Chapter 14 How to Build a Business Model

This chapter and the next two chapters illustrate the use of Catalysis on a sample problem and discuss the how-to of applying Catalysis. This chapter focuses on building a business model; this is usually the model of a problem domain or business and not of a software "business object" layer sitting behind a user interface. The software objects in the subsequent chapters are based on the business objects, and a naming scheme is used to associate them.

We start with a set of patterns that describe the process of building a business model and then illustrate their use in the case study.

14.1 *Business Modeling Process Patterns*

This section outlines the major steps in building a business process model.

Pattern 13.2, Reengineering, describes reengineering activities in general and discusses building as-is and to-be models to guide that effort.

Pattern 14.1, Business Process Improvement, discusses specifics that apply to business process modeling.

Pattern 14.2, Make a Business Model, covers how to go about constructing a useful business model.

Pattern 16.6, Separate Middleware from Business Components, discusses a specific and common concern in business modeling: designing and extending heterogenous and federated software components so that they directly track the business independently of changes in technology.

Pattern 14.13, Action Reification, introduces a common modeling pattern. When a use case is refined into a sequence of finer-grained actions, it is useful to model the abstract use case in progress as a model type in the detailed level, going through a life cycle as the detailed actions take place.

Whereas this section's patterns are about organizing the process of business modeling, Section 14.2, Modeling Patterns, covers some of the most essential patterns that are useful in the actual construction of a business model.

Section 14.3 shows how some of these patterns have been applied to the case study of a video rental business at an abstract level. Section 14.4 details this model using action refinement, showing the finer-grained actions involved in the business.

Pattern 14.1 Business Process Improvement

In this pattern, you abstract and re-refine to get an improvement in business organization.

Intent

The intent is to improve the organization of a business, a process not necessarily involving software or computing machinery. In the process, we can review roles and processes in the organization and can also require software systems development.

Considerations

A business improvement effort may be triggered by the installation or upgrade of a software system, a perceived quality problem in business performance, or the hiring of a new senior executive. Although the explicit request may be simply to work on a software project, the analyst's scope often expands to include business improvement. This is part of what the system context diagram (see Pattern 15.5, Make a Context Model with Use Cases) is about—the computer system as one element in the design of a business process.

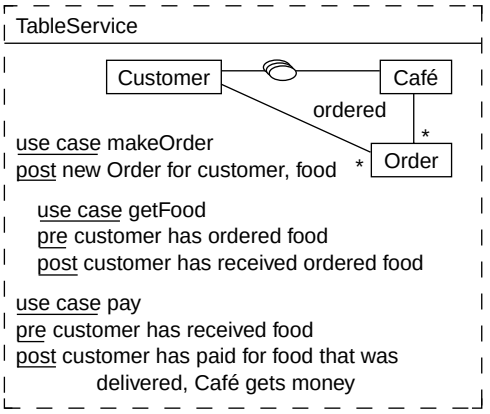
Companies, departments, people—and hardware and software systems—can all be thought of as interacting objects; so can the materials and information in which they deal. Designing the way in which a company or department performs its functions—whether making loans, manufacturing light bulbs, or producing films—is principally a matter of dividing the responsibilities among differentiated role players. It also involves defining the flow of activities and the interfaces and protocols through which they collaborate to fulfill the responsibilities of the organization as a whole (which is one role player in a larger world).

This activity is similar to the problem of object-oriented design, and the same notation and techniques can be applied. (Indeed, an effective help in deciding the distribution of responsibilities among software objects is to pretend that they are people, departments, and so on, although the analogy can be carried too far!) This similarity should not surprise us, because the big idea of OO programming is that the software simulates the business.

Strategy

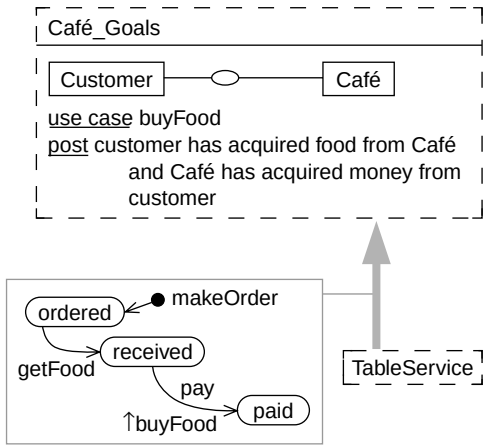
Approach this problem as a particular case of Pattern 13.2, Reengineering.

Make a business model (see Pattern 14.2, Make a Business Model), including associations and use cases, in which you reflect the existing process. This example is merely a sketch, less formal and less comprehensive than would be useful.



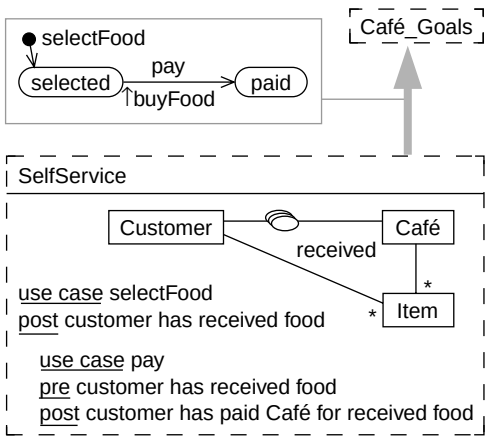
For any as-is element, ask, "Why is it so?" Abstract the collaborations to single use cases. Abstract groups to single objects (for example, individual person roles to departments). Specify abstract use cases with postconditions rather than sequences.

It may be useful to reiterate this to several levels of abstraction. Each abstraction generally represents a goal or goals of a part of the process or organization.



Refine the abstract model to a different design. Create the new design and document it as a refinement of the abstraction (see Chapter 6, Abstraction, Refinement, and Testing).

Evaluate the result in relation to quality aspects, including those not readily expressed as postconditions, such as error rate, cycle time, and costs. In the end, this technique can only suggest alternatives. Human, political, and many other constraints will influence the decision.



Consider how to plan the changes so as to cause minimal upheaval and at minimal risk (see Pattern 13.2, Reengineering).

Benefit

As always, there are many benefits to the act of formalizing your understanding of the situation in a well-defined notation, even at the business level.[1]

- It exposes gaps and inconsistencies and prompts questions you might never otherwise have thought to ask.
- It facilitates clear discussion and mutual understanding among those concerned.
- It provides an unambiguous method of encoding your current understanding even if your clients don't understand it directly (see Pattern 15.13, Interpreting Models for Clients).

The abstract and re-refine technique helps ensure that overall goals of the organization are still met insofar as they can be expressed in a functional notation.

1. Experience suggests the benefit might be even greater here, where we are conditioned to accept outrageously loose terminology.

Pattern 14.2 Make a Business Model

This pattern describes how to build a type and/or collaboration model that expresses your understanding of some part of the world, making it the final description of your understanding of your client's terminology.

Intent

The intent is to understand clearly the terms your clients are using before anything else and to ensure that they understand the terms in the same way that you do.

You can build a business model as a prelude to

- Pattern 14.1, Business Process Improvement
- System development, Pattern 13.1, Object Development from Scratch

Because a business model is not oriented to any single design, it can serve as the basis for many designs and structures within the business.

Considerations

The term *business* covers whatever concepts are of primary relevance to your clients and is not necessarily business in the sense of a commercial enterprise that makes money. If you're being asked to design a graphical editor, your business is about documents and the shapes thereon. Cursors, windows, handles, and current selections are possible parts of the mechanics of editing, but clients don't care about these mechanics except as an aid to produce their artwork. If you're designing a multiplexor in a telecommunications system, your users are the designers of the other switching components, and the business model will be about things such as packets and addresses. If you are redesigning the ordering procedures of a company, the business model is about orders, suppliers, people's roles, and so on.

There can be many views of a business. The concerns of the marketing director may overlap those of the personnel manager. Even when they share some concepts, one may have a more complex view of them than the other. See Chapter 6, Abstraction, Refinement, and Testing.

You may have frameworks available from which this model can be composed.

Strategy

You should end up with a model showing the types of main interest. They should have static associations and attributes, and they should have action links showing how they interact. The model consists of diagrams, invariants, and a dictionary (see Chapter 5, Effective Documentation). You should be able to describe any significant business event or activity entirely in terms of the model.

In making a business model, you will draw on a number of sources.

• *Existing procedures, standards documents, software, and user manuals*: When these exist, they should be consulted. But keep in mind that procedures as written often do not reflect the actual operations. User manuals for existing systems are a rich source of information. They act as a key input to reverse-engineering an abstract model of a system and therefore of a business.

• *Observation and interviews*: These include actual observation of procedures at work combined with interviews with the persons involved. Techniques such as CRC cards can be effective for eliciting information on as-is processes.

• *Existing relational or entity-relational (E-R) models*: These provide a quick start on the terminology and some of the candidate object types in the business. However, because of the mores or normalization and the exclusive focus on stored data, these models can often be considerably simplified to build a type model. Where used, triggers and stored procedures often encode many business rules.

• *Existing batch-mode systems*: Often, late-night COBOL batch jobs encode critical sets of business rules. Many of these rules can be captured as time-dependent static invariants on the type model (the batch job cuts in to make sure that no objects will be in violation of these invariants, come sunrise) or as effect invariants, of the form "Whenever this thing has changed, that other thing must be triggered."

• *Feedback from prototypes, scenarios, and models*: As always, any "live" media provides valuable feedback and validation of models being built. For example, storyboards, prototypes of UI screens, CRC-card-based scenarios, and model walkthroughs can all be used.

There are useful rules of thumb for finding objects, attributes, and actions. Looking through existing documents (business requirements, user manuals, and so on), map nouns to object types, verbs to actions, static relationships to associations, rules to static and dynamic invariants, and variations to subtypes or other refinements.

Focus on one type at a time and consider the interactions its members have with other objects. Draw use-case ellipses linked to participating objects. Some actions may have several participants of the same or different types. What roles are the participants playing within this collaboration? What information do they exchange?

What is the net result of an action? For example:

For an airplane: control—Pilot*; check position*—GPS, GroundStation*; lift*—Atmosphere.

Focus on one type at a time and ask yourself, At any one moment, what information would we want to record about this type of object? Write this information as attributes and associations sourced at this type. For each one, ask what the type of that information is, giving the target type. Don't confuse static information with actions; actions abstract interactions, whereas attributes represent information known before and/or after the actions. For example:

For an airplane: pilot, copilot: both of type Pilot*; airspeed, groundspeed:* Speed*; scheduled-to-land-at:* Airport*; previous 100 destinations: set of* Airports*; pilot's-spouse's-favorite-shoe-color:* Color.

Use snapshots to verify that all the information of interest can be represented by the static associations.

Determine a consistent level of abstraction in relation to actions: Are you going to worry about individual keystrokes or talk only about broad transactions? The highest-level action that could be useful should accomplish a business task or objective or should abstract a group of such actions. The lowest-level action that could be useful should constitute an indivisible interaction; if the interaction fails to complete successfully or otherwise is aborted, there should be no effect that would be useful at the business level.

Some business models focus on the types that a single client can manipulate, such as the Drawing example here. In this case, all the actions are shown localized in the Drawing and its constituents. Other models must be more concerned with interactions between objects, such as the Library example. Figure 14.1 illustrates both models.

The resulting business model acts as the central glossary of terms for all projects associated with it: business engineering, software requirements, and so on.

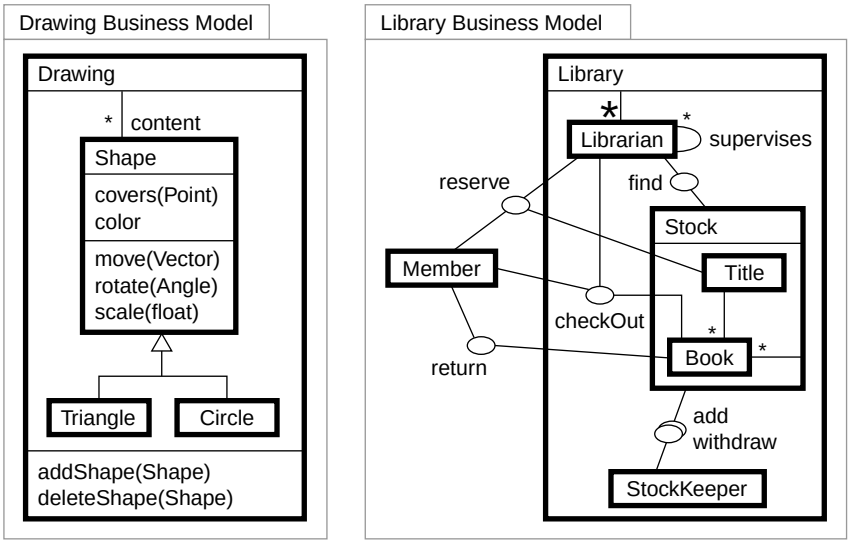


Figure 14.1 Range of business models.

Pattern 14.3 Represent Business Vocabulary and Rules

The purpose of the model is to represent both the terms used in the business and the rules and procedures it follows. Reduce business rules to invariants, generalized effect invariants, action specs, and timing constraints.

Intent

The intent is to reduce misunderstandings between colleagues about the terms and parameters of the business. Making some of these business rules explicit and precise aids identification of software requirements (which parts of the rules are dealt with by which component, which by users), type models (the models provide the vocabulary that helps specify the rules), traceability from software to business, and enterprise-level knowledge management (explicit representation of business knowledge and rules).

Strategy

Seek business rules in the current actual business process, the process as documented, and rules encoded in existing software systems (from user manuals, database triggers, and batch mainframe programs; see Chapter 17, How to Reverse-Engineer Types). Restate these business rules in terms of the following (in descending order of preference):

- Static invariants over the model
- Effect invariants that must apply to every action
- Timing constraints ("This must be done within . . .")
- Action specifications, including time-triggered actions

If an informal rule cannot be formalized in this way or starts to look complicated, you should suspect that the static model is inadequate and add new convenience attributes to make it simpler. Adding new named effects can also simplify things.

Wherever business rules are susceptible to change, exploit packages to separate them out. Remember that, in Catalysis, you can always say more about a type, action, and so on in another package; separating certain rules makes versioning and configuration management easier.

Pattern 14.4 Involve Business Experts

The idea here is to keep the end users involved in the business model.

Intent

The business model should be owned by the people who run the business. (The IS department is there only to coordinate writing it down.)

Strategy

Create the first draft of the model by interviewing experts and by observing what they do. Goals, achievements, tasks, jobs, and interactions of any kind should be sketched as actions; nouns are sketched as types.

For each action that appears on your sketch, ask, "Who's involved in this action? Whom does it affect? Looking at different specific occurrences of it, what could be different from one occasion to another?" These questions yield the participants and parameters. Don't forget to distinguish objects from types and parameters from parameter types.

For each type that appears on your sketch, ask, "What can be known about this? (Does it have a physical manifestation? What do you record about it? What creates it? What alters its state?" Or if it is an active object (a person or machine), "Whom does it interact with? What tasks does it accomplish?" These questions yield attributes and associations of the type as well as actions it participates in.

For each action, ask, "What steps are taken to accomplish this?" For each type, ask, "What constituent parts can it have?" These questions yield action and object refinements (see Chapter 6, Abstraction, Refinement, and Testing).

For each type, ask, "What states does it go through (or stages of development, phases, or modes)?" Draw state charts and check state transition matrixes (see Section 3.9.7, Ancillary Tables). The transitions yield more actions.

Remember to include abstract concepts (such as "Problem" or "Symptom") as well as concrete things (such as "Fault Report") and relate them together.

Ask, "What rules govern this action/state/relationship between types?" This will yield invariants.

With your analyst colleagues (who have perhaps been interviewing other experts concurrently) work out how to make the picture cohere. Formalize action specifications, define states in terms of attributes, make conjectures, and raise questions. Then go back to the experts with questions. (Don't be afraid to do so—people love explaining their jobs!)

After a reasonable (although still imperfect) model is drafted, hold workshops for the experts to review it. Take these in stages: static model first; then actions; then invariants and use case specs. Use snapshots to communicate the intent of the model.

Post parts of the model around the walls in a large room. Begin with a general presentation about types, associations, and actions so that your colleagues can read the diagrams. Then have everyone walk around the room—don't permit sitting—congregating around the subject areas they are most interested in. Have one of the analysis team on hand to walk people through the details. Also have scribes on hand to note all the comments.

Cycle until everyone more or less (although not entirely) reaches agreement.

Pattern 14.5 Creating a Common Business Model

Rather than build a business model first, you create a common model from several components in the business and from the definitions of the interfaces.

Intent

The intent is to deliver a business model, given that you have a variety of components, each with its own model. The components have been developed separately. You need to connect them, whether through a live interface, by enabling files to be written by one and read by another, or with a manual procedure.

Considerations

This problem is found wherever different software deals in the same area: Most of us have encountered the problems of translating from Word to or from some other documentation tool. The writers of the import, export, and translation modules must begin by building a model of the document structure.

It is also a common problem in large organizations; each department has bought its own software, and the IS department has the task of making them talk together more coherently. (See Section 10.11, Heterogenous Components.) The problem is particularly acute when two enterprises merge and they have two different systems talking different languages.

The most high-profile cases involve communication across the boundaries of organizations: from banks to the trade exchanges; from one phone company to another; or between airlines and ticket vendors.

The important thing is that we are not dealing with a model of any of the programs that communicate. Instead, at issue is a model of what they are talking to each other about, whether documents, financial transactions, aircraft positions, or talking pictures.

Strategy

First, agree with whomever is in charge of the various components that you are going to create a common standard. Try to ensure that the most powerful player doesn't just go ahead and do things his or her own way.

Second, accept that there will never be a standard model that everyone works to: There will always be local variants and extras. Adapters will be used to translate from one to another. Therefore, timebox the generation of the common model.

Now we come to the technical part. Consider the models of each of the components and write a common model of which all of them can be seen as refinements. Write abstraction functions (see Section 6.1.4, Model Abstraction) to demonstrate this, mapping each component model to your new model.

There will be interesting features within some of the components' models that not all of them can deal with. Not all televisions can deal with color signals; not all fax machines understand Group III compression; not all word processors understand tables; not all of the software components running a library will understand the concept of the acquisition date of a book even though most of them will understand its title.

Add to your common model these additional features (perhaps in different packages). Work out whether and how each component will deal with the additional information when it gets it and can't deal with it or doesn't get it when it expects it.

Pattern 14.6 Choose a Level of Abstraction

This pattern discusses the need to reach agreement with your colleagues concerning how much detail you're dealing with at each stage.

Intent

We have seen that it is possible to document objects and actions at any level of detail or abstraction. This is a powerful tool, but two problems commonly arise.

- It is very difficult for programmers (especially good ones) to keep away from the detail and implementation.
- Misunderstanding about the level of detail you're working at is a common source of arguments.

Strategy

Use the abstraction and refinement techniques (see Chapter 6, Abstraction, Refinement, and Testing) to explore the levels that are more abstract and more detailed than you have. For example, after you have identified some actions, ask yourself, "What more-abstract action or object are these part of?" And "What more-detailed actions or objects would form part of this?"

Make sure that for each action at any level, you at least sketch out the effects. Do this with sufficient precision to be aware of which types in the model are necessary to describe the effects.

Each layer of abstraction will have a coherent set of actions that tell the full story at that level of detail and will have a static model that provides the vocabulary in which that story is told. Different layers will have different sets of actions and static models. Programmers can write test code before writing implementations.

This process of exploration tends to provide insights leading to a more coherent model.

Begin by sketching the various layers. Decide in advance how much time you will spend in this exploration—anything from half an hour to one day, depending on the size of the model. By the end of it, you should be able to make a more informed choice about what level to focus on and elaborate completely.

14.2 *Modeling Patterns*

The preceding patterns have been about planning the development process. The next few patterns are more in the style of conventional analysis and design relating to the construction of a business model.

There are many good sources for patterns appropriate to this modeling. Our main concern here is to point out the patterns that we see as most essential. The Catalysis frameworks repository will eventually hold such model frameworks, and a project team can always define new model frameworks.

Pattern 14.7 The Type Model Is a Glossary

In this pattern, we ensure that every term used within the field appears in the model.

Intent

The intent is to ensure that the type model fully represents the business domain.

Considerations

The purpose of the business model is to represent all the vocabulary that is used by the people in the business. It can augment or form a more structured variant of the company or project glossary. Its advantage over the plain glossary or dictionary of terms is that it can summarize complex relationships readily.

On the other hand, in an alphabetical dictionary it is easier to look up the terms. But this loses the context and motivation for those definitions.

Strategy

Model the business by drawing the glossary explanations in pictures. Ensure that every concept written in documents that describe the business, or uttered by experts in the business, is represented somewhere. Generally, nouns map to object types, and verbs map to actions.

Remember that you are modeling the business, and not writing a database schema or program or modeling purely physical relationships. The associations are attributes drawn in pictures and not lines of communication or physical connections. (The latter would normally be drawn as an action.)

Feel free to include redundant terms: if people talk about it, include it. Write it as an invariant and describe how it relates to the other terms. An example is shown in Figure 14.2.

A level of detail you would have to optimize in an implementation is OK in a business model. For example, some pages in a library book may be torn. In an implementation, a list of integers, representing damaged page numbers, could be optionally attached to each

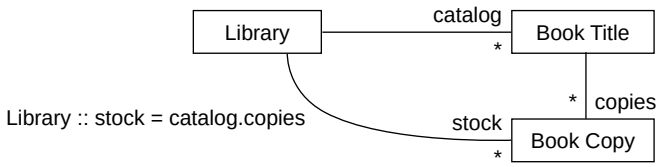


Figure 14.2 Business model with convenient terms and invariants.

book copy; but to model the world more directly, let's model a book as a list of pages and give every page a Boolean torn attribute.

Although the static type model tells what snapshots could be drawn at any one moment, a snapshot may contain plans for the future (schedules and timetables) as well as historical data (audit trails). So a list of past loans would be a valid attribute of a book.

Complement the pictures with explanatory prose. Attach to every type and action a glossary-style description of what it is. Have your tool dump an alphabetical list of these descriptions.

Divide the model into smallish type diagrams and write a narrative description of the business in which these diagrams are embedded. Use a modeling tool that supports embedding within narrative. (See Chapter 5, Effective Documentation.)

Pattern 14.8 Separation of Concepts: Normalization

Starting with a draft model, you enhance it by considering a number of rules.

Intent

Analysts who have experience with entity-relational modeling sometimes ask whether constructing a static object-oriented model is any different. Much is the same, and some aspects are different.

- In business and requirements modeling, the objective is not to design a database. Therefore, we do not need to be so strict about normalization, and we need not distinguish attributes from associations.
- Redundant links and associations are OK, if defined with invariants.
- Object models have sub- and supertypes.
- Object models include actions, whether joint or local to an object type. Sometimes the only distinction between one type of object and another is how it behaves (that is, how the effects of actions depend on it). An entity-relational model would not make these distinctions.
- In an entity model, each type should have a key: a set of attributes that together define the object's identity and distinguish one object from another. In an object model, every instance has an implicit identity: two objects may have all the same attribute values and yet still be different objects.

  Nevertheless, we can use some of the techniques from E-R modeling to improve a model.

Strategy

Look for the following triggers to re-factor your model.

- For a type having many associations and attributes, consider whether it should be split into several associated types (see Figure 14.3).

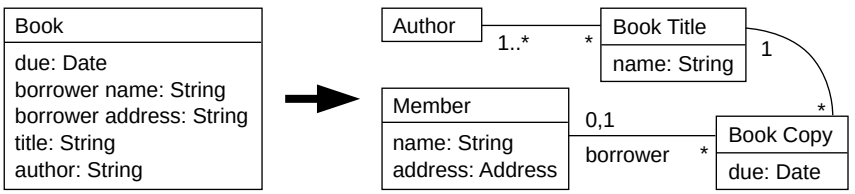


Figure 14.3 Splitting types.

- For any association, but particularly many-many associations, consider modeling it as a separate type (see Figure 14.4).

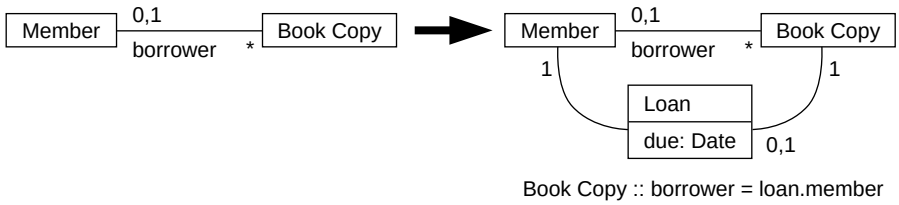
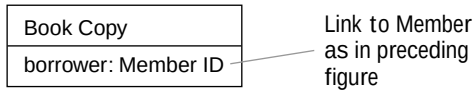


Figure 14.4 Reifying associations.

- For object identifiers, names, keys, tags, and similar attributes, use associations.



Pattern 14.9 Items and Descriptors

This pattern distinguishes things from descriptions of things.

Intent

When someone says, "I wrote a book" and someone else says, "I bought a book," do they mean the same thing by *book*? If the latter speaker is a publisher, then maybe; but when most people buy a book, they are buying one copy.

In a Library control system, we would have to distinguish individual Copies of books from the book Title. When Members borrow a book, they borrow a specific Copy; if they reserve a book, they're actually reserving a Title and don't usually care which copy they get. There are many Copies for one Title. Titles have attributes such as author and the actual title in the sense of a name (oops—another potential confusion here).

We often find the same potential confusion. A manufacturer's marketing people will discuss the launch of a new car; the customer will buy a car: But one means a model or product line, and the other means a specific instance of it. On the restaurant's menu, we see a variety of meals, with descriptions and prices; the meal a customer eats is not a description or a price but rather a physical item that conforms to the description. The flight you took the other day is one occurrence of the flight on the timetable.

The item/descriptor distinction is exactly the same as instance/class. (We use different words to avoid any additional confusion between the modeling domain and the software.) Indeed, in some programming languages (such as Smalltalk), classes are represented by objects, and each instance has an implicit link to its class. The class objects are themselves instances of a Class class, and new ones with new attributes can be created at runtime. This property of the language is known as *reflection*.

Strategy

In the most straightforward cases of items and descriptors, reflexive techniques are not necessary. The attributes of interest are the same for every book Copy in the library: who is borrowing it, what its Title is. Once we have spotted the distinction, there is no further complication.

A more difficult situation arises when we need to model a system in which the users may decide, while the system is running, that they need essentially new classes of items, with new attributes, business rules, and actions.

Few systems genuinely allow end users to make arbitrary extensions; those that allow it must provide a programming or scripting language of some sort. Normally there is some restriction; all the additions fit within a framework. An example is a computer aided design (CAD) system, in which new kinds of mechanical parts can be added and the users

can write operations that extend the code. Another example is a workflow system, in which users can define new kinds of work objects and their flows through the system.

In these cases, a framework can be defined (see Chapter 9, Model Frameworks and Template Packages) that imposes global constraints. The scripting language can be defined separately.

Pattern 14.10 Generalize and Specialize

Subtyping and model frameworks are used to simplify and generalize the model.

Intent

The intent is to make the model applicable to a wider range of cases and to provide insights to broaden the scope of the business or make it easier to understand.

Considerations

A supertype describes what is common to a family of types. A new variant can be created by defining what is different in the new type (see Chapter 3).

A model framework describes a family of groups of types. A collaborating group of types can be created by parameterizing the template. (See Chapter 9, Model Frameworks and Template Packages.)

The effort of generalization, either to a supertype or to a framework, usually leads to useful insights into the model. Also, it tends to simplify things: You find common aspects where you didn't notice them before, and you recast the model to expose them. Furthermore, the result is more easily extensible.

Strategy

Find common aspects between types in your model. Check that the similarities are real—that the users use a concept covering them both. Construct a supertype, and redefine the source types as its subtypes.

Find common aspects between groups of types in the model. Construct a template package containing everything that is common to the different occurrences, then rewrite them as framework applications.

Following are some specific triggers.

- Same types of attributes and associations in different types: form a supertype.
- Several associations with the same source and mutually exclusive 0,1 targets: form a supertype for the targets.
- Several *n*-ary associations with a common source, in which invariants and postconditions frequently need to talk about the union of the target sets: Form a supertype for the targets.
- Invariants and postconditions focused on type A use only some subset of the attributes of B, an attribute (or association) of A (see Figure 14.5): put the unused attributes in a subtype of B. (This reduces spurious dependency.)
- Similar "shapes" in different parts of the type and action diagrams: form a model framework that can be applied to recreate the collaborating groups.

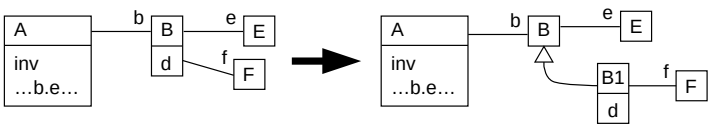


Figure 14.5 Introducing subtypes to decouple.

Pattern 14.11 Recursive Composite

Model an extensible object structure with a recursive type diagram.

Strategy

Let's look at a general framework for modeling extensible structures (see Figure 14.6). For a tree, set np to be 0,1; for a directed graph (with shared children), set np to *.

You must decide whether you will permit loops in the instance snapshots. If you will

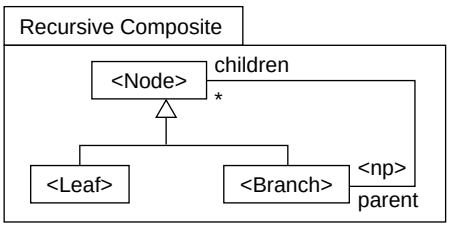


Figure 14.6 Template for recursive composite.

not, import instead the no-loops package (see Figure 14.7).

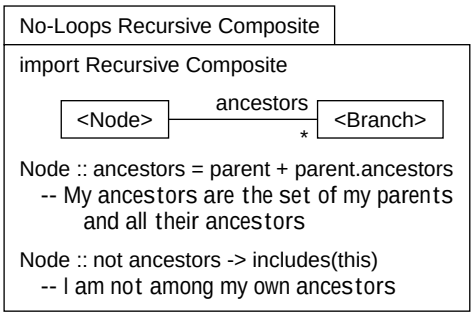


Figure 14.7 Recursive composite without instance loops.

Examples

The recursive composite pattern shows up in many problems. With np=1, no loops:

- *Programming languages*: Some statements (if, while, blocks) contain other statements; others do not (function calls, assignments).

- *Graphical user interfaces*: Some elements of a screen are primitive, such as buttons or scrollbars; others are composites of the smaller elements.
- Military or organizational hierarchies

With np=2, no loops:

- Family trees

With np=*, no loops:

- Macintosh file system
- Spreadsheets (each cell may be the target of formulas in several other cells)

With np=*, loops allowed:

- Road maps
- Program flowcharts

Pattern 14.12 Invariants from Association Loops

Loops in a type diagram suggest the possibility of an invariant.

Intent

The intent is to flush out useful constraints. You encounter this pattern when building a static type model (as part of a business model or component spec); it helps you to capture static invariants.

Motivation

A static invariant is a condition that should always be true, at least between the executions of any action that forms part of the same model. As a Boolean, it is composed of comparisons between pairs of objects. They might be < or = comparisons between numbers or other scalars; or more-complex comparisons defined over more-substantial types; or identity comparisons (whether one object is the same object as another).

Many invariants can be constructed when there is a loop[2] in the static type model: two different ways of coming at items of the same type. For example, a Library's books can be lent only to its own members. We could write

```
LoanItem :: borrower <> nil ==> borrower.library = library
```

We can see the loop easily (see Figure 14.8). Here's another:

```
Member :: age >= library.minimumAge
```

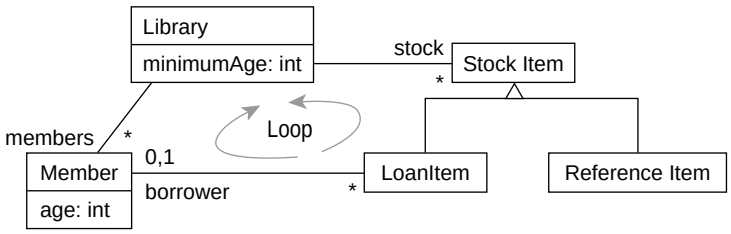
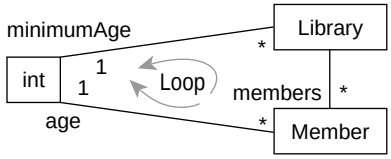


Figure 14.8 Association loops need invariants.

2. We are not referring to loops in instance snapshots here, as in Pattern 14.11, Recursive Composite.

Where's the loop? We said that attributes and associations are interchangeable in analysis. We could have drawn int as a type in its own box as shown here.



The only exception is to write a constant into the invariant itself.

Strategy

Whenever you notice a loop in the static type diagram, ask yourself whether there is any applicable invariant. The loop can have any number of links, from two upward; and a link can traverse up to a supertype.

14.3 *Video Case Study: Abstract Business Model*

In this section we begin our case study of a video rental business.[3]

14.3.1 Informal Description of Problem

The business sells and rents, or hires, videos to people through many stores. To rent a video from a store, a customer must be one of its members; becoming a member takes a few minutes. Anyone can buy a video without being a member.

Members can reserve a video for rent if all copies of it are currently hired. When a copy of the video is returned, the member will be called and the copy will be held for as long as three days, after which time the reservation will be canceled if not claimed.

Only a limited stock of videos is kept for sale, but a member can order a video for purchase. A store can order and acquire copies of a video from the head office.

The business head office sets the catalog of videos and their sale prices; this is common to all stores.

Each store keeps copies of a subset of the catalog for rent and for sale, and each store sets its own rental prices (to adjust for local competition). For strategic purposes, statistics are kept of how often and when last a video has been rented in each store. It is also important to know how many are available in stock for rent and for sale.

14.3.2 Concept Map

It is often helpful to start with an initial informal sketch of the main terms and concepts, drawn as a *concept map*. It serves as a concrete starting point for capturing the vocabulary used and the relationships between terms (see Figure 14.9). A concept map is simply a graph of labeled nodes and labeled (preferably directed) edges. We do not try to formalize the map or even worry much about distinguishing objects, types, actions, and associations. The concept map can serve as the starting point for the type model and collaborations.

14.3.3 Formalized Model

The aim is to get a more precise statement of the requirements. We segment the requirements into overlapping areas of concern, or subject areas, but focus in this

3. Thanks to Texas Instruments Software/Sterling Software for permission to use the video case study in the book; the authors originally developed this example for IT software.

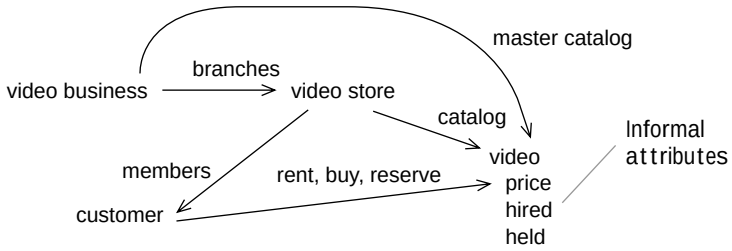
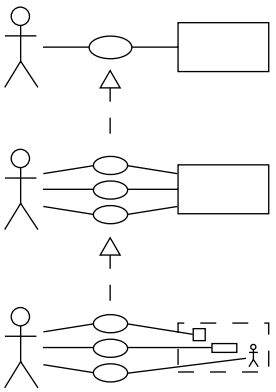


Figure 14.9 Concept map for video store.

description on the primary subject area of customer rentals and sales. There could be other subject areas for things such as marketing, purchasing, collections, and so on; they would define overlapping types and attributes and might add further specifications to common actions.

There are two distinct forms of abstraction at work on objects and actions. We must be clear about the boundary of objects and actions being modeled. We could model abstract actions, such as a complete rental cycle; or we could describe finer-grained interactions, such as reserve, pick up, return, and so on.

Similarly, we could model a video store as one large-grained object—an external view; or we could describe internal roles and interactions, such as the store clerk, the stockkeeper, the manager, and the video system—an internal view. We illustrate two levels of action abstraction in this chapter; the object granularity will be refined later, when we define the system context and user roles.



14.3.4 Subject Area: The Customer-Business Relationship

Rq 1 The business rents and sells videos through many stores.

The type model in Figure 14.10 summarizes the interactions between Stores and Customers. The elements on the diagram are as follows.

• *Object types*: The boxes represent types of objects. There can be many Video Businesses, Video Stores, and Customers in the world.

• *Associations*: The arc corporation is a link type (or association). It shows that for any given Video Store, there is exactly one Video Business, which is its corporation (although different Stores may have different Businesses). The * shows that there may be any number of Video Stores that share a single Video Business as their corporation.

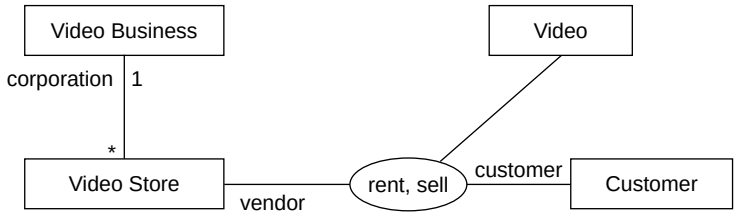


Figure 14.10 High-level business collaboration.

• *Use cases*: The ellipse (oval) represents a use case action called sell. (Actually, because exactly the same set of participants is involved in rent, we have written two labels in the same ellipse; but there are really two action types here.) Sell is an action each of whose occurrences involves one Video Store, one Customer, and one Video. (If there were several of any participant, we could use cardinality decorations.) We have not visually distinguished action participants from parameters here.

An occurrence of a use case action is a dialog of interactions, usually causing a change of state in some or all of its participants. The participants are all those objects whose states affect or may be affected by the outcome. An action is spread over time; when we look at it more closely, it is composed of smaller actions (such as scan shelves, choose, pay, and take away).

14.3.5 Dictionary

Accompanying the diagrams should be a dictionary carrying definitions of each object type, attribute, and action.

- Video Business A company with branches that sell and rent Videos.
- Video Store One of the branches of a Video Business.
- Customer A legal entity to whom videos may be sold or hired.
- Video An individual item that can be sold or hired.
- sell(VideoStore, Customer, Video) The interaction between a Video Store and a Customer whereby the Customer acquires ownership of an instance of a particular Video previously owned by the Business in exchange for money.
- rent(VideoStore, Customer, Video) The interaction between a Video Store and a Customer whereby the Customer is given possession, for some period of time, of an instance of a particular Video owned by the Video Business and normally kept at the store in return for a payment to the Store.

14.3.6 Use Case Actions: Precise Specs

It would be better to describe the use case actions more precisely. This cannot be done without the ideas that both VideoStores and Customers can own Videos and can possess money. Something like this:

use case sell (vendor:VideoStore, cust:Customer, v:Video)
post:
v is removed from stock of videos owned by vendor and is added to the videos owned by customer; the vendor's cash assets are increased by the price of v set by the vendor, and the customer's are depleted by the same amount.

What exactly do we mean by "the stock of videos owned by the vendor"? Let's augment the model to include this and then rewrite this statement more succinctly (see Figure 14.11). VideoStores and Customers are kinds of Owner; Owners have cash assets and own Videos. A VideoStore has a price for many Videos. We should augment the dictionary with these new types and attributes.

An attribute, like a link, is a read-only query. The main difference is that it is shown as text rather than pictorially. In theory, the two forms are interchangeable, but in practice the attribute form is used when the target type is defined in some other body of work. The inventor of Money does not know about Owners or VideoStores.

Now we can rewrite the description of sell. It should be written both in natural language (more readable) and precise terms (less ambiguous):

use case sell (vendor:VideoStore, cust:Customer, v:Video)
post:
-- v is removed from the stock of videos owned by the vendor:
vendor.owns –= v
-- and added to the customer's stock:
and cust.owns += v
-- The vendor's cash is increased by the vendor's price for v:
and vendor.cash += vendor.priceOf(v)
-- and the customer's assets are depleted by the same amount:
and cust.cash –= vendor.priceOf(v)

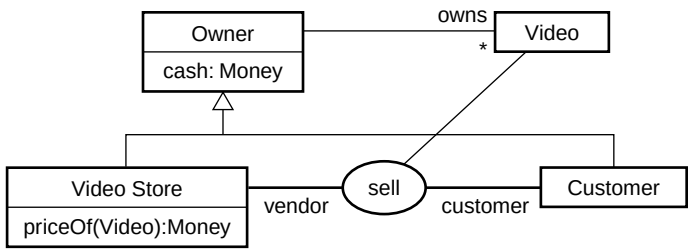


Figure 14.11 Revised and improved business collaboration for sell.

Every postcondition is a predicate (a Boolean, true/false condition) stating what must be true if the designer has done the implementation properly. So the clauses are connected by and, and there is no sequence of execution implied.

The purpose is to state properties of the result at the end of the use case, abstracting away from any details about sequences and how it is achieved. It is a relation between two

states, before and after the use case has occurred. In later examples, you can sometimes see the "before" value of a subexpression referred to as expression@pre.

The construct x += y is an abbreviation for x = x@pre+y. x is what it used to be, plus y; it is predefined as an effect. This is not an assignment as in programming language. Instead, it is only a description of a part of the relationship of the two states.

There are many uses for operations on sets in abstract descriptions. Because the traditional mathematical symbols are not available on most keyboards, OCL defines ASCII equivalents. It is easier to use + for union, * for intersection, and – for set difference (see Section 2.4.5, Collections), and in Catalysis these features can be extended by the modeler.

14.3.7 Preconditions and More Modeling

Now let's try to describe the rent use case. To be comparable to sell, the intention is to let it encompass the whole business of renting a video, from start to finish. We will separately describe how this refines into a sequence of constituent actions such as renting and returning.

There's a slight conceptual difficulty here. Although there's an obvious transfer of money from one participant to another, what you've gotten for it once you've returned the video is less concrete than in the case of a sale. The most you're left with as a souvenir is whatever impression the video has made on your mind. Nevertheless, there's no reason that we shouldn't model this abstract notion as a Past Rental and model the rent use case as adding to your list of them (see Figure 14.12).

In another part of the document, we've said that VideoStores and Customers are kinds of Owner, which have cash and own Videos; so it is unnecessary to repeat it here. Each

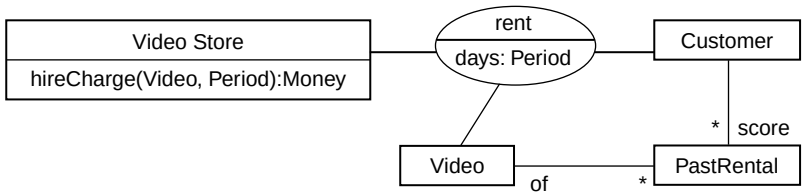


Figure 14.12 Model for rent.

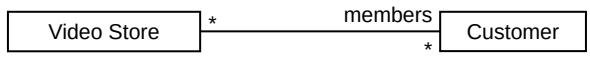


Figure 14.13 Incorporating the term members.

diagram is in effect a set of assertions about the elements that appear in it; if an element appears in several diagrams, then all their assertions apply to it. Rather than show all the

parameters pictorially by linking the use case to the types, we can show them as attributes within the use case ellipse.

One further complication is this requirement:

Rq 2 To rent a video from a store, a customer must be one of its members.

At present, there is nothing in our model representing membership, so we add it (see Figure 14.13).

Now we can state a precise description of the use case:

```
use case rent (hirer:VideoStore, cust:Customer, v:Video, days:Period)
post:
-- if customer is a member, then...
hirer.members→includes (cust) implies
    (    -- a new PastRental of v is added to customer's 'score':
         cust.score += PastRental.new [of=v]
         -- The vendor's cash is increased by
         -- the hirer's hire rate for v at the time of commencement of the hire:
    and hirer.cash += hirer.hireCharge@pre(v, period)
         -- and the customer's assets are depleted by the same amount:
    and cust.cash –= hirer.hireCharge@pre(v, period)
    )
```

Notice that nothing is said about the meaning of rent if the precondition is false. There might be another description of this use case elsewhere that covers that eventuality; but if there isn't, then we are saying nothing about what that might mean. Because we're describing rent and not "attempt to rent" or "inquire about renting," we leave it to a later, more detailed description to define precisely what happens when a nonmember demands to rent a video. It's part of the value of the postcondition approach that it allows you to paint the big picture, leaving the less important detail until later.

We could have modeled the hire charge as a per-day rate and multiplied it by the length of the hire. But doing it this way leaves the charging scheme open, allowing for reductions for longer rentals.

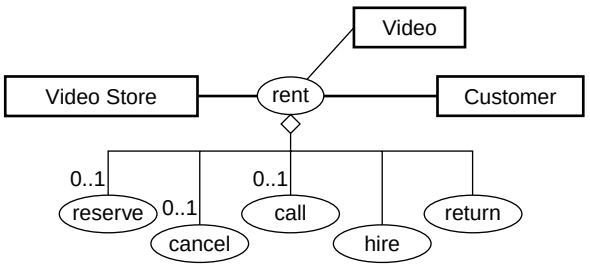


Figure 14.14 Refinement of rent.

14.4 *Video Business: Use Case Refinement*

Deciding we're interested in looking into one of the preceding use cases in further detail—let's choose rent as an example—we can show how it breaks into smaller constituents. The entire business of renting may involve reserving beforehand. So we could think of a rental as possibly reserving, followed by hiring and returning (see Figure 14.14).

The diamond indicates the relationship between an abstraction and its more-detailed constituents; it can include annotations such as multiplicity. Here it states that some combination of the constituent actions can be used to accomplish a complete rent; but we have yet to detail exactly how and in what order. This is only a summary diagram.

There may be other actions not mentioned here that can also be composed to make a rent; and these constituents may also be used in some other combination to make other abstract use cases. The constituent actions have their own participants, which we could have shown here or separately. They generally intersect with, or at least are mapped to, the participants of the abstract use case.

The relevant part of the informal business description is as follows.

> *Rq 3 Members can reserve a video for rent if all copies of it are currently hired. When a copy of the video is returned, the member will be called and the copy will be held for as long as three days, after which the reservation will be canceled if not claimed.*

To show the possible sequences of constituent actions and to specifically define which sequences correspond to an abstract rent use case, we model Rental as an object that goes through a sequence of states. In the transitions, s, v, c, and d refer to a store, video copy, customer, and rental period (see Figure 14.15). Frequently, such a reified action exists in the type model as an actual object, such as a progress record.

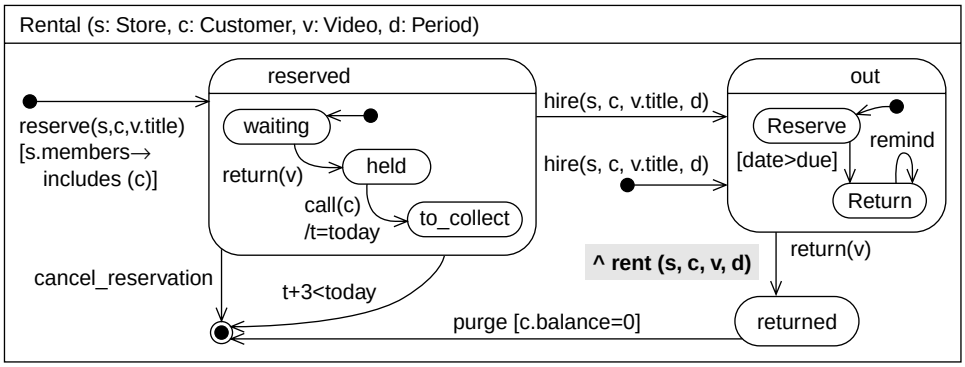


Figure 14.15 Refinement sequence for rent.

Round-cornered boxes are states, which can have substates; black dots are starting points. Arcs are state changes; they are labeled with the actions that cause them. The

square brackets are guards: the transition happens only if the guard is true. A forward slash (/) marks a postcondition. This diagram formally documents an action sequence refinement; any sequence of detailed actions starting from a top-level ● and ending with ^rent constitutes an abstract rent use case.

14.4.1 Refined Model

We should write down descriptions of the more-detailed actions; in this case we would likely still call each finer-grained action a use case. To begin with, let's keep it relatively informal, extending the dictionary with action definitions.

- reserve (VideoStore, Customer, VideoTitle) (At this point, we notice that there is a distinction between individual copies of a video—objects of type Video—and titles or catalog entries.) This use case represents the reservation of a given title by a member of a store. The title must be available at that store. The reservation will be recorded pending the return of a copy of that title to the store.
- return (Video) The return of a copy to its owning store. If there is a reservation for that title at that store, it will be held for the reserver to collect; otherwise, it goes back on the shelves. The hire record is marked "returned."
- call(VideoStore, Customer, Video) Applies when a returned Video has been held for this reservation. The reserver is notified by telephone that the Video can be collected. The reservation record is date-stamped, and, if the member has not picked up the copy within a fixed time, the reservation may be canceled.
- hire(VideoStore, Customer, VideoTitle, Period) A member takes away a copy of a particular title for an agreed period. If this customer had a reservation for this video and there is a copy held, then that copy should be the one taken. It makes sense only if there is a copy on the shelves beforehand or a held copy. A record of the hire is kept.
- cancel_reservation(VideoStore, Customer, Video) Occurs when the store becomes aware that the customer no longer wants the video. If a copy has been held, it is reallocated to another reservation or put back on the shelves. The reservation is deleted from the records.

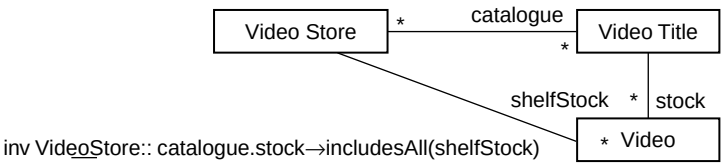


Figure 14.16 Clarifying video title and copy.

The distinction between Videos (individual copies) and VideoTitles has now become apparent. It gets documented together with an invariant (see Figure 14.16).

- VideoTitle The set of individual Videos having a particular title.
- VideoStore::catalogue The set of VideoTitles known to a VideoStore.

– VideoStore::shelfStock The set of Videos currently available for hire. A subset of the total stock of all titles.

We could also draw pictures showing the participants in each use case, duplicating the dictionary entries for the use cases. In practice, this seamless switching between text and diagrams may or may not be supported by particular tools.

14.4.2 Formalized Refined Use Case Specs

Looking again at the preceding spec of reserve, it's clear that we must represent the idea of a reservation in the model (to be able to state that the use case creates a record of one). The same thing applies to rentals. In fact, there may be some advantage in regarding these two records as two states of the same thing; then it will be easy to include any initial reservation as being part of the history of a rental.

A useful pattern here is Action Reification (see Pattern 14.13). We'll reify the use case as Rental but preserve the distinct idea of a Reservation as a state type (see Figure 14.17).

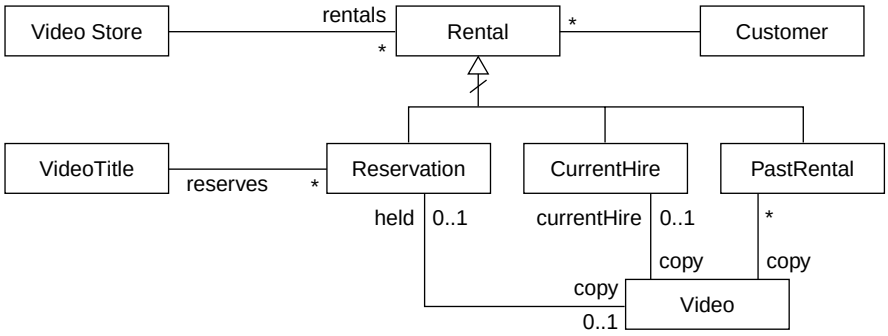


Figure 14.17 Using state types for a rental.

Now the finer use cases can be expressed more formally:

```
use case reserve (store:VideoStore, cust:Customer, title:VideoTitle)
pre:     store.catalogue→includes (title) -- The title must be available at that store,
    and  store.members→includes (cust) -- the customer must be a member
post:    store.rentals→includes (
           Reservation.new[ reserves=title and customer=cust and copy=null ] )
--There is a new Reservation in the store whose title and customer are as requested;
```

Notice that we are still talking about the interactions between the real-world objects here. The Rental is not an object in the store's computer system but rather is an abstraction representing the agreement between the parties.

Notice also that the postcondition restates (perhaps in more detail) what is shown in the state chart, given an interpretation of the states in terms of the model. This reminds us that

we should never write a state chart without defining the states in terms of the model attributes.

```
inv Rental::
    reserved self:Reservation -- reserved state is a Reservation state type
    and waiting = (reserved and copy=null) -- waiting means reserved with no copy
    and (held or to_collect) = (reserved and copy<>null) -- other substates
        -- we'll need a boolean attribute to distinguish held from to_collect
    and out = self:CurrentHire -- 'out' state is a CurrentHire state type
    and hired = (out and date=<due) -- substates of 'out'
    and overdue = (out and date>due)
    and returned = self:PastRental -- the returned state
```

Continuing with the other use case specs, return can be specified separately for its two effects on the state chart (of two separate Rentals):

```
use case return (v:Video)
pre:    v.currentHire<>null   -- v is rented out
post:   v.currentHire@pre : v.pastRental -- the rental is now part of v's past

use case return (v:Video)
pre:    v.title.reservation<>null and v.reservation.waiting
        -- v is a copy of a title that is the subject of a Waiting Reservation
post:   v.title.reservation[waiting@pre and held and copy=v]→size = 1
        --Of the resulting set of reservations for this title, there is exactly 1
        -- that was waiting and is now held for this video
```

The spec of hire can conveniently be split into a general part and two effects: one for hiring based on a reservation and the other without a reservation.

```
use case hire (store:VideoStore, cust:Customer, title:VideoTitle, period:Period)
pre:    -- title from catalogue, and customer among members
        store.catalogue→includes (title) and store.members→includes (cust)
post:   -- an available video copy of the title has been hired by the customer
        (   Video→exists (v | v.title=title and v.currentHire@pre = null
            and v.currentHire.videoStore=store
            and v.currentHire.customer=cust
            -- either based on a reservation, or without a reservation
            and ( hireFromReservation (store, cust, title, video)
                or hireFromCold (store, cust, title, video) )

-- we hire the video kept for this reservation
effect hireFromReservation
    (store:VideoStore, cust:Customer, title:VideoTitle, video:Video)
    =   (title.reservation@pre <> null and title.reservation@pre.copy=video)

-- we hire the video without a reservation
effect hireFromCold
    (store:VideoStore, cust:Customer, title:VideoTitle, video:Video)
    =   (title.reservation@pre = null)
```

Canceling a reservation clears it from the ken of customer, store, and the video:

```
use case cancel_reservation (r:Reservation)
post: (r.copy@pre <> null implies r.copy@pre.held = null) -- any held copy is released
    and r.videoTitle@pre.reservations –= r -- remove r from reservations for its title
    and r.videoStore@pre.rentals –= r and r.customer@pre.rentals –= r
```

We omit further detailing of these use cases and coverage of other subject areas.

Pattern 14.13 Action Reification

This pattern supports systematic progression from succinct abstract actions (or use cases) to detailed dialog, supporting the strategic aim to expose the most important decisions up front and keeping reliable traceability to the details.

Intent

A sequence of action occurrences can often be seen as a group with a single outcome, which can be documented with its own postcondition. (This sequence is not necessarily contiguous—there may be other unrelated actions in between them.) Conversely, when you're specifying a system, you should omit detailed protocols of interactions so as to understand overall effects.

Example

Consider the transaction of obtaining cash between a customer and an ATM. It has a clear postcondition, is often mentioned in everyday life, and can usefully be discussed between ATM designers and banks. The detail of how it happens—log in, select a service, and so on—can be deferred to design and may vary across designs.

Terms

The *finer* actions *refine* the *abstract* one, and the relationship is documented as an *action refinement* consisting of a *model refinement* and an action refinement sequence. Many refinements of one action may be possible.

A *sequence constraint* may govern actions, stating the possible sequences; it may be expressed in the form of a state chart. Of all possible sequences of finer actions, only some may constitute an occurrence of the abstract action; for example, the card reader and ATM keys can always be used, but only some sequences constitute a withdraw action. An *action refinement sequence* relates finer sequences to specific abstract actions, and can be expressed in the form of a state chart.

Strategy

Model the abstract action as an object—reify it. Implementations are not constrained to follow models, but refication often corresponds to a useful object.