Chapter 2

Three Models

Proposal

This chapter is the first of the four chapters that form the heart of the book. It will:

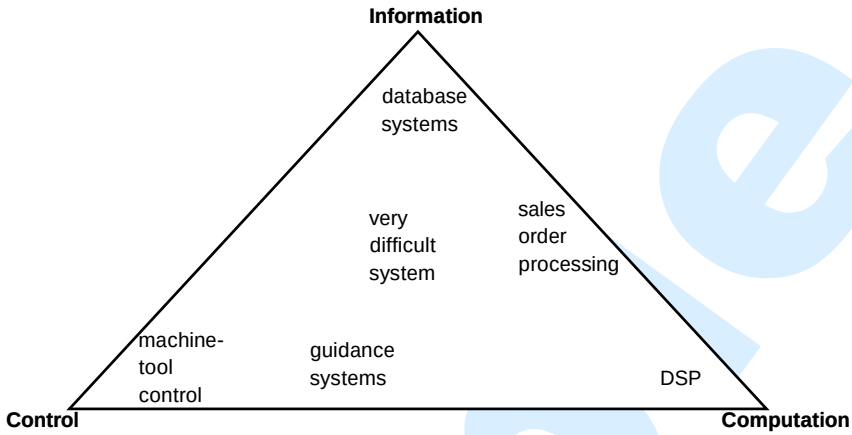
- introduce computer systems development as a modeling activity;
- describe the proposed modeling stages suggested in this book;
- mention the alternatives.

2.1 Models and Modeling

At various times for various purposes, software engineering is compared with hardware engineering. I've done it myself. There are significant differences however. Something that we can, and will, say about software engineering, that could not be said about most hardware engineering, is that all computer systems are models of something. A bridge is useful in and of itself; but to most of us a computer isn't much use until it's programmed. Think about it. Think about all the computer systems you are familiar with.

Whatever kind of system you choose, at its heart will be a model of something. In fly-by-wire, avionics software, there must be a little aircraft "in" the control software. The control software would not be able to decide what correction to apply to the control surfaces if it didn't have a picture of the current state of the control surfaces. And with regard to what? The airspace and the ground will be "in" there somehow as well. In an information storage system, there is a little world of facts in the machine. In straight computation, we use rules, equations or formulae that model something: addition models extension, multiplication models area, matrices model transformations, complex numbers model rotation, etc. Figure 2.1 portrays various computer systems classified as mixtures of control, information and computation.

Figure 2.1
Computer systems (in a "system space")



"Phase diagrams"

Diagrams like Figure 2.1 are quite useful little pictures, by the way. And you can get more elaborate if you want, with a square or a pentagon since the control/information/computer classification isn't the only one. It always helps me in the early days of overall architecture proposal to have an idea of whereabouts in that triangle a system would find itself. If your system is up at twelve o'clock, go and buy a well-regarded DBMS (database management system). If it's round at half-past four, buy a DSP (digital signal processor) chip or use FORTRAN; remember that very few systems are *nothing but* computation though. If your system is at three o'clock, you have, currently at least, the difficult decision as to whether you should program and worry about persistence, or buy a database and worry about programming, or whether EJB (Enterprise JavaBeans) might be the answer. If your system is right in the middle, good luck.

The success we have had with metaphors in computing furnishes some evidence that models with good correspondence are important. Over and over again when we examine successful computer systems, we find that they employ easily discerned metaphors. It is at its most obvious in good interfaces. Take any interface that people find easy and intuitive, and you find it employs a strong metaphor. The famous desktop metaphor of graphical user interfaces (GUI) is very common and very successful.

If you can think of a computer system that isn't in any sense a model, then either it wasn't developed by humans (a neural network or Darwinian system, perhaps), or it's an absolutely dire system that nobody understands and that cannot have one single bit changed without it reeling, writhing and fainting in coils.

Now, if all computer systems are models, then computer systems development is modeling. We are all modelers. And that picture of the job helps us organize the job.

2.2 The Three Models

Over on one side of Figure 2.2 we have the world – an adequate picture of our remit for most of us although not if your field is space or cosmology – and over on the other side we have the system-to-be. (If a tape drive is iconic enough for the movies it's iconic enough for me.)

Figure 2.2
Modeling

The question is, how do we get from one to the other? As a sculptor, unless you were terribly, terribly contemporary, you wouldn't start with the modeling medium and alter your subject to fit it, you'd take your subject and decide how to represent it in stone, bronze, wire coathangers, or whatever. Can we take lines of code as our modeling medium, and fashion a model directly in source code? That has been an expectation in the past, of course: "You lot start coding, I'll nip down the corridor and find out what they need." It can't work for anything but the smallest, most trivial, use once and throw away kind of system, however. We will decide to work from the subject to the model.

How many stages to our modeling will there need to be?

In 1956, in *The Psychological Review*, George A. Miller [Miller 56] wrote a paper suggesting that there was a limit on how much we could hold in the foreground of our mind, and reason about. He proposed that the limit was seven, plus or minus two. That number, 7 ± 2, has become pretty famous since then: George Miller's magic number. Today, we still believe that there is such a limit to short-term memory and foreground processing. Another expression used by cognitive psychologists for these kinds of limits, is that, if like me you are a word person rather than a picture person, the short-term memory or foreground processing area seems to be able to hold a two-second phonological sequence.

We could reasonably propose, then, that if a developer did *nothing other than* write one line of code after another, the longest program that could be developed in any confidence would be about nine lines of code. Some people find that difficult to believe, so here's a test suggested by Gerald Weinberg. I call it the "your money or your life" test. We make a bet. You tell me how many lines of code (expression statements, if you want to be a little more precise) you think you could write, for me to then run them, such that they did *exactly* what you predict they would do, no more and no less. The bet is €300 (£200, $350). Most people think for moment and then decline the bet; almost no-one goes for more than six lines of

code! (The "your life" version of the test is to picture yourself sitting back in your airline seat at take off, thinking about whether you're going to have a beer or some wine; and then, just as the engines are getting up to half power, the airframe is juddering and the brakes are about to be released, you realize that you wrote the flight software. Most people would be running down the aisle toward the door hoping they'd paid enough attention to the instructions on opening that door.) In reality we have, quite sensibly, very little confidence in pure coding.

To counter the bad news just delivered, there *is* something we can do. Cognitive psychologists call in *chunking*. We assemble groups of (7 ± 2) problem elements into chunks. These chunks can be pictured as single things, elementary things, from some perspective or other. Given the ambition and complexity of today's computer systems, we simply *must* employ a chunked model between the subject matter and the code. Will one extra stage to the modeling be enough? Few people say yes explicitly, although a lot say yes implicitly. I am going to say no, as you would guess from the book's title and from the introduction.

Our technology – object technology – is fairly complicated. One of the reasons for going object-oriented is certainly *not* that it's simpler than Fortran, COBOL, Pascal, or C. We will need guidance as to a likely optimum set of chunks. We also want a model that is understandable and malleable. So we have already proposed in the introduction that a chunked subject matter (analysis) model should precede a chunked design model, and that having the subject matter guide the formation of the first model will give us, a) a simple, straightforward development path, b) an understandable and communicable system, and c) a malleable system with a low impact of change.

In fact I am actually going to propose two design models, making three chunked models in all, as depicted in Figure 2.3. There seem to be several, commonly held, misconceptions in software systems development to be dealt with:

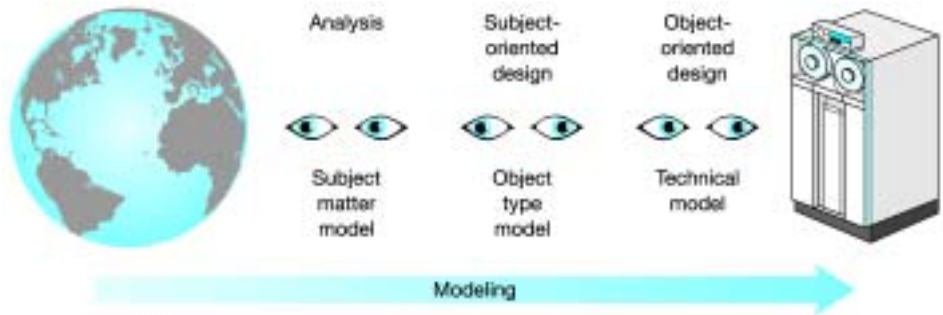
- There is confusion as to just what the analysis phase can and should do.
- There is far too much *class*-orientation, rather than *object*-orientation.
- There isn't enough emphasis on the interfaces of objects (amazingly).
- A lot of methods seem to suggest that good quality objects can be designed without thinking about the clients of those objects.

To try to counter all of the above, I am going to be suggesting that an analysis-using, instance-oriented, interface-oriented, client-oriented stage of design be distinguished from a detailed class design stage.

So there they are, the three models. Well four actually. It's quite reasonable to see the code itself as the final, executable model.[1]

1. It might be worth exploring this viewpoint a little further. Frederick Brooks [Brooks 86 or Brooks 95] pointed out that the two most successful uses of computers, against which all other efforts pale into insignificance, are the database and the spreadsheet. We will come back to the reasons for the success of databases or DBMS later. What is so powerful about the spreadsheet? It's a user-createable, machine-executable model.

Figure 2.3
Modeling stages



Incidentally, I am probably using the term *model* in a slightly broader way than the UML. The UML says that a model captures a view of a physical system. The word "physical" bothers me however. I want to use the UML for conceptual models. I have worked with models of finance systems and models of quantum field theories (which are themselves models of goodness knows what) and I am not comfortable with calling either of them physical systems. Also, the word "view" is rather weak. As we will see when we get into the detail of our first model, a model is more than just a view. It might be better to say that a diagram is a view of a model.[2] So, sometimes I will use the term model in a collective, mathematical sense, like the heading of the next section, and sometimes I will use the term model more like an engineer, for a particular kind of diagram, or even a particular diagram. It should be clear from the context; and it isn't all that important anyway.

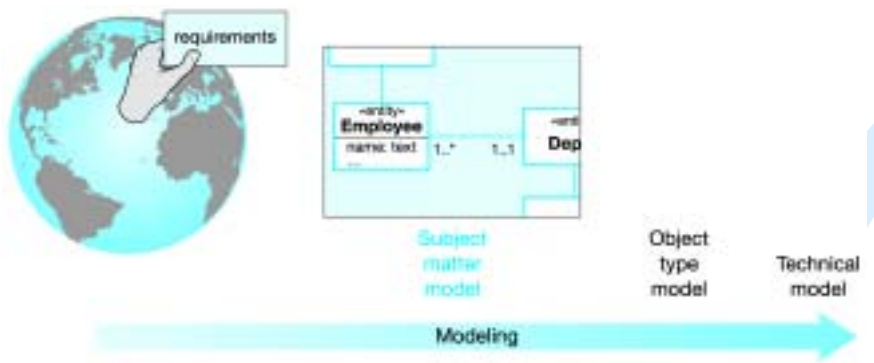
2.2.1 Subject matter model: first model

One of the reasons for proposing separate models was so that we could start with the facts, with the definitive. We want to defer the speculative for a while. So we start with an existing subject matter. Now almost anything we examine in the subject matter will support one of our purposes, which is understanding; but not everything we might look at will support our second purpose, which is making a good start on the architecture of the system-to-be. We want one model to furnish suggestions for the next model. We want correspondence or mapping from the elements of one model to the elements of the next. We want our analysis model to make good suggestions for our object architecture.

When we start our model, where would we expect to find correspondence? Our most obvious possibility, and expectation and hope, is in that there will be correspondence in the objects. If the subject matter for POIROT has a *Suspect* entity, do we expect to have *Suspect* object instances? Yes we do. It's reasonable and it works. That we perceive a relevant *Break*

2. Which the UML also says. It's all a bit confusing; and inevitable in a standard created as the amalgamation of three methodologists' ideas and now looked after by a committee. I don't want that to sound disparaging though. Without the effort that went into the UML, or something like it, software developers would be in a far worse state than they are.

Figure 2.4
First model



In entity, reasonably and usefully predicts *BreakIn* object instances. And it seems to work in all subject matters: *Employee*, *Invoice*, *Blood Sample*, *Chromatograph*, *Pump*, *Magnet*, *Electron*, *Curve*, *Polyline*, *Matrix*, *Vector*, *Function*, …

Are there any subject matters where we don't find entities, objects? I think the answer is "no". You can always find entities and they always suggest good objects for an object-oriented implementation. However, there are important qualifications. The first is: how easy is it going to be? Well it's not always easy. The second is: are you really considering an object technology implementation?

The latter is the easier to deal with, so I'll address that first. The most frequent example that comes up is compiler writing: "You can't tell me, Mr OOA Lecturer that I'm going to find an entity-based model useful for my compiler project!" No. Quite right. But then again, you'd be mad to write a compiler exclusively via an object-oriented programming language. Compilers are an example of "don't program, use a tool". We understand programming languages and compiler techniques well enough that there are compiler compilers (*yacc*, *bison*, etc.). One generates the bones of the compiler via a specification of the language, written using a well-known form – a context-free grammar – and a tool – a compiler compiler. One then adds the stuff that isn't easily generated like type checking. And yes, even there you could use an object-oriented approach; you could find entities in your first model: *type* entities, *declaration statement* entities, etc.

Now for the question of how easy it is going to be. When others and I first began talking about this kind of thing in seminars and classes during the early 1990s, I thought that there might be subject matters without entities. I am thinking particularly of numerical processing and of the FORTRAN programmers I spent much of my early object-oriented years in discussion with. There usually turned out to be one of two answers.

The first possibility was that although the code in question revolved around some tremendously difficult piece of mathematics, it was essentially a small piece of code and did only involve one or two objects. When one put that piece of mathematics in the context of the larger system using it, many more structure and control entities appeared. (If that piece of mathematics hadn't been part of a larger system, then it wouldn't have needed this book anyway.)

The second possibility was that our thinking at that time was already entrenched enough in flowchart thinking that the entities were there but were difficult to perceive; it required hard work and a change of mind set to perceive them; but they were there. In a large piece of mathematics, one can find function entities, domain entities, codomain entities, rule entities, matrix entities, etc.

There will be yet more subject matters where things are not so easy, but I think the above example will suffice to show the direction.

So, with varying degrees of effort, we find entities and we find correspondence in those entities. Where else would we expect to find correspondence between a subject matter and an eventual object architecture? The earliest days of systems analysis began with *processing*, flowchart-thinking if you like. With many software techniques we often used to start with a consideration of action, with processing. One of the very first languages – FORTRAN – gave almost no thought to data structure, focusing instead on algorithms and processing. It's an attractive viewpoint. The computer is there to *do* something, so surely *doing* is important. It's also an attractive viewpoint for the "lizard brain":[3] chase that, hit that, eat that.

In fact, thinking back to the early days of "object-oriented this", "object-oriented that" and "object-oriented everything else", perhaps it isn't as obvious as I made out, that we would base our models on objects. Some of the earliest techniques, probably because of the success of their data flow diagram predecessors, tried to make functions their primary model element. I think that we can safely say, a decade and more later, that all the suggestions[4] that started with something other than an entity – a data structure rather than a function – failed and have withered away.

Of course our eventual object instances will be very interested in what they have to do: the service interface that an object presents is just about the most important thing to get right.

But now for some, perhaps surprising, bad news. We can't base the things that our objects will do on what their subject matter counterparts do.

What might our subject matter *Suspect* entity do? Well presumably, some of the subject matter suspect entities commit crimes. Do we expect our Java suspect object instances to commit crimes? Not in the obvious sense, no. A Java object isn't going to leap out from behind a bush and snatch somebody's purse. It turns out, however, that *in no sense at all* would a Java *Suspect* object commit a crime.

Let's try to picture some of the program objects that a seasoned, object-oriented developer is likely to find reasonable. Let's picture a crime object of some kind, some suspect objects of some kind, and a terminal on an officer's desk with a reasonable interface, a GUI for example. Let's say that in the real world, a particular suspect did commit a crime. The crime object in the software would likely get created before the suspect objects and long

3. The most primitive part of our brain. The part that exists in most other animals as well. Sometimes called the limbic system. If you react instinctively, the limbic system is short-circuiting the higher functions.

4. Both GOOD and HOOD 3 attempted to feed an object-oriented design from a data flow analysis.

before the investigation revealed who the perpetrating suspect was. So the suspect object *cannot* have had anything to do with the creation, the *construction* of the crime object. In any reasonable system, it's the officer who would have input the details of the crime through the interface; then, over time, the details of one or more suspects would have been input or located by various officers, and possibly by some algorithms as well, and then one happy day an officer would have figured out the real perpetrator and told the system. If anything, the real world officer committed the Java crime!

I have been through this explanation with many groups over the years, and a common objection – because people are clearly uncomfortable with this suggestion – is that I am purposely picking silly, or quaint, examples. But I promise you that you will come to the same conclusion for any subject matter, and for any example, anywhere from the mundane to the exotic.

What does the subject matter *Crime* entity do? I mean the crime itself and not the criminal – the murder, not the murderer. Later on, when we come to classify entities, we might classify the *Crime* as an *event remembered*. Well here's our next problem. Many subject matter entities don't do anything! The crime itself is passive not active – it doesn't do anything. Does that mean that its software counterparts won't be doing anything? Of course not.[5]

Perhaps you've seen some example analysis diagrams elsewhere and your recollection is that they showed processing – or *operations* as UML calls them, perhaps for every entity. If you take a good look at such analysis diagrams, and particularly look at the processing or operations that the passive subject matter entities are shown as doing, you will find that they are inventions.

Now, of course, you *could* put inventions into this first, subject matter, model. My suggestion, however, is to have the first model be definitive and to make the speculative model a later model. The problem is that invention is more difficult. Some inventions are great and some inventions are not. Let's try to stick to the facts for as long as we can. Let's make the foundations of our architecture as solid as we can.

Think of some other models. I have in mind a clock and a sculpture. I use a clock because it's more convenient and feasible for me to look at the clock than to look for the sun and estimate its position in the sky. I have a bust of Johann Sebastian Bach because the real Bach is not disposed to come and stand on my desk. What do these models *do*? The sculpture does very little, unlike the real Bach who was rather prolific. The clock is more active but it certainly hasn't got a rotating lump of rock whizzing around a thermonuclear fireball. All models represent state much more faithfully than they represent action or communication.

5. In case you want more examples, or find this interesting, or amusing, let's gather a bit more evidence. Would you, could you, calculate the payroll by asking each human employee what they think they should be paid? Not in current political systems you wouldn't; yet that's an excellent thing to design the *Employee* objects to do. What does a subject matter customer entity do? Send money. Well that would be nice. What does a subject matter patient do? Get sick, take medicine, age, get well, … . What does a subject matter electron do? Nobody knows for sure.

If our primary first model elements are to be entities ("proto-objects") and if actions are not a reasonable secondary model elements, what *do* we have for the secondary elements of the subject matter model? We've just said: state. If you think about it, any and all models represent and have continuity with state. How, then, does state manifest itself?

The most obvious manifestation of state could be termed intrinsic state: the relevant and measurable properties of the subject matter entities. These are what we have often termed, and will continue to term, ***attributes***. If a suspect, for example, has an age and eye color, it could well be a good idea that the C++ suspect object, for example, exhibit an age and eye color. A C++ crime object would very reasonably exhibit a time and a date. A C++ invoice object would exhibit an amount and a number. A C++ patient object would exhibit a date of birth, an age. A C++ electron object would exhibit a mass and a charge. Intrinsic state, or attributes, are easily discovered and always going to tell us something useful.

There are more subtle manifestations of state as well. If, as we should, we decide to "chunk" our model, we are necessarily creating boundaries between the chunks. Looking out of my window I see what I call a tree, and in my mind it is separate from what I call the ground. In reality there is no clear separation: the tree has roots in the ground and the outermost cells of the roots have semi-permeable membranes with a continuous solution crossing the membrane. When we create boundaries, we take a little bit of the state of some thing, into some other thing. If it's an important and relevant piece of state, then we say that the entities are *related*. Ah! Now you see where I'm going. Relationships are just extrinsic state.

Relationships (specifically ***association relationships***) also exhibit good correspondence. If a crime is related to a suspect in the subject matter, then a corresponding C# suspect object will be related to the corresponding C# crime object. In a similar way, we would expect a C# patient object to be related to a C# doctor object if its subject matter counterpart was in a relationship with a subject matter doctor. In particle physics simulation, an electron object would be related to a magnetic field object.

In fact if you really want to be minimalist/elegant, you could say that an object/entity is just a packet of state that we grant identity to. (Identity turns out to an important property and we will return to it. An alternative term would be individuality.) So even the entities we began with concern state as well.

Are there any other manifestations of state? Yes: state *changes*. It can be useful to model what we call *state transition*. We can indicate that our entities exist in a finite number of abstract states. This aspect of the subject matter model is at its most interesting when there are constraints on the routes (transitions) from one state to another. If a suspect cannot reach the *convicted* state without having gone through an *arrested* state, we might want to depict that. In other words we do see the *effects* of actions even if we don't concern ourselves strongly, in this first model, with the actions themselves.

There are, however, problems with state transition models (or *state machines* as they are known). (It does rather seem that the closer we get to actions, the closer we get to problems.) State transition models make no direct predictions about architecture elements; instead they give us constraints that we must ensure our object instances abide by. Further-

more, state transition models turn out to somewhat difficult to get right. We have two whole chapters coming up on the subject matter model; and there will be more on state transition diagrams there.

Summary

In the model so far, demonstrating good correspondence to the system-to-be we have:

- entities (state packets with identity)
- attributes (intrinsic state)
- association relationships (extrinsic state)
- state transitions

We are not considering action as a primary or secondary model element. Is there anything else that we need to be careful with? Yes. Communication. (I should point out that the list of things I have chosen to talk about includes all of the common candidates that have been proposed for these kinds of models. I'm sure there are other possibilities in addition to these. I want to suggest some elements that I think are useful; and I want to warn about suggestions you might see, but that I have found to be less useful.)

Probably because messaging is a very important part of object technology, some methods have suggested that inter-entity communication should be part of the first model. Once again I am going to suggest that at best it doesn't help much, and at worst it's downright misleading. Many subject matters don't have much observable messaging. In a control system for example, it's straining things somewhat to perceive a control valve as messaging a fluid, or to perceive a magnet messaging a charged particle beam.

In other subject matters there are things that could be considered to be messages. An *officer* might be perceived as messaging a *suspect*, "You're under arrest." However, if we take that to mean that a Java officer object instance is likely to message a Java suspect instance, we will again be led to a silly design. Going back to our picture of a reasonable system-to-be, it would probably be the custody sergeant (desk sergeant) who updates the system such that an object instance is arrested.

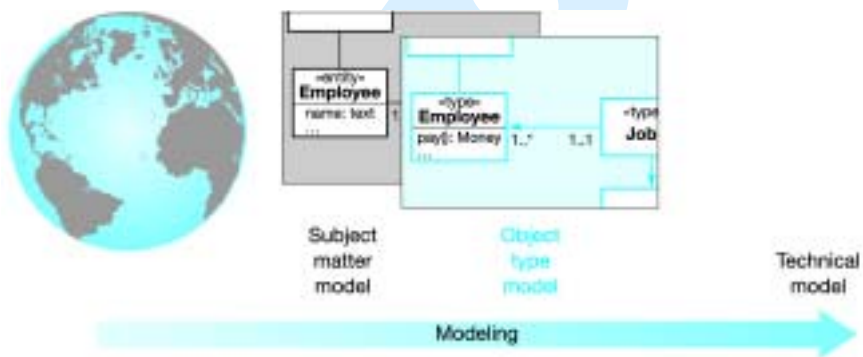
As we saw with actions, communications often turn out to be what we will term *events remembered*; important communications become entities rather than any kind of relationship between entities. If important communications get modeled as connections rather than things, not only is it going to be misleading as to eventual inter-object messages, but there is a risk that important entities, and therefore important objects, will be omitted.

You can include entity processing and communication – the UML certainly offers notation to represent it – but I counsel against it. You would need to make it crystal-clear to your readers that your subject-matter model is not proposing the allocation of processing or communication, but is instead providing nothing more than greater understanding of the subject matter. There are such big dangers down that road that I will be looking for another way of handling action and communication.

Apart from communications that are remodeled as events remembered, most of the detail of communication will be dealt with in the design models and in the CRC activity yet to come. What do we do about processing detail? There are two main possibilities. Which one you use depends on the structure of your development process. If your development process is iterative ("spiral"), then processing is allocated to objects in a design activity, and any subject matter processing questions posed by the design – "how exactly does one calculate a wind chill factor?" – are answered with a later iteration of analysis. If your process doesn't permit iterative development (then good luck, you'll need it), or doesn't permit "designers" to carry out subject matter investigation,[6] then you are probably best off describing important and non-trivial algorithms in non-object-oriented activity diagrams, as we shall see later (see *Dealing with Algorithms* on page 216).

2.2.2 Object type model: second model

Figure 2.5
Second model



There are enough issues at this point, as we are about to move from analysis into design, to warrant repeating and rewording a list from earlier in the chapter:

- We will have made a good beginning with our subject matter model. We will have the beginnings of a structure or architecture with object ideas and relationship ideas.
- This initial model makes no predictions or suggestions about some aspects of object technology – action and communication, in particular.
- The "orientation" in object-orientation should actually mean something.
- Object technology is complicated.
- Object orientation is a better start than class orientation.
- Outside-in design is better than inside-out design:
  - Interfaces are more important than implementation.
  - Think client. Messages are more important than methods.

6. I have known projects where bizarre project management edicts meant that designers were not allowed to interview subject matter users or experts.

Some of those points may be fairly obvious, and all of them will be or have been explained in other chapters; however, the last few points could probably bear a little amplification at this point.

A different orientation

To use object-oriented programming languages successfully, requires a different way of looking at things, whether our role is architect, high-level designer, or programmer.

For example, our overall picture is no longer a big "beginning, middle and end" program listing. The programmer doesn't call the shots any more; modern software tends to be event-driven or user-driven. Programmers and programs don't do things to the data any more; data does things for itself now.

A more complex technology

Many times, I've got the sense that organizations have adopted object technology because somehow the impression has arisen that software development is going to be made easier. Not at all. We've upgraded our technology to create more powerful software, better organized software, more scalable software, more reliable and robust software, and more malleable software; but that certainly doesn't mean it's going to be easier to develop. Even after all these years, it still seems to me that a significant number of organizations grasp at new technologies hoping that they will enable developers who have been trained and resourced to the level where they can build a garden shed, to suddenly and magically be able to bodge together a tower block.

Object orientation rather than class orientation

It's interesting to note that we call ourselves object oriented, and that for most people, object means object instance, and yet there is actually a great deal of class orientation about. It turns out in fact, that in many ways object-instance orientation is better and easier than class orientation.

Object classes differ much more from language to language than do object instances. A C++ object instance is pretty similar to a Smalltalk object instance (and a Java instance and a C# instance). A C++ class on the other hand is nothing like a Smalltalk class (with Java and C# somewhere between). A C++ class has just about no run time presence whereas a Smalltalk class is actually an object, present at run time, that happens to specialize in the making of other objects.

In all the languages, the class mechanism is quite complicated. This is partly historical. Classes began as fairly simple devices and then acquired more functionality as the years went by. In just about all the languages, classes can act as object makers and as types. In C++, the class mechanism is particularly complicated. It can make run time polymorphic objects; which is mostly what this book concerns itself with. C++ can also make compile-

time polymorphic objects through template classes.[7] C++ can use classes as types, it can use classes as object implementation resources, and it can use classes as both types and object implementation resources at the same time.

Given the tricky nature of a class, given the simpler nature of an instance, and given that it's the instances who will be there at run time doing the work, surely it's a winning idea to have the design start by seeking instances.

Does that mean we design individual instances? No that clearly isn't feasible. What we do is recognize that there will be populations of object instances that, *to their clients*, are alike in some important way. We will explain later that while internally, particular populations of object instances will have individualized state and may even have been defined by more than one class, externally – to their clients – they have the same *type*, they can be used in the same way and they fulfill the same services. This is ***polymorphism***, one of the pillars of object-orientation. If we have the earlier design phase aim at designing the outsides – the interfaces – of these object types – these populations as they are perceived by their client objects – we can then postpone the difficult questions of class choice until a little later. And at the same time, we satisfy several other of the issues listed earlier.

That is why this model is called the *object type model*. When we talk of the *type* of an object, what do we mean? There are entire books on the subject of type, but in essence, type gives us a handle on interchangeability. If two numbers are interchangeable in an algorithm's input, then we say that they have the same type: there are an infinite number of integers, but we can quite happily say that some algorithm inputs an integer. What makes object instances interchangeable? Accepting the same set of messages, i.e. having common interfaces, is what makes object instances interchangeable – gives them the same type.

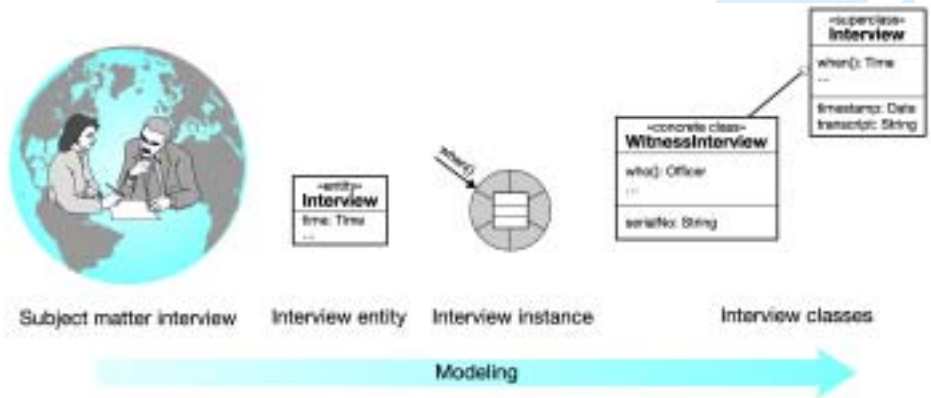
Object type never seems to me, to be a grand enough or obvious enough term for something so important, but that's the term that is used. What we are saying is, "we will need this kind of object, accepting these kinds of messages for this kind of client; and we will need this other kind of object accepting these other messages, …". But instead of terming them "object kinds", we're terming them *object types*.

Why is it more difficult to figure out what goes in classes than to figure out what object instances should look like? Apart from the complexity of the class mechanism, it's also partly because the subject matter model suggests the nature of the object instances of the system-to-be, rather better than it suggests the classes of the system-to-be. (One would have to suspect that that must be true anyway if one contrasts the variable nature of classes from language to language with the fairly constant nature of object instances from language to language.) The presence of an interview entity in the subject matter model tells us that there will probably be an interview object instance in the system-to-be. It does not so obvi-

7. Run-time polymorphism is typically what we usually mean when we refer to polymorphism in object-orientation. Compile-time polymorphism is a term we sometimes use to refer to C++'s and Eiffel's templates or generics. By the time this book is published, Java and C# will almost certainly have generics, i.e. compile-time polymorphism as well as run time polymorphism.

ously tell us that there will be an interview class. It certainly doesn't tell us what methods might go in, say, an `Interview` superclass rather than in `WitnessInterview` or `SuspectInterview` subclasses, as Figure 2.6 tries to illustrate.

Figure 2.6
Clients, types and classes



All of which suggests that we should first figure out which object instances we are going to need; and only then should we figure out what classes will be needed in order to make those instances.

Outside-in rather than inside-out

Good objects result from starting with the clients of those objects – the senders of messages to them – working from those clients to the messages they would send, then working from those messages to the methods that the objects would need in order to answer those messages, and then working from those methods to the instance variables that would be needed to support those methods. This is what we might term *outside-in design*. Outside-in design is good. Inside-out design – the completely opposite sequence – results in overly-complicated data structures masquerading as objects; this is bad, very bad; and it's very, very common.

Figure 2.7
Inside-out objects

One wouldn't say, "I've got these motors and gears I'm really keen to use, I'll just throw them in a casing and then ask sales if anyone wants to buy it." So why do object-oriented designers (and even some OOD methods) do this so often? Why do class designers so often

say, "I'll just take these instance variables, *x*, *theta* and *z* that will be completely encapsulated and that no-one else will ever know about, and I'll take a wild guess that *getX*, *setX*, *getTheta*, *setTheta*, *getZ* and *setZ* methods will be needed"? Figure 2.7 illustrates the difference.

We have a problem at this point, though. We've just finished saying that subject matter entities do not make good suggestions as to the actions of object instances; i.e. the subject matter model apparently isn't going to give us direct clues as to the contents of the object types' interfaces. However, we shall see in a moment that entities' attributes do help us out a little. And we did say that there was some inventing to be done; this is design, after all; we won't get everything handed to us on a plate. Does the inventing have to be a stare-at-the-ceiling-and-wait-for-inspiration kind of invention, or can we nudge it? Yes, we can nudge the process and make it a little more disciplined. Bear with me for two more paragraphs.

What then does happen to those attributes? How do they give us good clues? Do they predict the instance variables? No, they don't. That would be the inside-out way of proceeding. We can't possibly determine, or even hazard a wild guess, at this stage, whether we are going to store a property, calculate a property or ask another object to figure out a property for us. What we *do* know is that if an attribute is a relevant property of a subject matter entity, then we will want to query its object instance counterpart about that property. Thus we get part of the interfaces – we get the *query messages*.

And what happens to the relationships from the subject matter model, particularly those relationships that we will later call the *characterizing relationships*? Are they telling us about stored pointers or references? No, they suggest that the subject matter entities' solution object counterparts will have certain awareness of each other, and will be able to collaborate. And, lo and behold, it is this collaboration that gives us our way of taking the client-centric, outside-in, design route we wanted. The technique is called CRC or "class", responsibility, collaboration. (We have to "laughing-quote" the "class" part of the technique's historical name if we want to be completely pedantic, as we are going to use the technique first to develop *types* via responsibility and collaboration.)

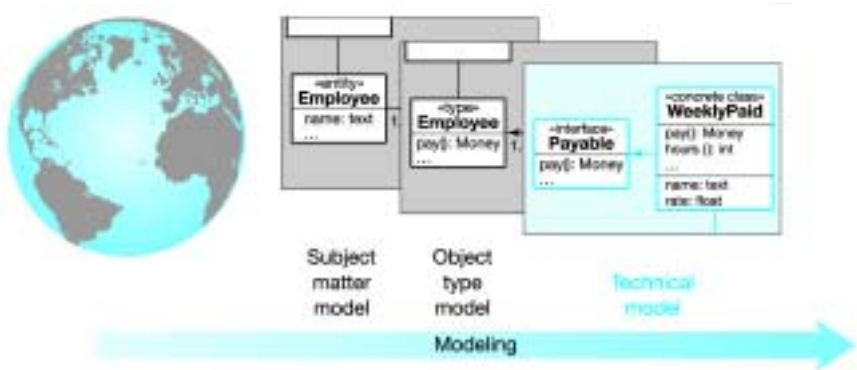
A CRC session takes a required responsibility (typically from the requirements capture), seeks out an object type that could, and should, take the responsibility – something that will be good to put in the type's interface – and then asks, basically using 7 ± 2, if the responsibility-taking object will need to collaborate, will require sub-responsibilities to be assumed by other objects. And such collaborations confirm or invent other message-accepting components of other objects' interfaces. CRC is the disciplined, inventive process we need.

Summary

In the object type model, we let the entities suggest the object instances, we let the attributes suggest the query services – the easy part of the interfaces – and we get the relationships to partly support an activity known as CRC workshops – establishing the difficult part, the invented part of the interfaces, the non-query messages.

2.2.3 Technical model: third model

Figure 2.8
Third model



This section gets quite deeply into the technology. If you are new to object technology you should probably still read the section, but not worry if you feel you're drowning in terminology by the end of it. When we reach the technical model chapters we will go over all this again. Appendix B gives a "thumbnail sketch" of object technology, which might be worth reading before we reach that stage.

Once we know what kinds of objects we will need, we will have enough evidence to begin planning the classes and their connection to the type system, though *abstract classes* and `interfaces`.

This will be the last model that isn't actually executable. We are still abstracting and modeling. We will still be planning our chunks, and drawing diagrams in order to understand and explore those chunks.

It's been a long time coming – and quite rightly – but here at long last are the classes. They weren't in the first model, and they weren't even in the second model.

As we said, there are quite a few different purposes served by classes, so we will have superclasses, subclasses, abstract classes, concrete classes, and perhaps some template or generic classes. Classes will be characterized by their interfaces, the methods that support those interfaces and the instance variables that support those methods. Notice that even when we reach classes, we want them to be designed outside-in.

Notice also the confusing uses of the word "interface". It's a common word in engineering. Many things have an interface of some kind or other – even you and I have interfaces – and a class has an interface – the signatures of its public methods. The term `interface` is also used more specifically in UML, Java and C#. An `interface` in those disciplines (and I will always style it that way, to try to avoid confusion) is a type that doesn't and cannot have implementation. C++ approximates it with a *pABC*, a *purely abstract base class*, which is a class with no data and no code. There is of course more, much more, on all this still to come.

Finally, and later than you might have thought, we will have inheritance coming in to the design, or more precisely, *implementation inheritance* (we also have to be a bit careful talking about inheritance because of all the ways in which it gets used).

Interfaces and Classes

When proposing the object type model just now, it was noted that the class mechanism has come to resemble the white knight and his accoutrements, with all sorts of bits and bobs being glued or bolted on (see *Object orientation rather than class orientation* on page 34). For the purposes of introducing the three models here, we'll stick to a fairly simple picture, one that includes and supports best object-oriented practices.

Think of a spectrum with a concrete class at one end and a pure type at the other end, as depicted in Figure 2.9.

Figure 2.9
A spectrum of object devices

Concrete classes

The concrete end appeared in languages early on, it was fairly obvious, it was fairly easy and it has similarities from language to language. The middle appeared pretty soon and also looks similar from language to language. The pure type end proved to be the most difficult, both technically and conceptually, and appeared later, and can look quite different from language to language.

A *concrete class* defines methods and instance variables. A concrete class can be *instantiated*, that is to say that we can create instances – object instances – of such a class; instances to go in variables[8] or to be pointed at (referenced). A concrete class might be completely standalone. This is common with what we will later call *value objects* or *attribute objects* – objects such as date objects, angle objects and complex number objects. It's likely, especially with *business objects*, also known as *entity objects*, that a concrete class will be in some kind of declared relationship with other classes (or better still *types*) – relationships such as *inheritance* or *implementation*. Concrete classes are simple and fairly obvious, especially when standalone, and if you have already learned your object-oriented programming, concrete classes were probably the first kind of class you learned about.

A concrete class can also act as a type: one can say of a variable, pointer or reference, "thou shalt only hold *RightAngledTove* instances". However, a concrete class isn't a flexible kind of type; and we usually welcome flexibility. Either because of the nature of some languages or because of a very good style guideline still to come,[9] by definition a variable or pointer of concrete class type can only ever give us access to one kind of object. In other words, our object-oriented temple is missing its polymorphism pillar.

8. Or travel as arguments or returns.
9. The guideline is that all base classes should be abstract. This is using C++ phrasing (base class rather than superclass) because of the nice abbreviation that results – ABC – abstract base class.

Polymorphism is information hiding flexibility. One kind of object can say that it needs to be in relationship with another kind of object without limiting itself to being in relationships with objects of only a single class. A *CriminalRecord* object, can say, "Actually, I don't care what kind of crimes my criminal has committed; to me a crime is a crime. Bring on the robberies, the blackmailings, the parking fines, the illegal stills, I'll list 'em all." Figure 2.10 has an illustration.

We would not be able to make a flexible yet type-safe relationship like that work if it mandated the implementations of the related objects; instead it must mandate the interfaces of the acceptable objects – the messages they can receive. This is why, for me, the central feature of object-orientation is the polymorphic use of messages, not the use of inheritance as some authorities would have it.

Figure 2.10
Polymorphism

Abstract superclasses (the ABC)

One kind of device that can act as this more flexible type, in most languages, is the superclass, or base class as it is known in C++. Using various spells and incantations, a programmer can arrange that a pointer or reference has superclass type, yet can point at subclass instances (derived class instances); i.e. instances of classes that have inherited from the superclass. That's the kind of class I've shown in the middle of the spectrum in Figure 2.9. But why abstract?

Well, we like a simple life – if one device does one job we are happy. If one device does two jobs, our life is four times as complicated. And if one device does three jobs, then our life is eight times as complicated. (This is a *factoid*. I couldn't give you the statistics to back that assertion up, but I'm absolutely sure it's true.) What is the job of a superclass? Partly it's to hold inheritable implementation resources – typically for more than one subclass; that's one of its jobs. Also, as we've just noted, any class acts as a type; so that's two jobs for a superclass. If we allowed a superclass to be instantiated as well, it'd be doing three jobs, and, worse, the last job – instantiation – would couple the superclass very tightly to the instantiating code. Any changes to the superclass would have a horribly large impact on the instantiating code. (I will explain that more when we get to the detailed chapters on the technical model, see *Abstract classes* on page 393.)

It turns out that there is no need whatsoever for a superclass to be instantiated, so we don't, and we make it a style rule not to. So all superclasses can, and should be *abstract*. Part of the meaning of "abstract" is "not instantiated". If we use the C++ term, we have the rather nice abbreviation ABC: abstract base class. The style rule is commutative: an abstract

class must be a base class (or it would be of no use) and a base class must be abstract or we get in a mess.[10]

So there is an ABC in the middle of our spectrum. It's partly a type and it's partly a source of inheritable implementation. As we've just said, even with the style guideline just mentioned, an ABC is still rather complicated and highly coupled. We would like to reduce its coupling further if we can. We can. We move on to the pure type end of our spectrum. Read on.

Interfaces

If we have nice, simple, concrete classes doing one job at one end – object making – would it be possible to have something that could just do the job of typing at the other end? That is to say is there something that could be used as a type for variables, pointers or references, but that doesn't have any implementation? In C++, although not in the very first versions, we can bring several mechanisms[11] together into an idiom known as the *pABC*. In Java and C#, that idiom became part of the language syntax, and is the `interface` we referred to earlier. An `interface` is a pure type, and it's splendidly simple: it's a named set of message signatures and nothing else. A *pABC* in C++ is also effectively just a named set of message signatures.

If we then decide that we should nearly always try to use `interfaces` or *pABC*s for variable (and parameter and return) types, and that we should avoid using *ABC*s as types (or instantiating them), then the potentially complicated *ABC* gets a little simpler, and, more importantly, gets greatly reduced coupling. The resulting *ABC* acts only as a type transmitter and as a source of implementation. To illustrate: a pointer or reference would have an `interface`/*pABC* as its type; the referenced object instance at the end of the pointer or reference would come from a concrete class; that concrete class might have inherited some of its implementation from a superclass/base class; and that superclass/base class would have declared that it was a subtype of the `interface`/*pABC*, therefore the instance of the concrete subclass/derived class carried the `interface`/*pABC* type and was therefore allowed to be the referent of the reference. This is illustrated in Figure 2.11.

I've gone into quite a bit of detail there, despite there being a whole chapter on this later on. It's complicated and important so I wanted to give a foretaste. If you're completely new to this, don't worry if you didn't quite get it all.

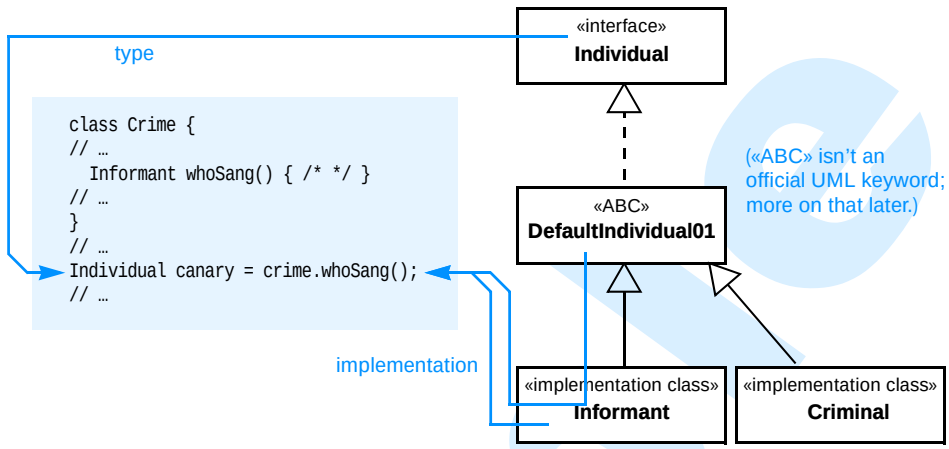
When you have:

- figured out all those types and classes;
- decided where to locate the methods and variables;
- minimized the coupling and maximized the cohesion as best you can;

10. You will probably see many textbooks, particularly programming textbooks, suggesting that superclasses do get instantiated. They say, for example, that a pointer of base type can point at base class (superclass) instances or it can point at derived class (subclass) instances. Well it can; but it shouldn't. More on this in the detailed design chapters.

11. Pointers, base classes, public inheritance, all member functions pure virtual, and no data members.

Figure 2.11
An instance getting its implementation and a type



- used types and messages to support polymorphism correctly, both for today and tomorrow;
- figured out the way to implement the characterizing relationships;

you've done the *structural* component of the technical model.

The CRC technique mentioned in the object type model, continues to be useful in the development of this model. We go on to using CRC to design classes, where previously it was helping us design types. The documented results of the CRC sessions – interaction diagrams – form a partial *behavioral* component of the technical model. It's only partial because, while a common characteristic of a model we deem *structural* is that it can be completed, a common characteristic of a model that we deem *behavioral* is that it can never be completed, it is only ever a sampling.

2.3 A Greater Continuity

One of the nice things about going object-oriented is that there is sufficient continuity from model to model that one can see development as a fairly continuous process. In the structured era that came before the object-oriented era, for example, we had structured analysis, structured design and structured programming, but the adjective *structured*, didn't really have a strong and similar connotation in the different models. Structured development had model discontinuities.

The proposal here then, was to begin with a model founded on a subject matter that exists in the absence of the system-to-be. Our perception of that subject matter gives us the initial form, and some of the substance of, a first model. We then transform that model. The transformation – into the object type model – takes our perception of the subject matter, but immediately and irrevocably departs from reality. Our subject matter creatures – entities we called them – evolve into populations of encapsulated, service-interface-present-

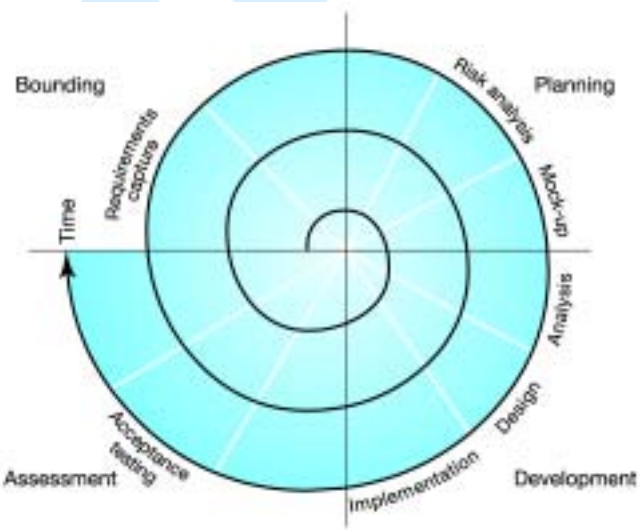
ing object instances that look useful to their clients and similar to each other. Our third model transforms this view of the objects into object technological types and classes. With the evidence of the useful outsides of the instances, we are able to plan the location and nature of the implementations – the concrete classes, their methods and their variables – and the contracts – the type system. In a sense, "all" that remains after that is to add a few curly brackets and semi-colons and we're done!

We should mention a tiny drawback to this continuity. In the old days – the structured era – one got to a clear point where one said, "Right, the analysis is done, let's get those analysis data flow diagrams baselined and published before we march off into the structure charts of design." Not only is it difficult to say where the boundary lies between analysis and design in terms of what should be in an analysis model and what should be in a design model, as was mentioned in the Introduction, but there is more of a tendency today that the analysis model doesn't get published in its own right, or put under configuration management. It *is* still a good idea to draw a line and say, "There, that's our (current) best understanding of the nature of the subject matter, let's get that on the shelf, such that we can come back to it, if necessary, in this project and in future projects."

2.4 Iterative Development

Let's recall a picture that was emerging earlier. Figure 2.12 shows a slightly elaborated version of our development model.

Figure 2.12
The spiral model



This is known, fairly obviously as the *spiral model* of systems development. In contrast with the original ideas for organizing a project of non-trivial size, the spiral model says that we don't usually get something right at our first attempt and that project organization and control should reflect that inevitable human fallibility.

Early suggestions for project phases were usually linear. Sometimes they were described as a waterfall, such as that shown in Figure 2.13, where there was definitely no possibility of going back to an earlier phase, unless you were some kind of software developing salmon, leaping back up the waterfall.

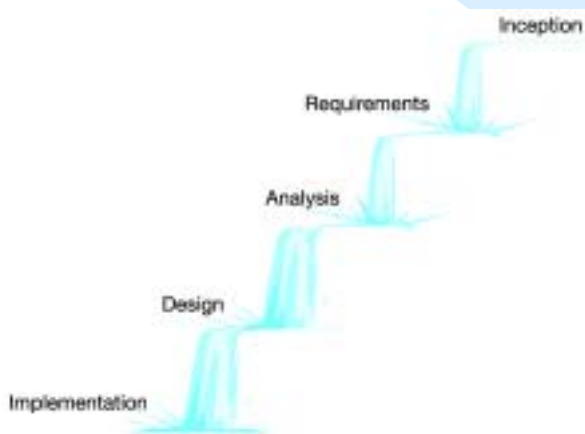


Figure 2.13
The old waterfall model

In those medium-scale and large-scale projects that used a linear model, it was often just a pretence. Secretly the developers would be reworking the fruits of earlier phases, and the project management was even more out of kilter than usual.

Over the last decade or two, reality has crept in. Development is planned and organized to be iterative. We won't go into the details, which are beyond the scope of this book, but will note that *ad hoc* iteration is uncontrollable, so some kind of structure is required and the spiral of two or three, sometimes four, sweeps gives the right compromise. The spiral graphic in two dimensions can be misleading though; it gives the impression that every activity is present in every sweep, and that every sweep gives more and more effort to every activity. Neither of those is true, of course.

Notice that the spiral includes more than the development itself. An important realization that also began to be accepted a decade or so ago, was iterative development *and delivery.* This supplanted the "big bang" delivery model, where the developers would isolate themselves in a bunker for *n* years and then spring out on the appointed day (inevitably actually quite a bit later than it should have been), only to find that the world and its expectations had been poorly appreciated at first anyway and had subsequently moved on as well, and thus rendering the product close to useless. Even if the system had been built right, the right system hadn't been built.

It's much better, if at all possible, to figure out layers of product that can be delivered and assessed while the next layer is being built. Again, the details belong in a project management book or a requirements capture book. We will return to this topic briefly when we take our look at requirements (see *Staged development* on page 75).

What are the impacts for us?

- packaging
- layering, interfacing and malleability
- support for the approach of this book

Packaging is important. We will need to organize and document the deliverable packages and their contents. We will be looking at packaging shortly.

Object technology assumes even more importance than it already had, because of its programming-by-contract nature, with its strong emphasis of, and support of, interfaces, and its reduction of the impact of change.

The final impact I'll mention is that we can feel comforted that one of the important suggestions, and some of the minor suggestions, of this book aren't forbidden from the outset. We will be needing to sort out some objects and to have an idea of the architecture or structures that they will inhabit, quite early on. Only when we have some ideas as to the objects and their collaboration pathways can we successfully apportion processing; and a consideration of the processing might lead us back to doing an analysis iteration after a design iteration.

There is also a "smaller" spiral during the development of the second model – the object type model – and its use of CRC ("class" responsibility collaboration). We will be discovering that you can't do this important design activity without some idea as to associations, but we won't be exactly sure in early iterations just where those associations will get housed. We will postpone worrying about the concrete classes that actually implement relationships in order to focus on good objects through their interfaces; but in order to find those interfaces, we will need to have collaboration and an important source of collaborating objects are those in the association relationships we haven't totally pinned down yet. So early iterations assume the presence of things that will be definitively established and detailed in later iterations.

2.5 An Agile Method

You should be pleased to realize that the proposals here are agile rather than ponderous or onerous; they are low-ceremony rather than high-ceremony. Many of the novel suggestions made, particularly in the analysis phase, tend to reduce the workload and improve the focus. There is a strange inertia–momentum phenomenon with analysis. It's sometimes difficult to

get a project to do analysis at all, but when you do succeed in getting a project into analysis, it's then difficult to get them to stop. The project enters what became known as *analysis paralysis.*

If we combine the danger of analysis paralysis with the possibility that the analysis phase might be creating something that's not actually going to be of any use to the design phase, we have a terrifying waste of time and energy leering at us.

There has been a welcome shift in the development community recently toward wanting methods that are practical and consist of the essentials; methods where the frills, furbelows and displacement activities[12] have been removed or put in an appendix (see Cockburn on agile methods [Cockburn 02] for example). If you're a music fan, and if the methods that had predominated up until now were the equivalent of pomp rock, where bands toured with an orchestra, a choir and 27 square feet of synthesizers, then perhaps we are moving, if not toward punk, then toward a more minimalist school.

When we get to the detailed chapters, you will see that our parsimonious attitude extends to the amount of UML that we deem useful or essential. We will focus on what gives results.

2.6 Other Approaches

(If you are not interested in a discussion of alternative approaches, this section can be skipped.)

2.6.1 Functionality driven methods

One of the ways in which one can classify the different approaches to system development is *architecture-driven* versus *functionality-driven*. This book elects to be driven by a desire for the best possible object architecture, and to ensure that said architecture provides the functionality of version 1, *and is ready to be modified and extended to support the functionality of versions 1.1 to version n.m*. The approach I am describing, for example, includes *use cases*, but it is not driven by them. There are alternative approaches that *are* driven by use cases. First of all, what is a use case?

We are going to be introducing requirements a little later (*Requirements* on page 67). Use cases (or scenarios) are an important part of requirements' descriptions. A use case is quite simply a description of the required system in action. To aid appreciation of the character of the system-to-be, and to help specify what is required of it, we describe a number of exam-

12. Displacement activities are dear to writers. I have just made another pot of coffee and have had to forcibly restrain myself from cleaning a smear from my office window. Roald Dahl said that he would sometimes sharpen an entire jam-jar full of pencils in order to postpone the moment when he actually had to write something. The UML gives us many, many things that we can occupy project time with, without actually accomplishing much. In one project I worked on, it became one of our phrases: "What are you doing?" "I'm pencil sharpening, I'm afraid. Tell me to do something."

ple scenarios of use, each one forming a reasonable and typical unit of interaction with, and unit of work of, the system-to-be. (If this is the first time you have come across use cases, you might like to have a quick read of *Use case scenarios* on page 73.)

Use cases were an important addition, in the 1990s, to development methods. In fact, it's quite astonishing to contemplate that a number of early methods didn't ask what the system would be used for. When use cases were first proposed, largely via Ivar Jacobson and his contributions to the UML, there was quite a polarized reaction to them. At one extreme there were people, for example Steve Palmer [Palmer 02], urging great caution in their use. At the other extreme are the use-case driven methods. I am going to put myself squarely in the middle. I believe that use cases are useful, probably very useful, but they shouldn't *drive* the development process.

The danger, should one decide not to consider use cases, is, I think, fairly obvious: that the eventual system, while possibly technically excellent, might find itself solving the wrong problem. Barry Boehm put this very nicely: you should build the right system, and you should build the system right. Building the right system mostly concerns requirements and analysis; building the system right mostly concerns design. To me, asking what use will be made of the system-to-be is very worthwhile. No matter how good the structural diagrams – entities, relationships, etc. – it's too difficult for the average mortal to form a good picture of what the system is actually *for*, if structural diagrams are all they have.

On the other hand, the danger of being driven by use cases is that the eventual system, while solving the right problem might have a poor object architecture, and not survive for very long in the face of correction, modification and extension.

Firstly, use cases in and of themselves are not particularly good vehicles with which to choose object candidates. Particularly – for those of you familiar with use cases – while actors might make good entities, they equally well might not. And symmetrically, many good entities might never have appeared as actors. Use cases are a requirements technique; not an analysis technique.

Secondly, use case driven development carries a strong and unnecessary risk, a risk that all functionality-driven approaches, including structured design, face: it's too easy to come up with an algorithm that satisfies today's needs, but that balks at modification for tomorrow's needs. Functions are less stable and more volatile than entities.[13]

The right objects support today's *and tomorrow's* needs, better than the "right" functions do. (Otherwise why have we bothered to become object-oriented? An object is more than just a function collecting device.[14] Being both object oriented and use case driven is perhaps contradictory.)

One can also look at the past and note that large-scale software has had little success with functionality-driven approaches. A good architectural structure gives the universal backdrop against which all else can be considered. I am reminded of another piece of his-

13. This may be, in part, why databases for information storage and retrieval are one of the more painless ways of using computers.

14. It's a self-processing creature with meaning for the problem at hand, and with individualized and persistent state.

tory. Early programmers had to be constantly aware of performance and continuously strive to write the tightest code. Scale eventually forced us to acknowledge that there was a locality phenomenon. Performance problems were nicely localized in well-structured software, and could be fixed; but -ability problems (understandability, maintainability, communicability, etc.) were never nicely localized in fast software, and could not be fixed.

The approach that I am suggesting is to write good use cases as part of the requirements, then to put them at arm's length while developing the first model – on the subject matter entity basis outlined earlier in the chapter – and finally to bring the use cases specifically into play to drive the CRC activities that are central to the second (and third) models. We can also use the use cases during requirements coverage checking. That way we ensure that we have built the right system – solving the problems at hand today – as well as having built the system right – it being amenable to modification and extension into a long-lived, happy, investment-returning system.

Finally, it isn't all that uncommon for a system to have but a single use case. Several years ago, Bistomotive Systems had to do a trail-blazing fingerprint matching system. Here is how one interview went.

"Tell me what you would do in a typical use of this system."

"We take an image of a print, scan it into the system and ask for any matches, ranked."

"And what else would you do?"

"That's it. We match fingerprints from the database."

"Aha! Doesn't that mean you add fingerprints to the database?"

"No, that's a separate system. There are legal issues surrounding putting prints into the database."

"Delete them?"

"Nope. Another system."

"I see …". Our interviewer walks away, nervously contemplating basing the entire design on one sentence's input.

We couldn't possibly contemplate driving the entire development from that single use case. Nor, for that matter, could we concern ourselves exclusively with "requirements analysis". We have to find something else as input. And of course, we have the subject matter of biology, chemistry, fingerprints, law and pattern matching.

Occasionally, typically with information-rich systems, it is not actually clear as to exactly how the system will be used. Bistomotive Systems were also asked to help with a system that would improve civil and military response to emergent threats. (I'm sorry if that sounds a bit vague; need-to-know basis, confidentiality agreements, and all that I'm afraid.) The thing was that no-one could say how the system would be used. It was pretty clear that getting the right data, in the right model, and coordinating with other systems, such as ports, airports and CCTV, would help. It was also clear that only by actually using the system would the best way to use the system be discovered. Also it was clear that priorities, the nature of the available intelligence and the nature of external systems would change month by month. Therefore, while there were a few useful use cases, the bulk of the work had to go into ensuring that there was a good and malleable object model.

The mission of this book is to be practical, not partisan, and there is another consideration, which is, in some sense, the opposite of the above. Some systems are function-rich and some systems are information-rich; you may recall that triangular "system space diagram", presented earlier (page 24) as one way of attempting to classify systems. If a project encountered a function-heavy system and found it overly challenging to follow an object-oriented approach, then a functionality-driven method might be their best chance of success.

I have encountered at least two projects with this flavor – as it happened both concerning work in three-dimensional spaces. The first was an airspace simulation and the second involved finding patterns – lines and curves – in points in three-space. In these systems, it was not at all obvious what the entities and objects were. While more experienced, object-oriented modelers might have been able to start by abstracting some useful entities, it was found to be challenging at the time. Rather than abandoning analysis and design and heading straight for the compiler, use case driven and function driven approaches gave them a way in. It must be said, however, that the eventual object model was ready for some serious refactoring pretty much straight away.