Chapter 15 How to Specify a Component

To specify a type means to describe the behavior required of any object that may claim to be of that type, regardless of its implementation. An object is a member of that type if its behavior complies with that specification.

Documenting a type specification involves describing the relevant actions an object participates in, and expressing the effect of those actions in terms of a type model of that object. The specification should be satisfied by any correct implementation, and the type model is an abstraction of any correct implementation.

In this chapter we describe concrete steps towards building a system or component type specification, to describe how an object behaves in response to any external such as a request to perform some action. The approach here is mostly top-down; Chapter 16, *How to Implement a Component* (p.627) describes how to design the insides of a component, and Chapter 18, *How to Reverse-Engineer Types* (p.671) covers how to reverse engineer a specification from an existing implementation.

This Chapter has two main parts. Section 14.1, “Patterns for Specifying Components,” on page 613, describes patterns often useful when building a component specification. Section 14.2 onwards illustrates the construction of a specification for the case study.

15.1 *Patterns for Specifying Components*

This section gives the major steps in building a component specification.

Pattern 14.30, *Specify components* (p.615) motivates the extra effort involved in separating specification of expected behavior from its implementation. Pattern 14.31, *Bridge requirements and specifications* (p.617) outlines a pragmatic view of requirements, as stated and understood by a user, and the more precise specifications that a developer might use to understand what must be built. Pattern 14.32, *Use-case led system specification* (p.619) explains why a use-case driven approach to requirements capture is prudent, but should always be interleaved with tools of more precise specifications.

Pattern 14.36, *Construct a system behavior spec* (p.628) — Pattern 14.38, *Using state charts in system type models* (p.636) describe concrete techniques to build the specification of the system of interest. Pattern 14.39, *Specify component views* (p.640) and Pattern 14.40, *Compose Component views* (p.642) explain how to show that separately specified components, when composed together in a particular way, realize the required behavior.

A development process should not leave large "leap-of-faith" gaps between models. Pattern 14.41, *Avoid Miracles ... Refine the spec* (p.644) describes how techniques of refinement and re-factoring can help to reduce such gaps to a manageable level.

And lastly, Pattern 14.42, *Interpreting Models for "Clients"* (p.646) provides concrete guidelines for reviewing formal specifications for clients or users.

Pattern 15.1 Specify components

Specify what you are going to build, separately from actually building it, and mostly before you build it.

Intent

- Identify conceptual problems and obstacles early;
- Reduce misunderstandings between developers, and reduce confusion in each developer's mind
- Lengthen product life

Considerations

Getting straight into the code can be very useful for building a quick prototype. It can also be all you need to do, in small projects (1 person-month or so). And when you start with existing components, a detailed specification may not be achievable with the parts you have.

But for anything bigger, it is useful to separate out the more important design-decisions from the less important ones: so that they can be thought about and discussed separately, before committing too much effort into generating the dependent detail. Even for existing components, a slightly more abstract spec is still invaluable.

After the code has been written, a spec is valuable for two purposes:

- So that those who will maintain the code after you will understand your concepts, and update it in accordance with your vision.
- So that users (whether external or software) will have a clear understanding of what to expect from your component. This is especially important for a component that will be reused in different contexts that you are aware of — we are no longer in the days when the designer of the only component that talks to yours is sitting in the next cubicle.

Strategy

Therefore, write a specification for each component — whether it is a complete software product, or a group of just a few objects in a larger program. Exceptions can be made for small components, and rapid-assembly products that are not expected to have a long life.

It is not always necessary to write the specification before writing the code. Indeed, it isn't unusual to write some experimental code, and then go back and try to make sense of what you've done.

Nor should you get the spec more than 80% done without some implementation.

Consequences

It is a very common experience with anyone who uses more formal specifications, that a single afternoon's work on a model at the beginning of a project can raise all kinds of subtle but important questions that would not have been noticed in writing the more conventional informal requirements; and that would often not have been noticed until the coding stage —þtoo late to solve the problem coherently.

The objection is sometimes raised, that it is easy to get tied up in writing a specification, in preference to getting on with the real work. Firstly, never wait until it is 100% done. Secondly, the decisions you are writing down in a spec are just the most important decisions that would have to be taken sometime in the design no matter how you approach it. We're only separating them out, to avoid needing to backtrack later. Any extra time spent writing and maintaining a document is repaid within the 70-80% of the product's life that comes after its first delivery.

Pattern 15.2 Bridge requirements and specifications

Requirements are for users; specifications are for analysts and developers.

Intent

Bridge the gap between users and developers, without loss of accuracy, when specifying a software product with non-software users.

Considerations

The notations we discuss in this book form a specialised language for developing and discussing software and other systems' behavior. It provides the benefit that we, the analysts and designers of the system, can use it to form a clear picture of what we intend to provide for the users.

Like the language of an electrical hardware specification, a formal specification (with diagrams and formal descriptions) is not directed at end users.

Strategy

Therefore, write and maintain two views of the specification: one in terms that users understand, and the other in more formal terms.

They may be in separate documents, or they may be interleaved in a single document, as we saw in Chapter 5, *Effective Documentation* .

There is a cycle between the two documents (or two views): use the formal statements to form questions you use to improve the informal statements. In this way, you build a good understanding of the users' views, help them understand what's possible, and keep them well involved in the development. (Any book on Rapid Application Development elaborates on this point.)

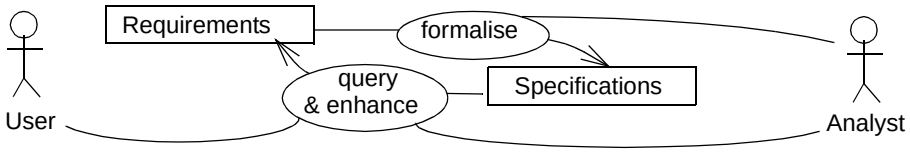


Figure 15.1 Typical user/analyst cycle with requirements and specificatior

Vocabulary

Requirements	A requirements document is a document written in terms that can be discussed with users.

Specification	A specification is a document primarily intended for our own use as developers. It represents our current understanding of what we're providing for the users.

Consequences

Requirements management. Use a suitable tool to link clauses in the informal narrative to the relevant elements of the specification (such as invariants and postconditions). These requirements-management tools perform a kind of configuration management function: if any modifications are made to one end of each mapping, the other end is highlighted. Many-many mappings are allowed, and the linked elements can be in the same or different documents. The tool cannot know what links to make, nor how the different elements should be brought into line after a change occurs: but they are very useful in tracing the propagation of updates.

Requirements and specification cycle. Requirements statements often look quite solid when you first look at them; but as soon as you start a specification, gaps and dislocations become painfully apparent.

Pattern 15.3 Use-case led system specification

Create a specification for the requirements of a system, working from use-cases.

Intent

Create a specification of a substantial software component or product.

Deliverable:	Unambiguous specification of system requirements.
Inputs:	There may be a business model — either written as a precursor to this step, or shared with other systems in the same business. See Pattern 14.2, *Make a business model* (p.551).

Considerations

The specification of a component goes hand-in-hand with the design for the process within which it is used. You cannot specify the component without having some idea of how it is going to be used; you cannot define how it will be used, without knowing something about what it does.

Strategy

Therefore, explore and document the context within which your component will be used, as well as defining its behavior. See:

- Pattern 14.34, *Make a Context Model with Use-Cases* (p.623)
- Pattern 14.36, *Construct a system behavior spec* (p.628)

Often, the context model will yield several classes of user, each with a different view of the system: see Pattern 14.39, *Specify component views* (p.640) and Pattern 14.40, *Compose Component views* (p.642).

Vocabulary

Context model:	A document describing the business processes whereby a component is used.
Behavioral spec:	A type description that defines the behavior of a component by treating it as a single object, and its interactions with the world around it as operations on the component.
System:	A component forming a substantially self-contained product.

Consequences

There is not usually a clear order in which to perform context analysis ("use-case analysis") and system behavioral specification. There is some element of negotiation between

the two: how the users will work depends on what will be provided; and what is provided depends both on how they wish to work, and on what is possible.

Therefore, don't attempt to finish one before proceeding to the next; and don't insist on doing them in one order or the other (though most people start with use-cases). Specifically, do not generate many detailed narrative use cases without iterating through a precise modeling cycle.

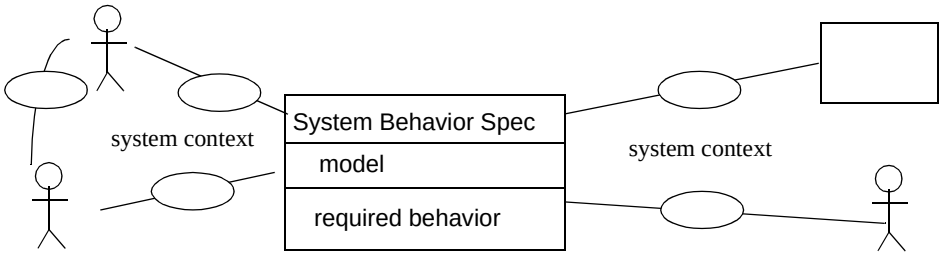


Figure 15.2 Use case model of system context

Pattern 15.4 Recursive decomposition — divide and conquer

Every specification is part of a larger implementation; and every implementation composes a solution from specified components.

Objectives

Practical design.

Context

Requirement to produce a design.

Considerations

We need to break up large problems into smaller ones.

We need to decouple the design efforts on the different partial problems, so that we don't have to keep them all in our heads at once — or indeed, in any one head.

Strategy

Construct the implementation to a specification as some form of composition of smaller components.

Specify each component. There may be many (or many potential) implementations of each component. Do not consider the internal details of the components, when devising the present implementation; they will have their own decompositions.

Do not confuse the use of a component with the definition of the component. One component may be shared between the implementations of several others. In each use, there may be particular restrictions or simplifications that apply; these are not part of the intrinsic definition of the shared part. Use packages to distinguish these.

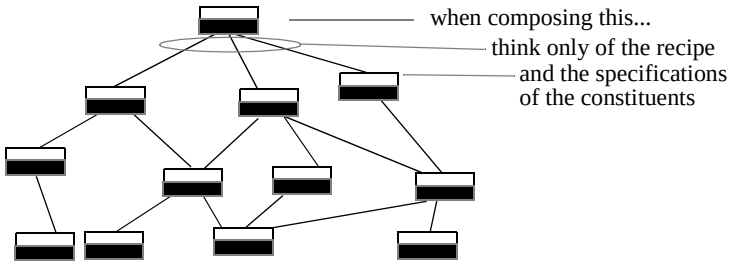


Figure 15.3 Each design should be in terms of specifications of its parts

It is not usually sufficient to say what the constituents are: the way in which they are combined is also required.

Vocabulary

Composite	A relative term: the object of which a constituent forms part.
Constituent	A relative term: an object that makes up a composite.
Recipe	The extra information that defines how the constituents are composed together to make the composite.

Known examples

- A subroutine fulfills a requirement. It is implemented as a sequence of invocations of constituent subroutines.
- An object's behavior is implemented with a composite of instances of smaller objects, linked together in a collaboration.
- A hardware component is a composite of smaller components, wired according to a defined scheme.
- The specifications of a component you are asked to design may come from a larger design. This situation contrasts with the more user-facing situation of an end-product design, where your team's responsibility includes documenting the system context use-cases (Pattern 14.32, *Use-case led system specification* (p.619)).

Pattern 15.5 Make a Context Model with Use-Cases

Capture collaborations with and around an object, component, or system, by building a context model, showing all abstract actions (use-cases) involving the system.

Objectives

To define the scope and boundary of a system and its relation and interactions with its immediate environment; to build the requirements spec for a system or subsystem to be installed; or to review the context of a legacy system.

Context

If this is a subsystem within a larger design, much of this work may already be done. Otherwise, we begin by understanding how our system works within a larger system to meet some larger objectives. For example, if we are asked to write an order payment application, we need first to understand the relevant procedures of the financial department, what role is envisaged for our software, and what functions will be performed by the users themselves, or other pieces of software.

Considerations

In general, the different parties that interact with an object each have a separate view of its concerns. A context model should cover all external roles and interactions, at least at an abstract level, even if split across multiple collaborations.

Many of the collaborations will follow stereotypical interaction patterns e.g. an interaction which consists of log in, conduct some transactions, and log out; or make a selection, and then operate on that selection. These can be abstracted as frameworks (Chapter 9, *Model Frameworks and Template Packages* (p.371)), then applied and composed to define the system context collaboration.

It can be difficult to understand all the operations on a system. It is easier to identify the main user roles, which can then help define the operations and flow of information across the system boundary.

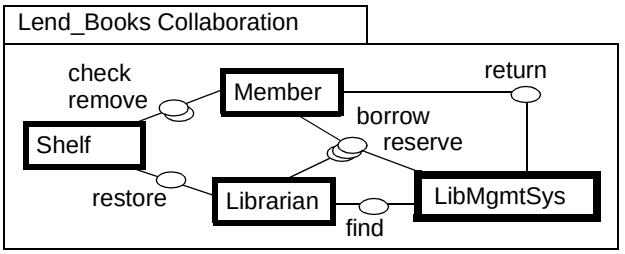
However, there is a subjective choice here: what should you model as 2 separate roles? For an interaction sequence: log in, conduct transaction, and log out, should you have three separate user roles? For order fulfilment, should the order placer be modeled as a separate role from the receiver?

How far out should you show actions? Should you just include actions that directly involve the system? How fine-grained should the actions be?

What about time-triggered actions that have no explicit external initiator?

Strategy

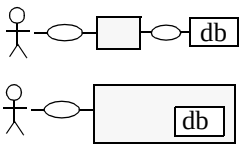
Scope the Context Model. A context model consists of one or more collaboration models (Section 4.6, "Collaborations," on page 203) focused on the object we are principally concerned with (typically a software system or component) and its interactions with others (typically people, other software, or hardware).



Summarise the interactions with the system as collaborations between the system being specified and other objects, building scenarios to validate the collaboration, and modeling actions at a consistent level of abstraction. Show all external roles as types — also called "actors" — that participate in these actions. Also, document those actions that can proceed concurrently on a single system, including constraints on concurrency. For example:

> *Concurrent actions: Any number of members can be borrowing or returning book copies, or reserving book titles at the same time; a given book copy can have only one borrow or return involving it at any time.*

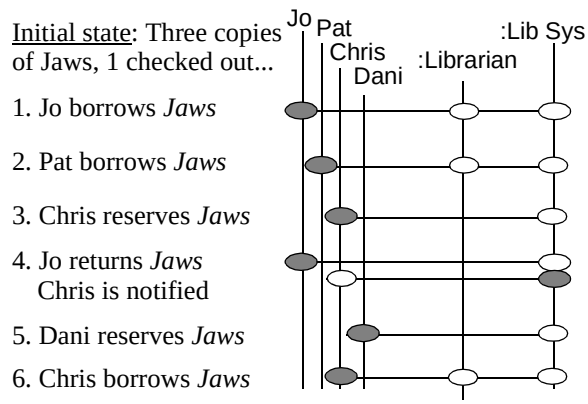
Besides all actions that directly involve the system, also show actions between the external actors if they are relevant to the collaboration. If there are relevant actions even further removed from the system, either show actors and actions one level further out, or model them as effects (Section 4.4, "Actions and Effects," on page 198) on the external actors already included.



Treat the system of interest as a single object, without internal roles or objects. Be precise about the boundary of the system you are describing in a collaboration e.g. is the database within, or outside of that boundary? If outside, you should describe interactions with the database; if inside, the database and all interactions with it are abstracted into a simpler type model and effects on that model (see Section 3.6.1 for an explanation).

Define for each object a set of responsibilities, and list them in the action section of that type. In particular, list and document the responsibilities of the system within the context of the larger organisation (or larger system); treat the system as part of the design of the larger context.

Scenarios. Construct scenarios and storyboards of the projected use of the system, starting from a known initial state. Pay particular attention to cases where the system's dealings with one object affect those with another, and cover interleavings of these in a scenario. Create interaction diagrams for the scenarios, combining scenario narrative with the diagrams; the interaction diagrams can depict directed arrows or joint action occurrences (Section 4.7); this example uses the joint action notation. A scenario should show named objects; it is usually sufficient to use generic names for them, like c1, c2, c3: Customer, and t1, t2: Title.



As a rule, each step of the scenario that involves the system should start with an externally initiated action and include all subsequent system response, including resulting system-initiated actions e.g. step 4 in the example above. This will serve as the starting point to identifying and specifying system operations.

Abstract and re-refine. Abstract some of the actions to understand goals more clearly; good high-level actions often reflect goals of the user. For example, suppose we are told that we must deal with separate actions:

made_order(agreed_price, items);

received_invoice(amount);

received_goods(items).

We could abstract this to a higher-level action buy_goods(price, items). The postcondition is that we get and have paid for goods we have ordered. Re-refining this, we can question whether the receipt of an invoice is relevant: so long as the goods have been ordered, we can pay for whatever we receive, cutting out some administrative work; this option can then be discussed with our client.

You can model the system context, and corresponding scenarios, system type model, and action specs, at any consistent level of granularity. The actions should be refined to the level where completion of the action accomplishes an atomic transaction with the system, and must be completed in its entirety by the actor to be meaningfully handled by the system.

Separate roles. Make separate context models for the system as used by different groups of collaborators. In general the separation works best by considering the abstract collaborations, rather than individual collaborators.
For example, the member and the librarian both interact with the Library system about the different stages of an abstract 'lend-books' action (which encompasses borrowing, return-

ing, and reserving). Theirs would be one view — the system plays one role in that collaboration. The stock-keeper's 'add-books' would affect the other collaboration of course — you can't lend books you haven't got — but it isn't part of the lend-books collaboration, and the system plays a distinct role. So in this example, one collaboration is separated from the rest.

Do not go overboard with separating actor roles. Specifically, if there are strong dependencies in state or attributes of the actor involved across two actions, do not needlessly split into two actor roles. For example, if the system requests authorization from an external system for a credit card, and follows with an approval request for a payment amount on that same card, it would be better to use a single external role for the authorization system. Similarly, it would not be helpful to distinguish 'reserver', 'borrower', and 'returner' as separate roles in a library. Map the user-roles in the system context to roles in the business model; perhaps eventually to job descriptions.

Time-Triggered Actions. In addition to explicit stimulus from exernal actors, a system may need to respond on its own to the passage of time. Model this also as an external input to the system, driven by a clock of arbitrarily precise frequency. Pick a suitable name for this action e.g. end_of_day, or tick. For clarity, separately define named effects to specify what happens at that time e.g. clear_outdated_reservations. As always, you can have multiple specifications for the action.

```
action System::end_of_day
    post:    clear_outdated_reservations
             -- all expired reservations have been cleared
effect System::clear_outdated_reservations
    post:    -- specification of the effect of clearing all outdated reservations
```

Many time-triggered actions are better specified as static invariants initially:

```
inv System:: -- there must never be any expired reservation in the system, or,
inv Reservation:: -- cannot ever be expired
```

Benefit

Many of us have experience of development where the designers don't get to meet the end-users: that is the analysts' prerogative. (They know how to wear smart suits, I guess.) The effect is that although they may implement the stated requirements, they may do so in a way that isn't very user-friendly — simply because they don't have a clear vision of the context in which the system works. The context model and its documentation gets around this, and provides a clearly scoped connection to the business model.

Pattern 15.6 Storyboards

Make pictures of the user interface.

Objectives

Animate the system use cases in a form that can be discussed with end users, when constructing a specification of a system that will have a graphical user interface.

Considerations

A rigorous specification expresses the common understanding that the developers have, of what they intend to provide for the users. The act of constructing it raises useful questions to put to the clients.

But it is not readily accessible to end-users who are not themselves computer people. We therefore seek techniques that animate the specification for the end users.

Users relate most directly to what they see on the screen.

Strategy

Therefore, make a slide show of what the users will see on the screen; or a prototype.

Summarise the different screens in a chart that shows all the possible transitions from one to another: this is the 'storyboard'. Annotate the transitions with the names of actions — which you should have documented with their effects on the system's state. Also annotate with the story — the reasons why people would choose different routes through the user interface.

The chart forms a state chart of the refinement of the user actions. (See Section 6.5, "Spreadsheet — Action Refinement," on page 279, and Section 14.13, "Action Reification," on page 576). Once the chart is complete, abstract the major actions from the GUI detail: the latter may change between now and the implementation, but the major actions are what you require at present.

When discussing the storyboard with users, let them get involved in the detail of the GUI at first: this will help fire their imaginations; but later, make the point that the detail can be changed later, and what we want now is the major actions.

(See section 14.2.3 (*Storyboards*), p.653, for an example.)

Pattern 15.7 Construct a system behavior spec

Treating your system as a single object, define the type of any system implementation that would meet the requirements.

Obective

To construct a type specification of (some view, or role, of) a system we are to build.

Deliverables: Type-specification of any implementation objects meeting the requirements of this system (in the chosen role), embedded in narrative documentation.

Dictionary with specific meanings of types in this model.

Snapshots are thinking tools rather than true specifications (since they only deal in particular scenarios) but may be used for illustration in the narrative.

Context

We already have a system context model (Pattern 14.34, *Make a Context Model with Use-Cases* (p.623)).

Considerations

The result will nominally be a type box that defines our system as if it were a single object (Section 6.6, “Spreadsheet — Object Refinement,” on page 288). Like all types, the

behavior presented to the external world — in the bottom section of the box — is specified in terms of the hypothetical static type model — in the middle section.

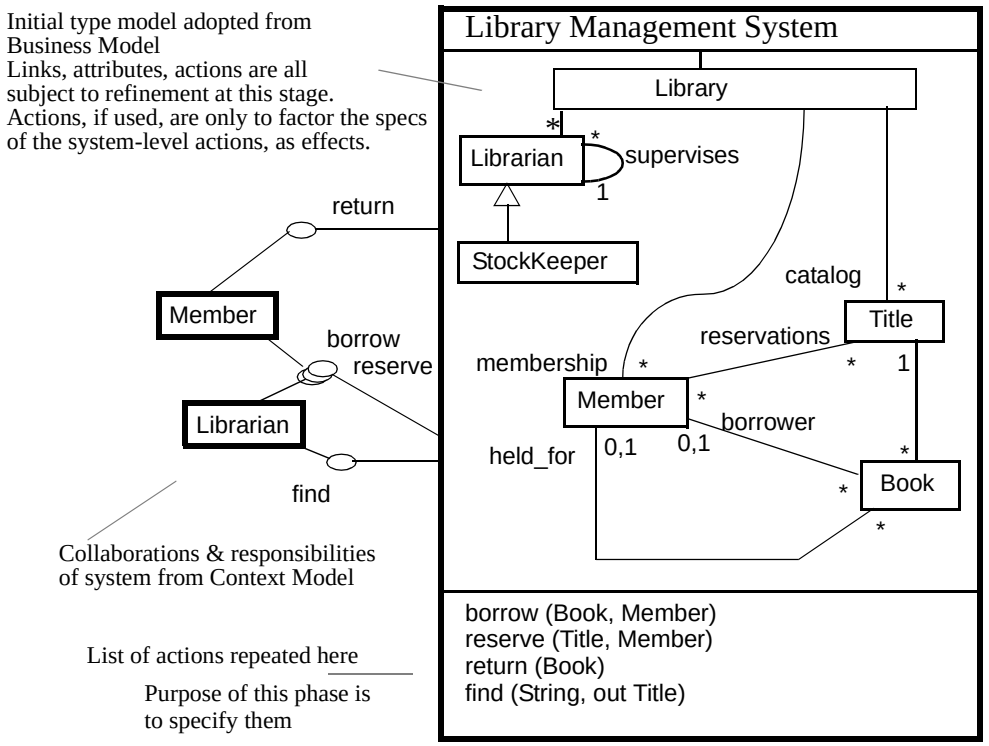


Figure 15.4 Initial type model derived from business model

Of course the whole thing won't usually fit in one box on one page. So we split into multiple drawings, or use "Refinement" (Chapter 6, *Abstraction, Refinement, and Testing* (p.245)) to separate into subject areas.

Strategy

Separate roles. As we noted in Pattern 14.34, *Make a Context Model with Use-Cases* (p.623), most objects play several roles, and, when specifying requirements, it is useful to focus on one at a time.

So in the example shown, we have listed only the operations from the lend-books collaboration, and will specify those operations. Ultimately the system's roles may be recomposed in preparation for designing it.

Summarise the set of actions for this role. This can essentially be read off the context model. Actions (and hence model) should be as abstract as reasonable at first cut —not individual keystrokes.

Each action needs to be given a parameter list. The parameters represent other information that will be available to the action when implemented. They could be the parameters of an API operation; or information that a user can be asked to enter as part of a dialogue to which this action is refined. They can usually be read off the participants and parameters in the corresponding action in the business model.
For example, reserve(Title, Member, Librarian) indicates that there is some way in which those pieces of information can be entered — we don't yet care how. (See Chapter 6, *Abstraction, Refinement, and Testing* (p.245), for how parameters relate to action refinement.)

In some cases, we may immediately decide that some information is not required — for example Librarian in the reserve operation. This doesn't stop you putting that information back in a later refinement —the refinement rules allow you to add parameters, but not to take them away; so by paring down the parameter list at the most abstract level, you are keeping your options open.

Adopt the business model as the initial type model. That is, the model part of the type-box defining the system. Note that this gives us two of everything: the external 'real' object and its model inside the system. To distinguish where this might be ambiguous, use LibraryManagementSystem::Member and ::Member. Note, however, that actions from the business model do not translate into internal actions that must be implemented in the system; they are candidate 'effects', there just to conveniently factor out parts of the system behavior specification.

The ultimate purpose of a type model is to support the action specs — expect to add to it, and eventually to drop parts as irrelevant. So for a drawing-editor, we might add the idea of a current selection. For a library management system, we might drop the business model's Librarian-Librarian 'supervises' link, since the actions expected of the system never use it.

An alternate approach is to start minimalist: the initial type model is not known i.e. empty, and you draw elements from the business model into the system type model as you uncover a need for them in describing the system behaviors.

Note that the model represents the state of the whole system from its external users' point of view — not just the business logic, and not just the software: hardware may be included too. So if a display shows the result of some calculation, and that display is part of the boundary the system, then that value can be an attribute in the model, even if the software forgets it immediately after sending it to the display. It's the system context model that sets this boundary.

It's important to be aware that at this stage, we are not intending to design the system internals. The type model is just a model: although the design may well be based on it, the designer is free to produce anything that exhibits the same list of system operations. And in particular, the actions adopted from the business model illustrate only the model-objects which may be involved in system-specified actions.

Exercise system queries. The most interesting actions are the ones that change the state of the system; but that state would be unnecessary if no information ever came out of the

system. It should be possible to represent in a snapshot, any piece of information that a user may wish to extract from the system. For example, from the library system type spec on the previous page, it would be possible to draw snapshots showing who is in possession of each book; but there is nowhere to record how long they have been borrowing it.

Build State charts of key types. This helps clarify what is required of each action and crosscheck the model for overlooked states, operations, and combinations of these. Each statechart shows the effect of actions in which the *system* collaborates on the *model* component of interest. See Pattern 14.38, *Using state charts in system type models* (p.636).

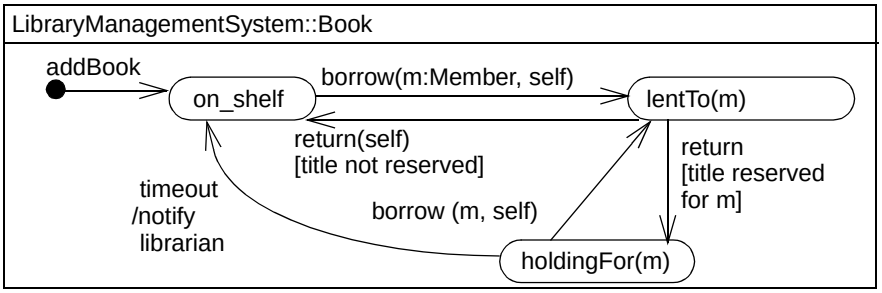


Figure 15.5 Statechart of a spec type

For each state, define an invariant that relates it to other attributes in the type model. States are boolean attributes, and can be parameterized just like attributes.

```
LibraryManagementSystem::Book::
        -- State definitions for the above diagram:
        lentTo(member) = self : (member.loaned)
            -- to be lent to a Member means to be in the 'loaned'
            -- set of books for that Member.
        holdingFor(member) = self : (member.held)
            -- wasn't in Business Model, had to add it!
        on_shelf = (~loaned = null) and (~held = null)
```

Specify each action. See Pattern 14.37, *Specifying a system action* (p.633). The result is an effects clause, and a review of the model.

```
action borrow (book, member)
    pre:    book.on_shelf
    post:   book.lentTo(member)
```

Exercise the spec with scenarios. Perform a walk-through of a senario and exercise the specification, starting from the initial state of the scenario. Check that the progression of states and system-responses is as expected from the action specs.

Review model. Delete attributes not used in any action spec; add attributes that are needed, or that simplify the specification, and relate them to other attributes by invariants. Delete types not used for any query or action parameter. Update the dictionary with defini-

tions that relate model elements to the business or domain model, its objects, and its events.

In the Library example, drop Librarian and StockKeeper from this model. They may be required in models for other roles of this system.

Benefit

The exercise of writing a type spec in an unambiguous language always raises issues that are relatively inexpensive to debate at this stage. In our experience, the more precision you aim for, the more questions you raise and the more gaps you fix.

Pattern 15.8 Specifying a system action

Specify an action using snapshots to help create the formal spec.

Objectives

Each system action, no matter how abstract or concrete, can be characterized by the relation between its inputs, outputs, and states before and after the action. We want to write action specs with pre and postconditions expressed informally and formally.

Context

We are specifying the type of a system playing a role in a collaboration; that is, the view that its collaborators have of it. We already have a initial type model. We may improve it in the light of what we need to say about this action, and iterate between specifying actions and updating the type model.

Strategy

Write the spec informally. State in natural language what is required of it. Think in terms of outputs, effects on external objects, and the initial and final internal state. Think also of preconditions — interpreted precisely as the conditions under which this particular postcondition make sense.

action return (book)

pre: — book is lent to someone

post: -- book is either back ready to be loaned again,

-- or, if someone has reserved its Title, it is held for them

Draw snapshots. Draw a snapshot conforming to your type model that shows the system in a state conforming to the action's precondition. Then modify it (preferably in a different color) to show the state after the action. Intermediate states, and the order in which things happen, are of no concern. Draw a snapshot regardless of how complex the system is, introducing convenience or parameterized attributes to simplify matters.

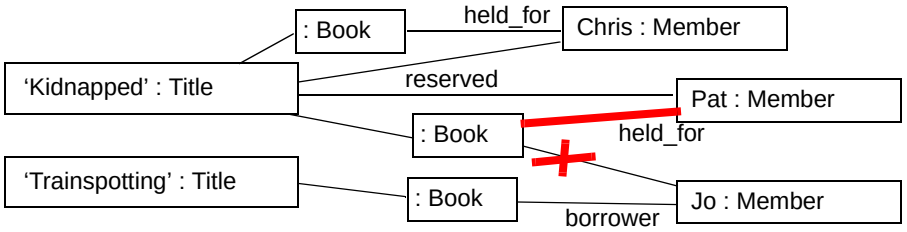


Figure 15.6 Snapshots of an action

Delete links rather than whole objects. If necessary, draw an entire sequence of snapshots for a scenario, starting from the initial state.

Write formal pre/postconditions. Using the snapshots as a guide, write pre/postconditions in the more precise terms of the links or states. Expect to discover gaps and inconsistencies. (See Section 3.4.1, "From Attributes to Operation Specification," on page 129, for detailed steps). Be aware of how your spec will be combined with other specs for the same action (Section 8.1, "Sticking pieces together," on page 350).

In this example, drawing snapshots helps us notice that an inaccuracy in the initial attempt at the informal postcondition.

```
action LibraryManagementSystem:: return (book: Book)
pre:        -- book is lent to someone
        book.borrower <> null
post:       -- book is either back ready to be loaned again,
            -- or, if someone has reserved its Title but doesn't already have a copy held
            -- it is held for them
        -- the book is definitely not loaned out any more
        book.borrower=null and
        -- if this title had more reservations than held books
        if book.title@pre.(reserved > book.title.books[held_for<>null])
            -- it goes on hold; the new holder used to hold no books of this title
            then        book.held_for <> null
                and     book.(not title.books@pre.held_for includes held_for)
        -- otherwise if goes back on the shelf
        else (book.held_for = null)
```

Look for cases not covered by the snapshots. The snapshots only represent a typical example, whereas the spec must cover every possibility within the scope declared by your precondition.

In the Library example, it would be easy to overlook the case of returning a reserved book. But drawing snapshots for the documented scenarios (Pattern 14.34, *Make a Context Model with Use-Cases* (p.623)) usually exposes these cases, as do state-charts (Pattern 14.38, *Using state charts in system type models* (p.636)).

Factor the spec into model types. If some parts of the pre or postcondition seem to be particularly associated with types in the model, write them there as effects that you use in the system operation specs (Section 3.8.3, "Effects factor common postconditions," on page 150). This helps avoid replication in the spec, and allows it to be polymorphic. Although the type model is not necessarily a blueprint for its design, this distribution of concerns obviously looks forward to the distribution of responsibilities in design.

```
effect LibraryManagementSystem::Book:: return ()
    pre: borrower<>null post: etc...
```

```
action LibraryManagementSystem:: return(book)
    post: book.return()
```

Don't try to state all the effects at once. In general, we don't specify every aspect of an action in one go, or in one place. We will be specifying its effect from other collaborators' viewpoints when we come to them; and we will probably also keep exceptional circumstances separate, just dealing with the main cases at first. An advantage of specification with pre/postconditions is that it is easy to combine different views later; a decent tool will readily present the combination of specifications when needed, yet the documentation can focus on different issues separately.

Benefit

The process of formalising the action spec always raises questions about ambiguities and omissions in requirements. It is far better to deal with them at this stage than at coding time. Formalisation also tends to expose shortcomings in the type model, and leads to a clear and concise vocabulary shared across team members, which will ultimately be the basis of the design.

Pattern 15.9 Using state charts in system type models

Building statecharts to find actions and effects and overlooked cases.

Objectives

Cross check the models and behavior specs. Tighten up the type model to capture different modes in which different combinations of attributes make sense. Take a different perspective on the behavior specs i.e. from the point of view of each specification type, what are all the actions that have an effect on it. Make statecharts of principal types in a model.

This is a supplementary approach to finding the actions and their effects when creating a system specification. Except for special cases, statecharts are not good at representing the whole story, since they form a single object's view of all the actions; designers ultimately work better from postconditions. But where there are clear changes in state, statecharts work well and provide a valuable cross-check for completeness.

Considerations

The states are as you choose them. By a 'state' in a statechart, we mean the truth or falsity of some predicate i.e. just like any boolean attribute, including parameterized attributes as well. A library book is or isn't out on loan; it is or isn't withdrawn from circulation; it is or isn't on the shelf. Some choices of states are obvious, because they make a significant difference to the postconditions and preconditions of the applicable operations. The point of a statechart is to illustrate these broad relationships between state and actions. But depending on your point of view, you can always choose a different set of states to illustrate — whether the book is or isn't in poor repair, whether it is over 10 years old, whether it is currently passing through the checkout desk, whether it is overdue for return.

Other state differences can be too subtle to record effectively on a statechart — for example, there is a difference between a book that has been loaned five times and one that has been out on six occasions; but unless the library's loan policy changes qualitatively at that number, it would be pointless, though valid, to draw separate states for 'loaned more (or less) than five times', and still more so to draw separate states for each separate number of times out.

Statecharts may (but need not) focus on a single type. In this pattern, we use statecharts to represent the state of the members of a particular type within a system type model. A statechart can potentially be about universal truths; but we usually quantify over one type e.g. any Book, so that the states depicted are truths about any one of its members. We also use state charts to describe action sequence refinements in Chapter 6, *Abstraction, Refinement, and Testing* (p.245).

When devising a system type model (Pattern 14.36, *Construct a system behavior spec* (p.628)) we usually play an extra trick. As we've said, the types within such a model are at

that stage just there for the purposes of specifying, and although the design would typically be based on them, there is no mandate to follow them provided the externally visible behavior is as modelled (although we usually want the design to closely follow the specification model, it may sometimes differ due to performance optimizations, or design reuse goals). The trick is to draw a statechart for one of these specification types — for example, Book in the Library system model — but in which the actions are those of the system being specified — for example, reserveTitle, returnBook, deleteMember. A statechart can be drawn for each of several types in the system model.

The thing to keep in mind is that when a system action occurs, it may affect several objects simultaneously. E.g. deleteMember may have the effect of marking 'lost' any and every book held by that member. It is as if the operations are broadcast simultaneously to all the objects inside the system, some of which take notice. The job of design can be seen as one of enhancing the performance of this mechanism, and of deciding how the information modelled as these objects will really be represented.

Strategy

Identify a type with interesting states and transitions. Some don't have any. Any boolean attributes are candidates for states. Similarly, if you have 'optional' attributes or associations, there are frequently states in which those attributes must be defined. Some continuous attributes may have values that signify a qualitative change in behavior e.g. speed > 75mph may qualify as a 'speeding' state. Look also for radical differences in behavior — eg, can be lent or not; and for equivalent manual system stages or phases; or temporary tags or marks; or locations of an item or piece of paperwork. E.g. Book on shelves or held at desk or with member. E.g. hotel reservation in 'future' folder or 'today' pile.

Draw transitions. At every state, decide the applicability of each operation, and link to its result. You may discover more states at this stage, but it is okay to leave some things underspecified for later refinement.

Label the transitions with the operations that cause them. Also mark preconditions — the transition is only guaranteed to occur if the precondition is true; may also write postconditions not indicated by the change of state — e.g. increment of counter.

Use only external operations — stages in the internal workings of the system are not states in this abstract picture. It is also sometimes useful to show state transitions for 'effects' that you introduced as a way to factor the action specifications. You may consider timeouts or the arrival of particular times as external operations.

Use snapshots to illustrate what is happening to each object on each transition. This is especially useful to clarify situations where different things are happening simultaneously to different objects.

Formalize. States and transition-labels are usually defined informally at first. Some attempt at formalisation raises more useful questions.

- Add each state to the Dictionary. Define its meaning in the users' world-view, taking care to exclude and include exactly what is intended. Does 'on shelf' include times when a browser is inspecting the book within the library?
- Define each state as a boolean function of the other attributes. It should be distinct from the other states in the diagram (except those explicitly parallel or nested). If it cannot be made distinct from another state, there is information missing from the model.
- Such state definitions are often combinations of a few boolean terms — some optional link present, some attribute > 0, and so on. Make a table of the combinations, showing how each state has a different combination, and identifying combinations that do not correspond to any state. The complete set of allowed states is an invariant that should be documented for the model.
- Define pre and postcondition labels of transitions in terms of links and attributes of this object in system model. Each statechart applies to each object of the relevant type, can refer to 'self' and to action parameters.
- As another cross-check, write a state-transition matrix. List all states on one axis, and all actions on the other. Check that each combination has been considered, and mark it as one of: specified, unspecified, or impossible.
- Be aware that statecharts have more than one interpretation. Does the absence of a transition between two states mean that there will never be any way in any member of this type (including all its subtypes) of getting directly from one state to the other; or does it just mean there are none we know about in this type, but subtypes might have some? If the former, mark the chart, or a state therein, as "closed" using a UML stereotype <<closed>>.

Integrate statechart information into specifications for each system action. Each transition implies something about what an operation does to the relevant type of object. Some operations will affect several types of object: AND the effects of these together.

Benefits

- The model is usually improved by defining states; some specifications become simplified by directly referring to states (including parameterized states), and letting the state invariants implicitly deal with the details. e.g.
- Actions cover more cases than initially envisaged. The state chart visually highlight less usual states in which actions can also take place, particularly when described as a state-event matrix. This fleshes out holes in the action specs.
- Model exercised in some detail, by adopting an alternate view of the behavior of the component, from the perspective of each of its specification types.

Pattern 15.10 Specify component views

Define partial views of a component, specifying behavior as seen from each interface.

Objectives

Create a specification of a system that has multiple interfaces, or many different users, without being forced into a single description.

Deliverable: Type model of a partial view of a component.

Context

Pattern 14.32, *Use-case led system specification* (p.619) or Pattern 14.33, *Recursive decomposition — divide and conquer* (p.621).

Considerations

1. When specifying a very large system (let's say, a telecoms network and its surrounding management software) , it is usual to find that different users have different terms, and different detailed models. What Engineering thinks of as a Phone has attributes like line_impedance; what Accounts thinks of as a Phone has attributes such as call_charges. What the StockKeeper thinks of as a Book includes its purchase date etc; but the Desk staff are interested in how long it has been on loan.

We need to keep these models separate, so that we can go back and talk sensibly to the different users in their own terms; though we also need to merge them at some point, so as to be able to implement an integrated system.

2. When specifying any component (with user or software interfaces), there are usually several interfaces. The different interfaces are aware of only part of the component's state and behavior. There may be several different classes of component that possess an interface conforming to a particular model.

We need to keep the models separate, so that we can pick which interfaces a new implementation conforms to.

In a hardware analogy, many different classes of device generate a video signal; and many different implementations accept it. Each of those devices (such as a monitor or projector) has other interfaces as well.

Strategy

Following Pattern 14.34, *Make a Context Model with Use-Cases* (p.623), separate the different interfaces, one for each external role (or one for each group of roles that always occur together). Alternatively, you may identify the component and its interfaces during Pattern 14.33, *Recursive decomposition — divide and conquer* (p.621).

Write a separate model for each interface. For each interface, consider only the concepts that the user of that interface requires to understand. Write the model in terms meaningful to that class of user.

Notice that one actor's view of the component often includes some understanding of the component's effect on third parties, if the actor has some means of access to them outside the system. For example, suppose I press some buttons on my phone, and then say "come to dinner at 8pm" to it: I do not expect the phone to appear at the table in the evening; nor do I imagine that my only access to my invitees is by shovelling the food down the wire. Instead, I understand that what I do to the phone has an effect on people with whom I have other means of interaction.

Pattern 15.11 Compose Component views

Join several views of the requirements of a component.

Objectives

Make a single combined model of a component, prior to implementing it.

Context

1. We have several separately constructed views — Pattern 14.39, *Specify component views* (p.640).

2. We are developing a component that has to conform to some existing interface specification — either from Pattern 14.33, *Recursive decomposition — divide and conquer* (p.621), or from an externally-defined standard.

Considerations

It is not unusual to have requirements provided from several sources, all of which are defined according to a separate set of terms and rules. The differing views can come from:

- different interfaces, to different external actors;
- different industry, legal and business constraints imposed by different organisations and expressed in different terms.

For example, a financial system typically has not only the bare functional requirements (of settling trades or whatever); but also must conform to the company's business rules; and the legal constraints; and the interface conventions of the external accountancy systems. Each of these sets of rules is defined with its own model, and each may impose its own rules on the system.

Strategy

- Construct a common static model encompassing the concepts in each of the constituent models. This will be the static model for the component.
  - The simplest way of doing this is to import the models into a single package, and then define invariants defining how the attributes from one view are related to those from another, just as you would for redundant attributes. However, you may then need a separate refinement stage to get to the code.
- The component model is a refinement of each of the views. Construct abstraction functions from the component model to the attributes of the views.
  (Section 6.4.2, "Documenting model conformance with a Retrieval," on page 271).
  If done fully, the abstraction functions provide a way in which the assertions of the

views can be checked against the implementation.
If you use the slam-together method, the abstraction functions are trivial.??

- Join the action specifications and invariants from different views, as decribed in *Chapter 9, Model Frameworks and Template Packages* (p.371). The covariant 'join' semantics permits different views to impose their own restrictions on the whole.

  An example may be found in Section 10.8, "Heterogenous components," on page 456.

Pattern 15.12 Avoid Miracles ... Refine the spec

Systematically go into more detail about actions and models, always mapping back to the abstract versions as a sanity check.

Objectives

Bridge the gap between abstract model and implementation

Context

Most people have seen the cartoon — a big complex diagram of some outfit's Methodology; it's got all sorts of purposeful-sounding tasks, numerous feedback loops and checkpoints, extensive forms and documentation templates to fill out, matrices to put check marks against and metrics to tell you how well you are doing. Inputs to analysis at one end, code out at the other. But in the middle of all this busy activity, all arrows converge sooner or later on the box that says "And Then A Miracle Happens"!

We don't have so much of that. The miracle is your creativity as a designer — there's no algorithm for that; but it is spread evenly over the whole process, without a phase transition in the middle. Partly, this is to do with the object approach: the software's structure mirrors reality. But we also can document refinements, which show the relation between a detailed description (often a design) and a more abstract one (often a specification), map between them, and justify choices made and key options rejected. We can also use 'frameworks' (Chapter 9, *Model Frameworks and Template Packages* (p.371)) to memorialize common forms of refinement as recurring patterns, again helping eliminate the magical miracle.

Strategy

Golden rule. Wherever possible, let your design reflect the model of the problem domain (Pattern 6.1, *The OO Golden Rule (Seamlessness or Continuity)* (p.307)). This makes it much easier to bridge the gap from business models to implementations.

Design review. Focus on refinement during a design review. The designer should explain:

- how the design is a valid refinement of the specification, including documenting conformance (Section 6.4.2, "Documenting model conformance with a Retrieval," on page 271)
- justifications for particular choices made, whether based upon performance, understandability, previous systems, etc.

The resulting description provides the context of the problem (the specification), the context and considerations involved (the justification), and the solution (resulting design).

Rearrange the spec. When the gap between the specification and design gets larger, try to re-arrange the specification so it's structure is somewhat closer to the implementation. It may be easier to rearrange the problem description itself and then map to the solution.

Frameworks. When you find a recurring pattern of transformation between spec and design, abstract that pattern into a framework.

Pattern 15.13 Interpreting Models for "Clients"

Use concrete examples with familiar notations, while still enforcing strict rules about terminology and definitions, to review and interpret formal models for customers who may not want to see models. Strictly separate external and internal views.

Intent

Building models helps clarify requirements and designs, but the models themselves may not be appropriate to present to the customers. You need a way to validate the requirements, or designs, as captured in your models.

Strategy

Concrete Review. Specification models capture all possible behaviors (action specs and statecharts) and all possible states (type model). It is always simpler to understand concrete examples, rather than generalized specs. To do this, review:

- scenarios: narrative and interaction diagrams that trace through a collaboration with the system, with named objects in the initial state and consistently named objects for any information exchanged with the system. If you show the 'graph' form of interaction diagrams, use a consistent layout and position for all objects involved; this helps comprehension across multiple interactions.
- snapshots: drawings of the state of the business/domain, or of the state of the system. Illustrate how the snapshots change through a scenario; this is the surest way to communicate their meaning to customers. The snapshots interpret the type model and static invariants; snapshot pairs interpret the action specs and dynamic invariants.

Positive and Negative. When reviewing concrete example of behaviors and states, include both an example that is valid according to the spec, and one that is invalid. Include the constraints imposed by static and dynamic invariants. Generalize from the invalid ones, and discuss how the informal specification prohibits them. Similarly, generalize the required valid cases, and discuss how the formal specification enforces them.

Informal Specs. The formal behavior specifications start with informal narrative, and end with much clearer narrative description (see Section 3.4.1, "From Attributes to Operation Specification," on page 129). Always review this narrative specification, and the definitions in the Dictionary. Explain to the customer that the terms reflect the system's view of the domain, not necessarily every aspect of the domain itself.

Familiar Notations. When showing objects, either in snapshots or scenarios, feel free to substitute graphical icons that are more suggestive of the objects involved. Use problem-specific drawings to show attributes and relatioships between objects (e.g. a batch of sili-

con wafers positioned inside a particular crucible in a machine), to explain the meaning of corresponding snapshots.

Draconian Dictionary. Despite using largely informal and narrative descriptions to review the models, you should be very particular about terminology. Treat the dictionary as the definitive glossary; terms outside it are suspect, and should be avoided as far as possible. Re-interpret the customer's descriptions in terms of the dictionary and have the customer validate your description; update the dictionary with terms to make communication easier (Section 3.8.2, ""Convenience" attributes simplify specs," on page 149). If discussions with the customer regularly evolve into long and unfocused discussions of ill-defined terms, or if the same issue is being repeatedly re-visited, take on a more proactive role: build models, and use them to suggest defintions and proposed interpretations of requirements.

User-Interface Review. Customers are happy to discuss what they shoud see on the user-interface. Start user-interface sketches early. However, use these appropriately:

- Do not get caught up in look-and-feel issues too early; focus on information exchanged, expected behavior seen on the UI, and the flow of actions and corresponding UI elements for the user.
- Any drawing of a user-interface is a particular visual presentation of a snapshot. e.g. attributes may translate into colors; an association between two objects may show up as master-slave lists, or relative visual positions of the graphical presentations of each object. Every element on the user-interface should map to some element on the underlying model, though the names may not be exactly the same: e.g. presentation labels map to attributes or parameters, window names map to types, buttons and menu items map to actions. Keep this mapping consistent, and make it a part of your dictionary (at least in the package that includes both the models and the user-interface specification or prototypes). You are responsible to ensure the user-interface bits are kept consistent with the underlying models.
- An appropriate review would include scenarios, corresponding user-interface prototypes and flows, and the dictionary corresponding to the type model.

External vs. Internal Review. Be very conscious of who you are presenting to, and why. It is a common mistake to review internal mechanisms and design decisions with a 'customer' who should never even know whether you implement things that way. This is probably the most common error made when reviewing models among developers[1]. Distinguish carefully between a review of a component from an external perspective (end-users; or clients who will call your code); and a review of the internal design (for the team that will implement that design of collaborating pieces).

1. As opposed to end users

The former is always centered around type models and operation specifications, and should very rarely discuss classes and inheritance. The latter is focused on collaborations between objects, and includes the corresponding type models of each one; it can include many more details of code structure and re-use e.g. class inheritance.

Prototyping and vertical slices. In short-cycle development (Section 13.3, "Short-Cycle Development," on page 543), the most effective way to get the users' feedback is to produce something that works minimally. The only drawback is the users' tendency to imagine that you have already done the whole thing.

15.2 *Video Case Study — System Specification*

15.2.1 System context

Here we introduce the idea of a software system we wish to build, and explore the roles it plays from the viewpoints of various users and areas of activity. Effectively, this means zooming in on the interactions of the Business Model, to understand exactly what goes on when, for example, a customer hires a video. It is often useful to build and compare descriptions of what happens now (with a manual or old system) and what will happen when our new system is installed; though only the latter is illustrated here.

Some of the techniques described in this section are things you might do to try to think ahead to how people (or other pieces of software) will work with your system; it is rare to start with clearly defined system behavior requirements; even in cases, such as telecommunications standards, where voluminous 'specifications' are available, a clear and understandable statements of requirements is usually not a part of these specs!

Building a prototype is an important option for many projects, and can be used as a vehicle for finding and asking the same questions. It can be built in parallel with the scenarios of this section and the system specification illustrated in the next section, and will be based on a set of classes that reflect the business model's types directly. See Section 13.3, "Typical Project Evolution," on page 530, and Pattern 13.3, *Short-Cycle Development* (p.543), for a discussion on prototyping.

15.2.2 Scenarios and User-Interface Sketches

A scenario is a story about how the system will be used. It has several benefits.

- It works as a medium for communicating with end-users, who would not be expected to understand the more formal notation of the system specification. Most scenarios should be defined *by* the users, with the goal of capturing their ideal task flow and sequence.

- It helps understand the likely frequencies and sequences of use of different operations, influencing the user interface design and internal architecture.

Jo hires Trainspotting

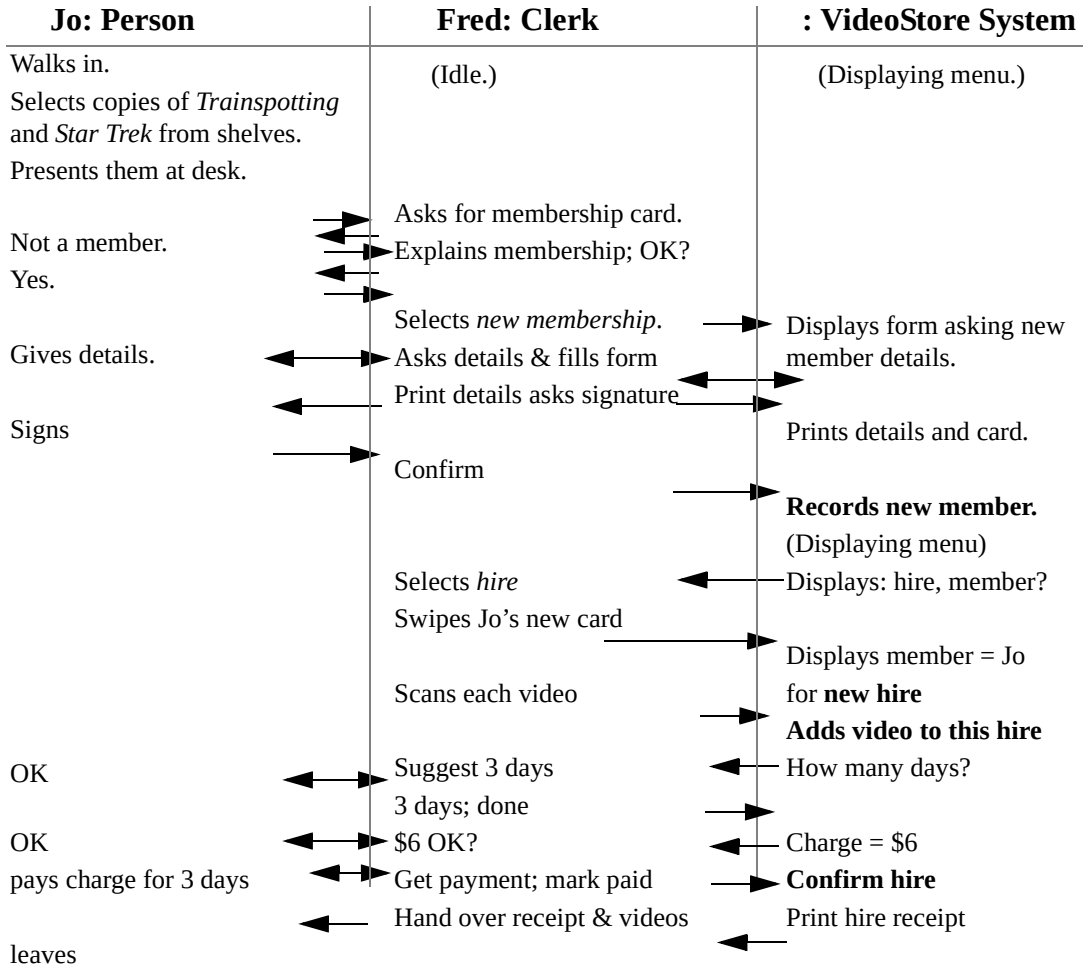


Figure 15.7 A scenario for a video rental

In this scenario, we have highlighted the commits of major transactions on the video store system: these are points at which the interaction is not just with the user interface, and not just reading the business core; but represent important changes to the state of business objects in the software. These will be the actions we focus on first when we specify the system's behavior.

The scenario makes it clear how the responsibilities are divided between the various parties. For example, the System expects the member card before anything else, and the

Clerk therefore finds out that Jo is not a member at an early stage. Contrast with this scenario, in which the Clerk breaks off the hire operation in order to record the new member:

Jo hires Trainspotting (2)

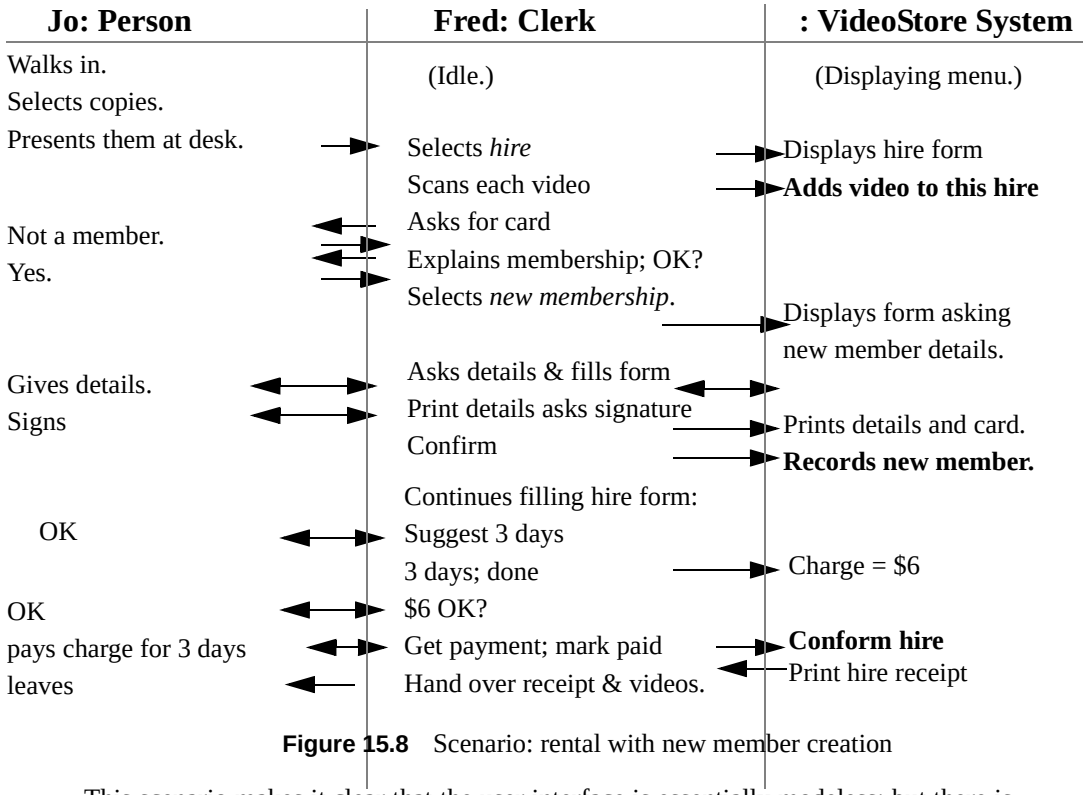


Figure 15.8 Scenario: rental with new member creation

This scenario makes it clear that the user interface is essentially modeless; but there is no reason why the system should not be able to accommodate both scenarios. Each is just a single example of the system's use: this semi-formal description does not capture all of the variations, so we need to do quite a few to get a clear idea of the main patterns. In the end, the system specifications of the next section are the definitive specification.

We like this style of showing scenarios as they distinctly separate what each participating object does, without using any technical notation. Popular modeling tools do not support writing narratives in this manner, but instead support interaction diagrams (Section 4.7.2, "Interaction Diagrams," on page 206). These can sometimes be annotated with narrative for each interaction arrow.

The next scenario describes a member making a reservation for a title.

Making a Reservation

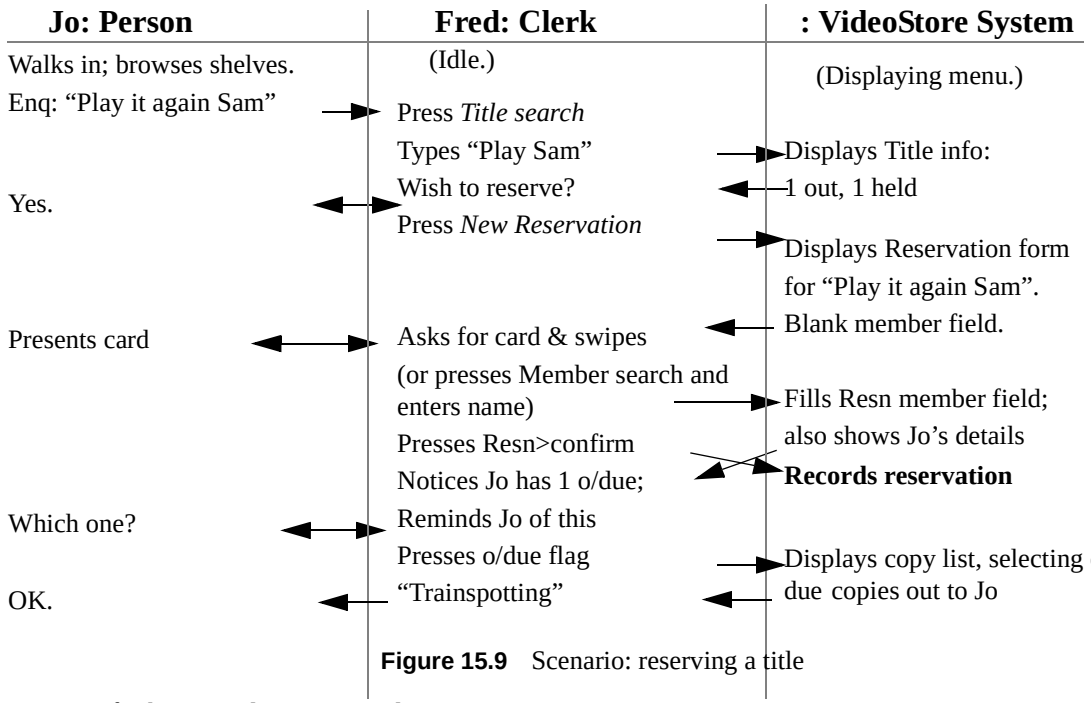


Figure 15.9 Scenario: reserving a title

A scenario for hiring and returning videos.

Returning and taking out copies

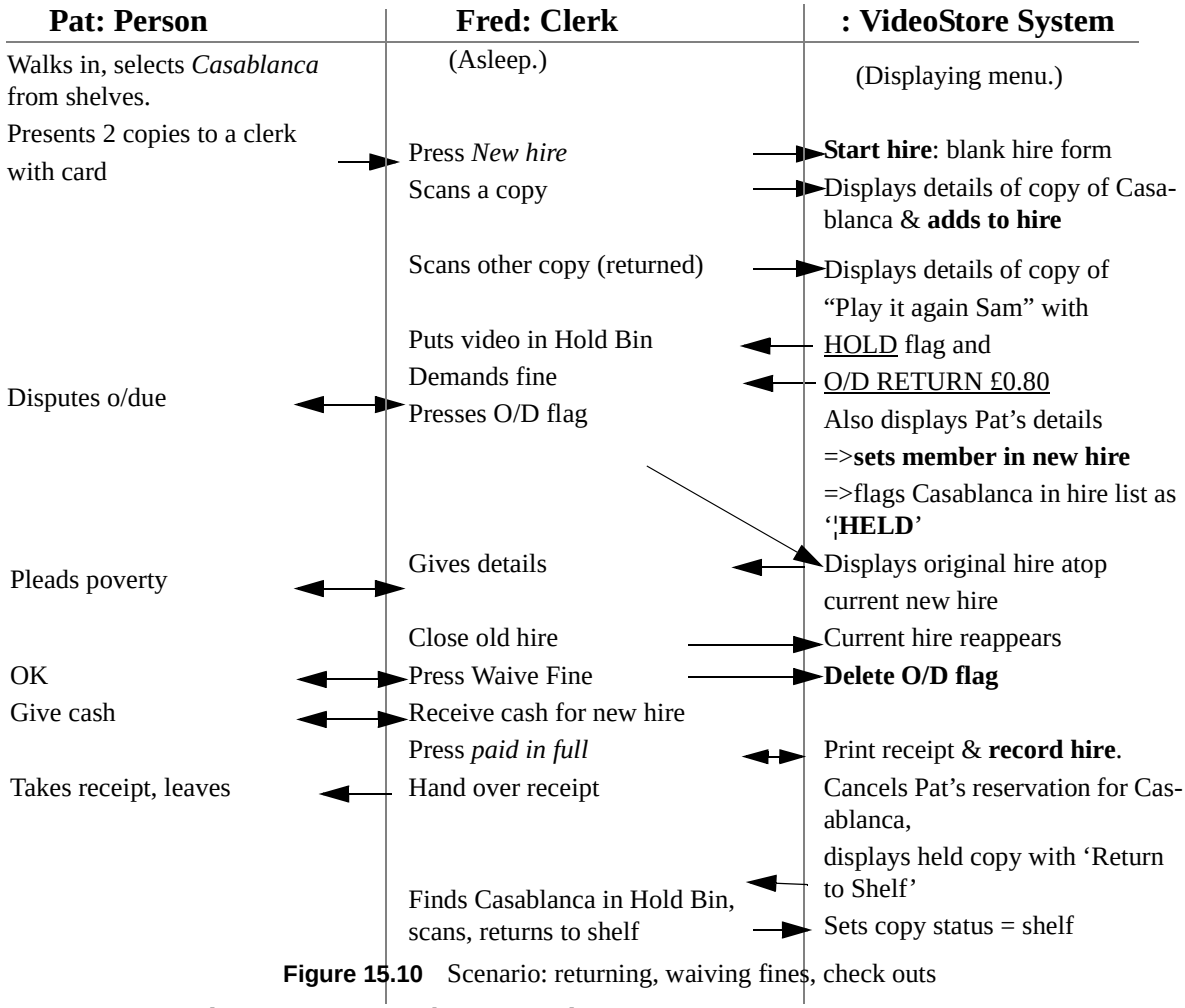


Figure 15.10 Scenario: returning, waiving fines, check outs

Reminding customers to pick up reserved copies:

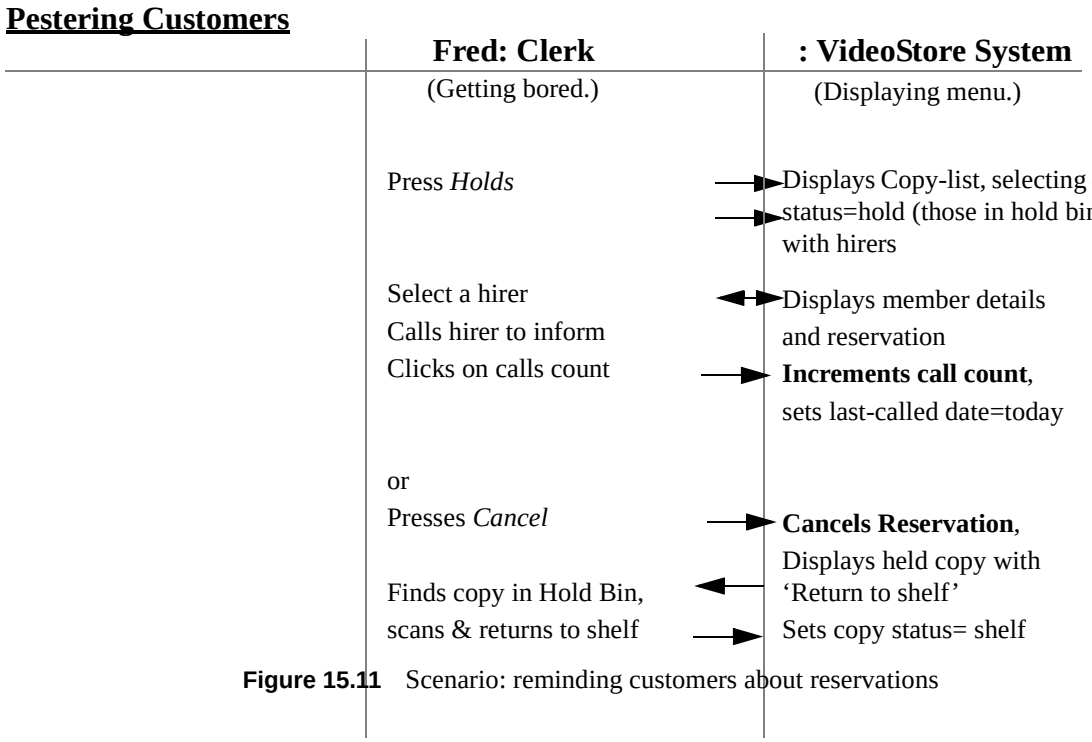


Figure 15.11 Scenario: reminding customers about reservations

15.2.3 Storyboards

Individual user interface windows and the flow between them can be captured as a storyboard. Each window is often associated with a particular task, at some level of abstraction; a flow sequence will refine a more abstract use case. Once again, users should be very actively involved in the actual design of the interface, even if that means teaching them a bit about what is achievable with the target system.

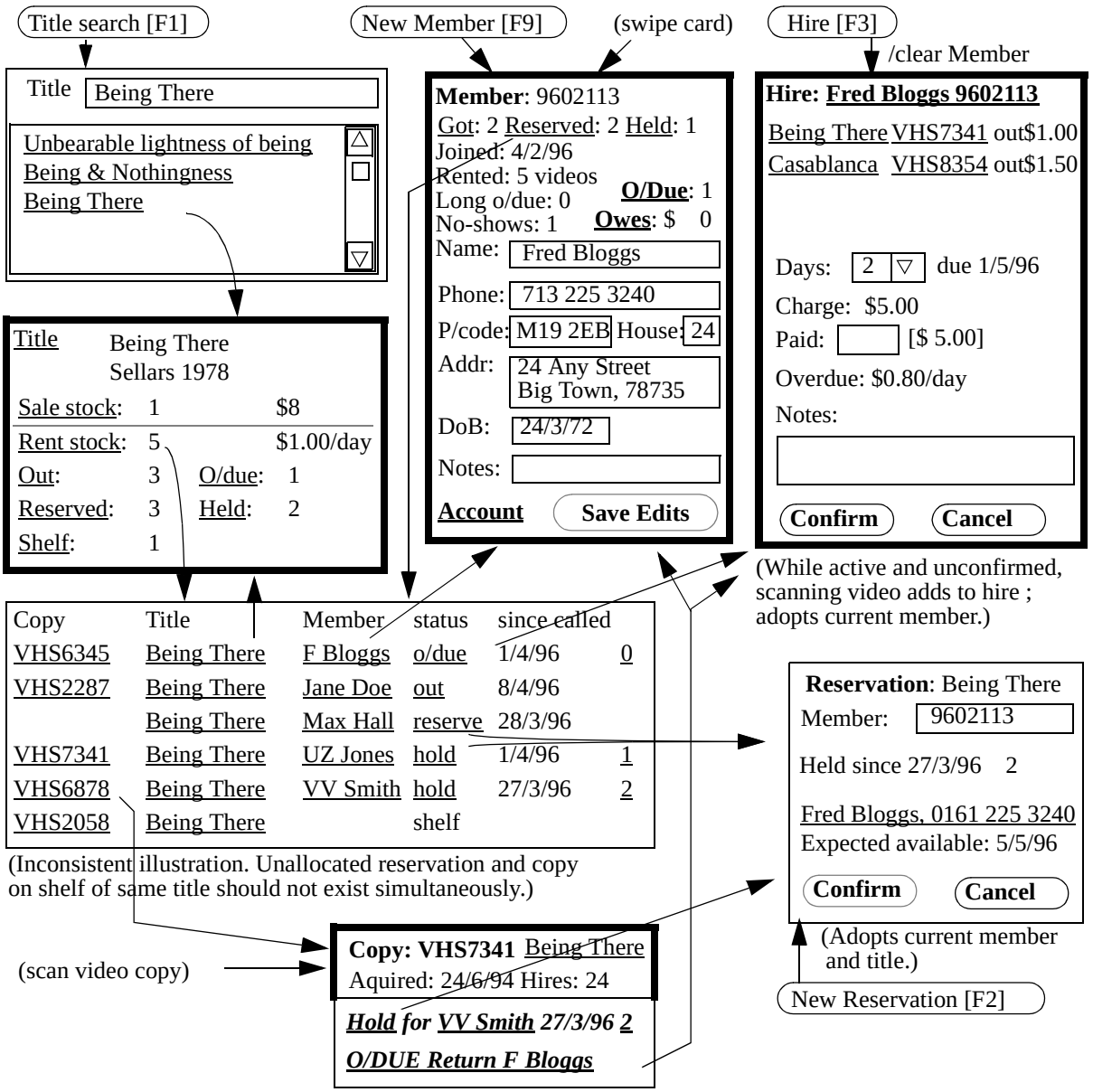


Figure 15.12 Storyboard: user-interface sketches and flow of windows

Of each type of window, at most one is visible at a time. Those strongly outlined are always visible: the current title, copy, member, and hire. Italics show information only displayed in certtain states. Underlined items are hypertext links —navigation as shown.

Only one subject area is dealt with here: the scenarios of the hires and reservations. Moreover, only those aspects immediately visible at the user interface are shown. If the system is expected to compile statistics about the monthly hirings of each title, that is not apparent here, and the user interface for it is not illustrated.

It is usually necessary to draw scenarios and storyboards dealing with several subject areas.

Animations and Prototypes. A slide show illustrating the scenarios is valuable for discussing requirements with end-users. Better still is a prototype which can be taken through many situations. A prototype should be viewed as an analysis tool, and is usually designed very simply from the principal types in the specification model (next section) together with a user-interface generator. In some cases, the core design of the prototype can be adapted to form the core of the system. However, it should be a clear part of the development plan, exactly which parts of the prototype are to be carried forward, and which are to be thrown away. Proper development practice should be applied to those parts to be carried into the design.

Predefined context design. Throughout this study, we conjecture that we're concerned with building a computer 'System' that will work within some human context. Of course, many of the contracts that software designers have with their clients or employers are about building components within a larger piece of software; or a system embedded within a complex design of many pieces of hardware.

In such cases, most of the work illustrated in this section should already have been done; and in considerably more formality than shown here. In fact the relationship betwen our system and its context is then exactly that of a Major Component as illustrated below in Section 14.3, "System context diagram," on page 656.

15.3 *System context diagram*

From the scenarios, we arrive at a picture of the system's situation within its context. We can summarise the major use cases that we identified as being core state changes.

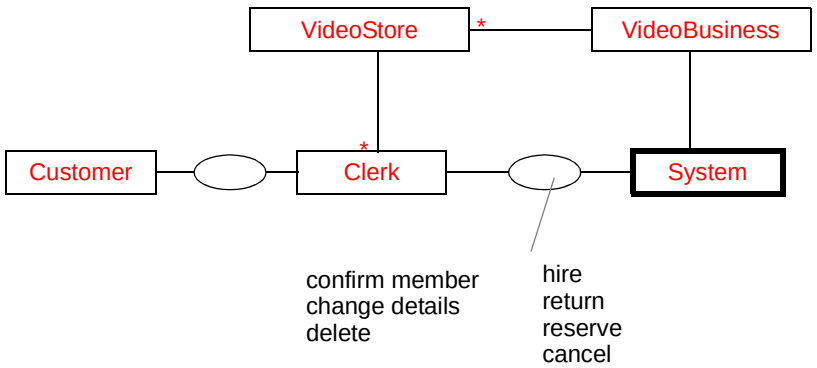


Figure 15.13 System context diagram

This makes it clear that we regard our System as spanning the entire business: there is one per VideoBusiness. It will probably be designed as a distributed system of independent subsystems in each store; but that apsect will be captured separately.

More generally, any single system can be seen as an object that interacts with its environment, which we can characterise with a type-specification. The operations it performs can be specified in terms of a model of its state. This system model is based on the business mdoel — it represents what the system knows about its surroundings.

15.3.1 Business vs. system context models

Software systems are a part of the business and its operations, and should be a part of business models — at least at some level of detail. So what is the link between business models, the system context for a particular software component, and the subsequence software specifications and designs?

Suppose we want to improve some part of the business, as shown in Figure 14.1. Suppose the video business today conducts its operations by some combination of manual and automated processes. Memberships are recorded by issuing cards with unique numbers on them; video rentals and returns are tracked in an Excel spreadsheet; and reservations are recorded by placing a slip with the member number into the cassette cover for the title. An *as-is* model of the business ('A') represents one refinement of the essential business requirements ('B). A corresponsing *to-be* model, incorporating the new target system ('S'), and revised business processes, will be an alternate refinement of the essential business model; this model is based on certain expected behaviors of the system (i.e. its specification) being used in particular ways by the business actors. Section 10.8.2.4, "Retrievals," on page 465, illustrates with an example how a model comprised of multiple

software components and manual business processes can be shown to map to the essential business model.

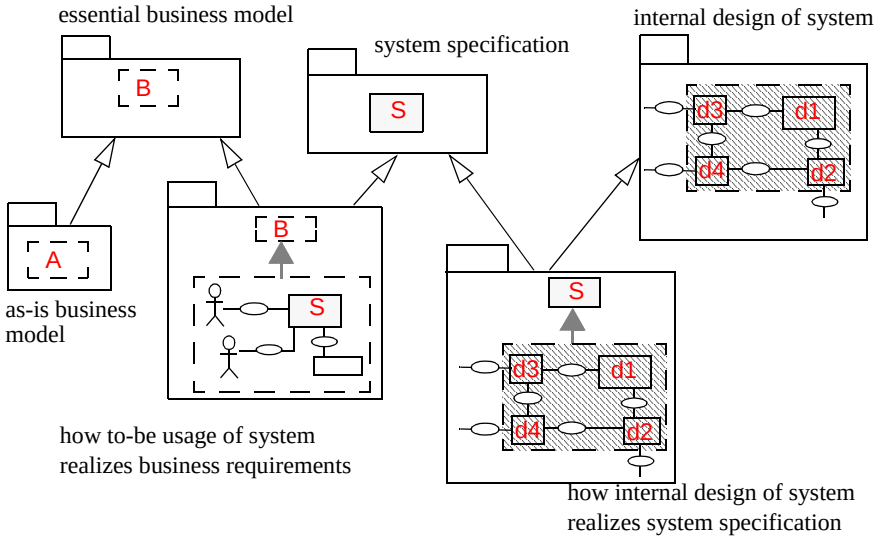


Figure 15.14 Relationship between business, system, and design models

Similarly, an internal design of the system is one refinement of the system specification. It, in turn, is based on the individual specifications of each of the internal components inthe design, configured so they interact in specific ways.

15.3.2 UI Scenarios vs. System Operations

In identifying the system context and use cases, we have actually considered some details of user-interfaces and fine-grained interactions; yet the system context seems to step back to a more abstract level of use case. This process can be rationalized as follows:

- It can be helpful to make scenarios detailed enough to animate the imagination about how the system might be used in reality, and to engage its users more actively in designing[2] this interface.
- However, realise that the GUI may change quite dramatically, from a traditional window/dialog-box approach, to direct manipulation, to a voice-driven command interface. We would like to specify the main functions on the system independent of the interface mechanisms used to use those functions.
- So we look at the scenarios and think "what are the real core state changes, and what are the less stable GUI aspects?" — what DB or IS people would call "transaction commits"; we have already begun the process by highlighting them in the storyline. Each of these comprise a “success units” which delivers some value to the user.

2. Yes, this is a design activity — albeit "external", or "business" design.

- Now we specify those more abstract actions, and separate the interface details to another package e.g. when a refinement adds a particular user-interface to the "business core".
- We also separate out queries on the core — how to look up member details, titles, etc. Typically these will not be specified in detail, so long as the type model offers the required information. The user-interface prototypes will capture the required query paths that the design must support.

15.4 *System specification*

This is the deliverable of the analysis phase, though it can be somewhat more detailed than the traditional deliverables of analysis, so as to provide a less ambiguous and more consistent understanding of what is required of the system. A feature of an OO analysis (by contrast with other rigorous methods) is that the description centres on a type model derived from the business model: it is therefore easier to relate to the business, especially when changes need to be made.

A Specification is best presented as a set of Subject areas — an informal division which relies on the idea that different things can be said about a type in different places: so we can focus separately on different aspects of a type's relationships and the behaviour associated with it.

For each subject area, we will present:

- an informal description of the system's role and knowledge in this area;
- a type model of the relevant aspects of the system's state; which is a vehicle for describing the operations the system can perform in this area.
- each operation-spec, given as a postcondition, written
  - informally and
  - formally, in terms of links in the system type model.

15.4.1 Subject areas
Membership

Rq 1 *The System records membership of each store, including making new members, changing their details, and deleting them. Members who have not been heard of for ages are periodically deleted.*

At present we are able to give only informal descriptions of these operations:

```
action confirmMember (store: Store, person: Person)
    -- If person is not a member of store, makes new membership relation between them
action changedDetails (m:Member, p:Person)
    -- Replaces old personal details for m with the new ones in p.
```

action delete (m:Member) -- Deletes m from records

action purge (Store, today:Date)

-- remove from members list all those who have not hired in > 2 years

More precise descriptions can be achieved if we model the state of the system using types adapted from the Business Model. In other words, we will assume that the system's internal state can in some way represent some aspects of the Business.

Parts of the Business model relevant to membership are:

Rq 2 *A video Business services its customers through its Stores. A Person may be a Member of any number of Stores.*

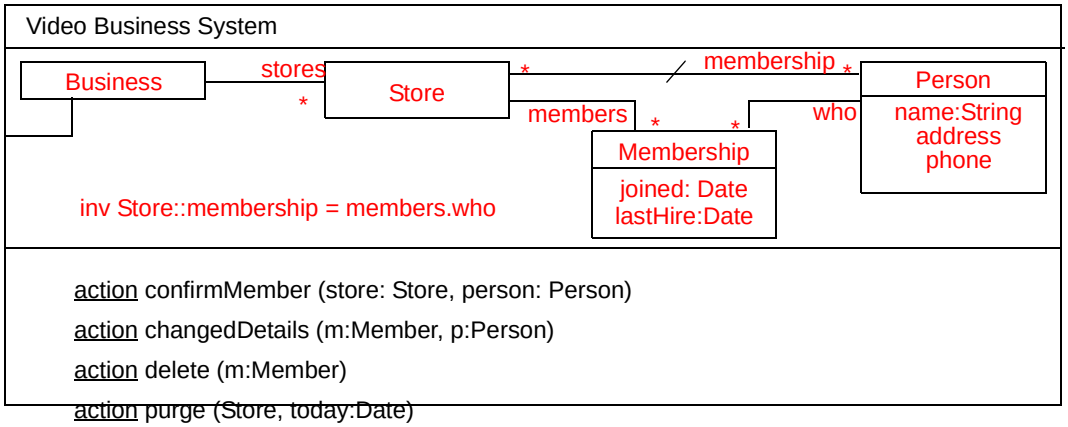


Figure 15.15 Type model for membership management

- As we said earlier, the (Video Business) System knows about exactly one Business, with its many stores.
- This is an abstract model of the state of the whole System, and says *nothing* about its construction — whether it is distributed or centralised, whether the links are database keys or pointers, whether the types can be found in the implementation as individual classes. These matters are all left to the design phase; a distributed object implementation, or a server-based one with remote screen-control clients would both be equally valid choices.
- We've represented the membership twice — as a direct association, and also refined to a type. The first makes it directly obvious how the relation works; the second makes it easier to attach more detail. The invariant tells how the two are related.
- This is of course only part of the system type model — more in a moment. The diagram can be interpreted as a set of assertions about Video Business Systems; and it so happens that there are other assertions presented later.

The actions can now be written in more detail, stating exactly what effects they have on the links in this abstract picture of the system's state. (They could all be written inside the diagram, but we'll move them outside to save clutter.) Snapshots help visualise the intention of the more formal statements:

action Video_Business_System:: confirmMember (store: Store, person: Person)

post If person is not a member of store, makes new membership relation between them

(store.members@pre[who=person]=0)

=> -- store members now include person, joined today

store.members[who=person] <>0

& store.members[who=person]::(joined=today & noShows=0)

To understand exactly what these statements are saying, it often helps to draw 'snapshots' of the relveant parts of the system's states before and after the operation occurs:

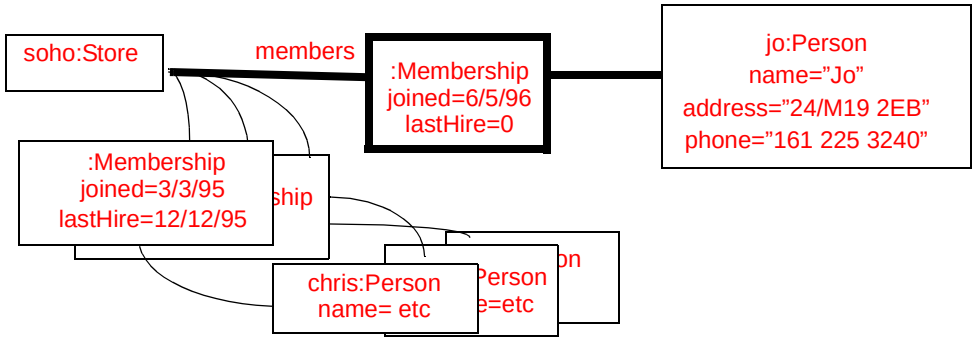


Figure 15.16 Snapshot: membership action

- From the Object Type Diagram, we see that each Store can have many links to Memberships, and the name of the sheaf of links is 'members'. This snapshot or Object Instance Diagram shows an example store with its members. Two states are shown, before and — in bold — **after** the action has occurred. In this case, a pre-existing Person object (perhaps set up by the user interface) has been linked to the store through a new Membership object.
- Snapshots show only an example of a particular occurrence of an action in a particular situation. They are therefore used only for illustration, and are not adequate documentation by themselves.

  To continue with the other action specs:

  action Video_Business_System:: purge (store:Store, today:Date)

  post remove from members list all those not hired for > 2 years

  store.members –= store.members[today–lastHire > 2.years]

  action Video_Business_System:: changedDetails (m:Member, p:Person)

  post m.who = p

  action Video_Business_System:: delete (m:Member)

  post remove m from list of members of what used to be its store

(m.~members@pre).members –= m

Rental

Rq 3 *The system records Rentals made by members. Each Rental is a contract whereby several Copies of a video may be taken away for an agreed period in exchange for a set charge. If any copy is kept beyond the date it is due for return, the system records a fine for it; the charges are made separately. The store's rates may vary, and are different for different video titles, and may include discounts for long periods; but rates charged for a rental are as they were at the time of hire.*

Rq 4 *Records of rentals are kept until they are no longer needed for statistical and accounting purposes.*

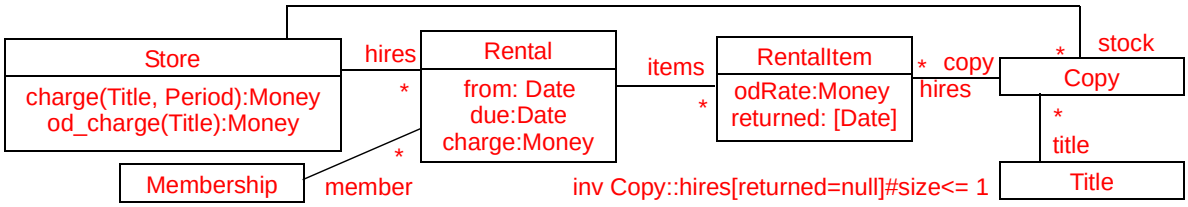


Figure 15.17 Type model: rentals

A Rental Item represents the hire of an individual Copy. Each Copy may have many RentalItems, though at most one may be not yet returned.

A Copy is said to be 'out' iff it has an unreturned hire; it is 'o/due' iff it is out and today is later than the rental due date. It is 'in' iff not out.

```
Copy:: ( isOut = (hires[returned=null] <> 0)
    &    isOD = (isOut & hires[returned=null].~items.due < today)
    &    isIn => not isOut )
```

A Rental is said to be complete iff all of its RentalItems are returned.

```
Rental:: isComplete = (items[returned=null] = 0)
```

System operations involving these types:

```
action hire (store:Store, to:Date, copies:Set(Copy), m:Membership)
    post      -- a Rental for these copies is added to this store's list of rentals
        store.hires += new Rental[member=m & from=today & due = to
                & charge = (store.charge(copies.title,to–today))..sum
                & items=(new RentalItem[copy:copies
                        & odRate=store.od_charge(copy.title)
                        & returned=null])]
action return (copy:Copy, store:Store)
    post      -- if this copy is out, this copy's rental is marked returned
```

```
        copy.~stock=store & copy@pre.isOut)
    =>  hires[returned@pre=null].returned=today
action purge(store:Store)
    post    -- get rid of all complete Rentals made longer than 6 years ago
        store.hires –= hires@pre[isComplete & (today–from)> 6.years]
```

Reservation

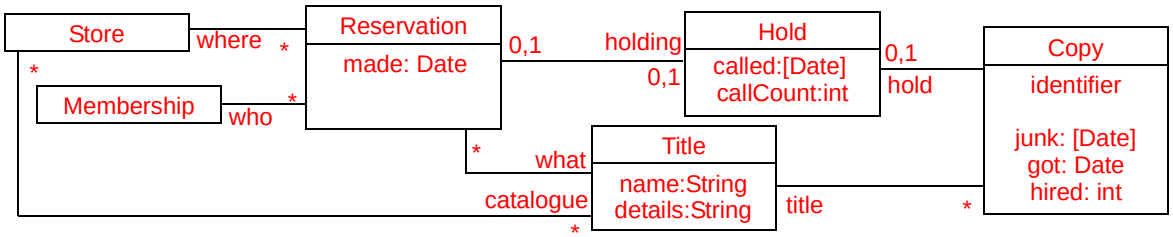


Figure 15.18 Type model: reservations

A Hold represents the fact that a Copy is being kept aside for a Reservation, waitiing for the Reserver to come and get it. A Hold not linked to a Reservation represents a Copy which is in the Hold bin, having been held; and for which the reservation has been cancelled, so that it should be returned to the shelves.

If it is 'in', a Copy is said to be 'held' if it is in the Hold Bin for some Reservation; 'hold-cancelled' if it is in the Hold Bin but has no Reservation; 'shelf' if available to be hired; 'sale' if it can be sold; 'junk' if it has been withdrawn from circulation.

```
Copy:: ( isHeld = (isIn & hold <> null)
    &   isHoldCancelled = (isHeld & hold.holding=null)
    &   isShelf = (isIn & not isHeld)
    &   isSale = (hired=0)
    &   isJunk = (junk <> null) )
```

Actions relevant to these types:

```
action reserve (member:Membership, here:Store, title:Title)
    post    -- if title is in the store's catalogue, add a Reservation for this title
        title : here.catalogue
            =>  new Reservation
                    [made=today & where=here & who=member &
                        what=title & holding = null ]
action cancel(res: Reservation)
post    delete res

action return (copy:Copy, here:Store)
post    -- if there are pending reservations for this title, hold copy for one of them
```

```
        copy.title.reservations@pre[holding=null & where = here].size>0)
=>      copy.hold.reservation:: (holding@pre=null & what=copy.title & who)

action hire (here:Store, to:Date, copies:Set(Copy), m:Membership)
post        -- clear any reservations for these titles and member,
            -- and clear the Hold of any copies actually taken away
        delete copies.title.reservations[who=m & where=here]
&       delete copies.hold
```

Account

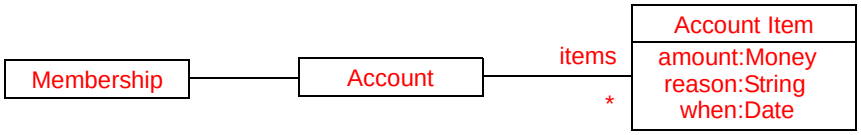


Figure 15.19 Type model: accounting

Operations relevant to these types:

```
action credit (ac:Account, what:Money, why:String)
post        ac.items+= new AccountItem[amount=what &
                                reason=why and when=today]

action hire (store:Store, to:Date, copies:Set(Copy), m:Membership)
post        -- add hire charge to account
            (store.hires–store.old(hires))::
            (member.account.items+= new AccountItem [
                amount = charge & reason="hire" & when=today])
action return (copy:Copy, here:Store)
post        -- add overdue charge, if any, to account
            let thisHire= copy.hires [returned@pre=null],
            (thisHire::(rental.due<returned))
            =>  thisHire.rental.member.account += new AccountItem [
                amount=thisHire::(odRate*(returned–rental.due))
            &   reason="overdue" & when=thisHire.returned ]
```

15.4.2 Putting it all together ...

Let's summarise the pieces of model we've seen so far in a single diagram. A good tool would make it easy to switch between the topic-specific views and this large view.

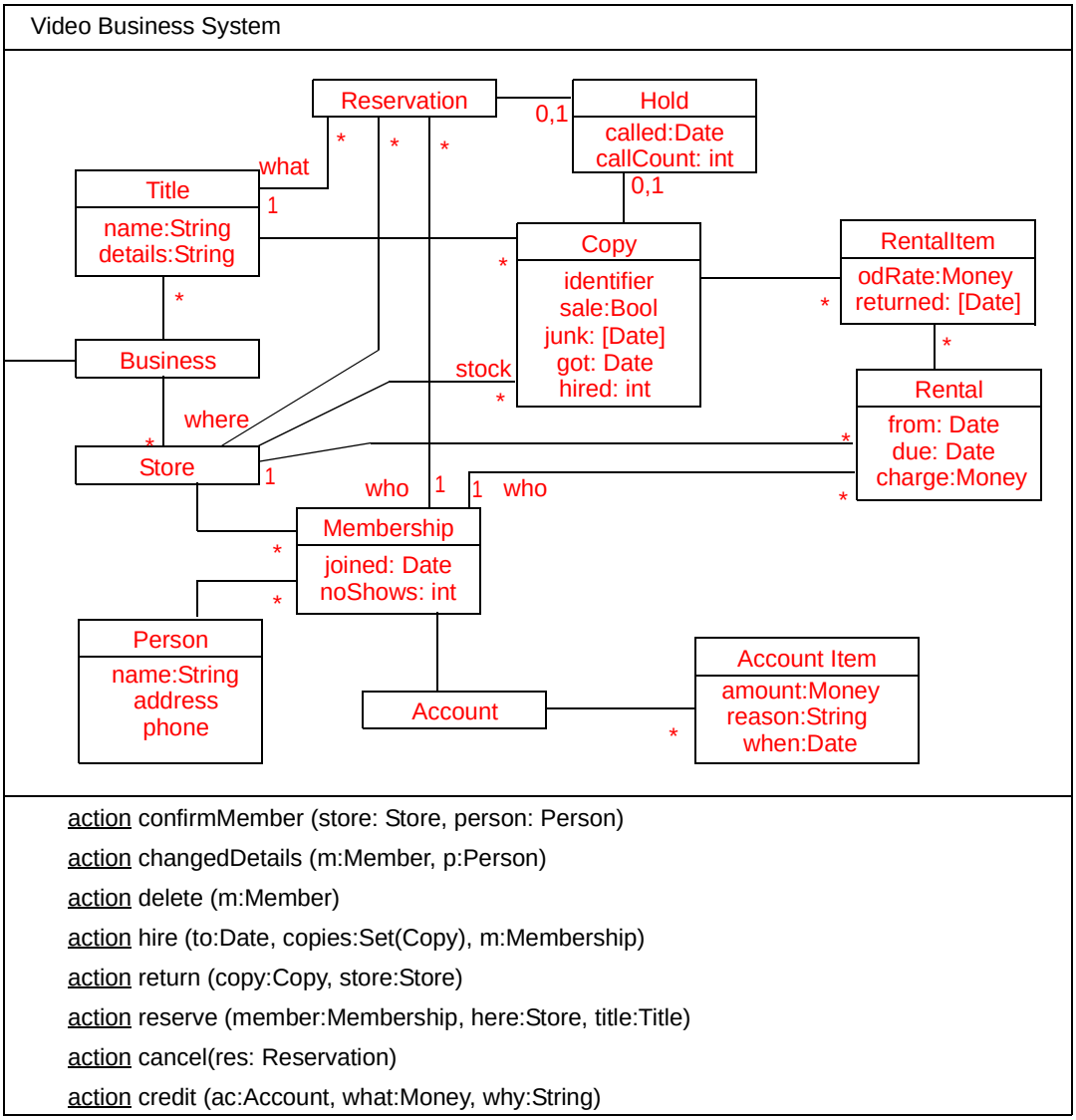


Figure 15.20 Combined type specification

We could also bring together the various pieces of specification we've seen for each action. For example, return has cropped up in several places. Its spec is the conjunciton of them all:

action return (copy:Copy, here:Store)

```
post
(
        -- if this copy is out, this copy's rental is marked returned
        copy.~stock=store & copy.isOut@pre
    =>  hires[returned@pre=null].returned=today
) &
(       -- add overdue charge, if any, to account
        let thisHire= copy.hires [returned@pre=null],
        (thisHire::(rental.due<returned))
        =>  thisHire.rental.member.account += new AccountItem [
            amount=thisHire::(odRate*(returned–rental.due))
        &   reason="overdue" & when=thisHire.returned ]
) &
(       -- if there are pending reservations for this title, hold copy for one of them
        copy.title.reservations@pre[hold=null & where = here]->size>0)
    =>  copy.hold.reservation:: (@pre.hold=null & what=copy.title)
)
```

15.4.3 Statecharts

It is often useful to express the behaviour of a system by showing diagrammatically the effect of each of its operations of some part of its state. The 'part of its state' we consider is usually (though not necessarily) one object in its model. By doing this for all the parts, we can build a picture of the behaviour of the whole thing.

Copy

We've already defined various states for Copies (under 'Rental' and 'Reservation' above). Here's a statechart for them. (Technically, it's a statechart for the entire system, where each state represents the truth of a predicate about any given Copy recorded within

it.). Most of the guards needed in a statechart can be made very simple by introducing convenience attributes, including parameterized ones, on the type model.

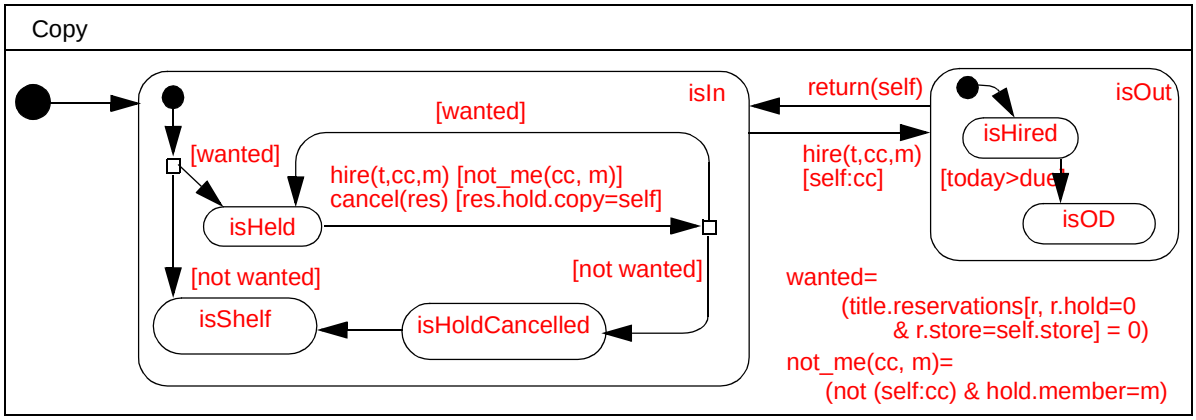


Figure 15.21 Copy statechart

When a copy is created or returned, it is either held for an outstanding reservation, or put onto the shelf. It is held if it is 'wanted' — that is, if there is a reservation for this title and in this store which does not have a Hold.

A copy ceases to be 'in' when it is hired (that is, when a hire event occurs in which it is one of the copies hired). If a copy is held and the reservation is cancelled, the copy is either reallocated to another reservation, or becomes Hold-Cancelled until the clerk checks it back to the shelf. The same happens if another copy of the same title is hired to the member it's held for.

A statechart frequently throws up questions not noticed previously. For example, how is a new Copy introduced? What is the operation whereby the clerk puts a hold-cancelled copy back into the shelf?