2 Modelling a process

SOME HEALTH WARNINGS

We have now looked at our needs as process modellers and at the aspects of the real world that we shall want to see in our models if we are going to satisfy those needs. It's time to look at the language for our process models.

But I want to start with two warnings.

We are going to begin by modelling a single process. But this begs a rather important question: how did we decide that this process exists? Put another way, how do we know it makes sense to call this particular 'pile' of activity a 'process'? In any organisation we might guess there will be many different processes. We might guess that they are related: that the process for purchasing goods is in some way related to the process for dealing with invoices; that the process for opening a new bank account for a customer is in some way related to the process for checking a person's credit; and so on. So how did we chunk all the activity in the organisation into those processes and then choose this one to model? These are questions we must, tantalisingly, leave unanswered until Chapter 6. Once we have the language necessary to model a single process, we can look at the different sorts of relationships that can exist between processes, and then build up a theory of 'chunking' that will allow us to answer these crucial questions.

This introduces a danger: if you don't read further than this chapter you may go off and start modelling things that you think are processes but which could only be called 'collections of activities', collections that don't have the coherence that exists in the real world. The axe man cometh.

That's the first warning: *be sure that what you are modelling is a process*.

Here's the second warning: *there is no single model of a process*. Our viewpoint will vary as our motives vary. If we are interested in why a process seems to bottleneck in certain areas, we might want to model the process from the point of view of how work is allocated to individuals. If we are interested in how the functional subdivisions of the organisation help or hinder the flow of a transaction through a process that crosses the functional boundaries, we might want to view the process in terms of those boundaries and the interactions across them, without worrying too much about *how* each function does its work.

There are as many models of a single process as there are viewpoints that we might want to take. The perspective taken, what is left in, what is left out – all these decisions rest on our judgement as modellers. Again, no simple rule will say what perspective we should take in any given situation, though in later chapters we shall draw up some guidelines. This is a point that I shall return to many times in the book. Relevance is in the eye of the modeller, too. A model is 'right' … if it helps. It helps if it *reveals* things we want to know, or helps us answer

questions, or can be analysed, or can be adjusted to test proposed changes, or simply aids understanding. We shall see in Chapter 8 that the most important thing at the start of a modelling activity is to be clear about the *purpose* of the model. The model itself cannot do anything – it is a tool that will work well in the right hands in the right situation, and badly otherwise.

Indeed, we might take several perspectives corresponding to the different perceived purposes of the process we are looking at. In our work for a pharmaceutical company, Tim Huckvale and I initially prepared two models of a particular process, essentially seeing the same process from two perspectives: one from that of the scientists doing the science necessary to take a new drug compound to market, another from that of the management pushing the development of the compound through the various stages of process scale-up and trials whilst weeding out those compounds that do not offer future success. In Soft Systems terms, these views could be considered to be *holons* which we 'put against the world' in order to learn about it (see for instance *Soft Systems Methodology in action*, P Checkland & J Scholes, Wiley, 1990; *Practical Soft Systems Analysis*, D Patching, Pitman Publishing, 1990). Each corresponds to a different idea of the purpose of the process (/system). The Research Chemist saw the purpose of the process we were looking at to be to produce a way of making the drug compound safely in the required quantities and to the required purity; the Regulatory Affairs Group saw its purpose to be the production of the information and audit trail of development which would satisfy the industry regulators; the Clinician saw its purpose to be the timely production of sufficient quantities of drug for the clinical trials they were planning; the senior management saw the purpose to be either to get a successful-looking compound to manufacturing as quickly as possible or to drop an unsuccessful-looking compound as early as possible; and so on.

If I walk into a map store and ask for a map, I'll hope I'm asked 'What do you want the map for?' I could have a number of reasons:

- To walk from Paddington railway station to Victoria railway station in London. I need a map to help me find my way around. In particular, it will need to be a fairly detailed map as I am interested in being able to trace my steps through London's network of streets. I shall need street names but won't need to know if streets are one-way for vehicles.
- To drive from Bath to Birmingham. Again I'm looking for help in finding my way around, but now I shall need a map on a different scale: one that shows me the broad shape of the country and the major roads will do. I won't need anything showing country lanes or small villages or the local topography of the countryside I shall be passing through. It would be useful if the map also had outline street-maps of Bath and Birmingham.
- To allow me to agree with someone on a spot in London where we will meet. In this case my reason for needing a map involves someone else: we will use the map as an agreed definition of something, we can agree on where we will meet in a pre-defined surrounding. If we are meeting at a pub, it would be good if the map showed the locations of pubs.
- To agree on a boundary to be drawn on the sale of some land. Here, we want to define something not already defined, and to place it in some larger context, with references to existing features.
- To decide where to move an existing footpath. In England, a footpath may well go back many centuries; its end-points and its route will have been determined by needs from the

past. Changes in surroundings might make moving it sensible. We need to know what the options are. We will be working at quite a small scale.

- ☞ To decide where to route a new road. If I am planning a completely new road from *A* to *B* I shall need to explore the options and the impact that each will have on things that I want untouched. I want a map that shows topography on a large scale, at a level of detail that allows the exploration of impact.
- ☞ To record the position of underground cables. Here, as a cable company say, I shall be maintaining my own maps of where my cables run relative to the infrastructure of the town. I need these maps to allow my staff to find the cables in the future.

The parallels between maps and process models should be clear. Before we carry out some process modelling we will need to know quite clearly what we want from the model so that we can choose the scale it is at, and the sorts of detail it shows.

KEY POINTS

Before modelling a process, be sure it is a process – always start with a process architecture.

There is no such thing as *the* model of a process.

Worse, all models are wrong … but some are useful.

To get to useful answers we must ask the right questions.

Our choice of model is guided by the questions we want to answer.

Before you start, answer the question 'What questions do I want to answer with this model?' and write your answer on the wall.

THE ROLE ACTIVITY DIAGRAM

A *Riva* process model takes the form of a *Role Activity Diagram* (RAD). A RAD shows the roles that play a part in the process, and their component actions and interactions, together with external events and the logic that determines which actions are carried out when. So, it shows the activity of roles in the process and how they collaborate.

I have summarised the notation for RADs – the various sorts of blobs – in Figure 2-1. This is the language we shall use. In this chapter, we shall look at each of the elements of this language, and take each one in two steps: how to use it diagrammatically, and what to do in different modelling situations. (The little spring-shaped symbol means 'don't care'. It is a sort of pictorial ellipsis … and you will find it used a lot in sample RADs throughout the book. If it appears at the start of a thread it means 'We don't care how we got here.' If it appears at the end of a thread it means 'We don't care what happens after here.' If it appears in the middle of a thread it means 'We don't care how we cross this gap.')

Figure 2-1 – The RAD notation

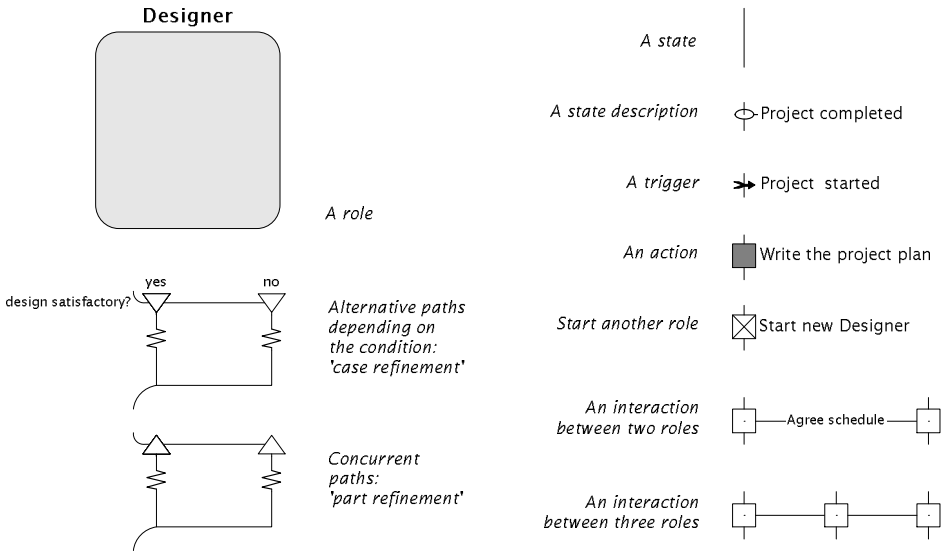
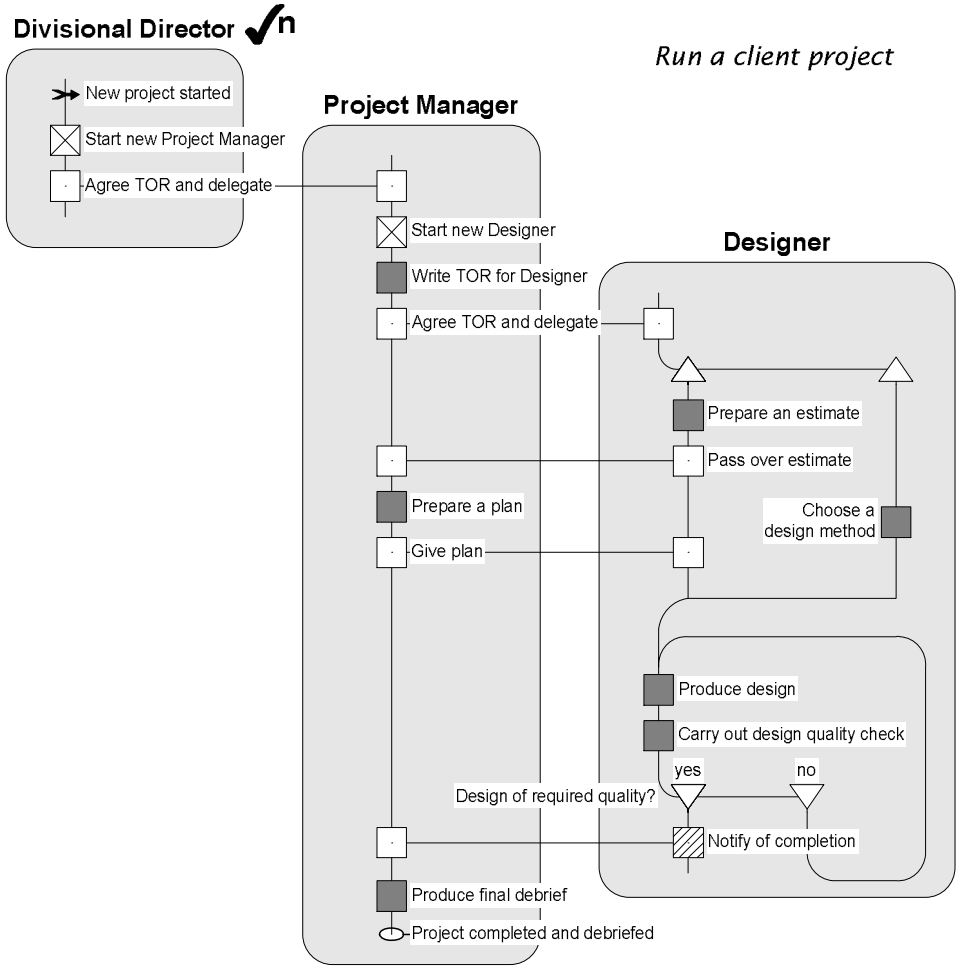


Figure 2-2 shows a RAD for a simple process, to give you a feel for what a complete RAD looks like.

Figure 2-2 – A RAD for a simple process



A RAD represents the whole of a process as far as we wish to capture it. Somewhere on the RAD we name the process it is modelling. This might be Develop a Software System for a Client, Develop a Portfolio of Products, Arrange a Payment of Benefits, or Manage the flow of Customer Queries. Remember, we haven't yet discussed just what constitutes a process, or how to decide what goes in one process rather than another, or how processes fit together – all of this will come later.

Let's now look in detail at the RAD notation and how we use it to capture the concepts covered in Chapter 1.

(Computer tools that support the preparation of RADs may use slightly different symbols but the shape is of no importance. It is the *meaning* we attach to those symbols that is important, and it is that meaning that this chapter addresses.)

REPRESENTING ROLES

Each role in the process is represented by a shaded block with rounded edges. Everything the role does appears in that box, and since only roles can do things, everything on a RAD is inside a role box. In Figure 2-2, there are three roles with the names *Divisional Director*, *Project Manager*, and *Designer*. All actions and interactions take place within those three roles. We can turn this around and say that everything is done as part of carrying out some responsibility or other.

The name of the role appears immediately above or below the block, whichever is convenient.

We sometimes draw a single role as a number of separate shaded blocks if there are indeed separate parts of the role and it makes the RAD easier to draw. Sometimes, as another modelling convenience, it is handy to let the boxes of different roles overlap, provided of course there is no ambiguity about what is in which role. Where we do overlap them we use different shading to distinguish the two roles – see Figure 3-5 as an example of both of these modelling conveniences.

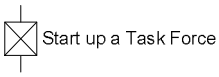
In Chapter 1 we examined several different sorts of role: unique functional group, type of person, etc. Our notation does not differentiate these graphically.

Although each of the roles in Figure 2-2 consists of just one 'thread' starting at the top of its grey box, a role might well consist of a number of separate threads corresponding to different things that it does. We shall see more of this later.

Representing new role instances being started – role instantiation

One role instance can *instantiate* another role, i.e. start a new instance of that role: this action is indicated by a square with a cross inside it. In Figure 2-2, the *Project Manager* role instantiates the *Designer* role. The caption against the crossed box identifies the role being instantiated.

Figure 2-3 – Instantiating the *Task Force* role



This idea of instantiating a role is of course a rather abstract one. In Chapter 1, I equated it with the idea of 'creating an area of responsibility', something separate from giving that responsibility to a real person, which, in *Riva*, we see as allocating an actor to a role instance – what we call 'casting', to follow the theatrical metaphor. We probably won't want to write *Instantiate Designer role* or *Instantiate the Task Force role* as the caption to a crossed box on a RAD. We are more likely to say things like

Start a Designer role

Create the responsibility for managing the project

Set up a Task Force

Get a Task Force going

Strictly, we should not say *Appoint a Task Force* as that really is about casting actors for the role instance. Indeed, whenever we instantiate a role, we have a modelling decision to make: do we want to model the casting of an actor to carry out the new role instance? Like all modelling decisions, the answer will depend on the purpose of the model. If we decide we do want to cover that aspect, we can expect the casting of the actor to be done after the instantiation: minimally an action such as *Nominate person to manage new project*, but potentially an interaction with other roles to come to a choice, or even with other processes responsible for resourcing.

In a RAD we have no separate symbol to represent the 'ending' of a role instance once its work is done. If we need to show this explicitly we simply use an action labelled something like *Close down this Task Force*.

Representing roles with pre-existing instances

We have seen that some role types have pre-existing instances: instances that are 'in place' when the process starts. For instance, we might expect that when the Handle a Customer Complaint process starts there is an instance of the role *Customer* already in place. It is important for us to distinguish those roles graphically for a very good reason: when we look at a RAD we want to be able to tell which roles have instances and hence where process activity can start. If a role does not have an instance in place when the process starts then there can be no activity to do with that role. The role must be instantiated first.

To mark a role with a single pre-existing instance we place a tick ✓ next to its name. If the role has exactly four pre-existing instances then we place the number *4* next to the tick: ✓ 4. If it has an indeterminate number of instances we mark it with '✓ n'. See Figure 2-4. If a role has no tick against its name we know immediately that, when the process starts, there are no instances of it and we can therefore expect to find it being instantiated by another role somewhere on the RAD.

Figure 2-4 – Roles with pre-existing instances

Choosing and modelling roles

Different types of role

We know that roles come in different flavours. Does it make sense to have differently flavoured roles in the same model? The answer to this question, and to many similar ones, is 'Yes, if it makes sense'! When we model a process, we have a purpose in mind. That purpose will tell us what makes sense. Let's start by looking at the different types of role and when we might use them.

- A unique functional position or post.

  Take the role *Chief Executive*. If we decide to have that role in our model we are clearly saying 'These are the responsibilities in this process that are placed on the desk of the person with the title "Chief Executive".' It's highly likely that the Chief Executive will have a host of other responsibilities in other processes. But a particular process model only shows their responsibilities in *this* process. The box marked *Chief Executive* in this model is not a full definition of the responsibilities of that post.

  Since such a role has only one instance we can expect that instance to be pre-existing and hence the role name will have an accompanying tick.

- A generic functional position or post.

  If all the Divisional Managers are in place when the process starts then we will put a tick against the role name. If we know there are six Divisional Managers then we can put a *6* next to the tick; if we don't care or only know that there is at least one Divisional Manager when the process runs, then we put an *n* against the tick.

  If there are no Divisional Mangers at the start of the process, then their creation must be part of the process.

- A unique functional group.

  The situation of a unique functional group is similar to the unique functional post: there will almost certainly be a single instance at the start of the process, and hence a tick next to the role name.

- A generic functional group.

  Generic functional groups work like generic functional posts. An example might be *Retail Branch* in the process Prepare the Annual Sales Forecast. There will be a number – possibly indefinite – of them at the start of the process, and we shall tick the role name appropriately. This sort of functional group typically has a more-or-less permanent existence – unless of course we are looking at the process for deciding to open new retail branches and close existing ones.

- A generic type of person.

  Customer is the typical generic-type-of-person role. In most situations, the role will appear with a tick indicating a single pre-existing instance.

- An abstraction.

  Here the situation changes. Let's take an example: *Project Managing*. With abstract roles like this, we are naming the contents of the box: 'The set of responsibilities represented by this box we will call *Project Managing*.'

  The term 'Project Manager' is ambiguous in that it is used in different ways. Like some organisations, we might use it as a synonym for *Project Managing*: an abstract, transient role, whose instances only last as long as the projects they manage, each being associated one-to-one with the project. I could say 'I am the project manager of the Battlebridge Project.' On the other hand we might use it as a badge, identifying people qualified to act instances of the

role *Project Managing*: I might say 'I'm a Project Manager' meaning that I get given projects to manage.

A more pointed example might be *Large Claim Approving*, where we are labelling the actions and interactions that carry out the responsibility of approving large claims. We have abstracted the role away from the organisation and its structure – we are not saying what post a person must hold to do these actions and interactions, or what qualifications they must have – we are simply labelling the responsibility.

An important abstraction that we shall see more of later is the sort that is also transient in the same way that a Project Team can be. Suppose I am a customer and make a complaint. In essence I generate a responsibility to handle my complaint. It is very common in *Riva* to identify that responsibility as a role which is instantiated when a complaint arrives, deals with that and only that one complaint, and vanishes once that complaint has been dealt with. If we have 127 complaints being dealt with at a particular moment then we will find 127 instances of the role *Complaint Handling* in the organisation. The moment a complaint is closed there will be 126 instances, and if two new ones arrive the count will go up to 128.

Abstract roles might or might not have pre-existing instances on a RAD.

(Ticking pre-existing roles is obviously important when it is not obvious which roles have pre-existing instances and when we need to know. In the many snippets of RADs in this book, I have not always used ticks, in particular where the situation is obvious or does not matter.)

Abstract and concrete roles

The last type of role in the above list – the *abstract* role – is important. We saw in Chapter 1 that such an abstract role is almost a definition of the responsibility itself, rather than a label of a post that typically gets to carry out that responsibility. In a sense, we are getting closer to the role itself when we think of it in such abstract terms. A recurring theme will be the difference between our *intent* and the *mechanism* we use. We can model a process in terms of intent or of mechanism. For instance, if you ask me what I am doing I might answer 'I'm pressing keys on a keyboard': it's true, I am. But you might have expected the answer 'I'm writing a book.' I'm doing that too … and the mechanism I am using is pressing keys on a keyboard. My intent is to write a book; I am doing it by pressing keys.

This distinction crops up when we choose the roles in a RAD. Do we want to talk about mechanism or intent? Is the mechanism that the *Finance Director* is the post that actually approves the payment of large invoices? Is the intent that the role executes the responsibility for *Approving large invoices*? It is probably both, so we have a modelling decision to make: *Finance Director* or *Approving large invoices*? The answer as ever will be obvious as soon as we remember why we are drawing this RAD. If we are preparing a process model as a work instruction then we had better be very specific about who does each job: we shall choose *Finance Director*. If we are trying to get inside a process and model what is really going on, or we are designing a new process and have no preconceptions about who does what, we shall choose *Approving large invoices*. But we shall see more of this decision and its answers in later chapters when we look at using *Riva* in different situations.

Committees and meetings as roles

It's often the case that a corporate body such as 'The Board' acts as a single role that performs various actions, monitoring, acting as an approval authority, planning, etc, whilst at other points in the process the individual members act in their own right: CEO, CFO, CIO, etc. The Divisional Directors might act collectively in one role – *Divisional Management Committee* – and individually in their own right in the role *Divisional Director*. In the latter case we might expect a Divisional Director to have an interaction with the Divisional Management Committee in order to submit a divisional plan as input to the corporate plan. If Shirley is a Divisional Director she will act both of the roles in that process, putting on the right hat at the right moment. She has two hats: one as *the* actor of *an* instance of the role *Divisional Director*, and another as *an* actor of *the* single instance of the role *Divisional Management Committee*.

Equally, it is sometimes useful to regard a regular meeting as playing a role in a process, particularly if it has some executive responsibility. A meeting might be ordained to happen monthly in order to agree the priorities of the coming month's activity, or to approve the expenditure of a department, or simply to ensure an exchange of information between the attendees. Such a role, e.g. *Monthly Planning Meeting*, might therefore appear on a RAD with a ✓.

People as roles

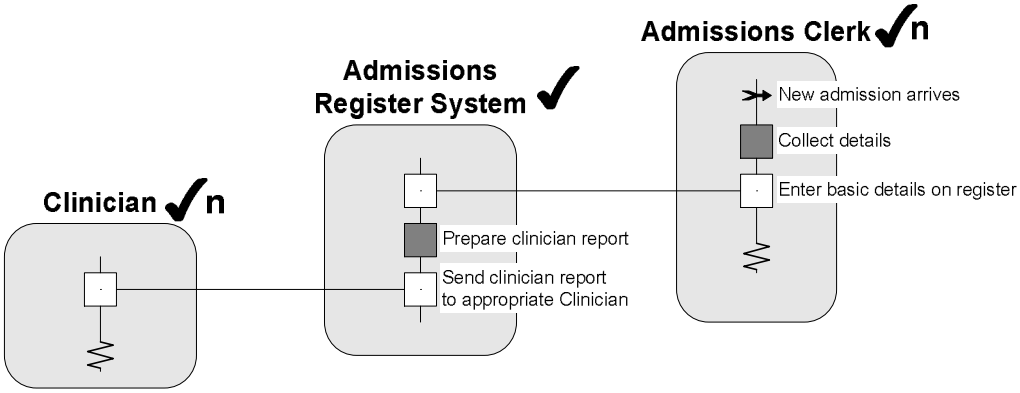
In the same way that a named department can be seen as a role, so can a named person. If we are modelling the Respond to Customer Complaint process in a very concrete way, and Mary is the one who is responsible for calling customers who have left messages, then we can feel quite at ease equating her with that responsibility. On the other hand, if we wanted to 'stand back' from the process and take a rather more abstract view of it, then we would probably not want to equate responsibilities with their current actors, and the role called *Mary* might appear as *Customer Recall*, say.

Computer systems as roles

We saw earlier that a computer can be an actor of a role (instance): in the work of the Accounts Department we will find people doing things (handling purchases, invoices, orders, cash advances, etc), but we might also find computers doing things such as automatically preparing lists of aged debtors each Monday. It may be that an information system running on a computer plays such a large part in a process that we could consider it as having a role of its own. There would be one instance of that role and, trivially, one actor: the box of tricks itself. Other people-acted roles would of course have interactions with it, either to put data in or get data out. There is a sense of course in which we cannot really say that the computer system 'takes on the responsibility' of doing whatever it does, but showing a significant system as a role can be a useful modelling choice.

Although we have not yet fully explored the RAD notation, Figure 2-5 should be readily understandable, and it shows a computer system which is the role *Admissions Register System* with Clinicians and Admissions Clerks interacting with it.

Figure 2-5 – A computer system as a role



KEY POINTS

A role is represented as a labelled grey box.

The name is annotated with or without ticks depending on whether it has or does not have pre-existing instances.

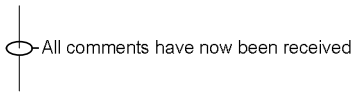
There is a rich variety of role types, including committees, meetings, and even computer systems.

REPRESENTING ROLE STATES

The vertical lines between blobs *within* a role are more than just ways of connecting the blobs. They represent *states* which the role can be in. Our understanding of a RAD is greatly enhanced if we see lines as *states* rather than just 'flows' from one action or interaction to another. This will become more and more apparent as we look more carefully at RADs and the way processes work.

Sometimes on a RAD we want to say what state the world is in at a particular part of the process; in other words we want to label the state. We do this by simply putting a little 'sensing loop' around the state line and annotating it, as in Figure 2-6. Similarly, in Figure 2-2, the final state of the *Project Manager* role is *Project completed and debriefed* (which is probably the goal of the whole process).

Figure 2-6 – Labelling a state



In real life we are quite used to the notion of state, even though we might not recognise it: 'How are you getting on with my expense claim?' is another way of saying 'What state has the processing of my expense claim reached?'. We happily ask 'Has authorisation to proceed been given yet?', or 'Has the Finance Director given his approval yet?'. In these too, we are asking about the state the process has reached. And someone answering the question will say something like 'Well, it has reached the desk of the Finance Director, but she hasn't had a chance to approve it yet,' or 'It's waiting for the Divisional Manager to provide some further figures.' We typically talk in terms of how far each role has got with the matter, especially if things have come to a halt on someone's desk!

Choosing and modelling states

History and potential

The first thing to say is that we do not need to label *all* the states in a RAD. This would be impractical and not useful. As ever, we label the ones we want to label, those that help us with our purpose in drawing the model.

We would label the *starting* state of a thread if there is something about the initial state of a thread that is important to the role or to its understanding: *Written complaint in hand*, or *Nothing yet recorded on the database*, or *The project has already been approved*. Some states in the middle of a thread in a role can also be important in some way: *All necessary materials now in hand*, or *The file can now be closed*, or *Everyone must now be informed*.

Note how some of these state descriptions say where we have been, and some say where we are going. In other words, we might summarise things up to this point: *All necessary materials now in hand*. Or we might say what is now possible or required: *The file can now be closed*, or *Everyone must now be informed*.

These dual aspects of a state – history and potential – reflect the fact that one action's post-condition (the past) can be another's activating condition (the future). When we label a state we might wish to signal either history or potential, or both.

Goals as desired states

In Chapter 1 we saw how goals are important in modelling processes. A goal is simply a desired state. The goal is reached when the state is reached.

Let's take the example of a process called Handle a Request for Quotation in an insurance company. When a customer requests a quotation for insuring a particular risk, the (main) goal of the process is to provide that customer with that quotation. We shall have achieved that goal the moment the customer has the quotation in their hand. So, in the role *Customer*, we should expect to find the state *Quotation in hand*, say, at some point.

We might be tempted to think that this marks the 'end' of the process: the main goal has been achieved. However, it might only be the end of the process as far as the customer is concerned. There may still be work to be done at the insurance company's end: database records to be updated, archives to be made, audit trails to be secured, information about the quotation to be added into the overall risk profile of the company, and so on. We might think of each of these as a 'minor' goal of the process: a further state that must be achieved before the process as a whole can stop. As such, it would be appropriate to label the states that correspond to

these minor goals– they will mark the 'ends' of various threads of activity that are raised by the single request.

These major and minor goals are *point-wise goals*: we can identify the points (states) in the process where they are deemed to have been achieved.

As we saw in Chapter 1, there is a second sort of goal: the *steady-state goal*. Let's take as an example the process Bring a New Product to Market in a software product company. We could imagine that a goal of this process is that the Marketing Department always knows the latest key features of the product. This is a steady-state goal: we want it to be true all the time. By definition, therefore, we cannot put our finger on a single point – or state – in the process model where this goal is achieved, so it is meaningless to try to *model* such a goal. Instead we will want to ensure that the process maintains that goal by *design*. This means knowing what in the process can perturb things (e.g. the software designers making a major change of functionality), how we could detect whether the Marketing Department was out of date, and how we could correct things if they got out of date. This is of course a design issue, rather than a modelling issue, so we shall pick it up again in Chapter 11.

Multiple outcomes of a process

When we come to examine the logistics of process modelling in Chapter 8, we shall look more carefully at the importance of establishing process goals at the start of a modelling project. Indeed, we shall widen the question to one of establishing process *outcomes*, some of which might not be goals. For example, we might have a process to develop a new pharmaceutical drug and take it to market. The *goal* could be said to get the drug to market, but more often than not the actual *outcome* is that the drug is withdrawn and the project is closed. Project closure was not the goal of the process, though finding out *as quickly as possible* if a drug will not be successful can be, in one sense, a goal.

So when we look for outcomes we may well find several for a single process. Let's take the example of a process to Handle a Customer Complaint for a retail store. We might brainstorm several possible outcomes:

> The customer receives replacement goods and agrees the complaint is closed.
>
> The customer receives a refund and agrees the complaint is closed.
>
> No agreement can be reached and the matter is referred to the industry ombudsman.

So we would expect to see each of these somewhere in the model of the process:

- The first outcome would correspond to the state after the receipt of notification from the customer that they have received the replacement goods and agree the complaint is closed. This state might perhaps be in a role such as *Customer Service Assistant*.
- The second outcome would correspond to the state after the receipt of notification from the customer that they have received the refund and agree that the complaint is closed. This state might perhaps be in the same *Customer Service Assistant* role.

☞ The third outcome would correspond to the state after the matter has been referred to the industry ombudsman. We might find this state in a role such as *Customer Service Manager*, say.

KEY POINTS

We mark interesting states with appropriate captions.

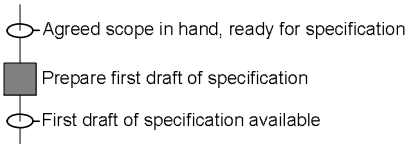
A state can express history (where we have been) or potential (where we can go).

Some states represent goals, or sub-goals, or outcomes.

REPRESENTING ACTIONS

An action in a role is modelled with a small black box, suitably captioned, as in Figure 2-8. The action can start when its activating condition is met. When it is finished its post-condition is true. In Figure 2-8, we have shown the activating condition and the post-condition, by annotating the states before and after the action, but remember that we shall only caption states where it's useful.

Figure 2-8 – An action in a role with its adjoining states

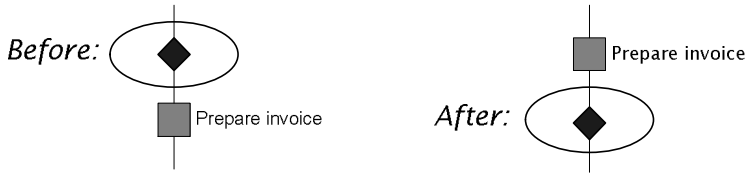


The fact that an action is shown as a black box is significant: it says that, as far as we are concerned *in this model*, we do not care how this action is carried out, so long as the desired state is reached after it. The question naturally arises as to whether one can 'decompose' or 'open up' a black box. This is an important issue which we shall cover in Chapter 4.

'Running' a thread

When a role is instantiated we can think of the new instance starting with a 'token' sitting on the initial state of each thread. As the process unfolds and the role instance proceeds through its actions and interactions, the changes of state are marked by changes in the positions of tokens on the states, what we shall call the *marking* of the model (to borrow some terminology from Petri Nets). For instance, suppose a role instance is in the state shown in the left-hand fragment of Figure 2-9.

Figure 2-9 – A role instance thread before and after an action



The token shown as a lozenge sitting on the state line before the action *Prepare invoice* indicates that the (/a) next thing that the role can do is carry out that action. Nothing else is needed for the actor to start the action. The state in front of the action represents what we have called its *activating condition*, that is, the condition of the role instance which will cause the action to start (strictly, to be instantiated). A token sitting on a particular state can be thought of as representing the potential future behaviour of the role instance.

When the action does start, the role instance is in the state of carrying it out (and the token essentially disappears, though you might think of a token sitting in the action box). When the action has finished, a token appears on the state immediately following the action to indicate that the role instance is now in that state. The state after the action represents what we have called its *post-condition*.

We shall use this idea of tokens flowing to illustrate what happens when a RAD 'runs'. It is important to understand just what happens when a RAD 'runs' so that we can make correct interpretations of the model for comparison with what happens when the real-life process runs, or to see how a proposed process would look when it does run. Above all, it will help us check the concurrency that we have modelled, both within one role (instance) and across the process as a whole. This is one of the features of RADs that distinguishes them from flowcharts and swim-lanes, where serial threads are the order of the day. We shall examine process concurrency in detail in Chapter 3.

Choosing and modelling actions

Actions as black boxes

An action is shown as a black box and it is useful to think of it in exactly those terms: 'This action is atomic, we are not saying how it is done, or what is in the box.' Anything we do wish to say about the action appears in the caption we attach to it. We are not likely to write an essay as a caption but we might have a sentence or two. For example, we might write

> Get deployment document, operational review, and support documentation signed off.
>
> Update file.
>
> Prepare annual forecast.
>
> Decide whom to involve in forthcoming project review.

When we choose to 'bottle up' some activity and represent it as a single box we are making an important modelling decision about the amount of detail that we want to get into. *Get deployment document, operational review, and support documentation signed off* may be regarded

as atomic for a particular RAD but it is of course potentially a 'big' complex action. My diatribe against hierarchies in Chapter 1 will have warned you that I am against decomposition as a basis for modelling. So, I would be very, very cautious about wanting to 'decompose' a black-box action – exactly what does 'decomposing an action' mean? This is in fact a very difficult question, one that process modelling methods that use decomposition ignore, with the result that they falsify their models. The question is important enough that we shall leave it to its own treatment in Chapter 4. Meanwhile, in answer to the question 'How much detail should I go to in my process model?' I would simply answer – of course – 'However much is useful given the purpose of your model.'

Concrete and abstract actions

I pointed out the difference between intent and mechanism earlier on, when discussing roles. A similar view can be taken of actions: we can model either the intent of the action, or the mechanism used to carry it out, or both. For example, we might caption an action as *Complete screen 24A* – this gives us no idea at all of what is intended by this action, but it does tell us what to do, what mechanism we should use. On the other hand, we might caption the same action *Record new customer details*, which tells us what we are trying to achieve with this action – a record of customer details – but gives us no indication at all of how to do it. The first caption treats the action in a concrete sense, the second in an abstract sense (we are abstracting away from mechanisms to what is intended). Of course, we might decide that the purpose of our model is best served by putting both the intent and the mechanism in the caption: *Record new customer details using screen 24A*. A purely mechanistic caption might be sufficient in a RAD serving as a work instruction or procedure. A purely abstract caption might make sense if we are using our model to really get an understanding of what the process is all about, irrespective of how we go about it physically. The combined caption could be used in training material where it is good to tell people what to do and why they are doing it. As ever, a process model must be fit for purpose.

Actions need verbs

Finally, note how the caption for an action starts with a verb. It is activity that we are describing, after all, so it's a good discipline to ensure that there is an active verb somewhere.

KEY POINTS

An action is shown as a black box.

An action is atomic for the RAD.

An action has an activating condition and a post-condition, and we might choose to model them.

An action can be described in terms of its intent or its mechanism, or both, as appropriate.

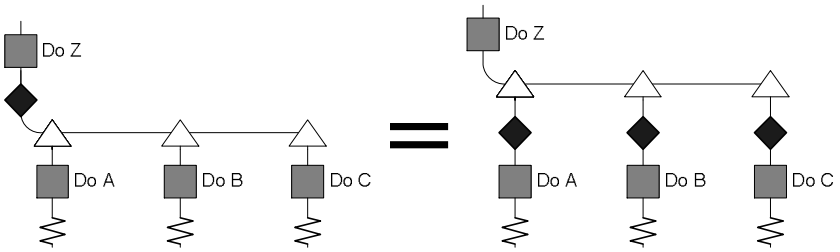
For brevity, I shall sometimes refer to the *pre-state* and the *post-state* of a process element, meaning the activating condition and the post-condition respectively.

REPRESENTING CONCURRENT THREADS OF ACTIVITY

There might be a point at which a role can start a number of separate threads of activity that can be carried out concurrently. 'Now that I have written all the chapters I can prepare the contents list and at the same time I can prepare the index.' This 'splitting' of a thread into two or more concurrent threads is represented in a RAD by the symbol shown in Figure 2-1 for *part refinement*. Strictly speaking, a state of the role is being refined (divided) into a number of separate parts. Another example of part refinement is the early split in the *Designer* role in Figure 2-2 where a designer starts two concurrent threads of activity. On one thread they choose a method; on the other they first prepare an estimate, then interact with the project manager to pass over the estimate, and finally wait for a second interaction to receive a plan back from the project manager. Part refinement can involve any number of threads of concurrent activity, depending on just how much concurrency is possible in the work of the role.

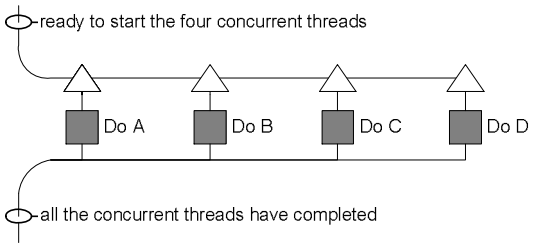
Using tokens again, we can think of the single token that reaches the part refinement becoming a number of tokens, each of which passes down one thread of the part refinement. In the left-hand fragment in Figure 2-10, the action *Do Z* has completed and there is a state token on the state line coming out of *Do Z* and before the part refinement. This marking is entirely equivalent to that shown in the right-hand fragment, where the token before the part refinement has 'turned into' one on each of the separate part threads.

Figure 2-10 – The marking before a three-way part refinement



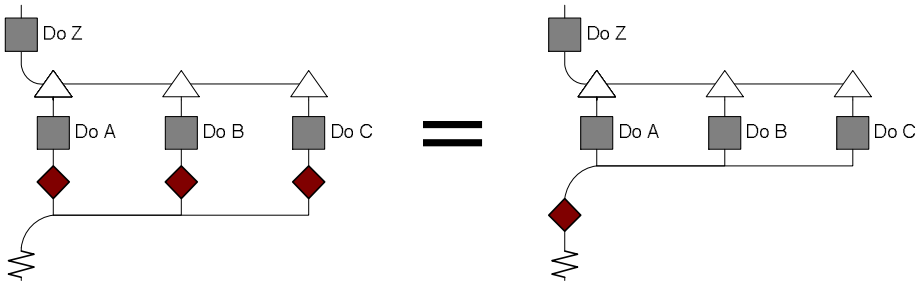
In some situations all the concurrent threads must complete before the role can proceed to further activity; in this case we use the representation in Figure 2-11, with the four threads being joined once they have finished (i.e. the part states are recombined).

Figure 2-11 – Closing a four-way part refinement



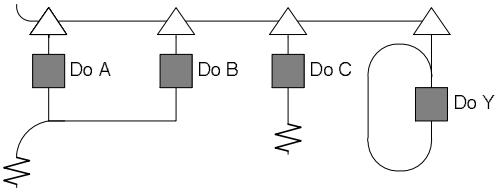
We can expect that at some point after the part threads have gone their separate ways, they will all have finished and hence there will be a token sitting on the state at the end of each thread, as shown in the example in the left-hand fragment of Figure 2-12. This is precisely the same marking as that shown in the right-hand fragment, where the part thread tokens have all been replaced by a single token on the state line immediately after the closure of the part refinement.

Figure 2-12 – State recombination at the end of a three-way part refinement



In some cases however, a role does not operate this way. A project manager's activity might be considered as two quite separate areas of responsibility, almost 'sub-roles': keeping staff occupied with work and liaising with the client. These might be the two threads of an early part refinement of the role, which need never recombine. In Figure 2-13 we see a fragment in which two of the four threads recombine, whilst two others never do, one endlessly looping on itself, and the other diving off to somewhere else in the role perhaps.

Figure 2-13 – A four-way part refinement where only two threads recombine



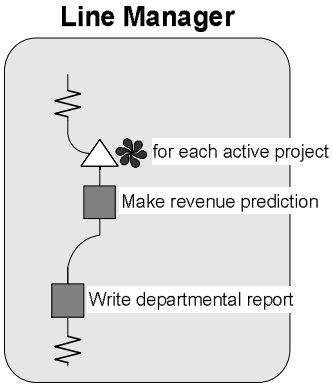
Replicated part refinements

Suppose a Line Manager wants to prepare a revenue prediction across all the projects they are responsible for. The number of projects they have active at any one moment may be variable, so we need a way of representing this. This is one example of a general situation where we want a thread of activity to be 'replicated' a number of times. Figure 2-14 shows how we do it.

The single thread that is to be replicated is shown within the usual part refinement structure, and the replication is indicated with an asterisk. The number of times the thread is replicated is captured in the caption, in this case *for each active project*. Because this is a part refinement, the action *Write departmental report* cannot start until all the replicated threads

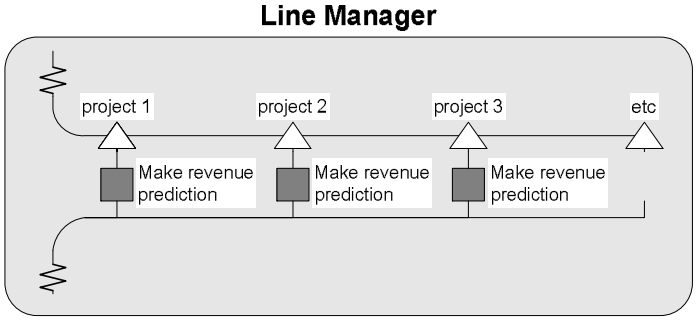
are complete, i.e. until the Line Manager has made a revenue prediction for all the current projects.

Figure 2-14 – A replicated part refinement



Note that *this is not a loop*: the Line Manager is *not* making a revenue prediction for one project, and then for the next, and then for the next, until they are all done, in a serial fashion. Instead, we are saying that there is a thread of activity – a simple one in this case consisting of just one action – which is replicated, say seven times, and all those seven threads can now start and run in parallel. Figure 2-15 shows the replicated part refinement expanded. As in a normal part refinement, the replicated threads might or might not recombine.

Figure 2-15 – The replicated part refinement in Figure 2-14 expanded



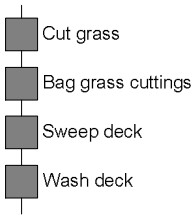
It is important to remember that the replication is done the moment it is reached. On one occasion a thread might be replicated seven times, and on another 70 times.

Choosing and modelling part refinements

The part refinement is an important construct because it is one way we model potentially concurrent action within a role. When people model processes, they all too easily fall into the trap of making all activity *sequential*: *A* follows *B* follows *C* follows *D* etc, thereby ending up with

a very sequential-looking process. If we are modelling an existing process, the resulting model might be true in that what we have drawn does capture the *time ordering* of actions, but it might not model the *necessary logic* of the process. Suppose action *B* must follow action *A*, and that action *D* must follow action *C*, but that those two threads – *A*–*B* and *C*–*D* – are independent. Because I can only do one thing at a time, you might observe me doing any one of the following sequences: (A, B, C, D), (A, C, D, B), (A, C, B, D), (C, D, A, B), (C, A, B, D), or (C, A, D, B). But none of those is the only or correct way. It would be wrong for us to observe things being done in the order A, C, B, D one day and then to show that sequence on our model. It would be an incorrect model of the process in general. And if we were *designing* the process and we drew it as in Figure 2-16 we would unnecessarily restrict people's options for carrying out that process in the future.

Figure 2-16 – An over-constrained process



KEY POINTS

Part refinements capture within-role concurrency.

Part refinements might or might not recombine once some or all of the threads have completed.

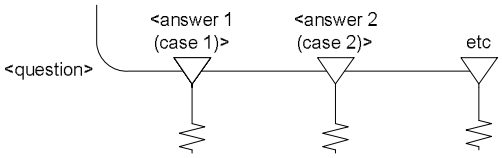
Replicated part-refinements model situations where the same thread is carried out a number of times in parallel.

REPRESENTING ALTERNATIVE COURSES OF ACTION

At some points in a process, what happens next in a role might depend on the state reached. For example, the way a clerk deals with an application for overtime might depend on the salary band of the claimant; how a chemist makes a batch of drug compound for a clinical trial might depend on which pilot plant has been allocated for the production; the way an order for a shrub is dealt with by a horticulturist might depend on the time of year the order is received and when the shrub concerned is best shipped.

We represent such alternative courses of action with the notation shown in Figure 2-1 for *case refinement*. Essentially, we are refining the state of the process according to different 'cases'. The general situation where there are two alternative courses of action is shown in Figure 2-17.

Figure 2-17 – Representing alternative threads of activity: case refinement



The preceding state line takes a bend to the right, and a downward pointing arrow appears at the start of each alternative thread. Typically, we label the bend with a question:

Has the invoice been paid?

What month is it?

Where is the package going?

We then label each downward arrow with an alternative:

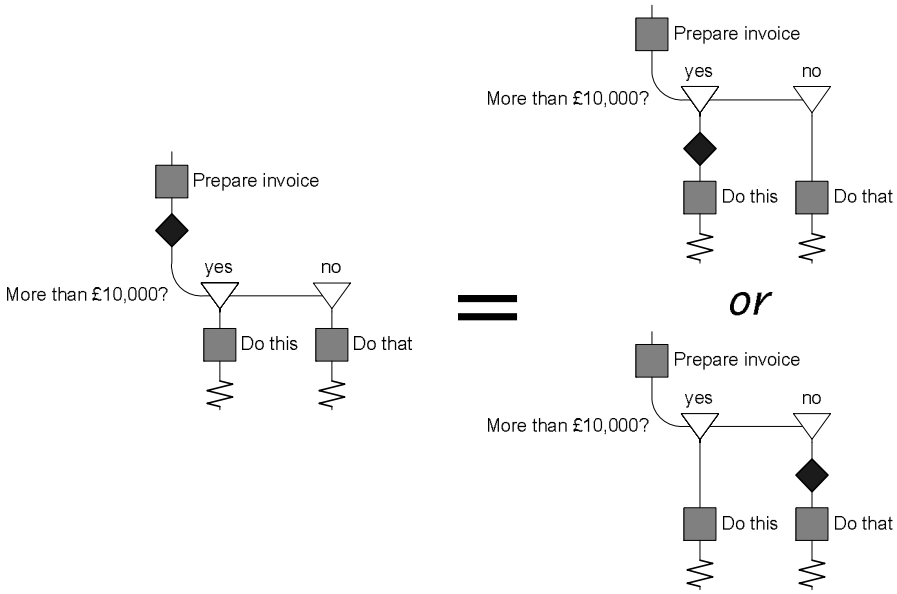
yes/no

January/February/...

UK/overseas

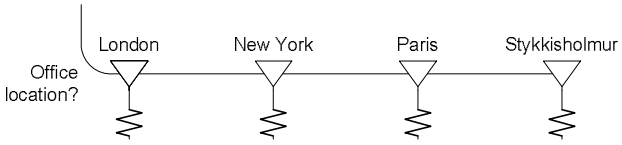
Using our token scheme, we can think of a token arriving at the case refinement, and then passing down the thread that corresponds to the predicate that is true; the role goes in different directions depending on the state of things at that moment. Figure 2-18 illustrates this. Immediately after the action *Prepare invoice* we can imagine a token on its post-state as shown on the left-hand fragment. The case refinement says that, if the predicate *More than £10,000* is true, this is equivalent to the upper right-hand fragment with the token on the corresponding state line; whilst, if *More than £10,000* is not true, it is equivalent to the lower right-hand fragment with the token on the other state line.

Figure 2-18 – The marking before a case refinement



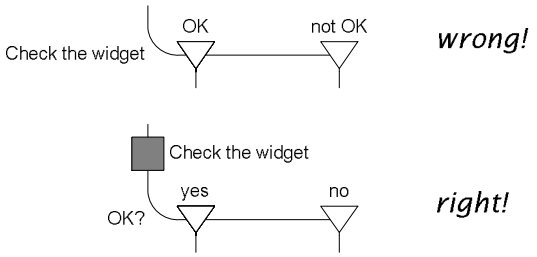
The two-way case refinement generalises quite naturally to *N*-way case refinements. The way an organisation carries out a particular part of the process might, for instance, depend on which of its offices it is carried out in. We would show this with an *N*-way case refinement such as that in Figure 2-19. Here, the process proceeds differently according to whether the location is London, New York, Paris or Stykkisholmur.

Figure 2-19 – A four-way case refinement



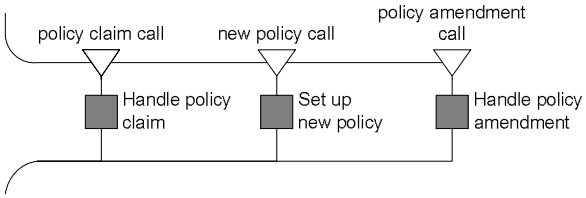
You could think of case refinement as a *case* statement in a programming language or as a decision box in a conventional flowchart. But it is important to note that, unlike a decision box on a flowchart, there is no activity going on 'in' the symbol for case refinement – no person or machine is doing anything to *make* the decision: the role instance is simply going in different directions depending on the state it is in. The upper fragment in Figure 2-20 would therefore be wrong: the caption is a description of an action and not a predicate (question) about the state. The value of the *OK?* predicate must be determinable as a result of some prior action or interaction in the process, such as the preceding quality control action *Check the widget* in the lower fragment; the case refinement does not itself 'contain' any activity to check the design.

Figure 2-20 – Nothing 'happens' in a case refinement



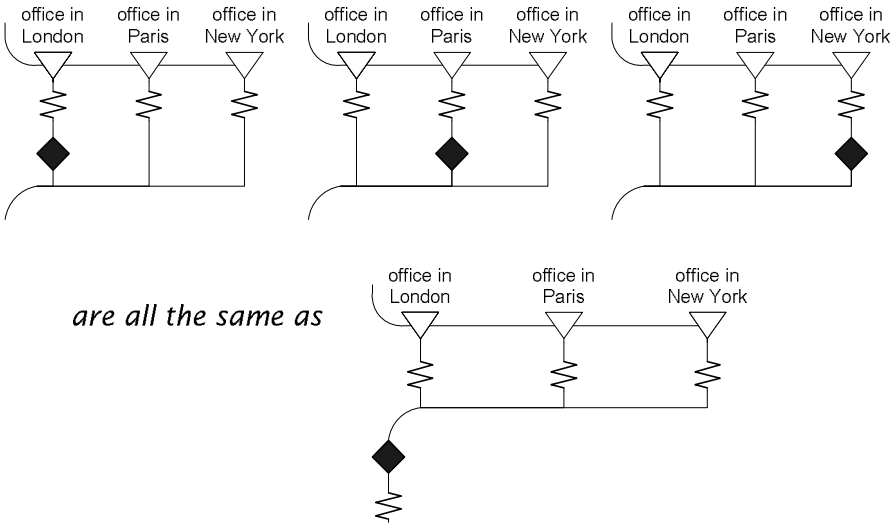
In some situations, whichever of the alternative threads of activity is followed, we want to 'return' finally to the 'main' thread of activity. In this case we use the representation in Figure 2-21, with the threads joining up again when they have finished (i.e. the case states are recombined). In this example we have also used an alternative labelling of the case refinement: we have omitted the question and simply labelled the different cases; if the meaning is clear, this is fine.

Figure 2-21 – Case refinement threads sometimes recombine



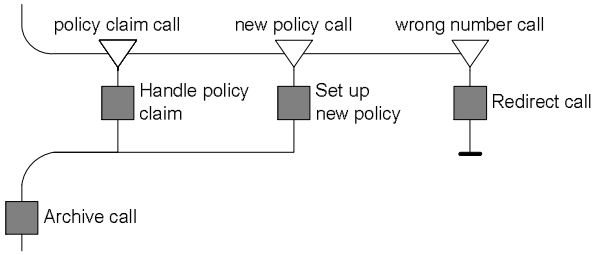
We can best visualise what this means by looking at it using tokens. In Figure 2-22, whichever of the three case threads is followed we require the main thread to be picked up finally. Thus each of the three markings shown in the upper part of the figure is equivalent to the marking in the lower part.

Figure 2-22 – Three case refinement threads finally return to a common thread



In other situations, the case refinement threads might not recombine. The example in Figure 2-23 is a case in point: if the customer call picked up at the call centre is about making a claim on an insurance policy or setting up a new policy, then it is dealt with and once it has been dealt with it is archived. If the call is a wrong number, we simply redirect it and finish the process there. (If this all seems hopelessly laboured and obvious, then you are probably not a software engineer: in software design and modelling methods that have come from that world, there is a strong emphasis on 'Dijkstra structures' where, in crude terms, everything closes off tidily. Unfortunately the world is not as tidy and 'block-structured' as we can make our software, so any modelling method that demands such tidiness will be of no use in modelling the real world.)

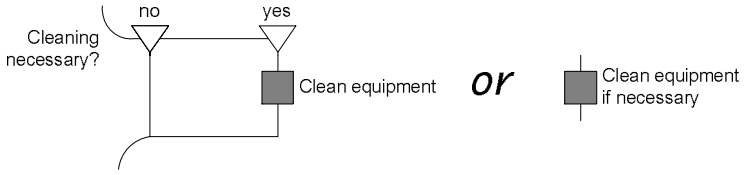
Figure 2-23 – Case refinements don't always close tidily



We sometimes adopt an abbreviation for simple case refinements. On the left-hand side of Figure 2-24 we show a case refinement with a single action only being done if cleaning is necessary, whilst on the right-hand side we capture the conditionality in the name of the

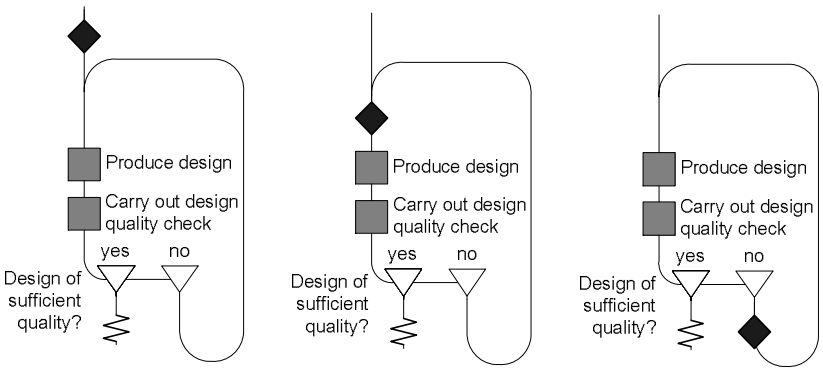
action. Words like 'if appropriate' or 'as necessary' might be useful. This can also be done with conditional interactions, but greater care is needed.

Figure 2-24 – Abbreviating a simple case refinement



Case refinement naturally allows us to represent conditional iteration within a role, as in the example in Figure 2-2 where the Designer repeatedly produces and checks a design until it is OK. Anywhere on a single state line is the same, so the three role instance markings shown in Figure 2-25 are equivalent in that the next possible action of the role is *Produce design* in all three cases. You might object that the first and third situations in the figure are quite different since in the first you don't have a design whilst in the third you do. True, but the RAD is telling us that, as far as this process is concerned, having a design that has failed its quality check is no different from having no design at all: in each case you have to produce a new design. The states marked by the tokens are equivalent in that *they define the same future behaviour* even if they define different histories. If this is not what we wanted to model then we have the wrong model.

Figure 2-25 – Three equivalent role instance markings



Choosing and modelling case refinements

Case refinements are not active

Case refinements are relatively straightforward. But we should always make sure that all the information needed to evaluate the predicate is available to the role concerned without further work. The more general question we should ask is 'Does the role have all the props necessary to answer the question?'

So, if the predicate is *Which office is the application being processed in?* there probably isn't a problem: the actor can look around and recognise they are in New York and not London. If it is *Is the applicant male or female?* and the applicant isn't sitting in front of the role concerned, then we should check whether the applicant's gender has been ascertained before this point in the process, and that it is available to the role concerned.

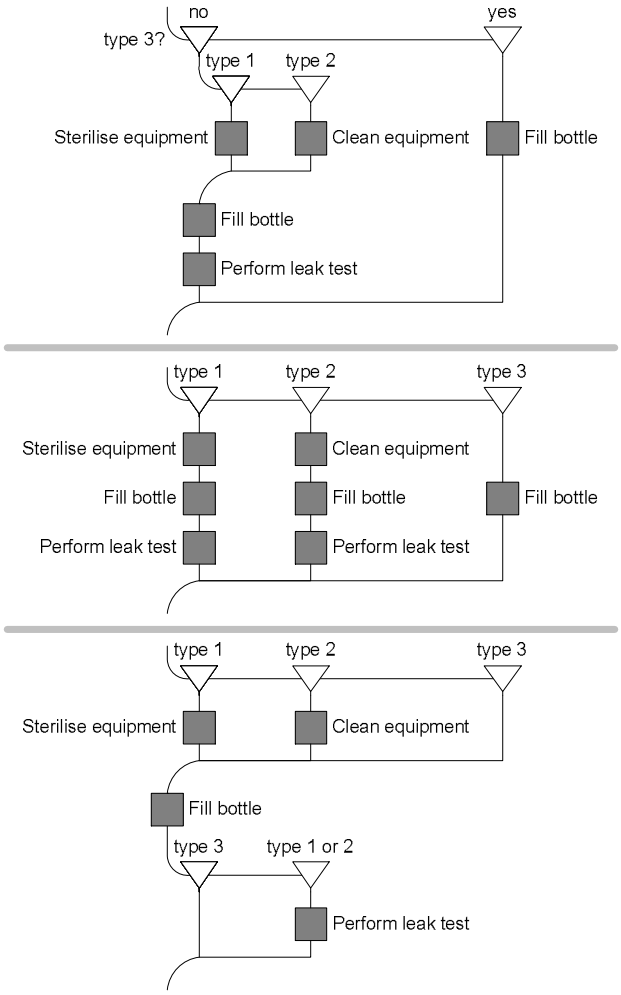
Complex case refinements

There are situations where we can model the case refinement in several ways. Take the following description of what happens in the pharmacy of a pharmaceuticals company:

> A pharmacy bottles three types of drug: (1) unformulated and aseptic, (2) unformulated and non-aseptic, and (3) formulated. Type 1 requires equipment to be sterilised before filling. Type 2 requires equipment to be cleaned before filling. Types 1 and 2 require bottles to be tested for leaks after filling. Type 3 only requires bottles to be filled.

We have several options, shown in Figure 2-26. They all describe exactly the same behaviour: for each type of drug the same actions are done in the same order. So, is there any reason to use one rather than the others? The first option might be appropriate if type 1 and type 2 drugs were prepared in one location and type 3 in a different location; the way that the threads are placed suggests the 'geography' of the process. The second option might be appropriate if the three types were prepared in different locations, so it is useful to elaborate the thread fully for each type. The third option might be appropriate if all three types were done in the same location and one wanted to emphasise that bottle filling was done in the same way (one single action) in all three cases.

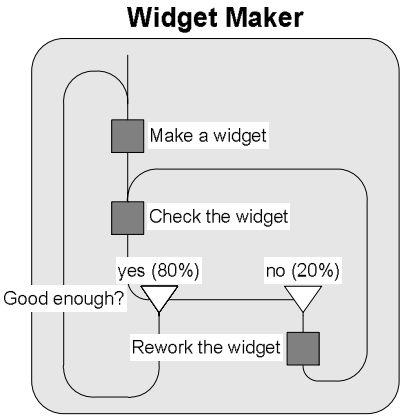
Figure 2-26 – Three different but equivalent models



Probabilities and case refinements

Case refinements represent alternative courses of action, and when we come to analyse the process it might be important to know how often the different alternatives are followed. In Figure 2-27, how often does the *Widget Maker* have to rework the design? We simply annotate the RAD in an appropriate way. Figure 2-27 shows a simple percentage, but there might be seasonal variation or it might depend on some other characteristic of the process (the depth of the items in the in-tray, the rate at which work is being done, etc).

Figure 2-27 – An annotated case refinement



KEY POINTS

Case refinements model alternative courses of action that arise in a thread.

The alternatives should be mutually exclusive.

Each alternative leads to a separate thread.

The alternative threads might or might not recombine.

There is no activity in the case refinement itself.

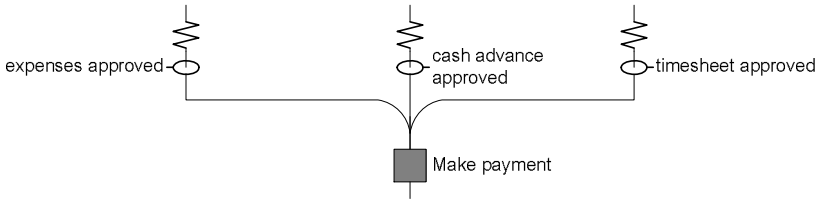
Representing merging threads

It is quite often the case that we want two or more threads in a role to come together in a single thread, even though they did not originate from a single thread. For instance, the Payroll Department will prepare a cheque for an employee and get it authorised at the end of a number of different procedures, all of which start from different triggering points:

> At the end of the month when the salary payment is due and the timesheet has been checked.
>
> Whenever a cash advance has been approved.
>
> Whenever the reimbursement of expenses has been approved.

To show these different parts of the role coming together to a single thread we simply combine the state lines in the curving fashion shown as in Figure 2-28. This says simply that whichever of those states the role is in, its future behaviour is the same: it makes payment. Each of the three threads defines a different history, but they all also define the same potential. (Note the different shape in the merge from the way that part and case refinements are closed.)

Figure 2-28 – Three threads combine to form one

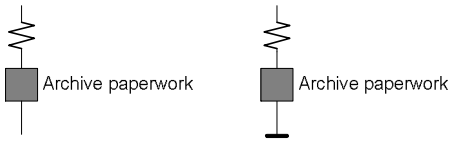


The threads can come from anywhere within the role. Organisational activity often ambles around, dividing and recombining, ducking and weaving, jumping off at tangents, sometimes coming back. Our notation must allow us to model the 'untidiness' of the real world. Within a role, threads of activity can divide, recombine, and switch to other threads without constraint – simply because that is the way the real world operates: as a network rather than a hierarchy. And roles can operate in a similar way using interactions as the mechanism for 'jumping'.

Representing the end of a thread

Do we need to signal the end of a thread? No. The left-hand fragment of Figure 2-29 shows a thread simply coming to an end. Once the action *Archive paperwork* has been completed there is nothing else to do and so nothing more will happen on this thread. In some situations however, it can be helpful to reinforce the fact that this is really the end of the line and not simply as far as we wanted to go in this model. In these situations we use a little 'stop' sign as in the right-hand fragment.

Figure 2-29 – The optional stop sign at the end of a thread



KEY POINTS

States in a role can merge: different past behaviours lead to the same future behaviour.

Optionally, we can mark the end of a thread.

REPRESENTING INTERACTIONS

Interactions are collaborative actions carried out by two or more roles. Let's take some examples:

The purchaser arranges finance with their bank.

The Project Manager and Line Manager review the Project Plan.

The QA Team lets the Project Manager know they have finished the tests.

The Line Manager asks for an update on progress.

Vendor and Purchaser agree on the delivery date.

The Sales Executive tells the Project Team of the contract change.

The Product Design Team give the Development Team the specification.

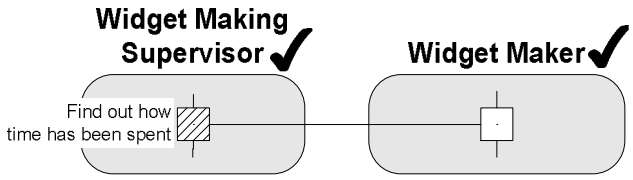
The Engineering Department drafts the design with the Production Engineer.

All the Divisional Managers meet to review the budgets.

Each Divisional Manager reports to the Board.

A vanilla *interaction* between roles is shown as a white box in one role connected by a horizontal line to a white box in another role. We refer to the white box in each role as a *part-interaction* – it is the contribution that the role concerned makes to the interaction. Figure 2-30 shows a simple interaction between two roles.

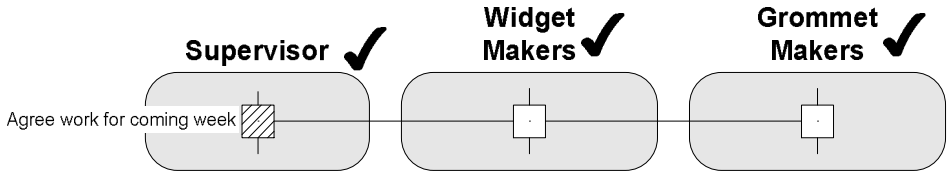
Figure 2-30 – A simple interaction



In this example, by shading in its part-interaction, we have also shown which of the roles 'takes the driving seat' and makes the interaction happen. We are saying that it is up to the widget making supervisor to take the initiative and ask the widget maker how they have been spending their time.

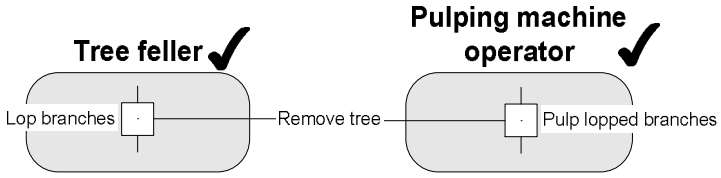
An interaction can involve any number of roles and signifies that the roles involved must pass through it together; as an example Figure 2-31 shows an interaction involving three roles. Note also how in this example we have lumped all the widget makers as one role, *Widget Makers*, and all the grommet makers as another, *Grommet Makers*.

Figure 2-31 – A three-party interaction



If it is useful, we can add further annotation to the interaction, perhaps captioning a part-interaction to describe its contribution, and even a further caption for the entire interaction. An example is given in Figure 2-32. We would typically model in this way when capturing a truly collaborative activity where each of the parties makes their own contribution to the overall action.

Figure 2-32 – Captioning the components of an interaction



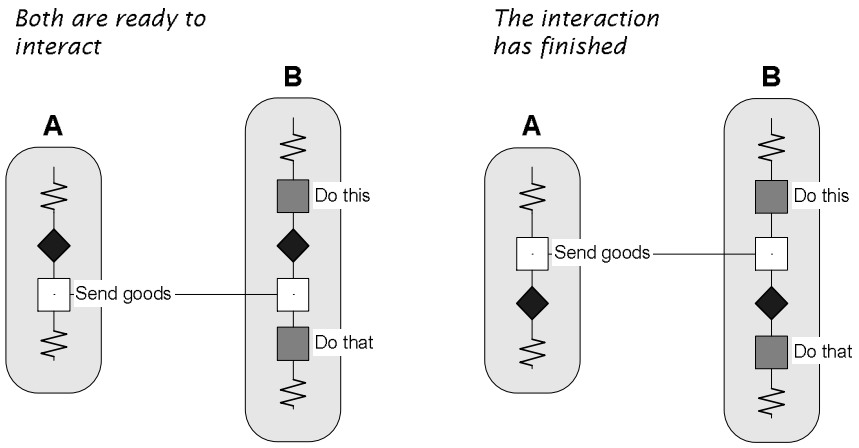
We always caption an interaction in a way that makes clear what is happening, and that would include whether anything 'changes hands' during the interaction, i.e. whether any grams are involved. Interaction lines do not carry arrows to indicate the 'flow' of any gram: we simply place an appropriate caption at the appropriate end. For example, in Figure 2-2 the *Project Manager* role receives an estimate from the *Designer* role, and this is indicated by placing the caption *Pass over estimate* at the *Designer* role end of the interaction line.

Interactions synchronise role instances

As with an action, an interaction has an *activating condition*, which is the condition corresponding to each participating role being in the state before its part-interaction. So the overall activating condition is effectively the state when all the participating roles are ready for the interaction.

The rules for an interaction say that an interaction cannot start until all the participating roles are in their respective pre-states, and that when the interaction finishes they all move into their respective post-states. Any role might get to the interaction before the others. If I reach my side of the interaction and you are not ready, I must wait until you are; as soon as we are both ready the interaction can take place. Put another way, *interactions synchronise the states of the participating role instances*. Figure 2-33 summarises this with tokens. In the left-hand fragment both roles are in their respective pre-states, i.e. their respective part-interactions can be activated. In the right-hand fragment, the two part-interactions have completed and so the two roles are in their respective post-states.

Figure 2-33 – Interactions synchronise role instances

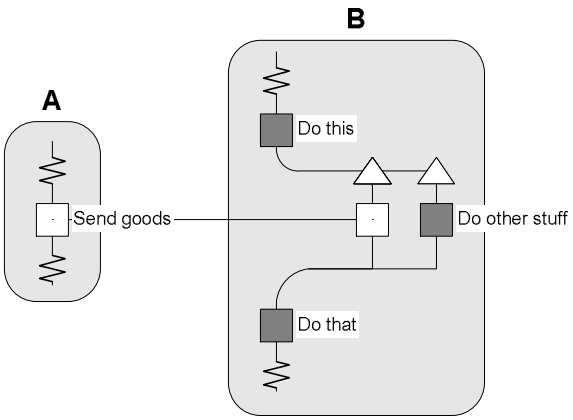


How much do we want to synchronise?

Remember that a RAD shows strict ordering. It is very easy to draw strict ordering when it is not actually required or present in reality. For instance, if role *B* must have got to a certain point but then can accept the goods any time after then, *and* must have them before certain other actions can continue, then we render this as shown in Figure 2-34a. This figure says that once *B* has completed the action *Do this* it can receive the goods while getting on with other tasks (*Do other stuff*) as shown by the part refinement. Once it has received the goods *and* completed those other tasks it can *Do that*.

We have recognised that there is a window during which *B* can and must take delivery, but that during that window *B* can be getting on with other things.

Figure 2-34a – Just enough synchronisation



Later, next to the water-cooler

Pupil: You've emphasised that interactions synchronise state, and … if I've got this right … an interaction therefore can't finish until all the parties have done their part-interactions?

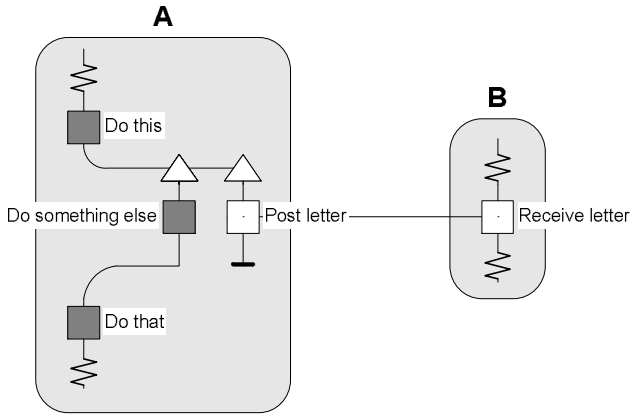
Tutor: Correct.

Pupil: But surely that's not like the real-world. Suppose I send you a letter. Sending you a letter is an interaction between you and me but I don't wait around until I've heard you've received it before getting on with something else.

Tutor: You have a knack of putting the answers to your questions in your questions! You're right that the interaction of my-sending-and-your-receiving-a-letter doesn't finish until – in particular – you've received it. But you told me that you don't want to wait around for the interaction to complete. In effect you told be that you want to do something else concurrently while that interaction is completing. Doesn't that sound like a part refinement?

Pupil: I guess it does. Shall I draw it on this handy whiteboard next to the water-cooler? (Figure 2-34b)

Figure 2-34b – Just enough synchronisation again



Pupil: I suppose that if I wanted to capture what we do once you have indeed received the letter then I would draw it after the interaction but I've drawn it as if once I've posted it that's it. Of course I can see now that it's very like Figure 2-34a, but reflected.

Tutor: Yes. Now, if we don't mind modelling mechanisms, we could recognise that when you post a letter you have a very short interaction with the post office immediately after which you can get on with other things. The post office then carries out the action of moving your letter from one post office to another, and then has a very short interaction with your correspondent to give

them the letter. We have decoupled you and your correspondent through what I would call a carrier function. But you can draw that for yourself – I'm going home.

Choosing and modelling interactions

Interactions change role states

Interactions can come in a great variety of guises:

I arrange finance with my bank.

We review the Project Plan.

He lets her know he has finished the tests.

She asks for an update on progress.

They agree on the delivery date.

She tells the team of the contract change.

I give you the specification.

We draft the design.

They report to the Board.

Reading these, we sense different things going on. Sometimes things are being passed over: information, documents, materials. Something that was in one role is now in another; or perhaps they both have it, as is the case of information. Sometimes, the interacting parties are jointly contributing to something: reviewing, agreeing, drafting. One or both of them now has something that neither had before the interaction: an agreed delivery date, for instance, or a design that they have jointly drafted. It is useful to think through the state changes in the participating roles: just what has changed in each role as a result of the interaction? How have the states of the props been changed? What props does the role now have that it did not have before? What props does it not now have that it did have before?

Concrete and abstract interactions

Interactions are as amenable to concrete and abstract descriptions as roles and actions are.

A caption on an interaction that read *Send completed form 195/5* would tell us nothing about what is going on except how the interaction is done. If we knew the form concerned we might know that it is the company's expense claim form and hence the interaction is in fact about claiming expenses. An abstract caption for the same interaction might be *Claim expenses*, which would not give us any indication of how to do it, but would tell us what was going on, the intent of the interaction. The caption *Claim expenses using a completed form 195/5* would tell us what was going on and how it was done.

Here are some other examples:

A Project Manager passes a budget report to the Line Manager … A Project Manager reports on the budget to the Line Manager.

The Client and Consultant speak on the telephone … The Client and Consultant agree the scope of the work.

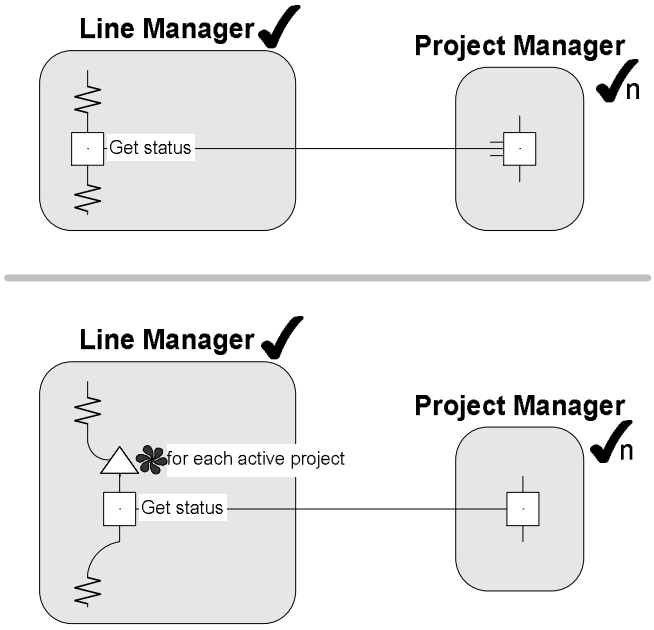
> The Customer presses the 'Buy Now' button on the web page ... The Customer confirms the purchase of the contents of their shopping cart.

When would we use these different styles of caption? Again, this is a discussion that we must leave until later chapters where we look at how *Riva* is used in different situations.

Interactions with replicated role instances

When a process runs, an interaction actually occurs between *instances* of the roles concerned, of course. (Strictly an *instance* of the interaction occurs between instances of the roles concerned!) So, one instance of *Divisional Manager* interacts with one instance of *Line Manager*. But what if the Divisional Manager wants to have the same interaction with each of the current Line Managers, perhaps to get the status of their projects? We need to be able to represent the fact that the one instance of *Divisional Manager* has the same interaction with all the existing instances of *Line Manager*, however many there are. The upper fragment in Figure 2-35 shows how we do this. You can think of this as shorthand for the lower fragment in Figure 2-35 in which the replication is shown at the *Line Manager* end, by replicating the thread containing the interaction for each Project Manager. The net effect is that the one instance of *Divisional Manager* starts a separate interaction with each instance of *Line Manager*.

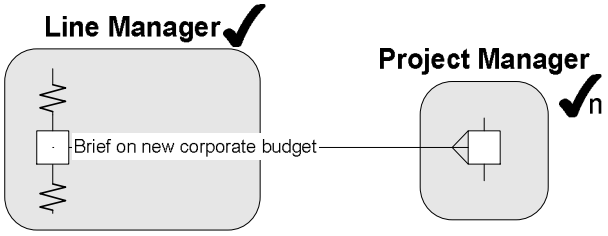
Figure 2-35 – A replicated interaction



Note that this is not the same as having a *single* interaction that involves all of them. An example might be that the Divisional Manager wants to brief all the current Line Managers on the new corporate budget. We show this using the notation in Figure 2-36. This single

interaction involves a total of $N+1$ role instances, where N is the number of instances of *Line Manager*.

Figure 2-36 – An interaction involving many role instances



Interactions are rarely as simple as we think

An interaction can be very simple (e.g. *I give you some terms of reference*), or very complex (e.g. *The three parties meet to negotiate and agree the price of a piece of work, drawing up the agreement as a legal document and obtaining financial securities from a bank*). As with actions, what we regard as 'atomic' in our RAD depends on why we are drawing the RAD. We always show whatever detail is appropriate to *that* model for *that* purpose. An interaction that is shown as a single line on a RAD might, if it were in some sense 'opened up', show the involvement of other roles not otherwise mentioned in this RAD, new interactions between them and other roles, and a whole mass of organisational activity. But by drawing just one interaction, we are saying 'One interaction is appropriate for this model.'

Contracts – a pattern of interaction

A contract is a common form of interaction between two parties and one that we shall see in various guises. Winograd offers a view of organisational behaviour in terms of a four-step contractual cycle between a customer and a supplier: preparation, negotiation, performance, assessment.

In the preparation step, either the customer decides what they want to buy (or, more generally, contract out for), or the supplier makes some offer to a would-be customer; this results in the customer making a request to the supplier. The two parties negotiate the request and two mutual promises result: the supplier agrees to provide something in return for something else, namely payment by the customer. The supplier then carries out the performance step which finishes with a declaration that the work is complete, an assertion that the customer tests, finally declaring satisfaction.

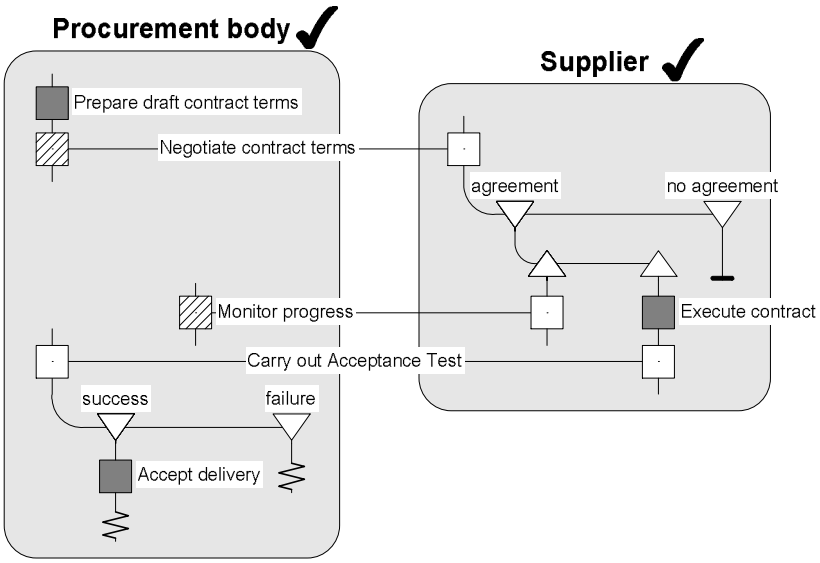
Each step can, in its turn, be carried out by a cycle of its own: performance might be broken down into sub-cycles for instance.

In the ideal world conceived by Winograd and Flores, or a new one that we intend to build, we build all organisational behaviour out of such contractual cycles. (See 'A language/action perspective on the design of cooperative work', T Winograd, *Human-Computer Interaction*, **3**, 1987.) It is doubtful that we can retrospectively *impose* a hierarchical structure of such cycles on an existing process or expect to find one there when we come to model it – we cannot

expect that the evolution of the process has preserved such neat conceptual integrity; in fact it would be far more realistic to expect that, as in any other physical system, entropy – the degree of chaos or lack of order in the system – inexorably increases over time unless energy is expended to reverse it (albeit temporarily). We could of course consider BPR to be an 'entropy-reversal' exercise, one in which the chaos and disorder that has built up over the years is recognised and, through radical change, replaced by a 'tidier' and 'simpler' system.

Nevertheless, the Winograd contractual pattern can be observed, and of course has a natural representation in a RAD: Figure 2-37.

Figure 2-37 – A contractual cycle in a RAD, driven by the customer



We can elaborate the model at several points to allow for situations where

- there is a breakdown in the negotiation;
- the *Supplier* fails to complete the performance;
- the *Procurement body* is not satisfied with the performance of the *Supplier*;
- the various steps are themselves the subject of sub-contracts.

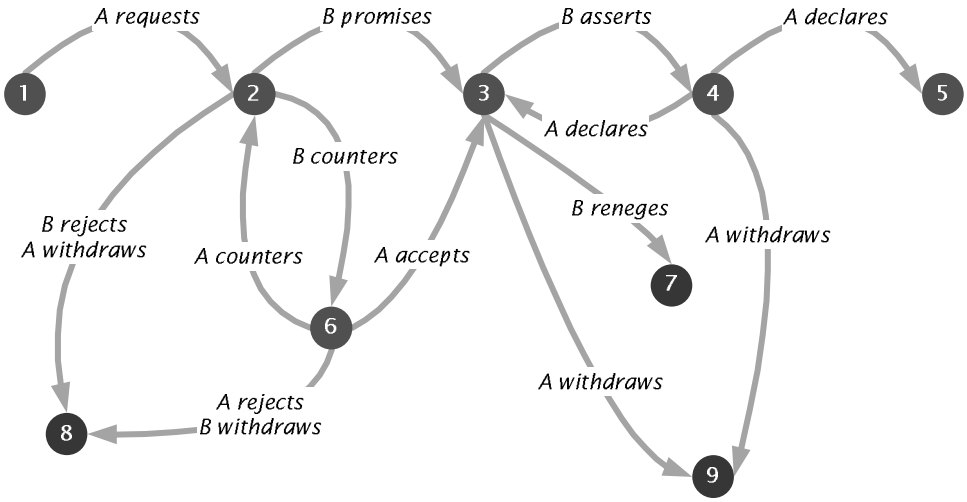
We shall find it a useful modelling discipline to spot interactions that represent acts of negotiation, assessment and so on, in order to see how far the complete cycle is present, and perhaps, if it is not, to ask whether it should be and whether the process could be improved by restructuring it to the 'standard' cycle.

So, whenever, in a modelling workshop, someone identifies an interaction by saying 'Then the employee gets the Line Manager to approve the expense claim', we shall now know to ask 'Do they always approve it?' If it really were the case that the Line Manager always approves the expense claim, we could identify an easy process improvement by simply removing that interaction – it seems to serve no purpose. What is more likely of course is that a more

complex conversation is being (poorly) summarised as one of approval, and we are probably ignoring the possibility that the Line Manager might reject the expense claim. And what happens then?

We can elaborate this idea a great deal further and recognise interactions as 'conversations for action', with the much more complex general pattern shown in Figure 2-38. The conversation between two parties *A* and *B* starts at state 1 and then moves to a new state depending on how *A* and *B* interact. The interaction therefore ends up at one of states 5, 7, 8 and 9; the last three of these represent a form of failure, the first success.

Figure 2-38 – An interaction as a conversation for action



This template gives us a useful pattern against which to assess each interaction when we come to it. I am not suggesting that every interaction we deal with should be elaborated to this *n*th degree, only that we should make a conscious decision about how far we want to go in *this* model, for *this* purpose, in unpicking *this* interaction into its component parts. Returning to our expense claim example, it would be entirely valid for us to wrap up a whole mass of such conversational detail in a single interaction that is indeed labelled *Get the Line Manager to approve expense claim*, if we allowed that that initial refusal, reworking, and resubmission were part and parcel of the interaction. That would still not allow the possibility of my withdrawing my expense claim altogether of course, and, if that possibility was important and relevant to the model we were drawing, then we had better draw it and not simply ignore it.

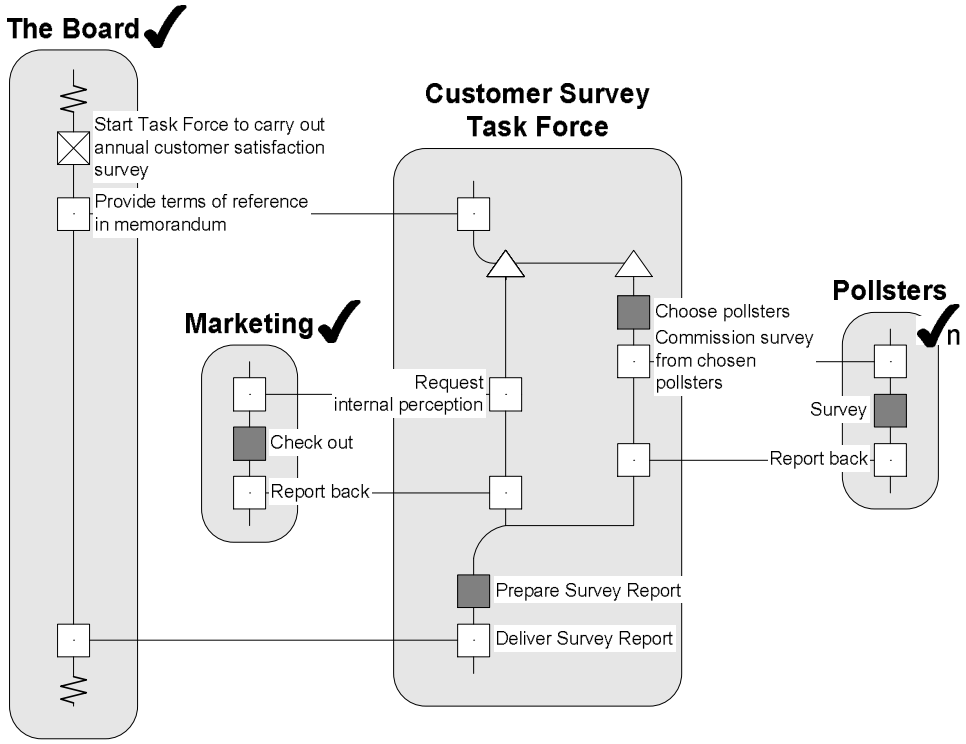
Delegation ladders

When we model a process with a RAD we seem not to take any explicit notice of one of the most important aspects of an organisation: its authorisation hierarchy. Most organisations – even those operating forms of matrix management – use some layering down from the Chief Executive; some *only* operate that way. In fact, in a RAD, although we might not model the hierarchy explicitly, we can model the way it makes itself felt: the business rules that operate

in terms of planning, delegating, reporting, authorising and so on. Indeed, it is normal for a RAD that uses functional positions or job-titles as roles to expose the hierarchical aspects of an organisation's behaviour in terms of interactions between 'superior' and 'inferior' roles. It is common to see the hierarchy running from left to right in the RAD: *The Board* appears as a role at the extreme left, passing instructions to the next level down, say *Divisional Director*, who in turn passes instructions to their right via interactions with *Project Manager*, and so on further to the right. We can see these as formal contracts of course as discussed above.

Delegation and reporting back are very common process patterns in an organisation and they have a very natural representation in a RAD: they are simply pairs of interactions. Take the process shown partially in Figure 2-39. If the *Board* starts up a *Customer Survey Task Force* to carry out the annual customer survey we will see an interaction between the two roles across which *The Board* gives the *Customer Survey Task Force* its terms of reference. In its turn the *Customer Survey Task Force* delegates parts of that responsibility to other roles: *Pollsters* and *Marketing*, say. The task is broken into smaller sub-tasks and delegated out to other roles. Note that *Customer Survey Task Force* is a transient role, an instance of which is created for the occasion.

Figure 2-39 – A asks B who asks C ...



As each role completes its sub-task, it might (or might not) report back to the delegating role to say 'I've done what you asked.' So we can expect to find a corresponding 'closure'

interaction for each delegation interaction. Or, at least, it is a useful modelling discipline when you see an interaction that represents delegation to look for a corresponding closure interaction in the real process; if there isn't one there you can question whether there should be; and if there is, you can question that too. Looking back at the sample (and not very sensible) process in Figure 2-2, we can see quite readily that the *Divisional Director,* having delegated the execution of the project to the *Project Manager*, apparently never expects to hear about the completion of the project – let's hope that in that model we simply weren't interested in the involvement of the *Divisional Director* after they had started the project off!

In some cases, one role instance might delegate a task to some other role instance which is pre-existent, such as a department or a post in the organisation:

> Each March, the *Board* delegates the annual assessment of effectiveness of the company's IT systems to the *IS Department*.
>
> After an accident in the plant, the *Divisional Director* delegates a review of plant safety to the *Health and Safety Manager*.

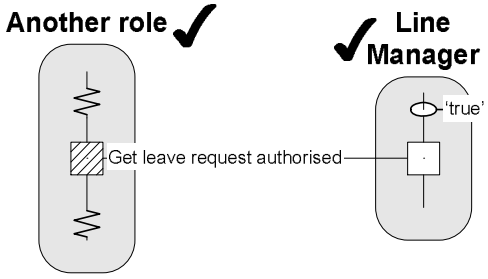
In other cases, a role might be instantiated for the purpose: we tend to call such roles 'task forces' or 'project teams'. Their job is to carry out the task and then disband. Figure 2-39 shows just such a role: *Customer Survey Task Force*.

Service interactions

Let's take this a bit further. We have seen how an interaction can be said to 'align' the states of participating role instances: that is, the participating role instances 'go through' the interaction together. As far as the RAD is concerned the interaction is atomic; once it has started we know nothing except when it has finished. But of course many things happen between parties without apparently any need to synchronise so explicitly. For instance, some roles provide some form of on-demand service: a Line Manager will authorise leave requests at any time, rather than only at some prescribed point in *their* work. So some part of the Line Manager's work must involve 'being available' in case such a demand arises.

This situation is modelled quite straightforwardly in a RAD by having a separate (i.e. concurrent) thread of activity in the service role 'hanging free' – see Figure 2-40. By definition, the activating condition for such a part-interaction on the server's side is 'true', i.e. the role is ready to undertake such an interaction on demand, at any time. Indeed, that thread can be activated/instantiated an indefinite number of times as requests for service come in, since the activating condition remains 'true'. This is often what we want. Not all activity is strictly sequenced in the sense of 'When *X* has finished do *Y*'; often the logic is more catch-all: 'Whenever necessary do *Y*.' The interaction with a hanging thread is a special case of this.

Figure 2-40 – A role ready to have an interaction at any time

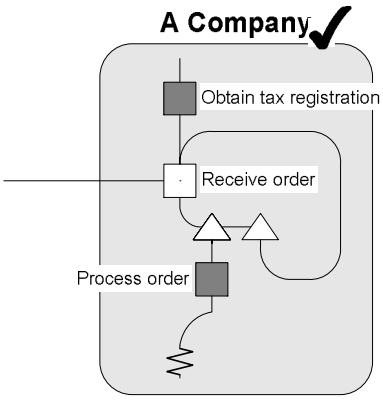


I like to describe free-hanging threads such as this as corresponding to parts of the role's brain: a bit of the brain that is always ready to have that interaction and start that thread.

Primed service interactions

In some situations we might wish to have an activity thread hanging free, ready to respond, but only after some prior action has been completed. For instance, once a business has acquired its tax registration it can process any number of orders arriving asynchronously. We would show this situation in a RAD as in Figure 2-41. When (the single instance of) *A Company* starts, the only thing it can do is obtain tax registration. The *Obtain tax registration* action takes place and the role then enters a state where it is able to accept an interaction (a purchase order). That interaction causes two threads to start: one to process the order, and another to wait for another interaction (another purchase order). While the first order is being processed, a second can arrive which again causes the two threads to start: one to process the second order and another to wait for the third; and so on.

Figure 2-41 – Many-at-a-time processing after initialisation



We can see this more easily by looking at it in terms of tokens. Figure 2-42 shows the states through which the RAD fragment passes (note the direction of the broad arrow at the back of the diagram):

1 Registration for tax has been completed; waiting for the first order.

2 The first order has arrived via the interaction.

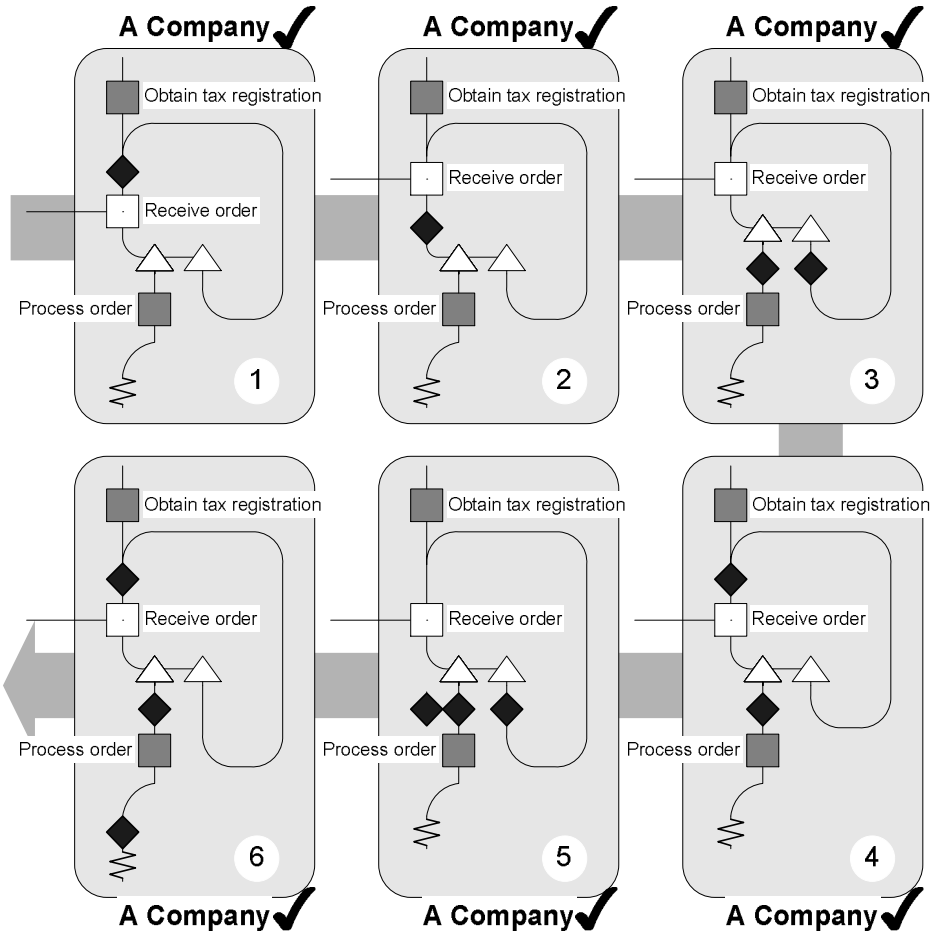
3 An equivalent state to state 2, using the definition of a part refinement.

4 An equivalent state to state 3, remembering that a state line shows a single state. The role is now ready to process the first order and also ready to accept a second.

5 A second order arrives but the first still hasn't been processed.

6 One order (it could be either) has been processed, the other is waiting for processing, and a new order is also being awaited.

Figure 2-42 – The successive states of the company after receiving tax registration

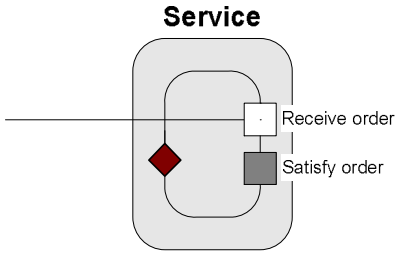


The thread to process an order is successively started as each order arrives. In fact, strictly, the actions on that thread are instantiated as and when their activating condition becomes true. We can imagine that the actions on the thread are being instantiated as many times as there are orders. Certainly each 'thread instance' proceeds independently, at its own pace. There is no implication that orders 'queue' to follow the thread. In fact, it is quite possible for orders to be processed at different speeds – the RAD says nothing about which will finish first. I have drawn one possible sequence of markings for the RAD, principally to show how a mass of concurrent activity can build up, indicated by the proliferation of tokens. RADs are about the concurrent activity in the real world.

Strictly sequenced service

In some cases it might not be desirable to have an activity thread hanging free in such a way that it can be activated at any time or indefinitely many times: we might wish to handle requests just one at a time in strict sequence. In this case a hanging interaction is clearly not what is wanted, and in order to serialise the processing of requests we put that processing in a sequential loop. The processing can no longer take place asynchronously: it can only occur when the server has finished serving the previous request. This situation is shown in Figure 2-43. We must assume that when the role starts there is a token sitting on the loop, as we have shown. Spend a moment checking that orders are dealt with strictly in rotation and a new one cannot be received until the previous one has been satisfied.

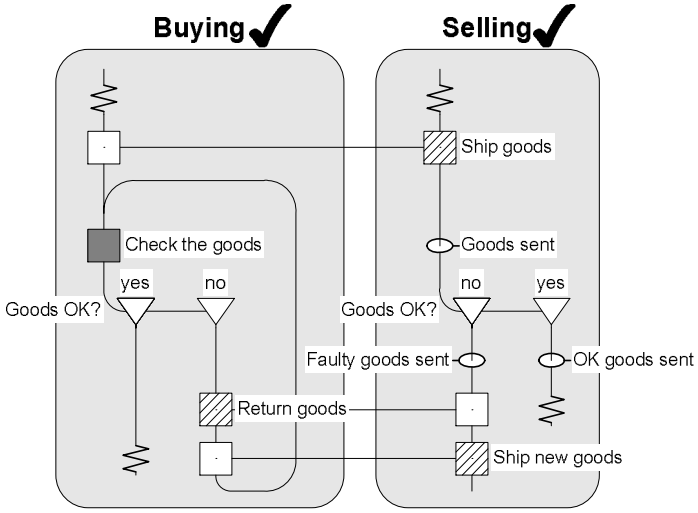
Figure 2-43 – One-at-a-time processing



Conditional interactions

It is not uncommon for an interaction to take place only under certain conditions. For instance, suppose that you (a buyer) order some goods from me (a seller). When you receive the goods from me, you check that they are OK, and, if they are not, you send them back to me. That interaction for returning the goods to me takes place only if they are not OK. How should we model this? Our first thought might be to draw a process as shown in Figure 2-44. Follow it through, especially on the seller's side.

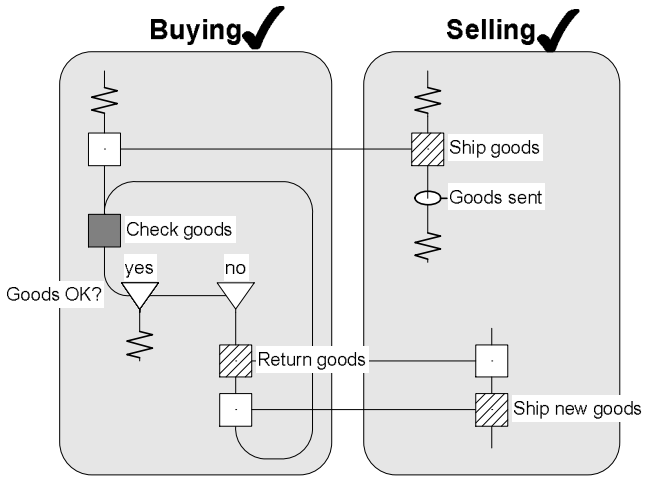
Figure 2-44 – A conditional interaction wrongly modelled



This would be wrong. Firstly, remember that a case refinement does not involve any action: it simply divides the state. So, the figure says that, in the *Selling* role, immediately after the goods have been shipped, we must be able to refine the state marked *Goods sent* either into the state *OK goods sent* or into the state *Faulty goods sent*. But if this really were the case it would mean that the seller must know immediately after shipment whether the goods are faulty or not and can be ready to respond appropriately! If we assume that the seller is not so underhand, we have modelled the wrong process. The problem is that the condition *Goods OK?* cannot be determined at that point within the body of the *Selling* role. Only the *Buying* role has the resources within its body to determine the value of the condition.

A correct model for this situation would therefore be that shown in Figure 2-45.

Figure 2-45 – A conditional interaction correctly modelled



Here, we show the case refinement only in the *Buying* role and a second interaction with the *Selling* role that only takes place if the *Buying* role decides the goods are not OK. The *Selling* end of that interaction takes the form of a 'hanging' thread: it can take place whenever necessary, and we do not need to tie it into the rest of the role. Only if the *Buying* role takes the initiative and 'forces' the interaction will the *Selling* role need to do anything about it.

If the seller had a standard practice of always checking with the buyer that the goods delivered were satisfactory then we would explicitly model this of course.

KEY POINTS

Interactions show the points where roles cooperate.

An interaction can involve any number of roles.

An interaction synchronises the states of all the participant role instances.

The result of an interaction can be the exchange of something, or joint activity, or both.

Replicated part-interactions allow us to model interactions involving all current instances of a role.

Interactions can be modelled in concrete or abstract terms, or both.

Interactions can be more complex than we think and the right level of detail must be struck.

Pupil: We've spent a lot of time on interactions.

Tutor: Yes, but don't sound so surprised: processes are about collaboration, and interactions are where collaboration takes place. If you're used to swimlanes, you

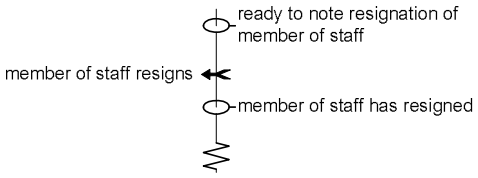
might suppose that, when two swim-lanes interact, all that happens is that the locus of activity moves from one to another. In fact, interactions are very rich in nature and we must be aware of that.

REPRESENTING TRIGGERS

Sometimes a role must wait for something to happen before it can proceed. To be more precise, sometimes a *thread* must wait for something to happen before it can proceed. We call that something a *trigger*.

We show a trigger by an arrow (➞) placed on the state line – see Figure 2-1 – with a caption that briefly describes the trigger concerned. In strict terms, a trigger moves the role concerned from the state preceding the little arrow to the state just after it – imagine this in terms of the movement of a token. In Figure 2-46 we see the start of a thread that waits for someone to resign. We have labelled the states before and after to emphasise the way the trigger changes the state.

Figure 2-46 – The before and after states of a trigger

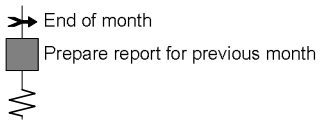


In Figure 2-2 the RAD tells us that a *Divisional Director* role instance must wait until a new project has been approved – somewhere outside this process – before it can proceed.

Triggers marking calendar time and clock time

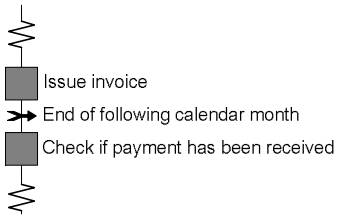
End of month might be the absolute time that triggers the start of the work to prepare the month-end paperwork (Figure 2-47). *1800hrs Thursday* might signal the start of work on the weekly wages cycle. *1st May* might signal the start of the annual budget round.

Figure 2-47 – A trigger marking calendar or clock time



A trigger of this sort is commonly found at the start of such time-related threads. But it could fall in the middle of a thread. Suppose we issue an invoice and then wait until the end of the following calendar month before checking that payment has been received. This would look like Figure 2-48.

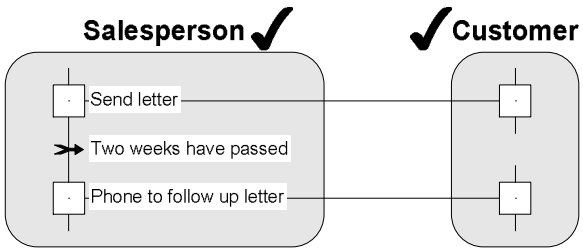
Figure 2-48 – Waiting for a time trigger in the middle of a thread



Triggers marking the passage of time

Sometimes work has to be suspended until a certain period of time has passed. Our little arrow symbol allows us to represent this. *Thirty days later* might be such a 'relative' time trigger: thirty days after an invoice has been sent out we might check that payment has been received. Figure 2-49 shows a situation where a letter is sent to a customer and the role waits two weeks before making a phone call to follow up the letter.

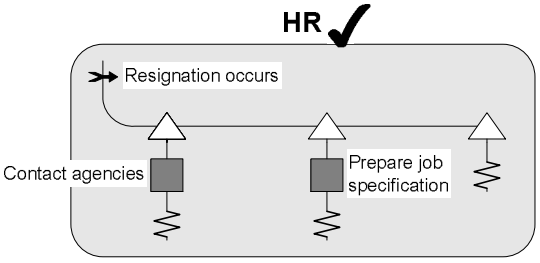
Figure 2-49 – A trigger marking the passage of time



Triggers marking external events

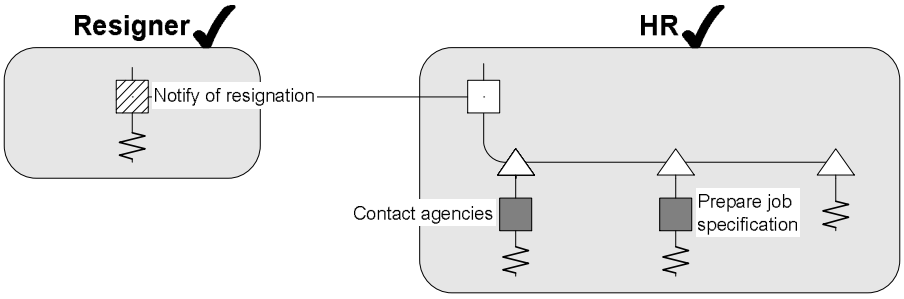
Sometimes, something happens outside our process which has an impact on its progress – we call this an *external event*. For instance, suppose we are modelling the recruitment process in a company. The resignation of a member of staff triggers a thread of activity in the *Human Resources* role in our process, preparing job specifications, contacting recruitment agencies, and so on. The person resigning would communicate their resignation to the Human Resources role in some way. We might not be concerned in our model to know the detail of the resignation, or how it was communicated or any details, other than that it has occurred and that a certain post is to become vacant. We would represent this using something like Figure 2-50.

Figure 2-50 – An event external to the modelled process



We could on the other hand adopt the perspective that the resignation is in some way an interaction between the *HR* role in our process and some other role, and we would have something like Figure 2-51. But this adds little and only moves the boundary of our model out to some portion of the role *Resigner*, leading us firstly to think (perhaps unnecessarily) about the identity of that external role, and secondly what led up to that interaction in the *Resigner* role! As it stands, Figure 2-51 shows a process which depends on the role *Resigner* deciding to use the *Notify resignation* interaction for anything to happen.

Figure 2-51 – Treating an external event as an interaction

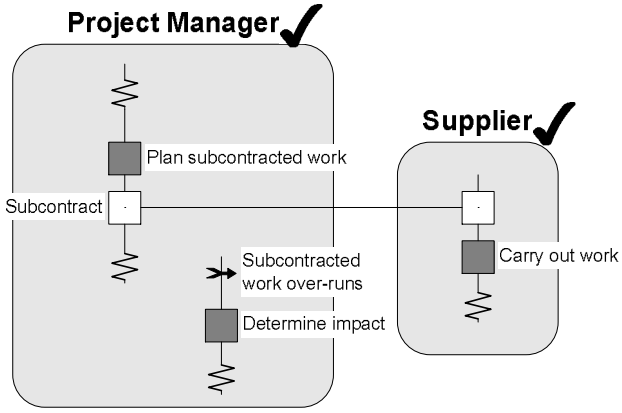


However, if we are concerned, for instance, to check the way we deal with someone who has resigned, it is obviously important that this interaction should appear, and, indeed, we might decide we want to capture precisely *how* that interaction can take place formally.

Triggers marking internal events

An external event spots something happening outside the modelled process that triggers action inside the modelled process. We can also use an event to spot something happening 'over there' in the modelled process that triggers action 'over here'. Figure 2-52 shows an example. In this case, an action has taken more than an allotted time: the subcontractor is late, and this triggers the Project Manager into action.

Figure 2-52 – Spotting an event internal to the process

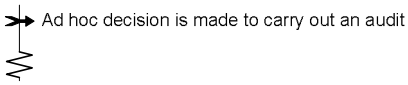


We need to be a little careful when using this way of spotting something happening 'elsewhere' in the process. When you do it, ask yourself the question 'How would this role find out that this event has happened?' If the answer is 'by asking so-and-so', then we might question why we did not model that interaction. The example in Figure 2-52 is fine: the Project Manager has in front of them a calendar and a note of when the work should have been finished. The trigger doesn't tell us *how* they spotted the over-run, simply that they did.

Triggers marking as-and-when events

Finally there is the situation where a thread of activity can be triggered at any time. It might be on a whim: 'Let's carry out an audit of the retail side of the business.' An example is shown in Figure 2-53. Indeed, if the thread can be started on a completely ad hoc basis then we have no need to show a trigger at all.

Figure 2-53 – An as-and-when event



Triggers can perform magic

An external event trigger is a way of defining part of the boundary of our model. We are recognising that something happens 'out there' that affects 'in here'. The actual effect can include a little magic: the triggering event might cause the importing of something into the process (strictly, into the role body), or it might change the state of something already in the process (strictly, inside the role body).

Figure 2-54 shows an event – *Application form arrives from customer* – that imports something into the role: the application form from the customer. The fact that we have shown its arrival as an external event says that we don't care, for the purposes of this model at least,

how this happens. It just does. The *Clerk* role now has that application form in its possession, in its role body.

Figure 2-54 – An event importing something into the role body

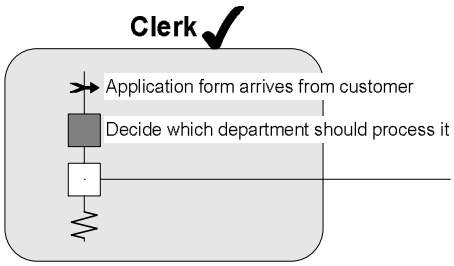
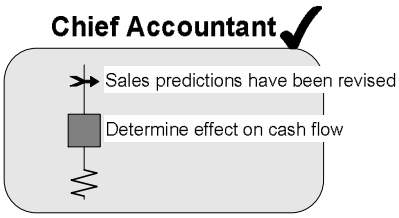


Figure 2-55 shows an external event changing the state of something already in the role body: the sales predictions in the possession of the Chief Accountant have, somehow (and we're not saying how), been changed.

Figure 2-55 – An event changing the state of something in the role body



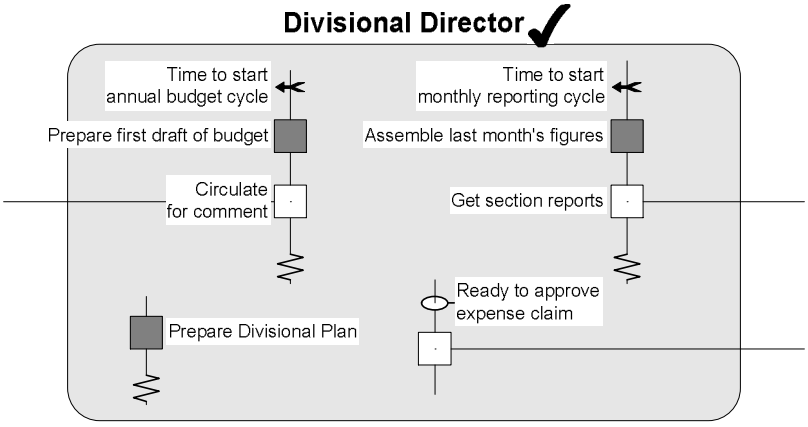
Again, when we use a trigger in this way, we should ask how the prop was changed: perhaps it was through an interaction with some other role. Should we be showing that interaction on this model?

Choosing and modelling triggers

Triggers and threads

This is an appropriate moment to remind ourselves that a role may have more than one thread. Figure 2-56 shows a part of a *Divisional Director* role which has four threads: one for dealing with the annual budget cycle and one to deal with the monthly reporting cycle; one waiting for an interaction with anyone wanting an expense claim approved; and another to get straight on with preparing their Divisional Plan. We can think of these threads as corresponding to four parts of the Divisional Director's brain: each waits for its own particular trigger to set it off.

Figure 2-56 – A role instance with four threads ready to start



Event vs cycle-driven threads

An *event-driven thread* is one that is triggered whenever a certain event occurs. It is a thread which represents our response to a certain situation. We can imagine many different sorts of triggering events. For instance, *Customer application arrives* and *Sample arrives for analysis* might be the triggers for threads that deal with individual cases: we will have a number of customer applications being processed at any one time, and there will be a number of samples at various stages of analysis at any given moment. *Budget change announced*, however, would be more likely to trigger a process that starts, proceeds and stops … and maybe is triggered again at some time in the future on the next budget change; but we are not going to be dealing with several simultaneously. In the worst case, we'll abandon any (over-running) attempt to deal with the last one and switch our attention to the one that has just come in.

A *cycle-driven thread* is an event-driven thread which fires regularly according to the clock or the calendar. There is probably (but not certainly) only one instance of such a thread in progress at any one moment and we fire an instance off at regular intervals. We will find events such as *End of month*, *5 April*, and *Midnight* at the beginning of such threads.

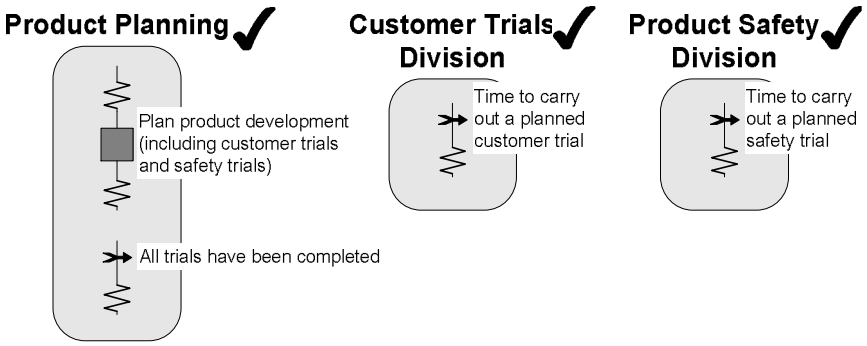
Plans and activities

It is not uncommon for certain parts of a process to be started off according to the dictates of a plan which is prepared *during* the process we are modelling. In other words, the *threads* of activity that will be carried out are known, but exactly which threads will be done and which order they will be done in might be decided on-the-fly, as the process proceeds. Those decisions – what and when – are what plans are all about. Our plan will tell us that, when a certain condition is right, a certain thread of activity should be started, but we cannot tell which or when until we are into the process.

Suppose our business is developing new electrical goods. At various points in the design and development of a new product we shall need to carry out various tests and obtain certificates of compliance with certain regulations. Precisely which tests are needed will vary from

product to product, and they will be different for a toaster and a hand-dryer. To model such a general Develop a New Electrical Product process we might therefore need to show threads of activity to do with carrying out tests and obtaining certificates, with those threads being triggered by internal events: effectively 'The plan says the moment is right.' Figure 2-57 gives an example. We have gone a step further and modelled the (internal) event that all the planned activities have been completed.

Figure 2-57 – Modelling a plan and its activities

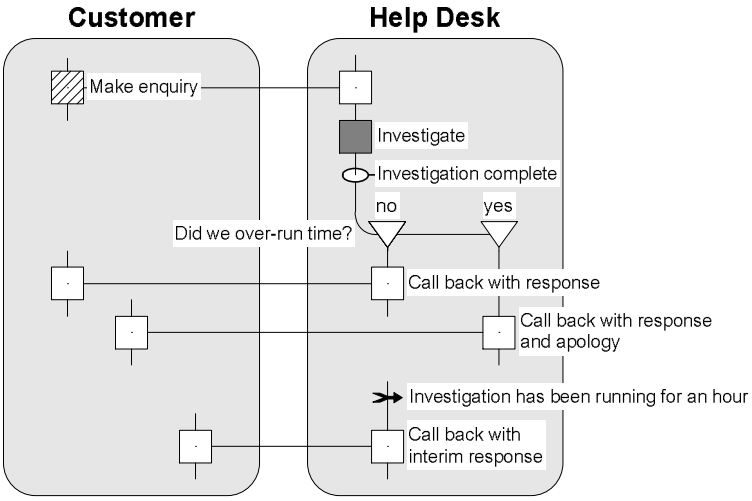


Modelling exception conditions – timing out an action

Suppose a Help Desk takes calls from customers. If the Help Desk person cannot answer the question there and then on the telephone, they tell the caller that they will go away and investigate the enquiry and call back with an answer within the hour. If at the end of the hour, they don't have an answer to give the caller, they need to give an interim report on things and perhaps give a time by which they will get back. We want to show both the intended process – immediate answer or call back within an hour – but also the process if an hour passes without an answer emerging from the investigation.

To do this we show the handling of a late action as a response to the event which is the timing out of the action. And when the action does finish (or gets aborted perhaps) we take an appropriate course of action depending on whether it has timed out – see Figure 2-58. We are using the ➞ to detect an appropriate event *inside* the process: *Investigation has been running for an hour*. This trigger sits on its own thread – it doesn't need to be 'tied into' other parts of the process.

Figure 2-58 – An action runs late



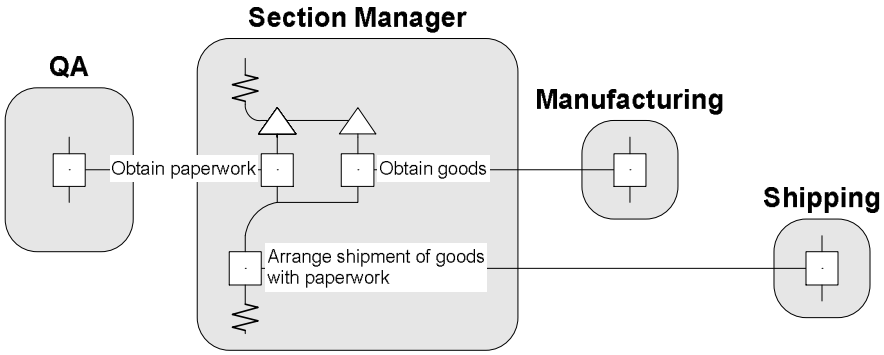
When people model processes, they all too easily fall into the trap of modelling the 'normal' or 'expected' behaviour. But if we observe people doing a process we shall often find them dealing with abnormal or unexpected situations. Should those behaviours be part of the model too? Well, the answer depends on the purpose of the model. But we might say that, if we want a full picture of an existing process, or a thorough model of a planned process, then we had better not forget those exception conditions. They will often reveal problems elsewhere in the process: earlier work that has not been completed on time, poor quality inputs such as forms inadequately completed, perhaps because they are too complicated, and so on.

Modelling exception conditions – timing out an interaction

Once a customer has placed an order, they wait for the interaction which is the delivery of the ordered goods. Do they wait for ever? Of course not: if the goods have not turned up after a certain period they will want to chase them. So, how would we model this exception condition and the way it is handled?

Suppose a Section Manager waits for goods from Manufacturing and for the corresponding paperwork from QA before passing the goods with the paperwork to Shipping for shipping. An optimistic view of this process is shown in Figure 2-59.

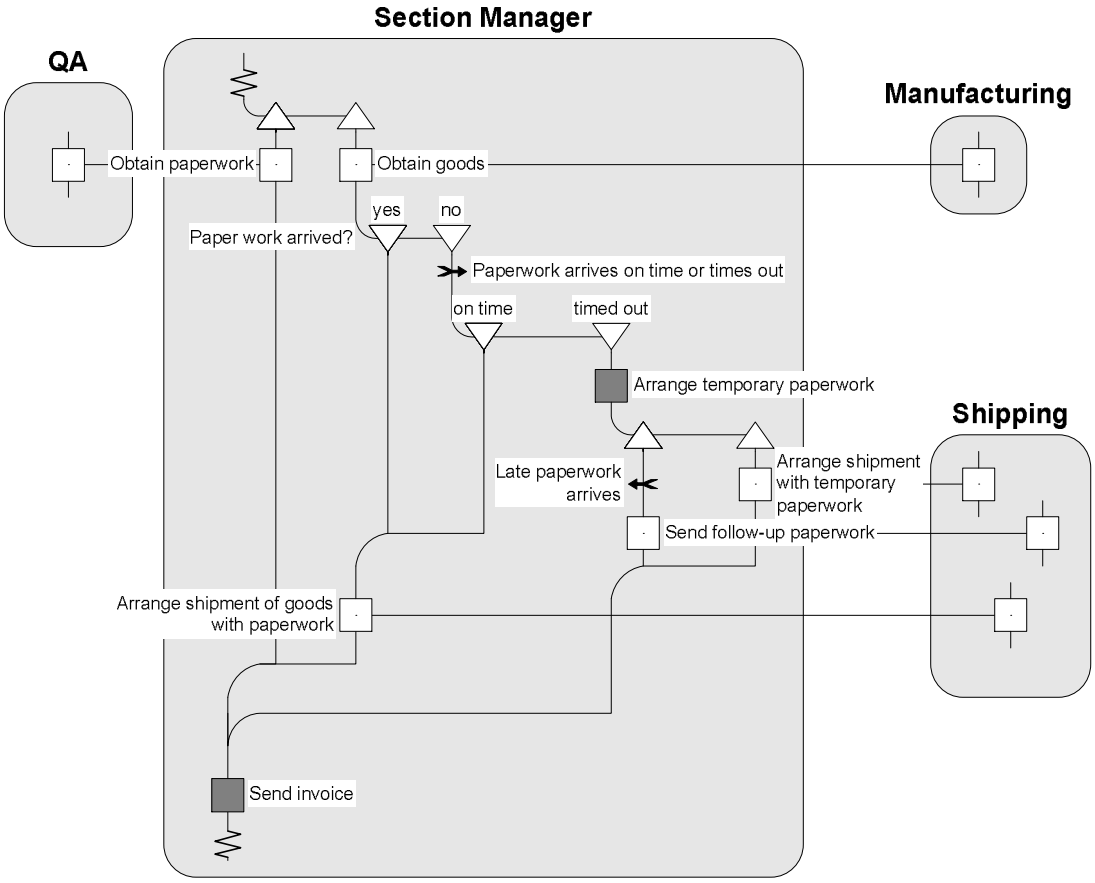
Figure 2-59 – An optimistic process



If we were modelling the process, and did not press our questions very hard, this is the story we might hear. But of course we might guess that the *Section Manager* won't wait for ever for the paperwork to turn up. Further questioning might reveal that the process as actually implemented takes a rather more pessimistic view of the world and has a work-around for when the paperwork is late, in other words, for when the *Obtain paperwork* interaction times out as perceived by the *Section Manager*.

That pessimistic – but realistic – process is shown in Figure 2-60. The *Section Manager* starts the two interactions with *QA* and *Manufacturing* as before, but now, having received the goods from *Manufacturing*, has two possible courses of action: one in the case where the paperwork arrives on time (on the other thread of the part refinement), and another in the case where that interaction with *QA* times out. In the first case, the *Section Manager* proceeds as normal, shipping the goods with the paperwork. In the second case, they ship the goods with temporary paperwork and, at the same time, wait (even longer, in fact indefinitely according to the RAD) for the real paperwork to turn up. When it does, the follow-up paperwork can be sent off and the process resumes its 'normal' course, with an invoice being sent to the consignee.

Figure 2-60 – A pessimistic yet realistic process

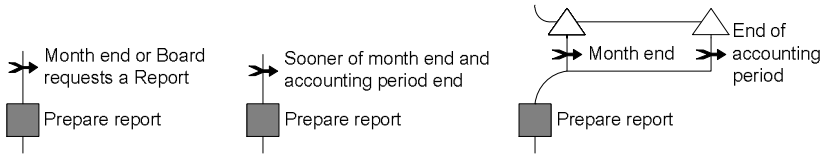


More complicated mechanisms can be modelled using a ➞ to pick up time-outs and respond to them. Spend a moment checking how the two triggers are used in Figure 2-60.

Triggers can be complex … in their captions

Figure 2-61 shows some typical situations modelled in different ways. The first RAD fragment shows a role that has to prepare a report at the end of the month or whenever the Board requests a report. The second shows the role writing the report at the end of the month or the end of the accounting period, whichever comes first. These make the point that an event can be quite complex but only needs to be described in English in its caption. The third shows the role writing the report at the end of the month or the end of the accounting period whichever is the later.

Figure 2-61 – Some tricky triggers



KEY POINTS

The trigger symbol is used to represent a variety of things that can trigger activity in a role:

- moments in time,
- the passage of time,
- events outside the modelled process,
- events somewhere inside the modelled process,
- as-and-when events.

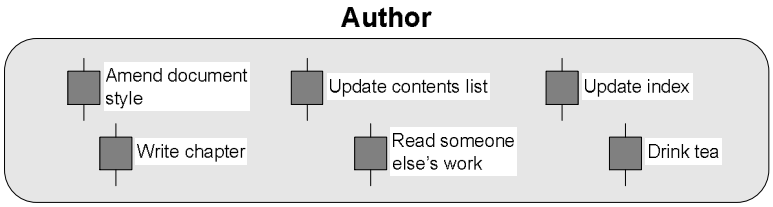
REPRESENTING THE AD HOC PROCESS

Pupil: So far we seem to have been describing processes that have some order, some sequence to them. OK, we've seen roles with several threads – separate parts of their brain for different parts of their responsibility, as you put it. But some processes are entirely … ad hoc. Things don't happen in a predefined order. Surely RADs aren't appropriate for them?

Tutor: Au contraire! You've just said how they can be handled: with separate threads! Yes, we can imagine a process where the order in which activities are carried out is decided by the actor on-the-fly, and can't be laid out beforehand. Writing a book is one: I flit between writing a chapter, changing a style across all the chapters, amending the index, updating the contents list, and so on. A part of my brain is ready to do any one of those things at any time. There can be many such parts, each corresponding to a thread of spontaneous activity, perhaps a single activity. With that thought, could you now sketch the RAD of the process I just described?

Pupil: Well, each of those threads must sit on its own … I'm not sure if they have triggers on them … I guess I don't need to show triggers if they are truly spontaneous as far as the model is concerned. How about the following (Figure 2-62)?

Figure 2-62 – A role of ad hoc activity



Tutor: Looks fine. Here's a role with six bits of brain each ready to start a thread when the actor feels like it – pure ad hoccery!

REPRESENTING PROPS

Processes involve things. The states of things affect the course of events and are affected by events. When I work on a chapter of the book, the state of that chapter changes. When the chapter is finished I can have it reviewed.

These things are the props for the role. Props can include the resources the role needs.

We might be happy to simply note the props concerned in captions to actions, interactions and decisions:

Prepare the Project Plan

Assemble the engine

Deliver the book that was ordered

Hand over the terms of reference for the work

Is the purchase over £100?

Was the employee working abroad?

But where are these things? Where are the Project Plan, the engine, the terms of reference? In the words of Bluebottle, everyone's got to be somewhere. The *Riva* answer to this is 'In the body of a role instance.' If I am preparing the Project Plan as (an instance of) Project Manager, the Project Plan is on my desk. If we, an Engine Construction Cell, assemble the engine, it is in our assembly area. If I am deciding if the employee was working abroad, something in my role body must allow me to decide. And that something must have got there somehow.

This is more than a matter of geography or cubicle politics of course: my desk is where I keep my resources to carry out my role; our assembly area is where everything we need to assemble engines – equipment, tools, parts – is held for our job. When parts are delivered (via an interaction) they end up in our role body. When we deliver an assembled engine, it passes from our role body to that of the Final Assembly Team.

If it's appropriate for the model in hand, it makes sense therefore to show props within the grey area of the role on a RAD, just as a simple list.

CASE STUDY

To close the chapter, let's take a look at a RAD of the sort that we might produce in real life. I have chosen it to illustrate a number of points. See Figure 2-63.

The process is of a type that we shall soon be referring to as a 'case process': it's a process that deals with one something, in this case a 'candidate product'. So the process is triggered when the Marketing Director detects the fact that a candidate product has been identified. A new responsibility is created to deal with it in the form of a Product Manager (the only role with no pre-existing instances). There is then a flurry of concurrent activity from a number of roles in the preparation of a dossier that goes to the Board. They might reject it (end of process), or ask for it to be resubmitted, or pass it for development. During development, the Board is kept up to date with progress, and finally Product Assurance run an Acceptance Test over the product before it is deemed ready for sale. The process therefore has two possible outcomes.

What is noticeable about this model is how the interactions outnumber the actions. It was drawn to emphasise the collaborative nature of this process, to stress how everyone's involvement has to be coordinated to ensure success. Since time-to-market is important for a product company, it also aimed to show where different roles would be operating concurrently, and there are part refinements within roles, allowing them to get on with several things at once. At one point, Development can be involved in at least three concurrent threads of work. Once again, interactions are important in getting material out to people and gathering it back in. Co-ordination.

In later chapters we shall look at how a RAD can be used to get to answers to the sorts of questions we might ask about a process. But, as well as providing us with an example of a RAD, perhaps we can also look at the sorts of things we can spot when something like this is drawn.

- The use of grey boxes to contain the activity of roles helps us see where roles fit in the process (much better than swimlanes, we might say).
- We might quickly observe that some key roles are involved early on, but disappear from the work in the later stages. The Marketing Director, for instance, has no involvement apparently once a project has been given the go-ahead by the Board – was this really the case, or just appropriate for this model?
- A glance tells us that the Product Manager moves into a rather responsive style of management once the project has been passed to the Board for approval and – if approved – on to Development for production. They appear to be on the receiving end of interactions from others. What are they doing to manage things proactively? Development seem to be in the driving seat.
- If the Board does reject the candidate, it seems to be a private matter: no-one else gets to hear about it. Perhaps, as far as this model is concerned, we weren't interested in the process for rounding off a rejection.
- Acceptance Testing can, it appears, only result in success. Do we ever get to the point where we decide that it has all been a terrible mistake and abandon the thing in Acceptance Testing? Indeed, we can imagine the product being abandoned at all sorts of

points: if costs spiral, or the timescale goes out too far, for example. We haven't shown those measures being monitored or responded to. This could be because there is no mechanism for that monitoring or because we chose not to model it.

- ☞ We presumably could have shown Development developing the product as a simple black box, but we have chosen to unpick it at least as far as showing a reporting cycle to the Board, and the need to develop in parallel a number of components that make up the product.
- ☞ The model is largely abstract in nature. Although it shows real posts and groups in the organisation, the actions and interactions are generally expressed in terms of intent: the Product Manager 'requests assessments', and we are not told how that request is made or what form the assessments take (written, emailed, verbal?).

These few observations should make it clear that as well as allowing us to capture the dynamics of a process precisely, a RAD is *revealing*: it allows us to see the nature, the style, the flavour of what is going on, and hence allows us to answer the sort of questions we shall want to ask about the process.

Figure 2-63 – Handle a candidate product