Chapter 5 Effective Documentation

The point of using an abstract, pictorial language for analysis and design is that you can say the most important things clearly. Each issue is dealt with only in as much depth as you require, unobscured by too much detail. Separate issues are presented separately.

The art of good documentation is the art of leaving things out; but in a consistent and useful way so that your documents are well structured, concise, and readable. In a sense, our ideal documentation would look like a cross between Knuth's "Web" of literate programming and a clear structure that separates specifications and external views of things from their internals.

The goal is not to create a nice, or large, or impressive-looking diagram; instead, the goal is to write an explanation of requirements, design rationale, and architecture. The diagrams help support the explanation, so you should structure the document with interleaved narrative, examples, and fragments of formal models and diagrams.

This chapter deals with how to use the notation to help convey your ideas both at the whiteboard and in documentation.

5.1 *What's It All For?*

Why are we drawing pictures and writing invariants and postconditions rather than writing program code? After all, it's the code that runs.

Well, first let's assume that you believe that there is something more to software design than just hacking the code; otherwise, you wouldn't be reading this book. We believe in early coding sometimes—for example, when we're building a small prototype from existing components. But for team work or for large projects and products that will last long enough to be updated, a proper design process is essential, and part of that is the use of a good design notation.

As we've already seen, program code is precise but tends to force you into too much detail. In design, you want to describe only the main scheme before proceeding to the detail. There are experimental ”wide-spectrum” languages that cover both specification and coding; but unfortunately, most of us must work with the the most widely available tools rather than the technically best. (It's the same with operating systems, programming languages, videotape standards, and mousetraps.)

On the other hand, natural languages and ad hoc drawings are expressive and can deal with any degree of abstraction you like, but they are prone to ambiguity to the point of having no predictable interpretation, making it easy to gloss over inconsistencies. Part of the risk is that readers of design documents and participants in discussions can come away with divergent ideas about the design.

But a more significant problem is that in the early stages of a design you can leave things unspecified that need to be decided up front. We've all been in a meeting, not long before the delivery deadline, to decide an issue that should have been sorted out earlier. Sometimes, different designers have addressed the issue on-the-fly and have done it different ways, and now there is a panic about how we're going to reconcile them.

Experience with formal notations shows that the precision of the notation tends to flush out these issues at the appropriate stage. Anyone who has spent a day or two in a specification workshop, creating a model for a business or a new system, is aware of how quickly important questions are raised. Many of these issues would never have been thought of had the requirements document been in plain natural language (English or your local favorite).

A good design notation makes it easy to highlight the issues you care about at the moment, leaving other issues until later. Just as important, it gives you a well-defined way of combining different issues when it's time to implement the design.

Notation is used for two principal purposes:

- Discussions at the whiteboard, presentations, and so on
- Documentation—a clear statement expressing the project's current state of understanding of the design

It is during discussions that many people gain their first real appreciation of the value of design notations, but much of the way you use it at a whiteboard or in presentations follows from how you use it in documentation. The rest of this chapter will concentrate on the latter.

5.2 *Documentation Is Easy and Fun, and It Speeds Design*

Documentation is arguably the most sleep-inducing word in the English language. In part, this is because it is often seen as an extra chore, something you must do in addition to the real job to satisfy fretful management or neurotic QA people. In Catalysis, we think of documentation more as a tool to help do the job well. Far from being a deviation from the fast path, it is a moving walkway toward a solid, lasting product. And, like all tools, it can be misused.

(In those cases when you don't want a lasting product, omit the documentation. This suggestion is not meant to be flippant. In a later chapter, we'll discuss short-life products that are rapidly thrown together from robust existing components. By contrast, the components themselves are carefully produced and maintained, and they merit accurate and in-depth write-ups.)

5.2.1 Bad Signs

Here are some bad signs that indicate the wrong approach to documents.

5.2.1.1 Bad Sign: No Documentation

When new people come on board, they must ask everyone how it all works or guess from the code and the work of people before them. Code and source code debuggers have never been particularly good tools for understanding requirements, problem domain concepts, or design intent. New developers easily get the wrong impression and soon start doing things to the code that are inconsistent with the rest of the design. From then on, it's downhill all the way.

The usual excuse is, "We'll do it soon—when we get some time." Developers are too busy sorting out problems and late updates to do the write-ups. It's funny how these projects are always the ones that require a lot of heroic overtime. See Ed Yourdon's excellent book *The Decline of the American Programmer.* In such a case, a possible damage limiter is to hire a suitably skilled person to spend a few weeks writing the models. This person may be able to clarify some inconsistencies and at the same time transfer some skills to the existing team. But a lack of documentation is a cultural problem and is not easily fixed.

5.2.1.2 Bad Sign: All Documents, No Code

These folks are taking the sober approach: "We'll do the design when we've signed off the specification." Now, we have been aficionados of specification for many years; one of us has done and enjoyed mathematical program proofs in his time. But we have never tried to write the whole spec before doing some code. For one thing, you can't execute a spec,[1] and executing something is vital to finding out whether what you are specifying is really what is required. For another, it is bound to be wrong before it is finished.

Producing code early is good for everyone's morale. One of the benefits of the high modularity of OO programming is that it is easy to do a sensible "vertical slice" of a program, try it out in isolation, and add the rest incrementally. Horizontal slices are also possible: What will ultimately be a database-resident distributed system can initially be

1. You may have heard the phrase "executable specification." Mostly snake oil. The point about a spec is that it states the requirements of a particular user, which are usually partial. You must add decisions to make it work. An executable thing is usually a program even if it's written in a high-level language, or in pictures.

written to run in the main memory of your laptop so that you can show it to prospective users.

The point about writing specifications and abstract designs is not to get them entirely set in concrete before implementing but instead to record what you've decided to provide. In the "How to" chapters in Part V, we look at the short-cycle incremental methods of delivery that have been found to be most effective in recent years. The main requirement of a specification is not that it be complete before the code is written but rather that by the delivery date it should be usefully complete and consistent with the code (and not too far off, in between).

The cure is to use RAD-style timeboxing: Put developers under a lot of stress to produce something demonstrably fast, with milestones for the specs and other documentation to get in sync.

5.2.1.3 Bad Sign: All Pictures

Pictorial models are a helpful part of documenting a design. Unfortunately, it is a common error to treat the pictures as the sole focus of documentation or design activities. Teams spend a real lot of money on modeling tools, draw lots of pictures using these tools, and then expect to press a button to generate well-structured, clear, explanatory documentation. Sadly, it does not usually turn out exactly that way.

Do not think of documentation as producing pictures; instead, you are trying to write a clear and concise explanation of the key decisions, whether requirements, design, or code. Precise definitions of terminology, coupled with well-written prose, goes a long way. The pictorial notations help summarize certain aspects of this; so do examples, tables, fragments of formalized specs, and informal sketches. You can sometimes write a clear explanation with an OCL fragment in a few minutes, whereas it would take an hour to create a diagram with the same information. We have seen extremely clear documentation in which the type model and action specs were described entirely in text, in the form of tables; a decent tool should keep such a view in sync with the underlying models just as easily as the pictures.

5.2.1.4 Bad Sign: Wall-Sized Documents

Sometimes you see a wall covered with a big class diagram that contains hundreds of little boxes. These diagrams are not very useful except for your boss to show off to her boss. It is difficult to find your way around such a diagram and difficult for people new to the project to know where to start. Diagrams should be used as part of a narrative explanation and not just on their own. Documentation should be structured so that the important, overall issues come first, with the detail filled in later.

The cure is to write a narrative document in normal requirements style. Every few paragraphs, use a diagram showing only those elements directly relevant to the text. The purpose of the text is to provide an immediately readable account; the purpose of the diagram is to disambiguate the text. Structure your document by separable issues.

The appearance of a type in more than one diagram means only that it has the attributes and operations gathered together from all its appearances. All OO analysis and design support tools allow you to separate big models into smaller diagrams. Some of them make it easy for the individual diagrams to be embedded within narrative text; with others, you must cut and paste.

5.2.1.5 Bad Sign: Big, Thick, Formal Documents

Sometimes we've come across a team busily engaged in writing a thick document that contains miles and miles of formally structured action specs. This is a write-only document, and it is likely to be a work of fiction by the time it is finished. Formal notation (pictures and formal text) is good at being unambiguous; pictures, in particular, are good at explaining complex relationships that would be difficult to get hold of in ordinary narrative.

Like software, documentation should be decoupled: the dependency relationships between different parts should be clear, and there should be not very many of them. Documentation should make it easy to find out about one particular aspect of the design; it should be clear what other things you must read.

The cure, again, is to embed the formal parts in narrative text. If the formal parts are getting too complex, look for ways to abstract, factoring out various aspects so as to better decouple your documents.

5.2.2 Good Signs

Not all documentation is bad! Let's look at some signs of good documentation in the sections that follow.

5.2.2.1 Good Sign: A Clear Document Structure

Document structure should be systematic, always separating external from internal views of things and further separating different aspects or areas of concern. Document structure corresponds to our ”packages” and has a clear and uncircular structure of reference between them, as discussed in Chapter 7, Using Packages.

5.2.2.2 Good Sign: A Mix of Formal and Informal

Each section contains a natural-language description as well as a more formal one, interspersed as far as possible. The formal fragments are specifically used to clarify parts that might otherwise be unclear.

Informal text can also be substituted for or used as well as any part of a formal expression. For example, it is useful to write postconditions at least in natural language and then in precise terms if required.

Similarly, sketches of situations or phenomena in the problem domain are encouraged. Use informal or formal notations, including drawings and rich pictures, tables, and snapshots.

5.2.2.3 Good Sign: A Clear Glossary

In addition to a narrative, it is useful to have an index of the vocabulary. The glossary's purpose is to link the formal terms back to the real world. Rather than a single monolithic table at the end of the document, the definitions could be introduced as needed in the context of the document structure, together with additional explanation.[2]

5.2.3 The Art of Abstraction: Keeping It Short and Sweet

Part of the art of good documentation is to achieve a consistent level of abstraction. What should be left in, and what deferred? What tools can we use to leave things out? Here are some tips.

- Treat several objects as one. In a sequence diagram, consolidate several vertical bars into one. An example is to treat a company as one entity rather than to see only the individual employees. Another example is to treat a system you're about to design as one object: During design, you split it into its internal components.
- Treat several actions as one. In a sequence diagram, consolidate several horizontal bars. For example, talk about a complex transaction as one action rather than see the detail.
- Ignore how actions are initiated. Treat the information exchanged in an action as being a parameter. Later elaboration will discover how that exchange occurs.
- Model every useful concept, even abstract ones, as an object. Remember that specification types need not appear directly in an implementation.
- Characterize actions by their goals or effects rather than by the detail of how they work.
- Partition different views or aspects of whatever you are describing. Remember that you can write multiple specifications for the same action and have the same type appear in many drawings.

5.2.4 Separating and Joining Definitions

A good document takes one issue at a time, and each type will be involved in more than one issue. For this reason, OO analysis and design tools usually support the idea that each type or action may make an appearance in more than one drawing. This means that anyone wanting a full understanding of a type must search out all its appearances or, better still, get the tool to show the type's definition in full.

The operation of bringing together the different appearances is called *joining.* For a type, you list all the attributes that crop up in any of its appearances and do the same for its associations. Operations or actions in which the type takes part are also listed.

Actions (whether localized to a type or symmetric) can also appear in several places, and action specifications can be separated. You can give one postcondition spec in one place and give different ones elsewhere. This approach helps you to discuss separately the

2. The table version could always be generated as an alternative view but not vice versa.

different constraints on the outcome of an action. For example, in one place you might say that the check_out action records that the given video is hired to the given customer; elsewhere, in a section about stockkeeping or marketing records, you say that check_out increments a count of the number of hires of the video. The join of the part specifications of one action is simply the AND of them all.

Invariants can also be separated: the joined actual invariant of the type is again the AND of the invariants given in its various appearances.

We will use the idea of join again when we discuss model frameworks in Chapter 9, Model Frameworks and Template Packages. Partial models kept in libraries can be composed to make complete models; their documentation need be kept only in one place.

Table 5.1 shows a few basic techniques for factoring models into parts that can be more easily recomposed. Well factored models have clearer documentation.

Table 5.1 Techniques for Factoring Models and Documentation

Construct	Usage
Effect	If actions have a common state change in their post-conditions; makes specifications more succinct and natural.
Invariant	To impose a static constraint on every action's post-condition; makes specs more succinct and natural.
Effect invariant	To impose a common state change on every action or on a specified subset using `change1 ==> change2`; makes specs more succinct and natural.
Convenient attributes '	To simplify all specifications; to make algorithmic query operations trivial; to make specifications more natural.
Parameterized attributes	To avoid data-modeling explosion of artificial types and attributes; defer variation in how an attribute is computed.
Collaboration	To define a partial view of the roles of a set of objects separately from other roles they might play.
State type	To treat as a type those members of another type that satisfy some predicate; to make specifications more natural.
Subjective model	To describe constraints on objects of some types that are specific to their being in some relation with others.
Derived specifications	To highlight aspects of a specification that are of particular interest even though they are already implied by what is stated.

5.3 *Reaching the Documentation Audience*

The documents are intended to ensure that everyone has a consistent understanding of what's being built. So the audience is everyone interested in the system.

5.3.1 The Development Team

The first audience is you and your colleagues on the development team. The documents express your understanding of the business, the requirements, and the design you are engaged in. By creating an explanation of these things, you clarify your understanding of them, raise questions that otherwise would not have come up until coding, and iron out inconsistencies that otherwise would have been glossed over. This is a repeatedly observed effect, and the more precise you try to be, the stronger the effect. It is therefore a good investment to take the time to write Catalysis models as early as possible.

5.3.2 The Maintenance Team

These people will be doing the most work on the product. (Under maintenance, we include everything done after first delivery.) Few of them will have the time to understand the entire project completely before making the updates they've been asked to do.

Traditionally, this means that a programmer reads the code, makes a good guess at how the design works in the region of interest, and makes an update, which may be inconsistent with the overall design principles. After some years of these patches, many spurious couplings are introduced, and it becomes increasingly difficult to tweak one end of the system without the other end falling over.

The most valuable purpose of documentation, over the life of the system, is therefore to give maintainers a clear understanding of the architecture and design of the system so that they can quickly make changes that are just as good as if the system had first been designed that way. Pretty pictures of architectures, with no precise interpretation, do as much harm as good. By documenting it well, you are significantly extending the prospective life of your design.

5.3.3 The Clients

The people who will use your product need to see the requirements, although not the design. How this works depends on what sort of clients they are. If they are another design team using your software component, you can expect them to understand the documents directly. Still, it is essential that you clearly separate the external view from the internal workings. The requirements documents form part of the client's contract with you.

If they are not software people, your role as analyst is to interpret between them and the formal models. The models represent your team's clear understanding of what you are to provide. By writing them, you are taking the fuzzy and inconsistent desires of the client, crystallizing them into a more precise form, exposing the questions you need to ask, and going back to the client with questions, scenarios, storyboards, proactive proposals of precise definitions of terms and requirements, and prototypes (see Figure 5.1). The cycle continues until all the parties are satisfied that they know what is being discussed; your model should at that stage represent what is in the client's head (see Pattern 15.13, Interpreting Models for Clients).

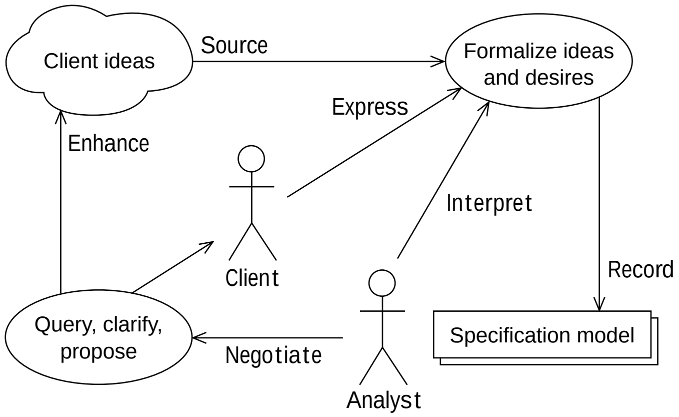

Figure 5.1 Typical interactions in building models.

The contract between you and the client can take the form of a requirements document of the traditional kind written in natural language. But having been through the process of formalization, both sides can be confident that it is achievable and consistent and masks no big unknowns.

5.3.4 Documentation Timing

As we've said, writing models improves your understanding of the system and raises useful questions. It therefore makes sense to start writing models as soon as possible. Once you are used to the technique, it becomes easy to sketch your ideas in this form. Sessions around the whiteboard with colleagues are valuable at an early stage, and Catalysis provides the language for it.

Although it is good to start writing as early as possible, it is not necessary or desirable to complete the models early. For one thing, this approach can lead to your getting hung up on writing specifications; for another, many of the details will change as time goes on. Don't expect to get the documents much more than 80% consistent or complete except at big milestones such as requirements sign-off and product delivery.

Don't insist on drawing a line under the spec before going on to the first prototype development. The RAD idea of timeboxing applies to each phase of development, including determination of requirements. Set a date by which requirements will be signed off, but start designing before then so that early storyboards or prototypes can be achieved. Section 10.11 presents an excellent example, using a system built from heterogenous existing components, of how and why models, designs, and implementation should not be done using the waterfall approach. The example, nevertheless, shows the value of models prior to code.

Formal documents take longer to write than traditional informal notes. And the more precise you make them, the longer it takes. The big fear from a managerial point of view is that people could get hung up on writing this sort of thing instead of getting on with the real job of writing the code.

To begin with, we believe in early delivery of prototypes and vertical slices, as you'll see from the "How to" chapters in Part V. And understanding what it is you are building is a major part of the real job of writing the code, and it is a lot more effective than just getting in there and hacking.[3]

Think of the code as embodying all the requirements and design decisions that have been made during the development: They're all mixed up in there somewhere. Imagine pulling out the biggest decisions—the most important ones on which most others depend.

3. Medieval architects would be shocked to see the amount of paper that gets produced before a stone is laid these days compared with their own methods; but we are aware that modern architects generally achieve a more reliable result—the landmarks we see today are the ones that didn't fall down—and can build bigger and more versatile buildings even if they are no more pleasing to the eye.

Now rinse off any smaller ones that may be sticking to them. What you have left in your colander are the main requirements and the biggest design decisions.

By writing models up front, we identify these major ingredients. The more formality and precision we achieve now, the more we clarify the picture and ensure we haven't missed or mistaken any important ingredients and hence that we don't make a cake of it further down the line, when changes will be more expensive. Anyone writing formal models soon discovers that more questions are raised and clarified in the first afternoon of a modeling workshop than are often raised in a traditional development until the code has been written.

The main visible effect is that 100% completion of the requirements comes later than in the traditional process, but there is less work to do after that stage. Writing the code is relatively easy, because the big decisions have been made. However, because we overlap the phases, code certainly gets written before the requirements are complete.

The main worry is that requirements sign-off with the client could be delayed. In practice, this tends not to happen. Determination of the requirements is a cyclic process, and if it is reasonably well managed the main features can be brought out and agreed on first. If the client initially approaches you with an informal requirements document, the essential goal is to get most of its points modeled and clarified. In consultation with the client, you cycle through your model, regenerating the client's requirements from the model. An estimate, and hopefully a contract, can be formed on the basis of the first iteration, with the understanding that further questions will be discussed later.

5.4 *The Main Documents: Specification and Implementation*

There are two main kinds of documents.

• *Specifications:* A specification describes actions using postconditions (perhaps drawn with state charts); the postconditions are written in terms of a vocabulary of objects and their static relationships, which is described in a type model. These documents must be inspected at *external design reviews.* Types, rather than classes, are the main focus.

• *Implementations:* An implementation shows how a component works, describing the objects that are inside, how they interact, and in what order things happen. These documents are the internal counterpart to the specifications and are shared by the *design team.* Together with types, internal components and classes become an important element.

Specifications are defined by types. In an implementation, the interactions are expressed with collaborations, and the complete prescription of the design of any one component is called a class.

A component is anything you can design as a unit, whether it is a single C++ class, a complete software application, a part of an application, a piece of hardware, or even a business department.

Implementations always include some element of specification (but not vice versa), so the two kinds of document are never entirely separate. First, an implementation need not be complete down to the finest detail; it might interconnect a few smaller component specifications, leaving their implementations to be chosen separately. Second, the concepts the participants talk about when they interact can also be specified rather than implemented; using types, we can define what information must be sent without deciding its representation.

5.4.1 Spec and Implementation in a Typical Project

You typically start a design project by looking at how the existing business organization works—that is, how it is implemented. At this stage, you may reorganize the business (on paper at first) , including as interacting components a new or modified department, piece of hardware, or piece of software. You specify this new component so that it supports the interactions you require it to take part in and then go on to implement it. Figure 5.2 outlines this stage with new components B1 and B2 included in the envisaged business model.

So the following three main documents generally crop up in a standard software development project.

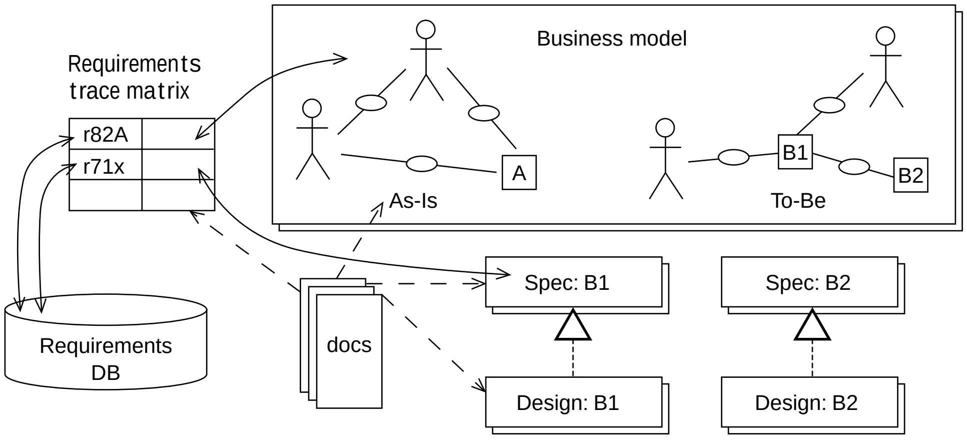

Figure 5.2 Typical separations of the documentation structure.

- *Business model:* The “outside.” An implementation describing the users’ picture of the world they work in: what concepts there are, how they are related, and how they interact. ”Business” here means any part of the world in which you’re interested, including software systems.
- *Component specification:* The “boundary.” Specification of the behavior of a component, saying nothing about its internal design. The document may also show collaborations between the component and the objects around it (whether soft, hard, or live). It is

frequently structured into different subject areas, which focus on different aspects of the system.

• *Component design:* The "insides." Implementation showing how the component works, detailing how responsibilities are distributed among the internal components.

These three levels, with various names, are found in most books about OO design. In practice, you might use only some of them, or you could introduce in-between layers such as an abstract spec, a detailed spec, a high-level design, or a detailed design.

Once you have decided on the levels, you should keep them separate. The business model will act as the basis for your component design, so you can edit a copy of the former until it becomes the latter; but use a copy rather than lose the original. When the business changes, update the business model and then update the derived models correspondingly. Good tools (which by no means includes all of them) let you keep track of the derivation trace from one model to the other, flagging the parts of the derived diagrams that need attention after changes have been made.

One business model may be common to many component designs. For example, an insurance company with many products and procedures may construct a business model to gain a clear understanding of what goes on in the company. The model can be used both to help improve the business organization and to serve as the basis for the specification of various support software. The business model will therefore be unbiased toward any particular software design.

In other cases, there is no strong need to construct a business model before you get on with a component spec, particularly if the software focuses on only one aspect of a very general business. Sometimes it is useful to create the static part of the business model, without any collaborations. For example, before designing a word processor, you first define what a document is. This is the business world in this case—the subject matter with which the software will deal but modeled separately from any concerns of what the software will do with it. This approach is particularly useful when there are standards of interchange between components: Web pages, RTF files, floppy disks, and TV pictures all have standard models of the objects without designing the equipment that handles them.

5.4.2 Continuity of Notation

One of the great things about OO analysis and design is that you use the same notation for every level of model and retain the problem domain concepts through to your code. That makes it easy to adhere to the Golden Rule of OO design: that the software have the same shape (more or less) as the business. It is possible to analyze or design the business organization, hardware, software, or any combination of them in essentially the same notation. The business, the tools that support it, and the interfaces between them are all seen as different parts of a single design.

We'll now look at the purposes of these three major models in more detail.

5.5 *Documenting Business Models*

A business model records your understanding of a business. As an analyst, you record your clients' views of the world in which they work. Suppose a client tells you, "Every customer has a name and an address; several customers may live at the same address; sometimes they move." You busily sketch a type diagram (Figure 5.3).

Type diagrams represent the static relationships between objects and the actions intrinsically applicable to them. Collaborations represent dynamic behavior, as when your client says, "A check-out means that a customer chooses a video from the shelves, shows it to the assistant, and takes it away." You can draw the diagram shown in Figure 5.4.

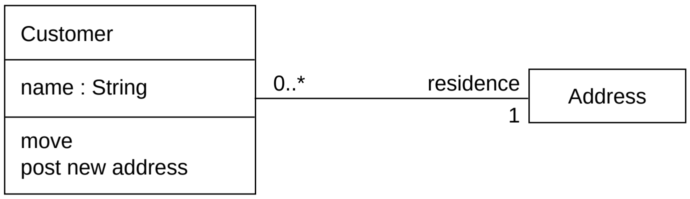

Figure 5.3 Simple business type model.

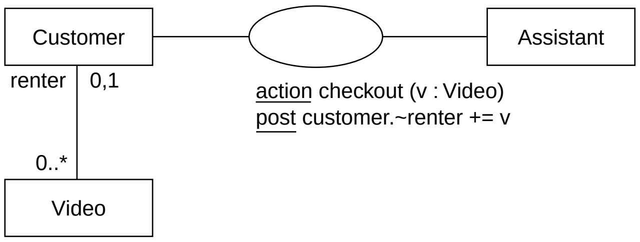

Figure 5.4 Actions, or use cases, in the business.

Part of your job as an analyst is to reconcile different views when they conflict and to clarify concepts when they are fuzzy. Suppose your client says, "Several customers may be renting the same video. Customers can reserve videos, and when a reserved video is returned, we hold it off the shelves for the reserver to come and collect it." Somewhere in there are two uses of *video*: one meaning an individual copy, and a second one meaning a title. Your job is to identify the two types.

You should also faithfully record redundant relationships and views. Your client sometimes speaks of the renter of a Video, which is a Customer, as shown in Figure 5.4. But you may also be told, "We record the date a video went out, whether a deposit was paid, when it is due back, and who rented it: That is called a rental." So the renter of a Video is the Customer named on the Rental; this is a derived relation (see Figure 5.5).

5.5.1 The Glossary in a Business Model

Annotate each type and collaboration with a summary of what its instances represent. Be careful to make it clear which real-world objects and events are included. For example, does a Customer include all the people we have ever sent brochures to, or is it only people we've actually done business with? Is a separate check-out deemed to have occurred for each separate video, or is a bundle of four (with one free) a single check-out?

The same principle applies to attributes and associations. Is the renter linked to the person who last rented the video, or does it exist only while the video is out on hire? Is the Customer associated with a check-out the person who takes the video away, or is it the person whose membership is used to rent it? (Or does a business rule prevent these from being different people?)

5.5.2 Document Different Degrees of Abstraction

For a clear understanding, it is important and useful to zoom out and in from the detail of the processes. For example, several actions can be considered to be part of the use case of

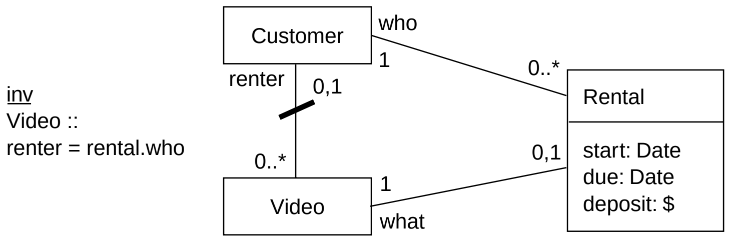

Figure 5.5 Business terms as redundant attributes.

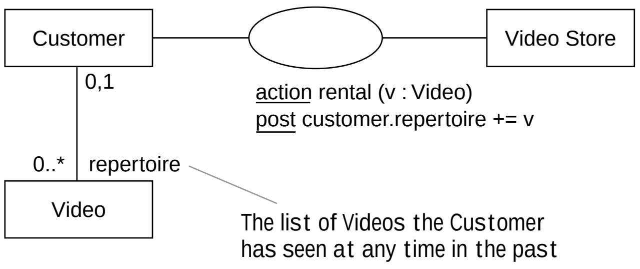
Customer
Video Store
0,1
action rental (v : Video)
post customer.repertoire += v
0..*
repertoire
Video
The list of Videos the Customer
has seen at any time in the past

Figure 5.6 Abstract business actions.

rental, the business of allowing a customer to have access to a video for a period of time (see Figure 5.6).

We should show how this zoomed-out view of things relates to the more fine-grained operations such as check-out and return. One way to do this is to associate a set of event charts with the rental action. They show examples of the ways in which the more abstract action can occur (see Figure 5.7). Each detailed action, such as checkout, should be specified somewhere.

An abstract action, together with a refinement, is called a use case. The story accompanying the event chart is called a scenario. Some tools can show the breakdown of the video store into its internal parts directly in the event chart; others make you show it in a separate diagram.

You'll find more on collaboration abstraction and refinement in Chapter 6, Abstraction, Refinement, and Testing.

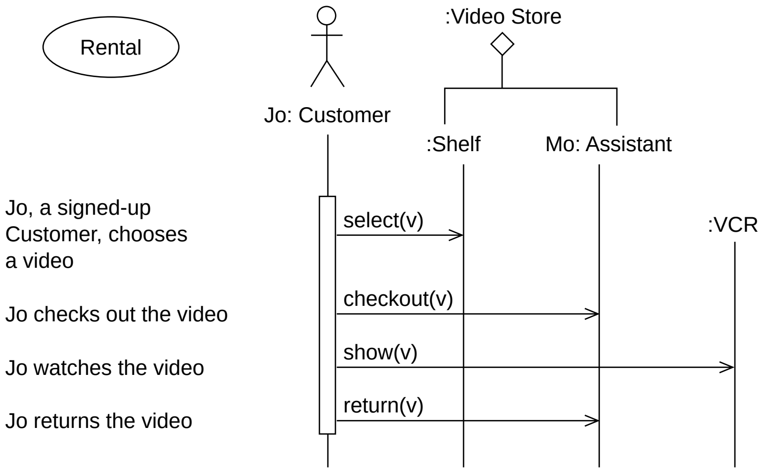

Figure 5.7 Interaction diagram can show refinement.

5.5.3 The Reengineered Business and Component's Context

After you analyze the business context, the typical next steps are to redesign the business interactions and to "install" a new component. This may involve redrawing the diagrams if the business process is to be much changed, or you may need only to refine the picture to sufficient detail to see how the abstractly specified business processes are supported by the system you intend to build. (More on the procedure will follow in the next chapters.)

The result should show how the business works, with the component of interest being one element. A component can (and usually does) play several distinct roles, providing a different interface to each of several other objects, or *actors*, as they are often called in this context; as usual, they may be other pieces of software, people, or equipment. A collaboration diagram can be used to illustrate the different interactions (see Figure 5.8). This is often called a system context diagram; it is traditional to draw the external roles as stick figures, although type boxes are also OK.

The collaborations rental support and stock keeping represent the overall business transactions our system is directly involved in. As we've seen, we can refine them separately to individual actions, using event charts and so on to show the relationship between abstract and detailed collaborations. The more-detailed collaborations can be summarized as shown in Figure 5.9.

5.5.4 Constructing the Business Model

You write business models and requirements documents by talking with people who know the business and will use the product that you are designing. They may be end users, or

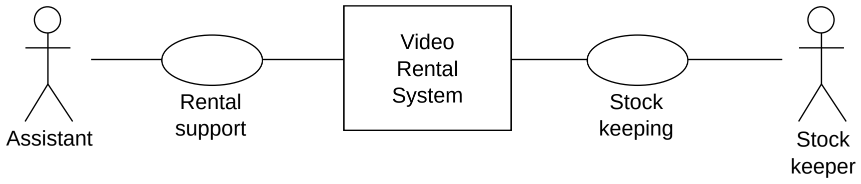

Figure 5.8 System context with abstract actions.

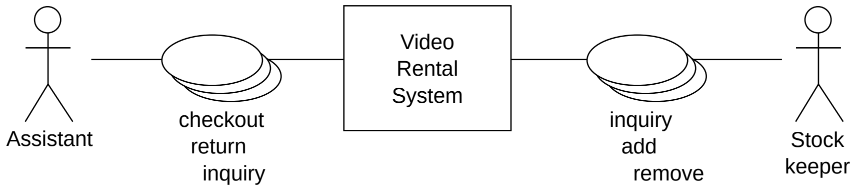

Figure 5.9 System context with refined actions.

they may be the designers of a larger system of which your component is a piece.

We recommend that you look at some of the excellent books on requirements elicitation (for example, [Graham]). The main message is to keep the users involved. Work on a short cycle: Interview, draft models, and send them back to users. Recast the story in your own words, and feed it back to them for comment.

Focus on scenarios: what people actually do in their work. Keep the scope broad to begin with. As they describe the business, construct static type models to represent their vocabulary of concepts; construct collaborations (use cases) to represent processes and tasks and the people and things involved in them. Follow use cases across departments; identify roles rather than follow departmental structure. Sketch snapshots whenever there is any questionable situation.

You will find that different users in a large business have different views of the business. Some of the differences reflect different areas of concern, others reflect differences in vocabulary, and still others are misunderstandings. Begin by making different models. Many of the differences can be resolved by writing invariants: The simple rental relationship to one user is a detailed object to another. You may find that your activities result in a clearer understanding by the clients of different parts of their own business and create a more consistent language between them.

On a practical note, it is worth making it clear, when you interview the client's staff, that your project is engaged in improving support for them in their work and that they are to be the prime consultants about what is required. After this is clear, enthusiastic cooperation is usually forthcoming. Nearly everyone loves the opportunity to explain what he or she does to a receptive ear. For that reason, do not be afraid to go back and ask again. Make it clear, when you leave the first meeting, that the usual procedure is to digest the information in your own terms and return later to ask for clarifications.

Another practical tip: to find out how a business really works, interview the people who do it and not their managers, who don't know the half of it. And don't believe what people say they do: Watch them doing it. It usually takes several visits to do the job properly.

5.6 *Documenting Component Specifications*

The purpose of a component specification is to say what the component does rather than how it works inside. (Once again, a component may be hardware, a software system or object, or a person or organization, on any scale.)

The component is treated as a single object, and its specification is a type spec. The actions that the component can perform are listed in the bottom section of the type box; they can be separated into protocols grouped by the more abstract use cases they form part of.

The component's attributes are used to help describe the actions. The type spec can be shown pictorially (see Figure 5.10). Typically, we show the types nested inside the model part of the component spec because they represent what it knows about those types, which may be different from what another component knows about them. This nested model is only a model of the component's state and says nothing about how the component might be implemented (except, of course, that it must conform to the spec).

The full name of the Video type as used in the Video Rental System's model is Video Rental System:: Video. The full signature of operations of the component follow the form Video Rental System :: checkout(video, customer). This follows C++ convention.

5.6.1 Factoring Operation Specs

Operations attached to types inside the component model have a special "factored" interpretation. Everything the model says is part of the behavioral spec of the component: There are no internal parts. So if we attach a return operation to the Rental type, we mean that it is part of the abilities of the Video Rental System to deal with a return relating to a particular rental. Because this is only a model, the implementor is under no obligation to assign the return operation to the Rental class or even to have such a class. (A sensible OO design from scratch would do so in the interests of mirroring the spec; but the designer might reuse existing designs that do things a different way.)

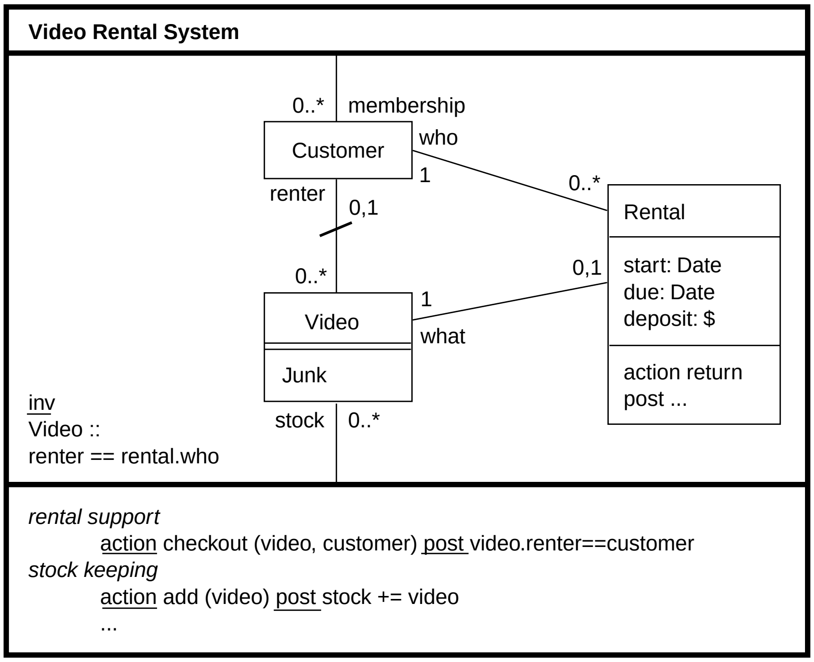

Figure 5.10 Type specification for a system.

The full signature of the return operation is Video Rental System :: Rental :: return ( ).

5.6.2 State Charts in Component Specifications

A state chart in a component spec refers to the component's behavior (see Figure 5.11). It can focus on one of the component's model types, but again, there is no guarantee that such a class will exist in an implementation, only that the information it represents will be implemented.

All the operations are those that the component as a whole makes available to the outside world. Some of them may be factored into a model type, and, in those cases, the diagram shows which objects the operations should relate to for the transition to occur.

One occurrence of a component operation may affect several model objects simultaneously. For example, a return puts the relevant rental into the "done" state and simultaneously (as we can see from Figure 5.11) sets the related Video into available. The diagram states that this occurs only for the Video whose rental is the one being returned.

In some complex cases, we want to specify that when a particular transition in one model object occurs, then some other transition should also occur in another. We do this by raising a *private event* in the component, writing ^ eventname against the causing transition (or in a postcondition). Its meaning is that all the effects of eventname elsewhere in the component should take place. For the specifier, the cause and the targets of the event are decoupled; for the implementor, it is necessary to find all the targets and ensure that the appropriate effect takes place.

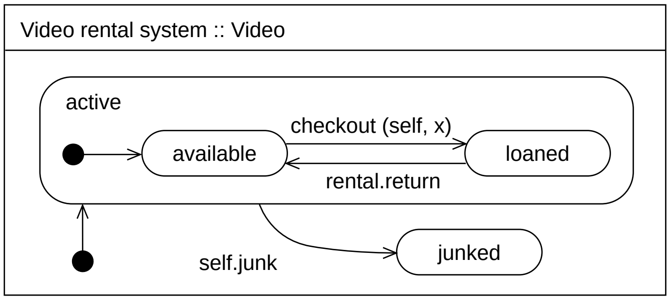

Figure 5.11 State chart of a specification type.

5.6.3 The Glossary in a Component Specification

The types, attributes, and associations should be annotated with what they mean to the component, rather than globally. This is only a shift of emphasis and might not be very different from the business model. However, because one business model may be associated with many component models, there will typically be a difference in what is included.

The glossary entries for the actions are the pre- and postconditions (formal or informal) with a general natural-language description.

5.6.4 Writing a Component Specification

As with business models, the main message is to keep the users well involved. The worst method is to take an informal requirements document, go off into a back room for three years, and then proudly present users with something that isn't what they wanted in the first place and is inappropriate now. You need to make it clear to users that their ideas are invited, take steps to stimulate ideas, and keep them in touch with the process.

Make a model by adapting the initial statement of requirements, by interviewing prospective users, and by using domain knowledge and existing systems and standards. Much of the static type model will be taken over from the business model. The big difference is in the system context: what the system is required to do and, more important, how people will do their jobs after it is installed.

Workshops are a good mechanism for stimulating and gathering ideas. Bring together key representatives of the user departments involved. The workshop also tends to encourage people to identify with the project and to help different groups of users to understand the issues of other groups.

A workshop should aim to produce scenarios, the most accessible notation to general users. Different small groups can work on different use cases. The main job of the facilitator is to show the groups how to abstract both actions and objects and not to get hung up on details. Once people understand it is acceptable to omit detail, they are reasonably good at it, especially if sessions are timeboxed.

In a system that has a GUI, storyboards—sketches of successive screens—are a well-tried method. Play-acting workflows and business interactions is also both effective and fun, as are snapshots.

Once the party is over, it is your job as analyst to abstract and refine, find generalizations, clean things up, and write a proper specification. This process will probably require much more detail on some of the collaborations. The workshop having established the main ground, you can go to individuals to ask for the details.

As you formalize the action specifications, you will find that the static types adopted from the business model contain superfluous information in some parts and inadequate information in others. Notions such as ”currently logged-in user” are relevant only to a computer system. Adapt the model as required; but the changes should be extensions and deletions rather than alterations. If you find yourself altering something, it is probably a mistake in the business model that you should fix; or it may be that you are dealing with a slightly different concept, in which case you should give it a different name and model it separately.

Once a clear model of at least one aspect of the system is obtained, use it to construct slide shows and prototypes and to reconstruct the scenarios from the workshop in more detail. The process will have generated plenty of questions. Discuss them with the appropriate users.

Cycle until time runs out; meanwhile, implementation should be starting, beginning with key parts that can be delivered for trials ahead of the rest.

5.7 *Documenting Component Implementations*

An implementation uses collaborations to define interactions; types to define interfaces and roles between objects; and classes to define the implementation of individual objects.

5.7.1 Internal and External Views of the Implementation

Always be clear about who the document is for and why it has been created. A document, even of an implementation, that is meant for a client of the implementation should not bother with showing internal implementation decisions. Instead, it should show only the types that the client needs to know about along with their type specifications.

Documentation of a framework—a set of abstract and concrete classes and types that must be extended by the client before using it—should also explicitly include the superclass/subclass interface and describe what is expected of anyone extending and overriding methods in the framework.

5.7.2 Classes and Roles in Design

A class diagram is just a type diagram with a different emphasis. The purpose of a class diagram is to represent the classes that appear in the code and the roles that they take in the working software.

The emphasis in design shifts to the responsibilities of the instances of each class (or larger component). Each class has a defined purpose for its existence that determines which operations and information should reside in its instances. This purpose should be documented succinctly in the annotation of a class. A standard rule of thumb is that better designs have classes that can be more briefly described.

Because a class diagram is closer to code, navigability arrowheads appear on the links. Annotations relating to privacy can also be added.

One class of object often plans several distinct roles. Good documentation is structured to describe these roles separately as collaborations, and then shows how they are synthesized in one class. For this purpose in Catalysis, we use the join relationship between the roles. Different tools support different ways of doing this. For example, in Rational Rose, roles are shown as actor symbols that take part in collaborations (use cases); a class can be shown as implementing several roles (see Figure 5.12). (The «join» stereotype shows that this is not pure subtyping.)

5.7.2.1 Documenting Designs versus Specs

A design document should also explain how the design realizes the specification—that is, how the individual type models of the designed components together realize the specified type model—and how sequences of interactions in the design realize the required external behavior. The details are covered in Chapter 6, Abstraction, Refinement, and Testing.

5.7.2.2 Class Diagrams and Code

At the lowest level, class diagrams can correspond directly to program code: attributes and associations turn into pointers or (in languages such as C++) contained objects. Many OO design tools provide the means to turn a class diagram into skeleton code and vice versa. However, we should be careful not to use this facility too readily; it is useful only at the very bottom layer of design.

For example, your code could include mention of various classes and interactions that deal with making objects persistent; the extracted diagrams will also include this persistence information. However, at a reasonable level of abstraction, I'd like to design my collaborations ignoring the practical need to move some things to and from disk. There are some very good packages, OO databases, and tools that help make this kind of thing reasonably transparent; unfortunately, they are not yet very well integrated with the OO analysis and design tools. Before you can generate code, you must take the design down to the level at which some persistence mechanism is explicit.

Code generation is nevertheless a useful facility, at least for creating prototypes. It is easy to take the packages from your business analysis and turn them into the skeleton of a rapid prototype.

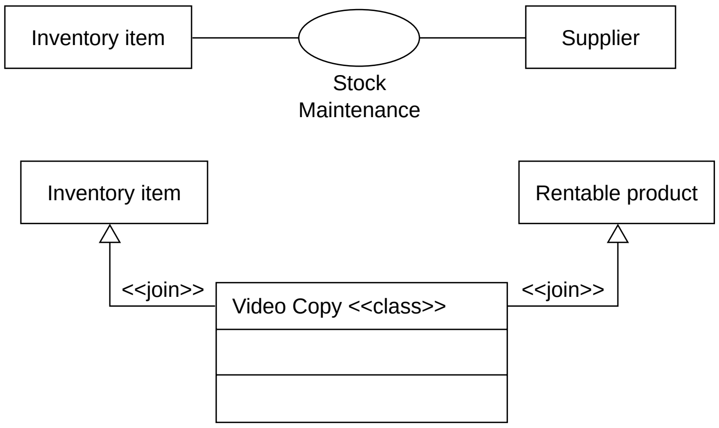

Figure 5.12 Classes can "join" roles in different collaborations.

5.8 *Summary*

The purpose of documentation is to explain the essential parts of a requirement, design, or implementation in a way that would not be obvious from the code itself. Think of all the documentation parts of development as writing this explanation. It includes model diagrams, examples, informal sketches, formal specifications, storyboards, tables, and so on only because they make the explanation clearer, uncover hidden issues and problems, and help you focus on some things while deferring details of others.

No less than the code, the documentation must have a clear structure. Documentation that is separated and decoupled typically follows a prototypical form.

- *Business model:* A (section of a) document that describes the business or problem domain, precisely capturing object types, attributes, invariants, and actions. This document may itself be structured based on
  - *Subject areas:* Different views of areas of concern in the business. For example, a video rental business may have subject areas for rentals, accounting, and marketing.
- *System or component specification:* One or more documents, each of which describes a black-box view of a software system or component, specified as a type. In turn, it may be structured as
  - *System or component contexts:* A section that describes the use of the system by external agents to accomplish business tasks. It includes system contexts as collaborations, scenarios, user-interface sketches, and prototypes.
  - *System or component type specifications:* A section that uses static type models and action specifications to specify the type of each system.
  - *Subject areas:* Different views of each component, including those of different users or interfaces it offers.

- *Internal design:* One document for each system or component, showing how it is built internally.
  - *Technical architecture:* The domain-independent pieces of middleware, databases, libraries, and so on that will be used in the implementation. This is documented with package diagrams and with collaborations.
  - *Application architecture:* The major components, how they partition the business logic, and how they interact, defined primarily with collaborations and types.
  - The main types offered between components, separated from the corresponding classes that implement those types.
  - Documentation of the refinement between the design and the specification.