Chapter 15 How to build a Business Model

Outline

This chapter, and the next three, illustrate the use of Catalysis on a sample problem, and discuss the *how-to's* of applying Catalysis. This chapter is focused on building a business model; this is usually a model of a problem domain or business, not of some software "business-object" layer sitting behind a user-interface. Of course, the software objects in the subsequent chapters will be based upon the business objects, using some naming scheme to associate them.

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

15.1 *Business modeling process patterns*

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

Pattern-11, *Re-engineering* (p.542) described re-engineering activities in general, and building of as-is and to-be models to guide that effort. Pattern-14, *Business Process Improvement* (p.551), discusses specifics that apply to business-process modeling. Pattern-15, *Make a business model* (p.553), covers how to go about constructing a useful business model. Pattern-45, *Separate Middleware from Business Components* (p.637), discusses a specific and common concern in business modeling — designing and extending heterogeneous and federated software components so they more directly track the business independent of changes in technology.

Pattern-26, *Action Reification* (p.577) introduced a common modeling pattern: when an 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 lifecycle as the detailed actions take place.

While this section's patterns are about organising the process of business modeling, Section 15.2, "Modeling patterns," on page 561, gives some of the most essential patterns that are useful in the actual construction of a business model.

Section 15.3 shows how some of these patterns have been applied to the case study of a video-rental business at an abstract level. Section 15.4 details this model using action refinement, showing the finer grained actions involved in the business.

Pattern-14 Business Process Improvement

Summary

Abstract and re-refine to get an improvement in business organization.

Objectives

Improve the organization of a business —not necessarily involving software or computing machinery. In the process, we may review roles and processes in the organization, and may also require software systems development.

Considerations

A business improvement effort may be triggered by the installation or upgrade of a software system; or 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 some of the former. This is part of what the System Context Diagram (Pattern-31, *Make a Context Model with Use-Cases* (p.589)) is about — the computer system as one element in the design of a business process.

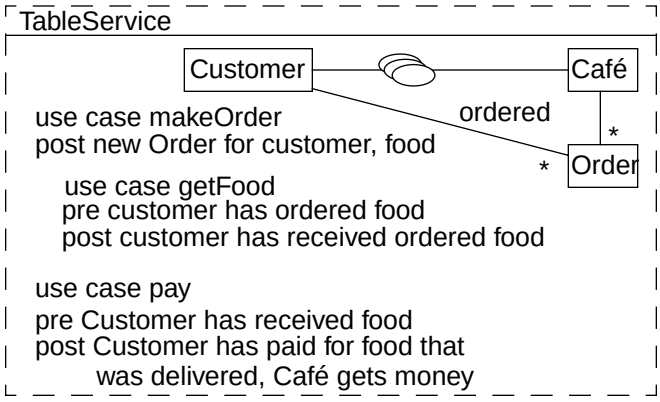
Companies, departments, people — and hardware and software systems — can all be thought of as interacting objects; so can the materials and transactions in which they deal. Designing the way in which a company or department performs its functions — whether making loans, manufacturing light-bulbs, or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

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

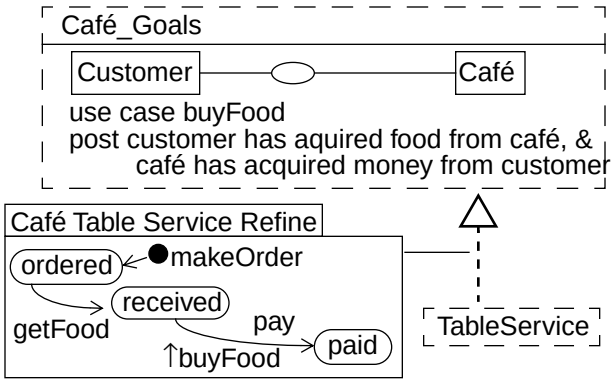
Strategy

Approach this problem as a particular case of Pattern-11, *Re-engineering* (p.542).

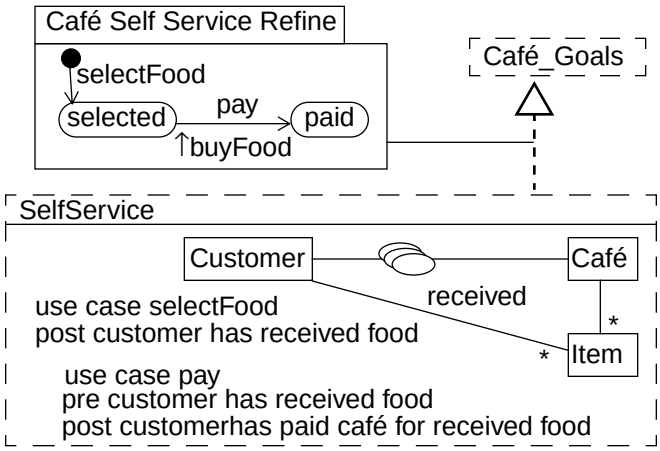
- Make a business model (Pattern-15, *Make a business model* (p.553)) including associations and use cases, in which you reflect the existing process. This example is merely a sketch, less formal and complete than would be useful.



- 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 re-iterate this to several levels of abstraction. Each abstraction generally represents some goal or goals of a part of the process or organization.



- Refine the result to a different design. Do the new design and document it as a refinement of the abstraction (Chapter 7, *Abstraction, Refinement, and Testing* (p.265)).



- Evaluate the result in relation to aspects not readily expressed as postconditions, such as error rate, cycle time, costs. Acknowledge that there are a great many of these. 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 (Pattern-11, *Re-engineering* (p.542)).

Benefit

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

- 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 (Pattern-39, *Interpreting Models for "Clients"* (p.606)).

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-15 Make a business model

Summary

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.

Objectives

Understand clearly the terms your clients are using before anything else; ensure that they understand the terms as well, the same way as you do.

You may build a business model as a prelude to:

- Pattern-14, *Business Process Improvement* (p.551)
- System development, Pattern-10, *Object Development from Scratch* (p.541)

A business model is not oriented to any single design, and may serve as the basis for many designs and structures within the business.

Considerations

"Business" covers whatever concepts are of primary relevance to your clients; 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 and windows and handles and current-selections are possible parts of the mechanics of editing them: clients don't care about those except as a aid to produce their artwork. If you're designing a multiplexor in a telecoms system, your users are the designers of the other switching components, and the business model will be about things like packets, addresses, etc. If you are redesigning the ordering procedures of a company, the business model is about orders, suppliers, people's roles,....

There may be many views of a business. The concerns of the marketing director and the personnel manager may overlap, even where they share some concepts, one may have a more complex view of them than the other. See Chapter 7, *Abstraction, Refinement, and Testing* (p.265).

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

Strategy

Aim. You should end up with a model showing the types of main interest. They should have static associations and attributes; and action links showing how they interact. The model consists of diagrams, invariants, and a "Dictionary" (Chapter 6, *Effective Documentation* (p.237)). You should be able to describe any significant business event or activity entirely in terms of the model.

Sources.

- Existing procedures, standards documents, software, and user-manuals: where these do 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: actual observation of procedures at work, combined with interviews with the persons involved in these procedures. Techniques such as CRC cards can be very effective at eliciting information on as-is processes.

- Existing relational or E-R models: these provide a quick start on the terminology and some of the candidate object types in the business. However, due to 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 those 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 no objects will be in violation of these invariants, come sunrise); or as effect invariants, of the form “any time this thing has changed, that other thing must be triggered”.
- Feedback from prototypes, scenarios, models: as always, any “live” media provides very 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.

Techniques.

There are some basic yet useful rules of thumb for finding objects, attributes, actions: looking through existing documents (business, requirements, user manuals, etc.) map nouns to 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 what interactions its members have with other objects. Draw use cases ellipses, linked to participating objects. Some actions may have several participants of the same or different types. Ask: What roles are the participants playing within this collaboration? What information do they exchange? What is the net result of an action?

> e.g. for an airplane: control — Pilot; check-position — GPS, GroundStation; lift — Atmosphere.

Focus on one type at a time and ask: At any one moment, what information would we want to record about this type of object? Write these as queries (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, while attributes represent information known before and/or after the actions.

> e.g. for an airplane: pilot, copilot: both of type Pilot; airspeed, groundspeed: Speed; scheduled-to-land-at: Airport; previous hundred 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.

Decide a consistent level of abstraction in relation to actions: are you going to worry about individual keystrokes, or just talk about broad transactions? The highest level action that could be useful should accomplish a business task or objective, or abstract a group of such actions. The lowest level action that could be useful should constitute an indivisible interaction; should the interaction fail to complete successfully, or otherwise be 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, the actions are all shown localized in the Drawing and its constituents. Other models need to be more concerned with interactions between objects, like the Library example.

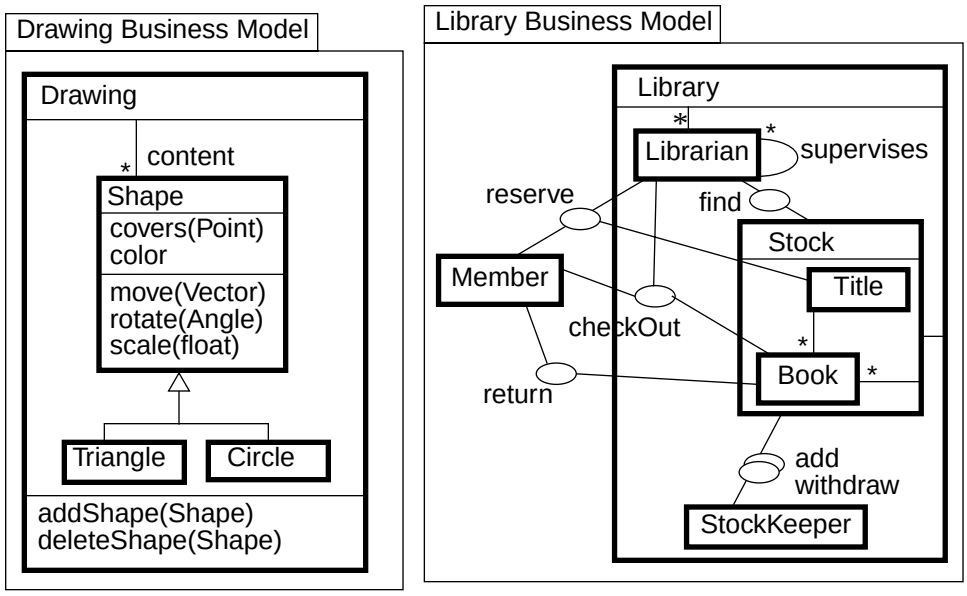


Figure 269: Range of “Business” Models

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

Pattern-16 Represent business vocabulary and rules

Summary

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.

Objectives

Reduce misunderstandings between colleagues, as to the terms and parameters of the business. Making some of these business rules precise directly aids identification of software requirements (which parts of the rules are dealt with by which component, which by users), type models (the models provides the vocabulary that helps specify the rules), and traceability from software to business.

Provide a well-understood set of terms and relationships, to which all other documents can be related.

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, batch mainframe programs; see Chapter 18, *How to Reverse-Engineer Types* (p.671)). Restate these business rules in terms of (in descending order of preference):

- invariants over the model
- generalised effects-specs that must apply to every action
- timing constraints "this must be done within ..."

If an informal rule cannot be formalised in this way, suspect that the static model is inadequate, and add material.

Pattern-17 Involve business experts

Summary

Keep the end-users involved in the business model.

Objectives

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

Context

Creating a business model.

Strategy

Create the first draft of the model by interviewing experts, and by observing what they do. Goals or achievements, tasks or jobs or 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? Who 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, parameters from parameter types.
- For each type that appears on your sketch, ask "What can be known about this? (Has it 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), "Who does it interact with? What tasks does it accomplish?" These questions yield attributes and associations of the type, and 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 yield action and object refinements (Chapter 7, *Abstraction, Refinement, and Testing* (p.265)).
- For each type, ask "What states does it go through? (Or stages of development, or phases, or modes.) Draw statecharts, check state transition matrices (Section 4.9.5, "Ancillary Tables," on page 180). The transitions yield more actions.
- Remember to include abstract concepts (like "Problem" or "Symptom") as well as concrete things (like "Fault Report"), and relate them together.
- Ask "What rules govern this action / state / relationships between types?" This will yield invariants.

With your analyst colleagues (who have perhaps being interviewing other experts concurrently) work out how to make the picture cohere. Formalise 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!)

Once a reasonable (though still imperfect) model is drafted, hold workshops for the expertsto review it. Take these in stages: static model first; then actions; then invariants and use-case specs.

Post parts of the model around the walls in a large room. Begin with a general presentation about types, associations and actions, so that they 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 agreement is more or less (though not entirely) reached

Pattern-18 Creating a Common Business Model

Summary

Rather than building a business model first, a common model is created from several components in the business, and from the definitions of the interfaces.

Objectives

Deliverable: a business model.

Context

There is a variety of components, each with its own model. They have been developed separately. You need to connect them toegether — whether through a 'live' interface, or just by enabling files to be written by one and read by another.

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 and export and translation modules have to begin by building a model of document structure.

It is also a common problem in large organisations: each department has bought its own software, and the IS department has the task of making them talk together more coherently. (See Section 11.7, "Heterogenous components," on page 472). The problem is particularly acute when two enterprises merge, and they each have their different systems talking different languages.

Finally, the most high-profile cases are those which involve communication across the boundaries of organisations: from banks to the trade exchanges; from one phone company to another; or between airlines and ticket vendors.

And the important thing is that we are not dealing with a model of any of the programs that communicate: it 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 the most powerful player doesn't just go ahead and do things their own way.
- Second, accept that there will never be a standard model that all work 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 (Section 7.1.5, "Model abstraction," on page 269) 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 book-library will understand the concept of the aquisition date of a book, even though most 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 the lack of it, when it expects it.

Pattern-19 Choose a level of abstraction

Summary Agree with your colleagues how much detail you're dealing with at each stage.

Motivation We have seen how 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:

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

Strategy Use the abstraction and refinement trechniques (Chapter 7, *Abstraction, Refinement, and Testing* (p.265)) to explore the levels more abstract and more detailed than you have. For example, once you have identified some actions, think "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; and make sure that you do so with sufficient precision to be aware what 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.

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

Begin by sketching the different layers — decide in advance how much time going to 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.

15.2 *Modeling patterns*

The preceding patterns have been about planning the development process.

These next few are more in the conventional analysis and design style, relating to the construction of a business model.

There are many good sources on patterns appropriate to this modeling (for example [Fowler 97b] and [PLoP]). Our main concern here is to point out what we see as the most essential ones.

Pattern-20 Type model = glossary

Summary Every term used within the field should appear in the model.

Objectives Ensure the type model fully represents the business domain.

Motivation 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 summarise complex relationships readily.

On the other hand, the alphabetical dictionary is more easy to look up.

Strategy Therefore model the business by drawing the glossary explanations in pictures:

- Ensure every concept written in documents that describe the business, or uttered by experts in the business, is represented somewhere. Generally, nouns –> object types; verbs –> actions.
- Remember you are modeling the business — not writing a database schema or program; and not modeling purely physical relationships. The associations are attributes drawn in pictures — 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 as an invariant, how it relates to the other terms. For example:

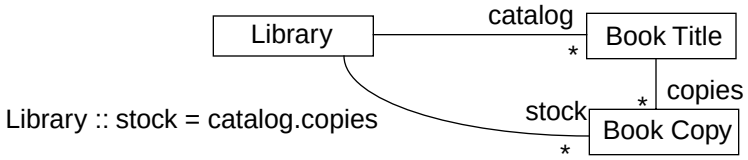


Figure 270: Business model with convenient terms and invariants

- A level of detail you would have to optimise in an implementation is OK in a business model. For example: ome 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 Book Copy; but we want to model the world as directly as possible, so 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 in time, 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 these things are. 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 6, *Effective Documentation* (p.237)).

Pattern-21 Separation of concepts — Normalisation

Summary

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

Motivation

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

- In business and requirements modeling, the objective is not to design a database. Therefore we do not need to be so strict about normalisation; and we need not distinguish attributes from associations.
- Redundant links and associations are OK.
- 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 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 take over some of the techniques from E-R modeling, to improve a model.

Strategy

Therefore, look for these triggers in your model:

- For a type with many associations and attributes, consider whether it should be split up into several associated types.

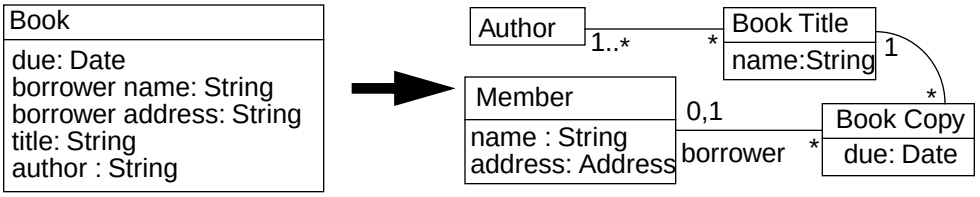


Figure 271: Splitting types

- For any association, but particularly many-many associations, consider modeling it as a separate type.

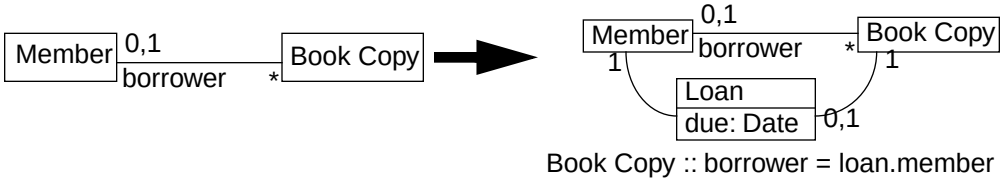
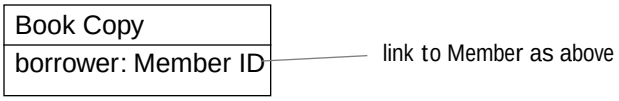


Figure 272: Reifying associations

- Object identifiers, names, keys, tags, and similar attributes: use associations.



Pattern-22 Items and Descriptors

Summary Distinguish things from descriptions of things.

Motivation 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 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 a Member borrows a book, they borrow a specific Copy; if they reserve a book, they're actually reserving a Title, and don't usually care what copy they get. There are many Copies for one Title. Titles have attributes like 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, while 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 a more 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 run time. 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 comes about where we need to model a system in which the users may decide, while the system is running, that they need essentially new classes of item, with new attributes, business rules, and actions.

Few systems genuinely allow end-users to make arbitrary extensions; those that do need to provide a programming or scripting language of some sort. Normally there is some restriction — the additions all fit within some framework. An example is a CAD (computer aided design) system, in which new kinds of mechanical part can be added, and the users can write operations that extend the code. Another is a work-flow system, in which users can define new kinds of work object, and their flows through the system.

In these cases, a framework can be defined (Chapter 10, *Model Frameworks and Template Packages* (p.389)) that imposes global constraints. The scripting language can be defined separately.

Pattern-23 Generalise and specialise

Summary

Use subtyping and model frameworks to simplify and generalise the model.

Objectives

Make the model applicable to a wider range of cases; provide insights that will help broaden the scope of the business, and/or make it easier to understand.

Motivation

A supertype describes what is common to a family of types. A new variant can be created just by defining what is different in the new type. (See Chapter 4, *Behavior Models — Object Types and Operations* (p.129)).

A model framework describes a family of groups of types. A collaborating group of types can be created by parameterising the template. (See Chapter 10, *Model Frameworks and Template Packages* (p.389)).

The effort of generalisation, 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 recast the model to expose them. Furthermore, the result is more easily extensible.

Strategy

Therefore,

- 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, and then rewrite them as applications of the framework.

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, where invariants and postocnditions frequently need to talk about the union of the target sets: form a supertype for the targets.
- Invariants and postconditions focused on type A only use some subset of the attributes of B, an attribute (or association) of A: put the unused attributes in a subtype of B. (Reduces spurious dependency.)

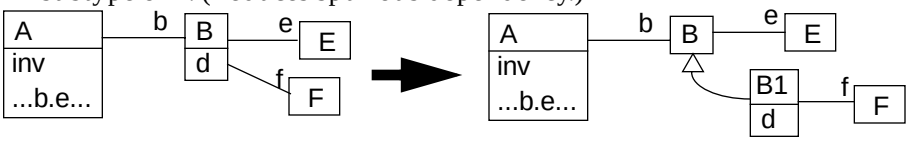


Figure 273: Introducing subtypes to decouple

- Similar 'shapes' in different parts of the type diagram and action diagrams: form a model framework that can be applied to recreate the collaborating groups.

Pattern-24 Recursive composite

Summary Model an extensible object structure with a recursive type diagram.

Strategy This is a general framework for modeling extensible structures:

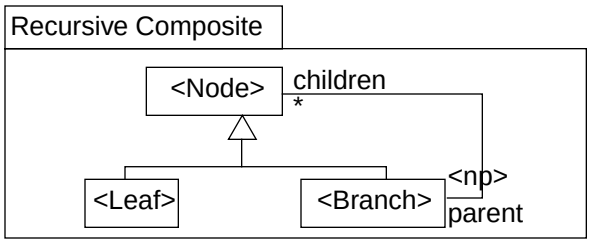


Figure 274: Template for recursive composite

For a tree, set np to be 0,1; for a directed graph (with shared children) set np to *.

An issue to decide is whether you want to permit loops in the instance snapshots. If not, import instead the no-loops package:

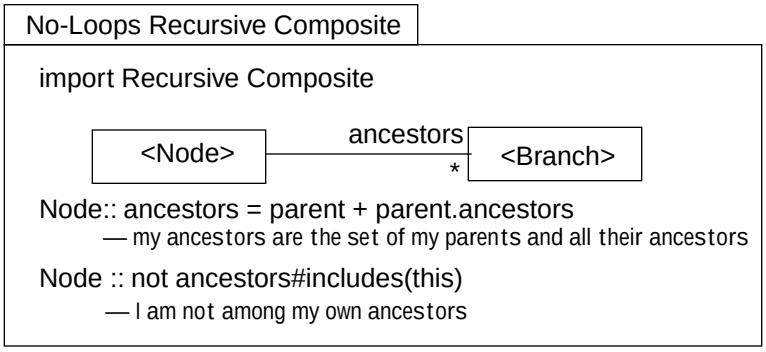


Figure 275: Recursive composite without instance loops

Examples np=1, no loops:

- Programming languages: some statements (if, while, blocks) contain other statements; some 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.

np=2, no loops:

- Family trees.

np=*, no loops:

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

np=*, loops allowed:

- Road maps
- Program flowcharts

Pattern-25 Invariants from association loops

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

Objectives Flush out useful constraints.

Context Building a static type model (as part of a business model or component spec).

Motivation A static invariant is a condition that should always be true — at least in 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 (is that the same object as this).

As a comparison, each term of an invariant must take two values of the same type (or at least with a common supertype). So an invariant can only be constructed when there is a loop[1] in the static type model: two different ways of coming at items of the same type.

For example, a Library's books may only be lent to its own members. We could write:

```
LoanItem :: borrower != nil => borrower.library = this.library
```

We can see the loop easily:

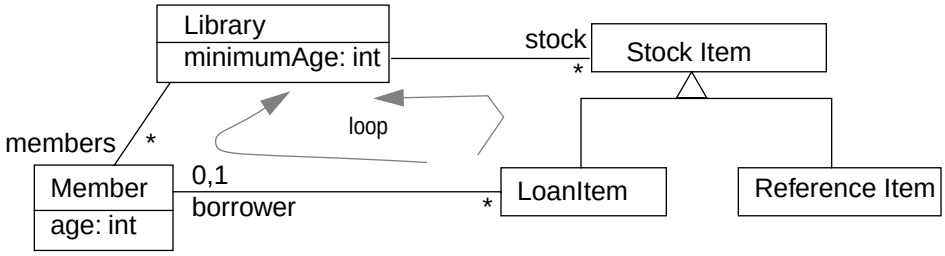
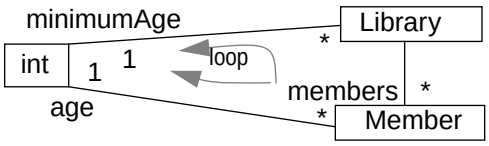


Figure 276: Association loops need invariants

Here's another: Member :: age >= library.minimumAge



So where's the loop? Well, we said that attributes and associations are interchangeable in analysis. We could have drawn int as a type in its own box, like this:

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

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

1. Not refering to loops in instance snapshots here, as in "Recursive composite" on page 566

15.3 *Video Case Study[1] — Abstract Business Model*

15.3.1 Informal description of problem

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

Members can reserve videos 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 up to three days; after which the reservation will be cancelled if not claimed.

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

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

Each store keeps copies of a subset of the catalogue 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."

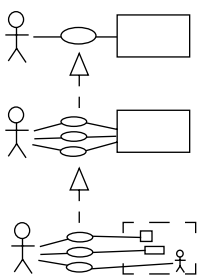
15.3.2 Formalized model

We want to make the requirements precise, across subject areas

The aim is to get a more precise statement of the requirements. We will segment the requirements into different overlapping areas of concern, or subject areas, but focus in this description on the primary subject area of customer rentals and sales. There could be other subject areas for things like marketing, purchasing, collections, etc.; they would define overlapping types and attributes, and might even add further specifications to common actions.

Object and action abstraction at work

There will be 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, etc. 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 stock-keeper, and the manager — an internal view. We will illustrate two levels of action abstraction in this chapter; the object granularity will be refined later, when we have to define the system context and user roles.



Subject Area — The Customer-Business relationship

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

1. 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 them.

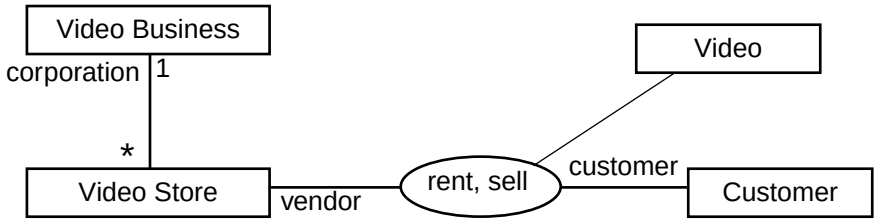


Figure 277: High-level business collaboration

This type model summarizes the interactions between Stores and Customers. The elements on the diagram are:

Object types — The boxes represent types of objects. There may 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 (though different Stores may have different Businesses). The * shows that there may be any number of Video Stores which share a single Video Business as their corporation.

Use cases — The ellipse represents a use case action called 'sell'. (Actually, since 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 involve one Video Store, one Customer, and one Video. (If there were several of any participant, we could use cardinality decorations.) We have not distinguished action participants from parameters visually here.

Use case recap

An occurrence of a use case action is some 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, and will actually, when we look at it more closely, be composed of smaller actions (like 'scan shelves', 'choose', 'pay', 'take away').

15.3.3 Dictionary

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

Video Business — A company with branches through which it sells and rents 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 which 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 some money.

rent(VideoStore, Customer, Video)

The interaction between a Video Store and a Customer whereby the Customer is given possession of an instance of a particular Video owned by the Video Business and normally kept at the store, for some period of time, in return for a payment to the Store.

15.3.4 Use case actions — precise specs

Make use cases precise

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.

Introduce attributes

So what exactly do we mean by 'the stock of videos owned by the vendor'? Let's augment the model to include this, then rewrite this statement more succinctly.

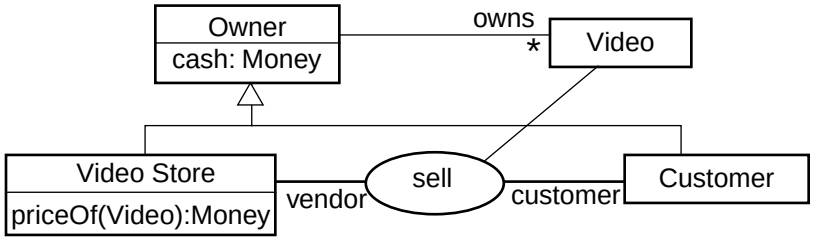


Figure 278: Revised, improved business collaboration for 'sell'

VideoStores and Customers are both 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.

Attributes vs. links

An attribute is, like a link, a read-only query. The main difference is that it is shown as text rather than pictorially. The two forms are in theory 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)
```

No sequence in 'post'

Every postcondition is a predicate (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 sub-expression referred to as 'expression@pre'

'+=' abbreviations

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

Extensible 'built-in's

There are many uses for operations on sets in abstract descriptions. Since the traditional mathematical symbols are not available on most keyboards, OCL defines ascii equivalents; it is easier to use + for union, * for intersection, – for set difference (Section 3.4.5, "Collections," on page 111), and in Catalysis these features can be extended by the modeler.

15.3.5 Preconditions and more modeling

Complete 'rent' use case

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 like hiring and returning.

Need model of watching videos

There's a slight conceptual difficulty here. Although there's an obvious transfer of money from one participant to another, what you've got for it once you've handed the video back 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 why we shouldn't model this abstract notion as a Past Rental, and model the rent use case as adding to your list of them.

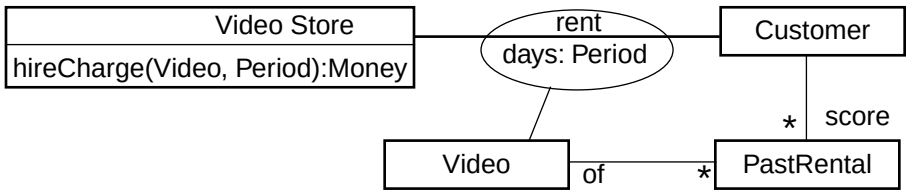


Figure 279: Model for 'rent'

Multiple appearances in little diagrams

We've already said in another part of the document, that VideoStores and Customers are kinds of Owner, which have cash and own Videos; so it is unnecessary to repeat it here. Each diagram is in effect a set of assertions about the elements that appear in it; if an element appears in several diagrams, then all of their assertions apply to it. Rather than show all of the parameters pictorially by linking the use case to the types, they can be shown as attributes within the use case ellipse.

One further complication is the 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:

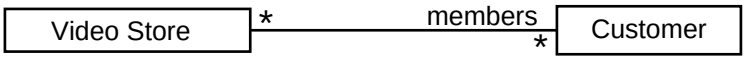


Figure 280: Incorporating the term 'members'

Final spec

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(v, period)@pre
         -- and the customer's assets are depleted by the same amount:
    and cust.cash -= hirer.hireCharge(v, period)@pre
    )
```

It does not cover all cases

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 just saying nothing about what that might mean. Since we're describing 'rent' and not 'attempt to rent' or 'enquire about renting', we leave it to some later more detailed description to define precisely what happens when a non-member 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.

The construction 'Type.new [predicate]' means a member of Type which didn't appear anywhere beforehand, and conforming to the predicate. The use of @pre in relation to the hireCharge allows for a possible change in the price list during the period of the hire.

Deferring detail with parameterized attributes

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.

Use Case refinement on 'rent'

Deciding we're interested in looking into one of the above 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:

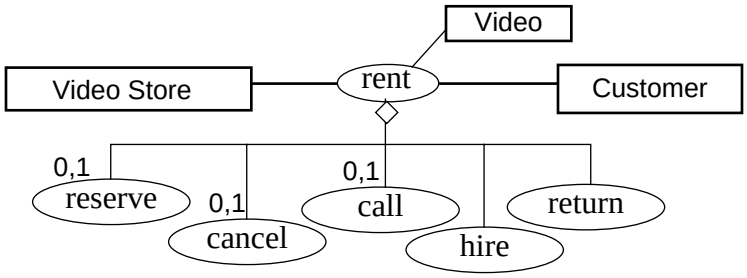


Figure 281: Refinement of 'rent'

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 just a summary diagram.

This need not be the only way to rent

There may be other actions not mentioned here which 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 will generally intersect with, or at least be mapped to, the participants of the abstract use case.

The relevant part of the informal business description is:

> *Rq 3* *Members can reserve videos 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 up to three days; after which the reservation will be cancelled if not claimed.*

Show the states of an abstract rental use case

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

Notation

Round-cornered boxes are states, which may have substates; black dots are starting points. Arcs are state changes, labelled with the actions that cause them. The square brackets are guards: the transition happens only if the guard is true. / marks a post-condition. 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.

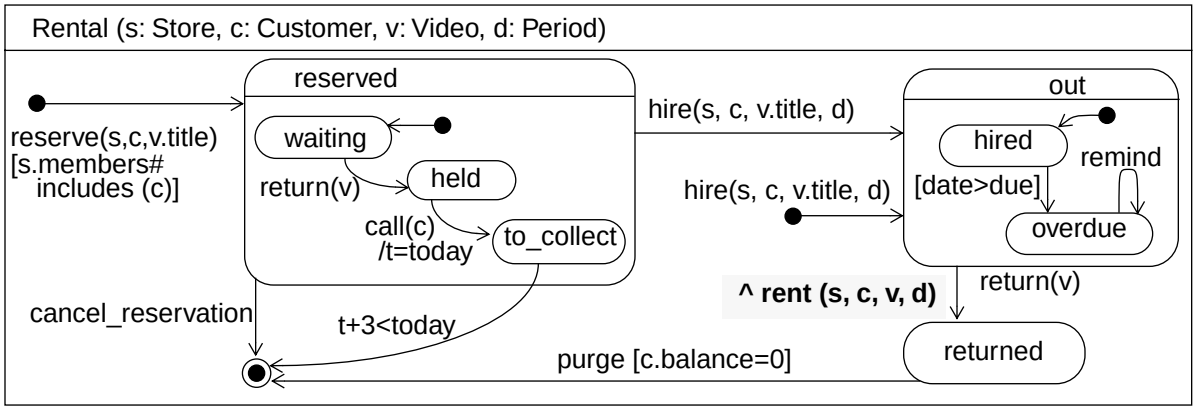


Figure 282: Refinement sequence for ‘rent’

15.4.1 Refined model

Specs of detailed actions

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 catalogue-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 cancelled.

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. Only makes sense 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.

The distinction between Videos (individual copies) and VideoTitles which has now become apparent gets documented, together with an invariant:

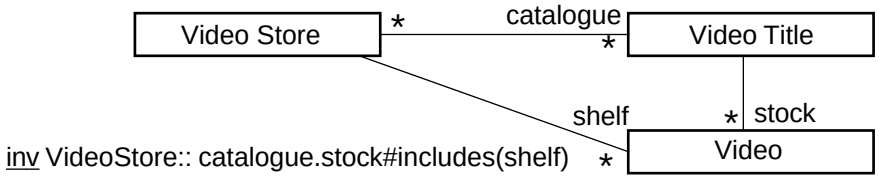


Figure 283: Clarifying video title and copy

VideoTitle The set of individual Videos having a particular title.

VideoStore::catalogue

The set of VideoTitles known to a VideoStore.

VideoStore::shelf

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

Refined use case diagram

We could also draw pictures showing the participants in each use case. This would just duplicate 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.

15.4.2 Formalized refined use case specs

Need to track the state of a 'rental'

Looking again at the spec of **reserve** above, it's clear that we must represent the idea of a reservation in the model (in order to be able to state that the use case creates a record of one). The same 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.

There's a useful pattern here, called *Action Reification* (see box). We'll reify the use case as Rental, but preserve the distinct idea of a Reservation as a state type:

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;
```

Pattern-26 Action Reification

Objectives: Systematic progression from succinct abstract actions (or use cases) to detailed dialog, supporting the strategic aim to expose most important decisions up front, and keeping reliable tracability to the details.

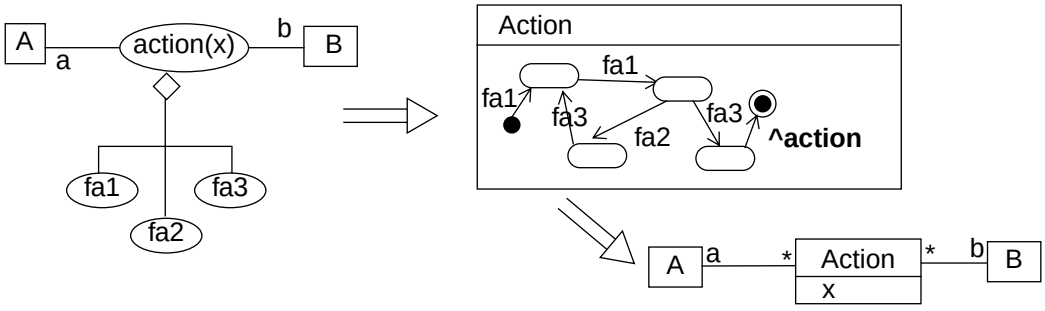
Context: A sequence of action-occurrences (not necessarily contiguous — there may be other unrelated actions in between them) can often be seen as a group with a single outcome — which can be documented with its own postcondition. Conversely, when specifying a system, detailed protocols of interactions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

Example: The transaction of obtaining cash, between a customer and a bank 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, etc. — 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*. There may be many possible refinements of one action.

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 keys at the ATM 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 statechart.

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



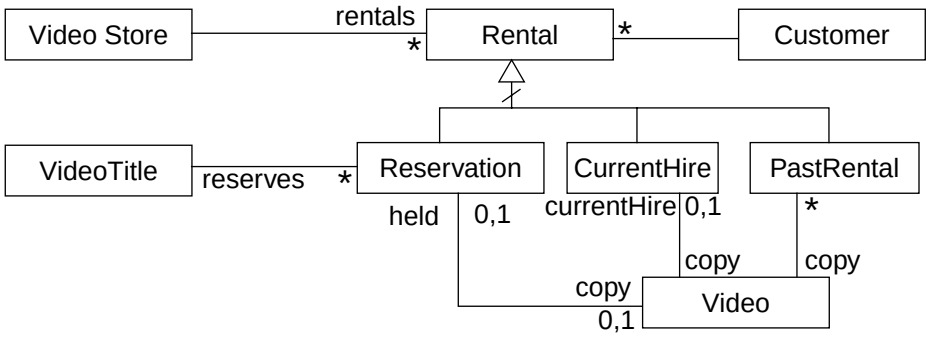


Figure 284: Using 'state-types' for a rental

No software objects — as yet

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 an abstraction representing the agreement between the parties.

Relate states to attributes

Notice also that the postcondition restates (in more detail perhaps) what is shown in the statechart — given an interpretation of the states in terms of the model. This reminds us that we should never write a statechart 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 substate: 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
```

Other use cases

Continuing with the other use case specs, return can be specified separately for its two effects on the statechart (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
```

Splitting out effects

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

```
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)
```

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

```
use case cancel_reservation (r:Reservation)
pre:
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.