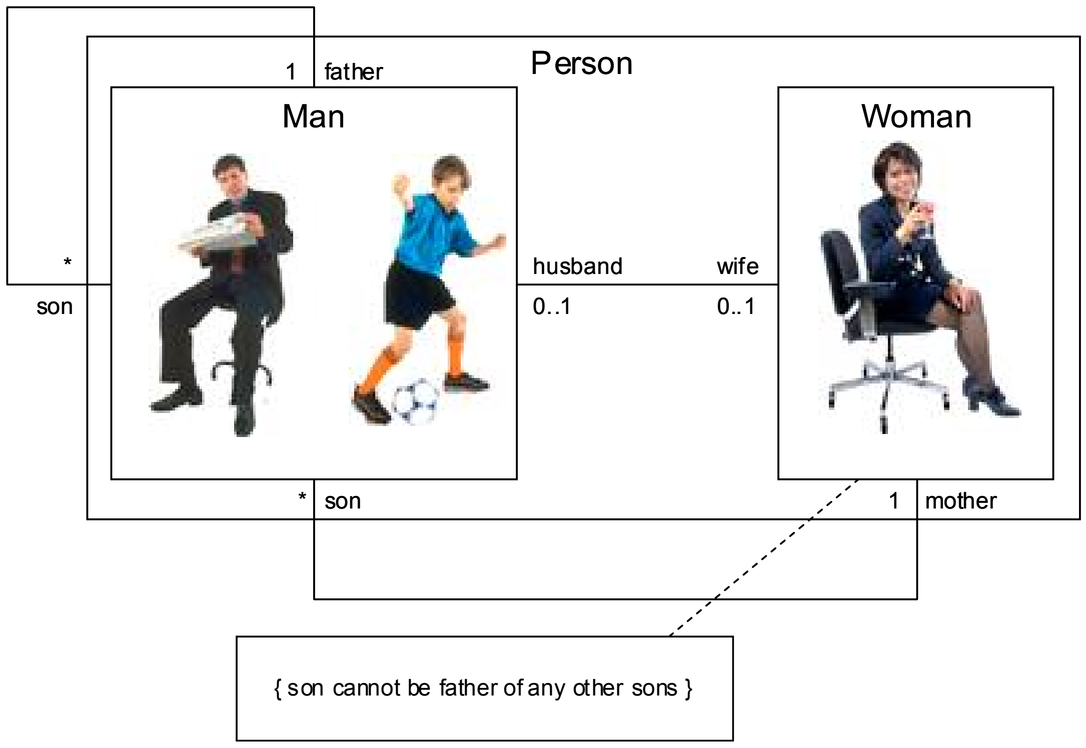


Fig 1.6. Rules can be added to type models to refine our understanding.

Adding constraints to our models helps us to more completely describe the logic of our systems – whether they be business rules (eg, “Every customer must have at least one account with a balance greater than zero”) or architectural rules about the design of a software solution. Just as objects are the basic currency of object oriented systems, rules figure very highly, too. All will be explained in later chapters…

So far we have seen how UML can be used to model *structure*. We can also use UML to describe *behaviour*. One very useful application of UML is in the modeling of workflows and processes. These might be business workflows, or they might be the control flow of program code. We can use UML to describe how objects *transition* through different stages in their lifecycle (eg, how a book goes from being an initial draft to a finished publication you can buy from book stores). In this kind of model, these transitions occur as a result of *events* happening to the object. An event is some significant action or some change that happens to the object that takes it from being in one discrete stage – or *state* – to another. For example, the action of withdrawing funds from a bank account could take it from being in credit to being overdrawn.

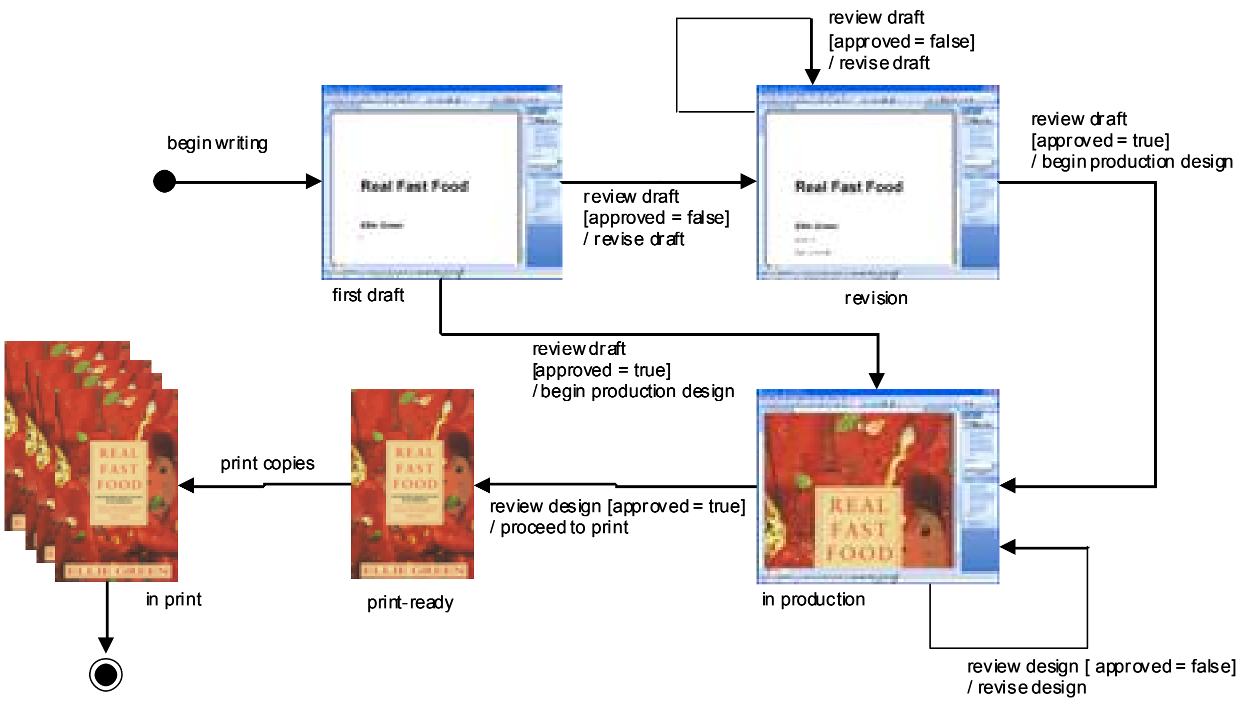


Fig 1.7. State transition models allow us to model object lifecycles and event-driven behaviour

State transition models, as we call them, are very useful for modeling objects that have a distinct and well-defined lifecycle – like a book going through the development and publishing process – or systems that have particularly event-driven logic to them (like a user interface where we want to show how the system responds to actions the user does using the mouse and keyboard.) This kid of model is especially well-suited to describing document workflow.

We can use constraints that apply to transitions to show how a certain event triggers a certain transition from one state to another only when some condition is true. For example, when the publisher reviews the draft of a book in development, it could take the book into production design – but *only* if the publisher has approved the draft.

We call constraints on transitions *guard conditions*. In UML, guard conditions are written in square brackets after the event.

Optionally, we can also show how some action is executed as a result of a transition – for example, we might want to show that the designer should revise his design for a book if his last design was not approved when the publisher reviewed it. Actions appear after that event and the guard – if there is one – for a transition. We will cover state transition diagrams in more detail in the next chapter.

We can also use UML to describe workflows or processes that are not especially event-driven or that do not necessarily apply to objects that have distinct lifecycles. An activity model, which we will explore in a later chapter, is a very simple kind of state transition model where every action is immediately followed by the next action. Activity models are useful for describing simple batch processes where we don't need to worry about user intervention or other kids of events.

State transition and activity models are very closely related, and many people mix them together, quite legally, in their models.

One other thing we can model using UML, and this plays a very significant part in the object oriented design process, is how objects *interact* with each other to complete useful pieces of work.

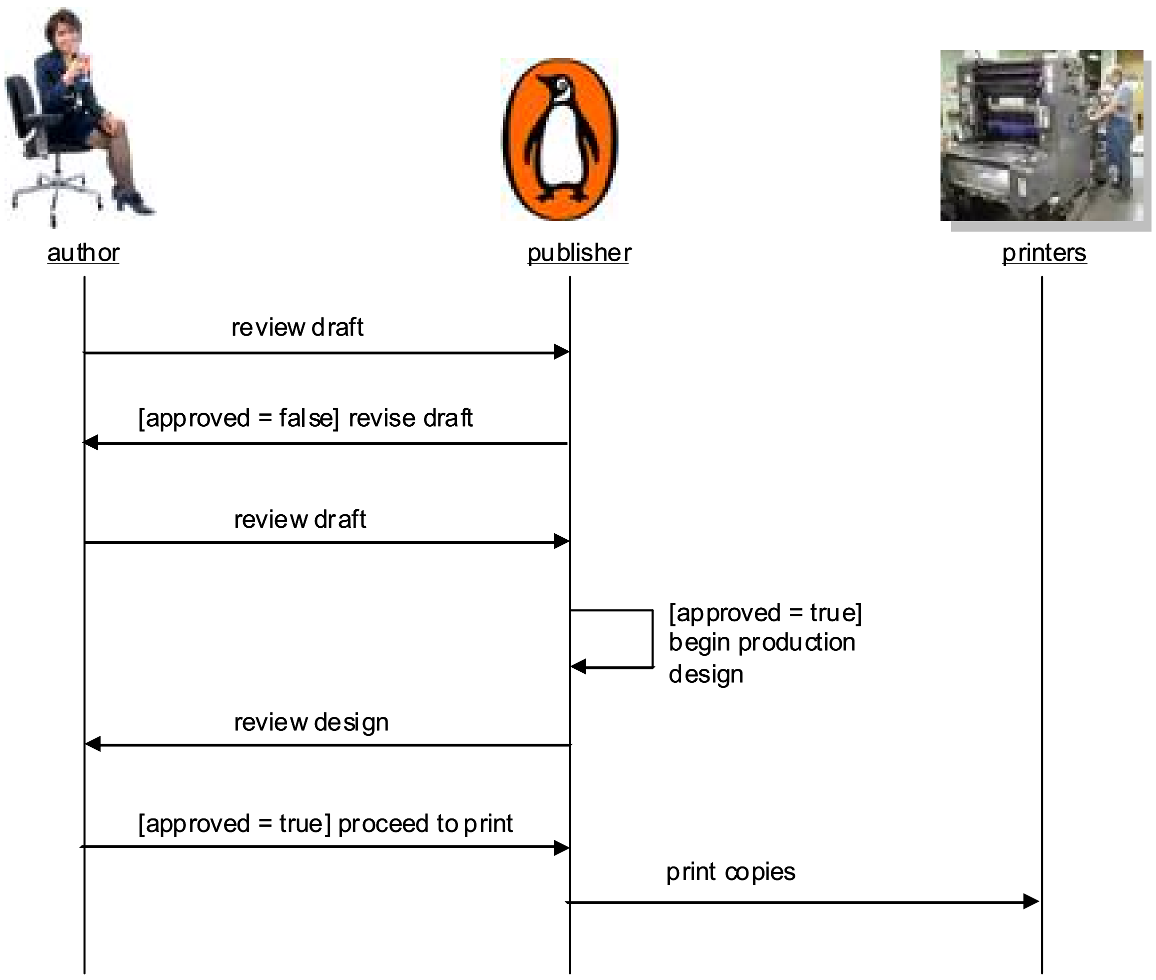

author
publisher
printers
review draft
[approved = false] revise draft
review draft
[approved = true] begin production design
review design
[approved = true] proceed to print
print copies

Fig 1.8. A sequence model describes how objects interact over time to achieve some goal

Once we have identified the objects involved and their relationships, we must now decide which object is taking *responsibility* for what action in the execution of a process.

Sequence models describe how objects send *messages* to each other – through well-defined *interfaces* – asking them to do some useful piece of work. Each type of object has responsibility for providing a set of *services*, and the interface for each type public face through which other objects can request those services. In UML, we call those services *operations*.

A sequence model shows how, over time – and for a specific scenario (a specific instance of a process – or a pathway through that process) – the objects involved use each others' operations to get the job done.

Assigning these responsibilities is a key part in the object oriented thought design process, and we will see in later chapters how these models can be used in a well-defined and rigorous object oriented development process.