Chapter 1 A Tour of Catalysis

As a software professional you have just been charged with the following task:

> *Build an application for a seminar company. Clients call and request courses, and the company says yes or no based on the availability of qualified instructors. Instructor qualification is based on exams instructors take and results from the courses they teach. Your solution must integrate with the existing calendar package (which is currently used for vacation planning) and the database of clients.*

Where do you begin attacking this problem? What objects should you have? How should you use the ready-made components? When are you finished?

This chapter takes you through the principal stages of a Catalysis development, covering the main features. We recommend that if you read a single chapter of this book, this should be the one. For a greatly abbreviated sound-bite version of the tour, see Section 1.15, Summary.

The rest of the book deepens and generalizes the ideas developed in this chapter so that they are applicable to a wide variety of problems. In reading the rest of the book, if you feel lost you can reorient yourself by coming back to this chapter.

1.1 *Objects and Actions*

Object-oriented development (OOD) bases the software structure on a model of the users' world within which the software will work. One benefit is that when the users talk in their own vocabulary about changes in their requirements, it is easier to see which parts of the software are relevant. The mirroring is not always exact—because of the constraints of platform, performance, and generalization—but if the differences are localized and clearly documented, the benefits of OOD are not lost.

Object-oriented analysis and design therefore use the same basic concepts to describe both the users' domain and the software. In Catalysis, these basic concepts are the *object*, representing a cluster of information and functionality; and the *action*, representing anything that happens: an event, task, job, message, change of state, interaction, or activity (see Figure 1.1). Catalysis places the action on an equal footing with the object, because good decoupled design requires careful thought about what actions occur and what they achieve.

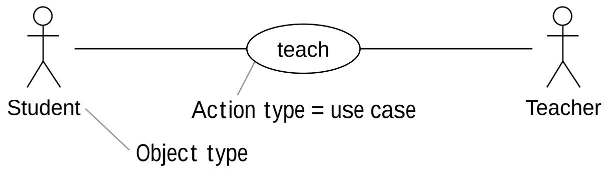


Figure 1.1 Objects participate in actions.

This analysis is done separately from focus on any one object. (Diagrams such as this one help make clear the principal relationships in a description but are insufficient in themselves. They should always be accompanied by an explanation, in a separate dictionary or embedded in narrative, of what the elements represent.)

Unlike some object-oriented design methods, Catalysis does not always begin by assigning responsibility for actions to specific objects. We believe in not taking decisions all at once. We first state what happens; then we state which object is responsible for doing it and which one is responsible for initiating it; and finally we state how it is done.[1]

1.1.1 Actions Affect Objects

Not only do objects participate in actions, but they are also used to describe the actions' effects on the participants.[2] Actions are characterized primarily by what they achieve and only secondarily by how they achieve it; there might be many different ways. For example, we might say

```
action (student, teacher) :: teach (skill)
post     -- this skill has been added to the student's accomplishments
```

This description uses new terms to describe the effect, or *postcondition*. It implies that every Student has a set of Skills called her accomplishments. We can draw this relationship or can write it textually as an attribute of Student (see Figure 1.2).

The stars indicate that every Student can have any number of accomplishments, and every Skill can be the accomplishment of any number of Students. The stick figure and the box represent types of objects; the use of the stick figure instead of a box is optional and highlights the expectation that Student may be one of many roles played by any one object.

To visualize an occurrence of an action, we can use a *snapshot*.[3] Figure 1.3 shows sample instances of objects in two states: immediately before and after an example occurrence of the action. In this book, the alterations in the "after" state are shown in bold; you might prefer to draw them in two colors.

1. Details on refinement are in Chapter 6.
2. Defining actions is covered in Chapter 4 and specifying actions in Chapter 3 and Chapter 4.
3. Snapshots are described in detail in Section 2.2.

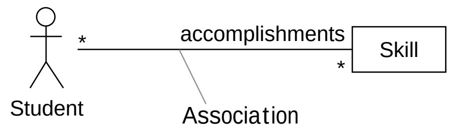


Figure 1.2 Participants have associations.

The associations may represent real-world relationships: if we asked Jo some deep question about lettuce curling, she should now know the answer. Or associations can represent software: there is now a row in a database table saying that Jo has completed the lettuce-curling course, or Jo's name is in a record somewhere. The useful aspect of associations is that we don't have to say exactly how they are realized, but we can still make meaningful statements about how actions affect them.

This use of the associations, or a *static model*, to provide a vocabulary for the actions, or a *dynamic model*, gives a clear guide as to what objects you need: those that are required to describe the actions.

1.1.2 Precise Specifications

Associations provide a vocabulary in which it is possible to describe the effects of actions as precisely as in programming language:

```
action (student, teacher) :: teach (skill)
post     -- this skill has been added to the student's accomplishments
         student.accomplishments = student.accomplishments@pre + skill
```

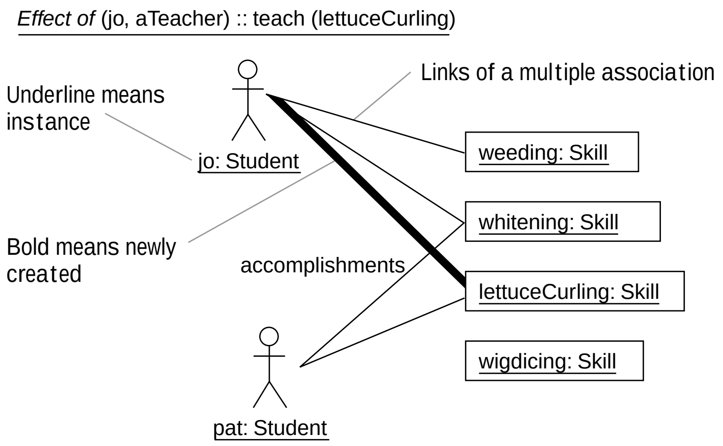


Figure 1.3 Action occurrence and snapshots.

A *postcondition* is a relationship between the before and after states; @pre refers to the before state.

Postconditions allow us to make precise statements about what happens in a business or what is required of a piece of software even though we are working at an abstract level. Unlike natural-language requirements documents, these specifications are abstract and yet precise. Experience shows that the effort of writing them exposes inconsistencies that would be glossed over in natural language. Although there is extra effort involved, it saves work further down the line and focuses attention on the important issues.

1.2 *Refinement: Objects and Actions at Different Scales*

Figure 1.4 shows a picture of some interacting . . . let's just call them "things" for a moment. Each one has a set of tasks it can perform, and each one performs its tasks in part by making requests to others. You can look at the picture in different ways and at different scales.[4] The picture might represent the seminar business, the boxes representing departments interacting to achieve the corporate goals. We could look inside any department and find the same sort of picture, with different people sending one another memos and documents and exchanging phone calls. Some of the actors in the picture might be pieces of software such as the scheduling system; we could zoom in on one of these boxes and find the same kind of picture again, showing major components such as the vacation planner.

Looking inside any component, we can identify the same kind of structure again: if it's an object-oriented program, the units will be (we hope) neatly encapsulated objects. Each object has a set of tasks it performs and data that it uses to perform these tasks—just as, at

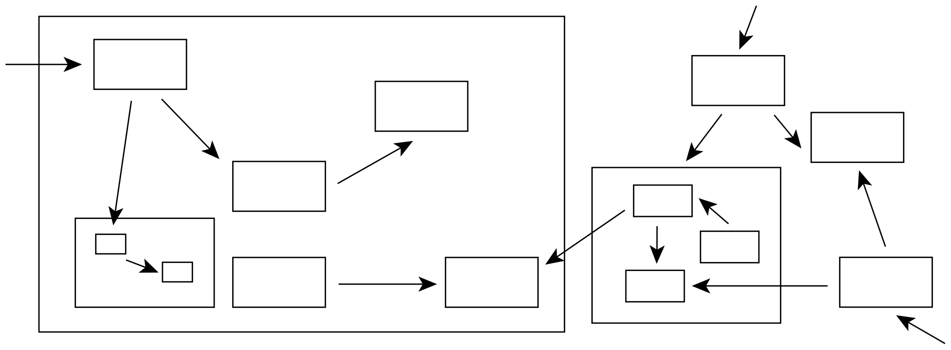


Figure 1.4 Objects at all scales interact.

the larger scale, the people have their jobs to perform, and their filing cabinets and Internet browser shortcuts are full of (we hope) relevant information.

4. For more than you ever wanted to know about refinement, see Chapter 6.

We consider the essential issues in design to be the same at all scales, with some differences only in detail. To this fractal[5] picture, we apply a fractal method. In this book, we call all these units *objects:* business departments, machines, running software components, programming language objects. The interactions between them are called *actions*, a term that encompasses big business deals, phone calls, bike rides, file transfers, electronic signals, taps on the shoulder, function calls, and message sends in an object-oriented programming language. Like objects, actions contain smaller actions and are parts of bigger ones.

1.2.1 Actions at Different Scales

The seminar system is part of a larger organization, some aspects of which we must understand before we can design it. The company teaches courses to clients. We show the interaction as an ellipse (see Figure 1.5). We can *zoom in* on the action to show a more detailed picture, as in Figure 1.6. We can draw another diagram to relate together the two levels of description, stating that the teach action is a composition made up of the smaller ones. Or we can overlap them on the same diagram (see Figure 1.7).

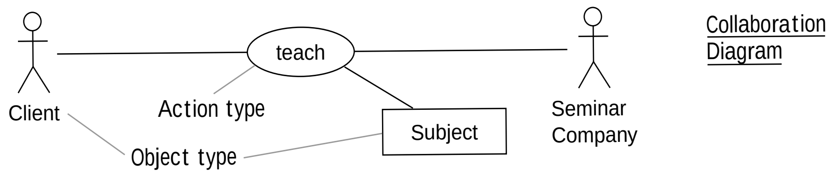


Figure 1.5 An abstract action.

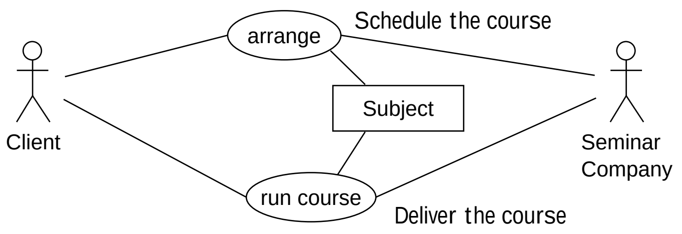


Figure 1.6 Zoomed-in actions that constitute teach.

We haven't detailed here how the composition works. The smaller actions might happen in parallel or in sequence, or they might be repeated: we give that detail separately. One way to do that is with a *sequence diagram*. This is used to illustrate a typical occurrence of an action (see Figure 1.8). The horizontal bars in these diagrams are occurrences of actions; the vertical bars are instances of objects.

5. A fractal picture has the same appearance at all scales.

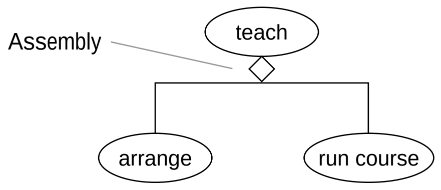


Figure 1.7 Showing action constituents.

By contrast, the ellipses in the previous diagrams are *action types* representing definitions of all the interactions that can take place. The actors they link are *types* of object: each type describes a role that specific instances can play. (We don't always put arrows on the actions at this stage, because each action may represent a sequence of smaller interactions between the participants; there may be several alternative sequences, each of which has a different initiator. We'll see shortly how we distinguish who does the teaching and who gets taught.)

Notice that the different levels of detail correspond to the way we normally account for things in everyday life. You might say, "I had a Java course last week from SeminoMax" to a friend who doesn't care when or whether you arranged it in advance, or whether they grabbed you off the street. But such detail is appropriate when you work out exactly how to become taught.

Of course, the arrange or the run_course can be further detailed into finer and finer actions, some of which might involve or be inside computer software.

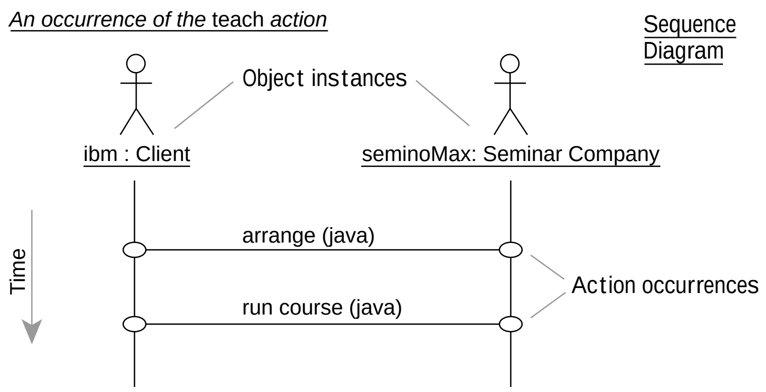


Figure 1.8 Sequence diagram.

1.2.2 Objects at Different Scales

The objects can also be detailed or abstracted. A seminar company has a Sales department and an Operations department; a Seminar System, which helps coordinate activities; and a portfolio of courses, each of which deals with a number of Subjects (see Figure 1.9). As before, boxes are the default representation of object types. We can use stick figures

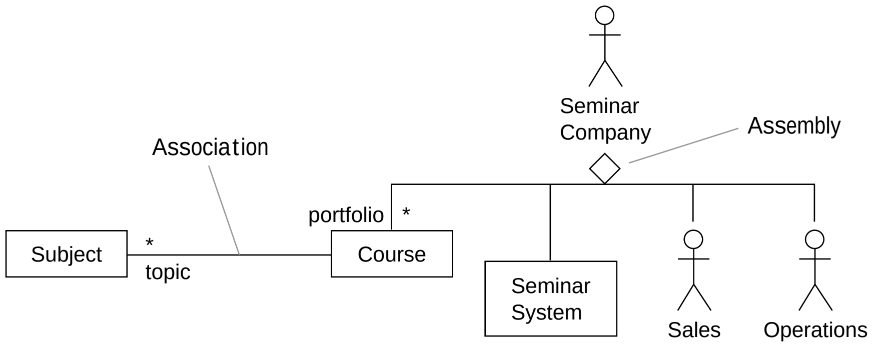


Figure 1.9 Showing object constituents.

instead to highlight roles; several roles can be played by one object. There is no implication that the stick figures represent individual people, although there is usually some human element involved.

The line from Course to Subject is an association; it represents the fact that you can ask about a Course, "What topics does it cover?" and the answer will be a set of Subjects. By implication, you can ask which Courses have any one Subject as their topic. The association says nothing about how the information is represented or how easy it is to obtain. Like objects and actions, associations can be refined, going into more detail about the information they represent.

At this level of refinement, we can show that the arrangement of courses is dealt with not only by the Seminar Company as a whole but more particularly by Sales, assisted by the Seminar System; and Operations run courses, also with the help of the Seminar System (see Figure 1.10).

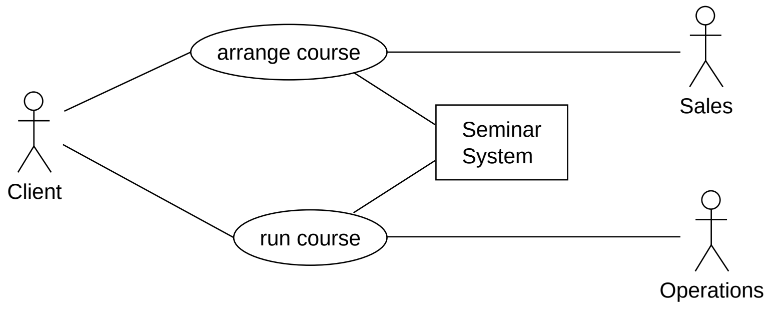


Figure 1.10 More-detailed description.

Normally we zoom actions and objects at the same time. Notice that at the coarser scale, you are not compromising the truth but only telling less of it. Everything you say in the big picture is still true when you look at the finer scale.

This technique of zooming in and out is useful when you're seeking to understand what a business does and why it does it. It is also useful for understanding the requirements on a software component separately from the internal design and for understanding the overall design separately from the fine detail. The ability to see not only the big picture but also how it relates to the fine detail is essential to coherent, traceable, maintainable design.

1.3 *Development Layers*

Abstraction is the most useful technique a developer can apply: being able to state the important aspects of a problem uncluttered by less-important detail.[6] It's equally important to be able to trace how the more-detailed picture relates to the abstraction. We've already seen some of the main abstraction techniques in *Catalysis*—the ability to treat a complex system as one object and to treat complex interactions as one action and yet state the outcome precisely. This approach contrasts with more-traditional design techniques in which *abstract* also tends to mean *fuzzy,* so you can't see whether a statement is right or wrong because it might have many different interpretations.

Different projects will use these abstraction techniques differently. Some project teams may work with a simple design that needs little transformation from concepts to code. Others, starting with an existing system and existing procedures for using it, may need to abstract its essentials before devising an updated solution.

A "vanilla" development from scratch is typically treated in two to five layers of abstraction:[7] bigger projects have more abstraction layers, each one separately maintained. Typical layers are as follows.

- *Business model (sometimes called domain or essential model):* Describes the users' world separately from any notion of the target software. This model is useful if you are building a range of software all in the same world or as a draft to transform into a requirements spec.
- *Requirements specification:* Describes what is required of software without reference to how it is implemented. This spec is mandatory for a major component or complete system. It may be approached differently when the complete system is created by rapidly plugging kit components together.[8]
- *Component design:* Describes on a high level how the major components collaborate, with references to a requirements specification for each one. It is needed if there are distinct major components.
- *Object design:* Describes how a component or system works, down to programming language classes or some other level that can be coded (or turned into code by a generator). It is mandatory.

6. For an overview of the development process, see Chapter 13.
7. There are many "routes" through the method—see Section 13.2.1.
8. Heterogenous components: see Section 10.11. Homogenous component kits: see Section 10.6.

- *Component kit architecture:* Describes the common elements of the way a collection of components work together, such as standard interaction protocols. It is needed to allow a variety of developers to build interoperable components that can be assembled into families of products.

1.4 *Business Modeling*

The examples we have looked at so far are typically part of a business model.[9] The main objective is for the model to represent how the users think of the world with which they work. A great many of the questions that arise during a project can be uncovered by good business modeling.

The construction of a business model is a subject for a book in itself. Typically, there is a cycle of reading existing material and interviewing domain experts.

- The starting questions are “What do you do?” and “Whom do you deal with?”
- Every time a verb is mentioned, draw an action and add to your list of questions “Who and what else participates in that? What is its effect on their state?” The answers to these questions lead you to draw object types and to write postconditions; these in turn lead to associations and attributes with which to illustrate the effects.
- Every time you introduce a new object type, add to the list of questions “What actions affect that? What actions does it affect?” Or ask it individually about the type’s associations and attributes.
- Go up and down the abstraction tree. Ask, “What are the steps in that action?” “What are the parts of that object?” “What does that form part of?” “Why is this done?”

Here’s an example.

ANALYST THINKS: *What is the postcondition of* (Client, SeminarCo) :: arrange (Subject)*?*
ANALYST SAYS: *What is the result when a client arranges a course with you?*

CLIENT: *We find a course that suits the subject they’re interested in. Then we must find an instructor qualified to teach that course. We assign him or her to do a run of the course for the client on a date the client is available. An instructor is available when not on vacation or doing another course.*

ANALYST: *Presumably it’s a rule of the business that instructors can teach only courses they’re qualified for; and an instructor can’t be on more than one holiday or assignment on the same day.*

ANALYST THINKS: *So we need to refer to instructor qualifications and their schedules, including vacations and assignments to course runs along with the dates, courses, and clients for any assignment (among other things).*

9. A more detailed discussion of business modeling is in Chapter 14.

1.4.1 Static Models and Invariants

This interview yields the static relationships and attributes[10] shown in Figure 1.11. (The analyst decided that CourseRun and Vacation are both kinds of InstructorOutage—that is, situations when that instructor is not available.)

These business rules can be written as *invariants:*

```
inv  -- for every CourseRun, its instructor's qualifications must include the course
     CourseRun :: instructor.qualifications -> includes (course)

inv  -- for any Instructor, and for any date you can think of, the number of the
        instructor's outages on that date is never more than 1
     Instructor :: d:Date :: outage[when=d] <= 1
```

An invariant, like a postcondition, is written by following the links from a given starting point. The invariants are described informally and are written in a simple language of Boolean conditions and set relationships. Again, their power lies in the ability they give you to write unambiguous statements about abstract descriptions.

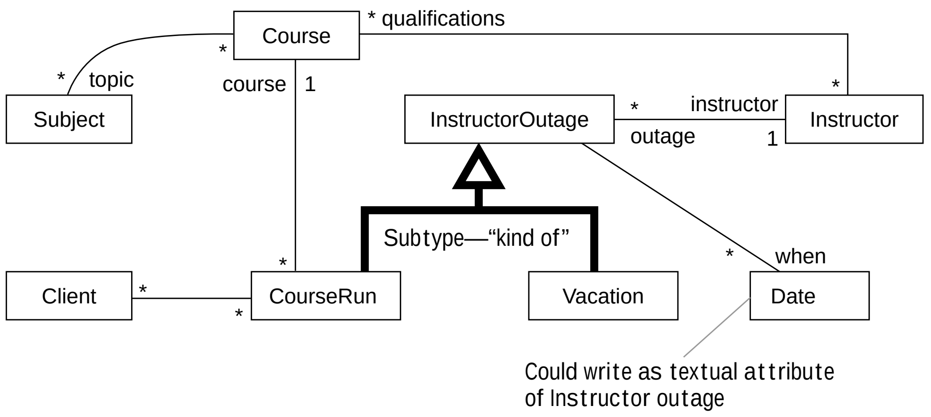


Figure 1.11 Static type model.

The action's postcondition can be written in terms of these static relationships. We can draw snapshots to help visualize the effect (see Figure 1.12). This might lead to some follow-up questions—"What actions affect the Instructor's qualifications? What other actions create an outage?"—and so on.

```
action (c: Client, s: SeminarCompany) :: arrange (t: Course, d: Date)
post       -- a new CourseRun is created with date, client, course
           r : CourseRun.new [date=d, client=c, course=t,
                       -- assigned instructor who was free on date d and qualified
                          for the course
```

10. Static invariants are detailed in Section 2.5; precise action specification is discussed in Section 3.4.

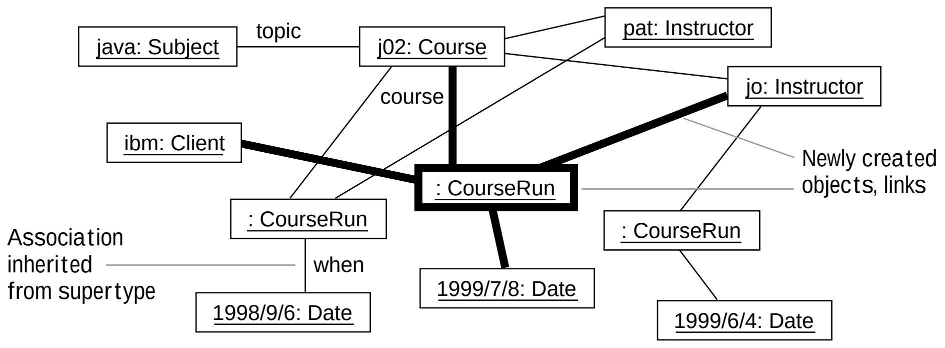


Figure 1.12 Action postconditions.

```
instructor.outage@pre[when=d] = 0 &
instructor.qualifications -> includes (t) ]
```

1.5 *Model Frameworks as Templates*

Some of the same patterns of relationships and constraints crop up frequently in modeling and design. In a package separate from our project-specific models, we can define a generic modeling framework for resource allocation; it acts as a macro-like template that can be applied in many places. A template can contain any of the modeling constructs in the form of both diagrams and text. Additionally, any name can be written as a <placeholder>, which will be substituted when it is applied to a model.[11]

The resource allocation constraints of the seminar company are quite common. They can be generalized as shown in Figure 1.13. Now we can re-create the original model by using the generic model for resource allocations. We can easily add allocation of, say, rooms (see Figure 1.14).

The framework is applied twice, with placeholder names substituted as indicated. Each substitution for <Resource> defines a derived type <Resource>_Use. The framework applications can be *unfolded* to reveal the complete model. The benefits of frameworks are that they simplify a picture, and, having been tried and tested, they often deal with matters you might not have thought of (such as the possibility of a Room being closed). The details and precise specifications have already been worked out for you.

Templates can also contain actions and can be used to generalize interaction protocols between components.

11. Such frameworks can be used in a great many ways; see Chapter 9.

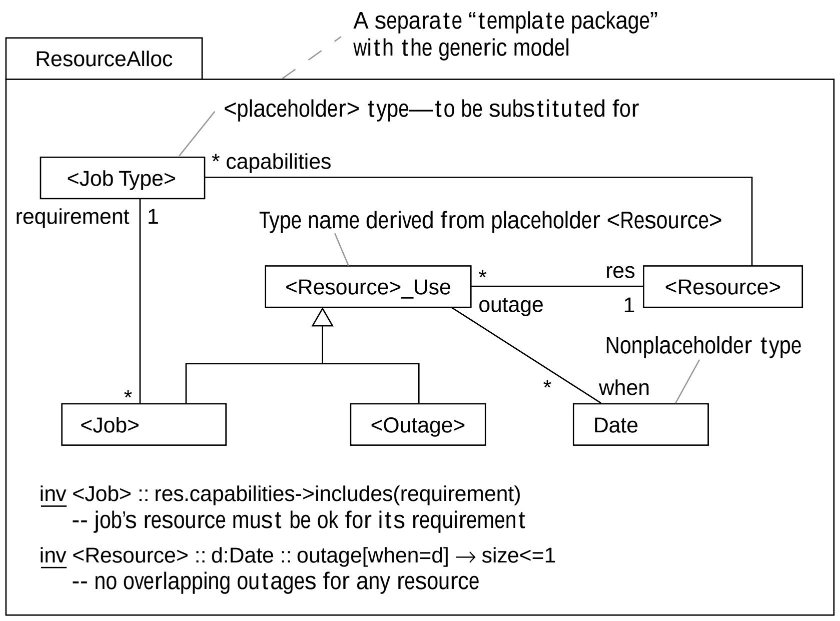

ResourceAlloc
A separate "template package" with the generic model
<placeholder> type—to be substituted for
<Job Type>
* capabilities
requirement 1
Type name derived from placeholder <Resource>
<Resource>_Use
* outage
res 1
<Resource>
Nonplaceholder type
* when
Date
*
<Job>
<Outage>
inv <Job> :: res.capabilities->includes(requirement)
-- job's resource must be ok for its requirement
inv <Resource> :: d:Date :: outage[when=d] → size<=1
-- no overlapping outages for any resource

Figure 1.13 Generic framework model of resource allocation.

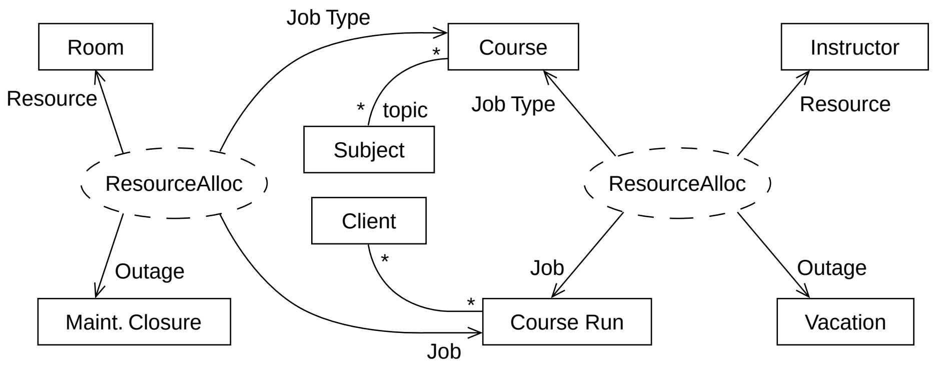


Figure 1.14 Applying the resource allocation framework.

1.6 *Zooming In on the Software: System Context*

The seminar company uses a software system[12] to run its courses. If we proceed with our successive refinement, we will get down to interactions between people and the Seminar

12. For information about describing a system's context and proceeding to specify it, see Chapter 15.

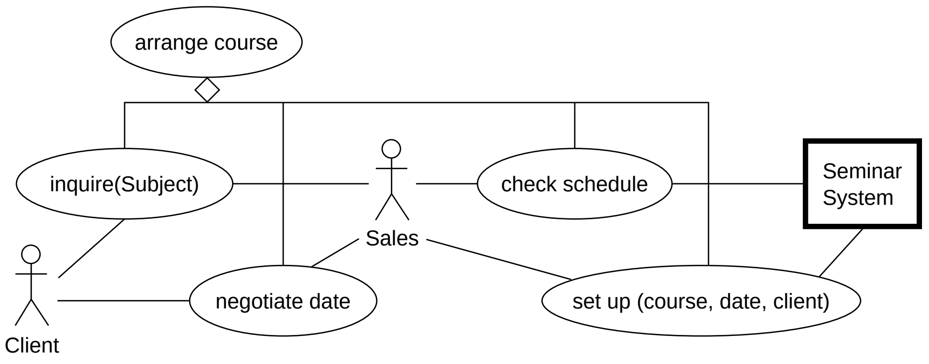


Figure 1.15 Zooming in to the software system boundary.

System, which we initially treat as one object (see Figure 1.15). Once again, we can draw a sequence diagram to illustrate how the refinement works (see Figure 1.16).

What other roles does the Seminar System play? The Operations department uses it to schedule the various tasks involved in running a course and to record the qualifications and availability of instructors. Also, the administrators perform various systems management and peformance tuning tasks on it (see Figure 1.17).

We can show separately how these objects and actions are part of what we've already seen. Some objects and actions, such as those associated with the system management tasks, may not be relevant to what we've seen at the abstract level (see Figure 1.18).

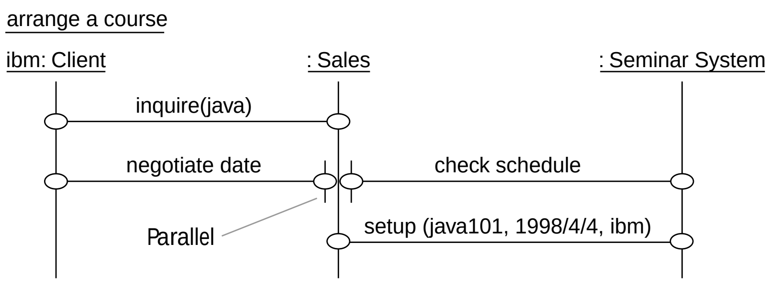


Figure 1.16 Sequence diagram, including the role of the software system.

So we can now see the actions in which a Seminar System takes part from the viewpoints of several different users. (Notice that even when we focus on the system we're proposing to design, we don't limit ourselves to the actions that the system is immediately involved with. We also show, for example, teach course, to emphasize that this is not among the responsibilities of this particular object but is something that will be taken care of by someone else.)

We can also trace the software requirements back through the assembly (or refinement) links up to the business goals: print materials is part of run course, which is part of teach.

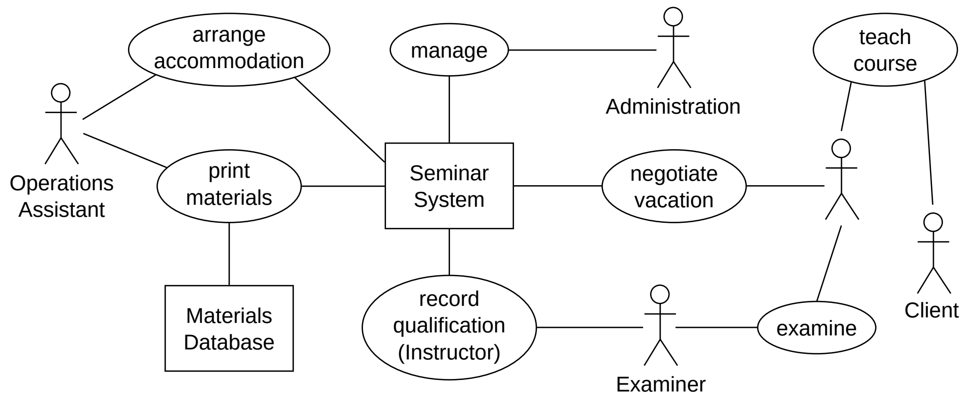


Figure 1.17 Other roles of the software system.

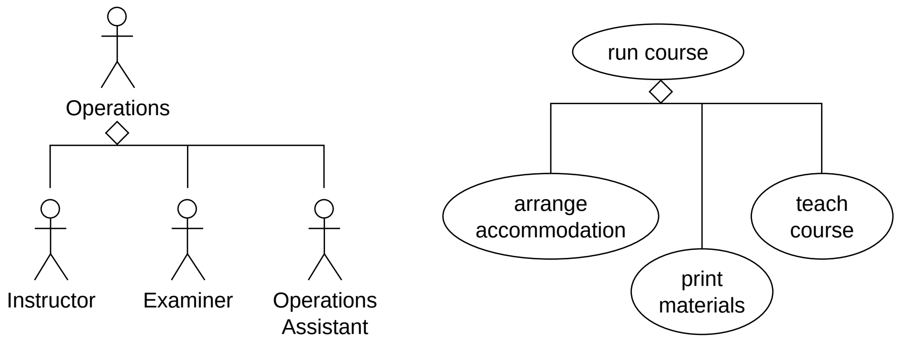


Figure 1.18 Object and action constituents.

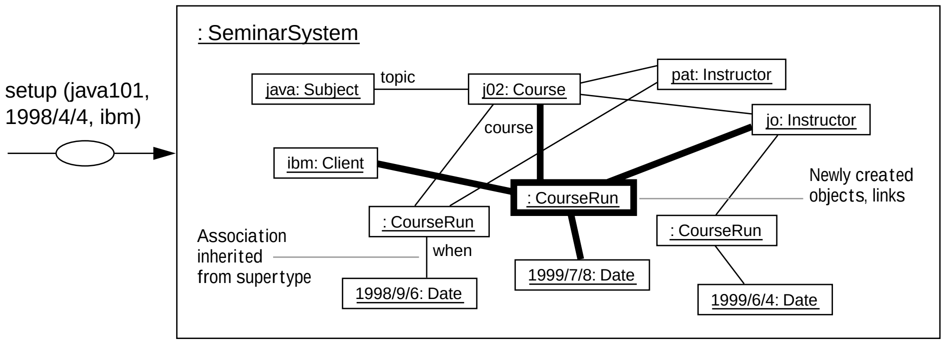
Presumably, we could go further up and see that teach is part of make_money or maybe satisfy_ego—who knows what motivates pedagogues?

1.7 *Requirements Specification Models*

Now that we know which actions we want our software to take part in, we have the option of making the descriptions more precise. We can describe the effect of each action on this system.

As in the business model, we can illustrate an effect of an action with a snapshot; but in this case, the objects in the diagram denote not the real objects but rather the system's own representation of them (see Figure 1.19). For that reason, we draw the objects inside the system boundary, whereas the actions themselves are still actions on the whole system. This reflects the fact that we are not yet designing the internals of the system.

A system requirement spec often reflects the business model closely, as in this case. Differences occur where the business model is general to several systems or where there are special mechanisms of interaction with the user apart from any domain concepts (for



action SeminarSystem :: setup (Course, Date, Client)
post A new CourseRun has been created, linked to the client, the date, the course, and any Instructor available on that date qualified to teach that course

Figure 1.19 Snapshots of software system state.

example, a word processor's clipboard has nothing to do with the basic model of the documents it manipulates).

Gathering all the specs for the actions the system is required to take part in and the static models needed to draw snapshots for those specs, we compile a formal functional requirements model. Notionally it looks like the drawing in Figure 1.20.

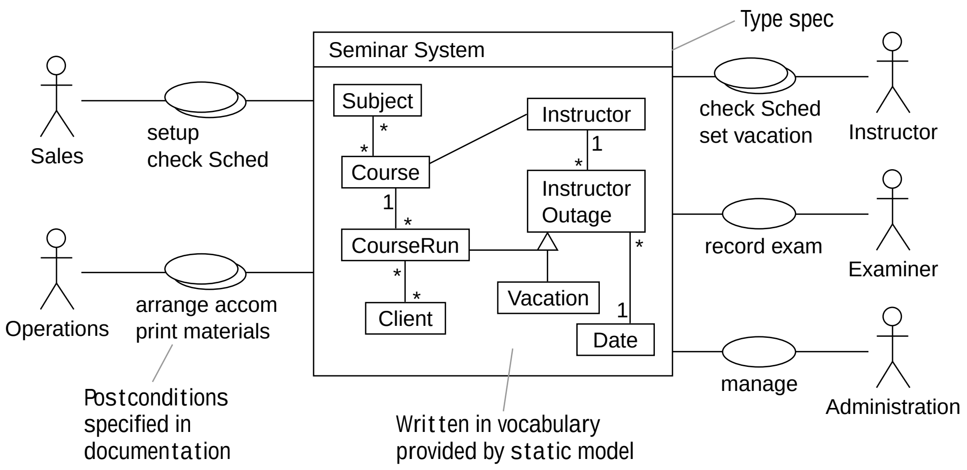


Figure 1.20 Requirements model—basis.

The static model is a hypothetical picture created for the purpose of explaining the system's externally visible behavior to its users. There is no absolute mandate on the designer to implement it exactly with classes and variables that directly mirror the types and associations in the spec. For one thing, there are usually complications such as managing a database, and there may be performance and decoupling issues. Also, there may be a layer of partitioning between different components. Nevertheless, it is preferred to keep to this model as much as possible to minimize the conceptual gap between the users and the implementation.

The principal difference between the requirements spec and the implementation is therefore that the latter defines how the objects inside the design collaborate to achieve the effects specified by the former.

1.7.1 Packaging

A *package* is a container for development artifacts; it is roughly like a section in a document or a directory in a file system.[13] A package contains some quantity of information, whether models, software, refinement relationships, or information about the structure of other groups of packages. Development work should be separated into packages. Typical packages might contain a component specification; a component implementation; a reusable collaboration framework; and a single type specification. Documentation is part of a package; document structure parallels the package structure.

A real requirements document[14] will be spread over many pages, covering action specifications, narrative, rationale, and so on. Different external views of the system might be partitioned across multiple packages, separated from packages that will reflect internal design decisions (see Figure 1.21). One package can import another: if one set of terms refers to another, in code or in models, you should import its package; if you want to say something to relate two other descriptions, use a package that imports them both.

It is often useful to manage dependencies on the package level, checking that the number of dependencies from and to any one package is not too large.

1.8 *Components*

The actions we've so far documented at the Seminar System boundary are abstract in two dimensions. First, they show us nothing of how they are implemented inside any of their participants, and in particular our system. Second, each of them still represents what will turn out to be a more detailed dialog; by refining in this dimension, we ultimately get down to menus, keystrokes, and mouse clicks. But let's take the implementation track and peer inside the system.

Our sample system consists of several major components.[15] They could be modules within a single program, or they could be running on different machines. There is a seminar scheduler and a separate vacation planner (and that is just the way we use a more general calendar program).

13. Chapter 7 says more about packaging.

14. For guidelines on how to structure documentation, see Chapter 5.

15. More discussion of component-based design is in Section 10.11 and Section 16.3. A broader discussion of component technology is in Chapter 10.

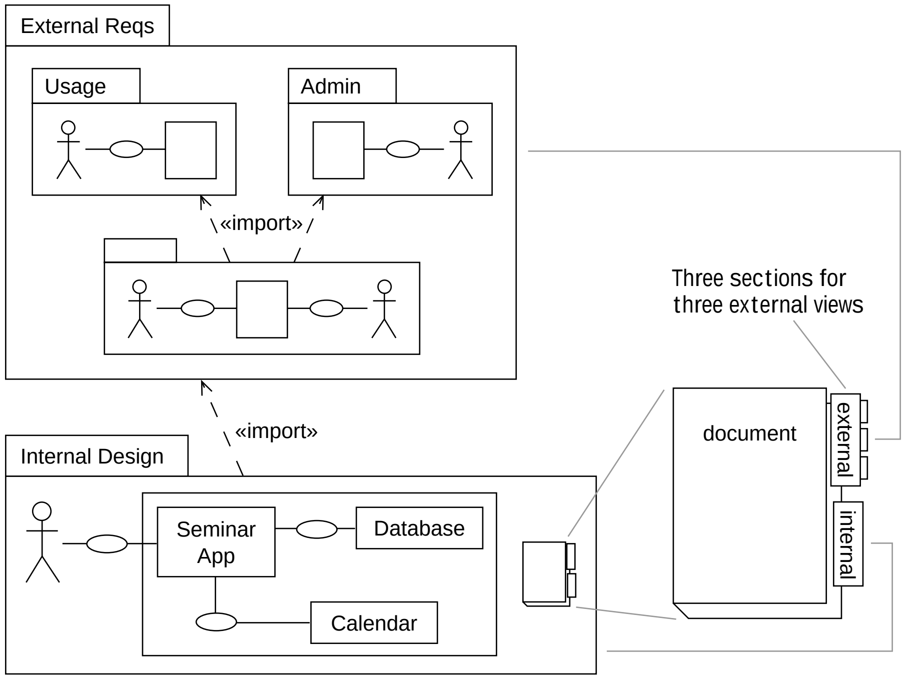


Figure 1.21 Multiple packages and document structure.

1.8.1 Two Versions of the System Design

Actually, there are two designs for this system. In version 1, the vacation calendar, the qualifications database, and the seminar planner are not significantly coupled in software: it's up to the Sales people to make sure that they don't schedule a course for an unsuitable or recuperating Instructor (see Figure 1.22).

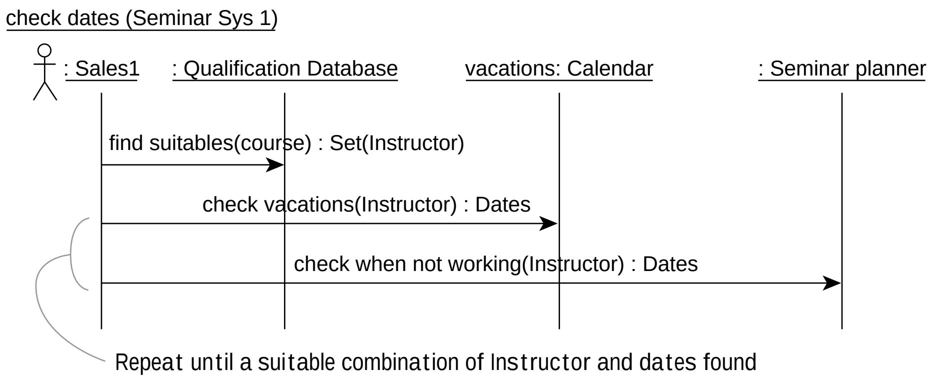

check dates (Seminar Sys 1)
: Sales1
: Qualification Database
vacations: Calendar
: Seminar planner
find suitables(course) : Set(Instructor)
check vacations(Instructor) : Dates
check when not working(Instructor) : Dates
Repeat until a suitable combination of Instructor and dates found

Figure 1.22 Sequence diagram including internal software components.

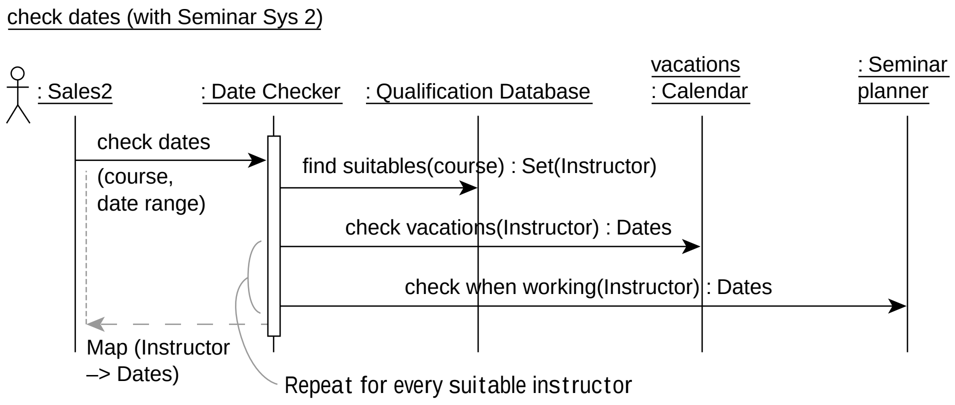

check dates (with Seminar Sys 2)
: Sales2
: Date Checker
: Qualification Database
vacations : Calendar
: Seminar planner
check dates
(course, date range)
find suitables(course) : Set(Instructor)
check vacations(Instructor) : Dates
check when working(Instructor) : Dates
Map (Instructor –> Dates)
Repeat for every suitable instructor

Figure 1.23 Alternative sequence diagram, design version 2.

Version 2 of the system does some of this work: it compiles a list of the available instructors within a range of dates (see Figure 1.23). We introduce a role called Date Checker, which might be combined with others in some object not yet decided, such as a Schedule; it may be useful to create such a role explicitly. Again, we can summarize the breakdown of actions and objects, as shown in Figure 1.24.

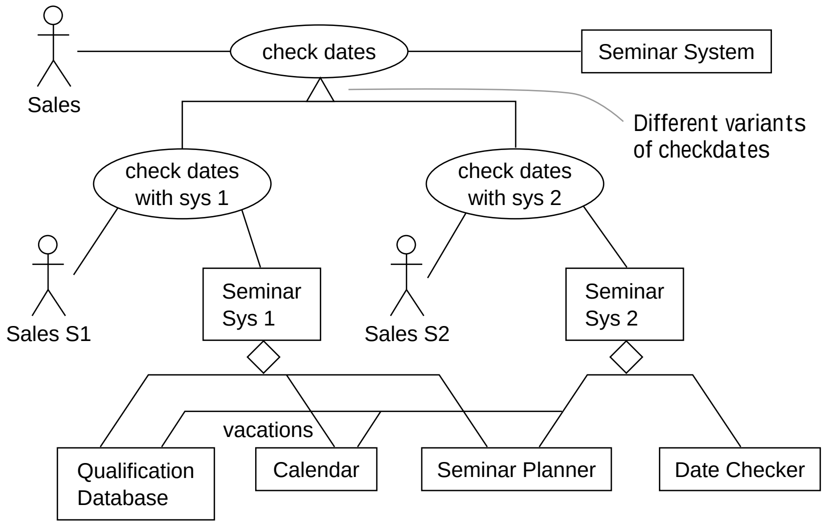


Figure 1.24 Summary of action and object constituents.

Not only are the two variants of the system different, but so are the users. Although the effects achieved by the overall action are equivalent, the details of using the two systems are different—human salespeople would need retraining, and mechanical ones would need reprogramming—so we give them different type names. Nevertheless, many of the System components are the same (that is, different instances of the same designs).

(For the sake of the illustration, we've cut a corner here. The different system variants imply different operating procedures throughout, so we should strictly go up to the topmost level of analysis at which the system was introduced and break it all down separately.)

1.8.2 Roles

We have begun to put arrows on the actions because they have definite initiators. At this stage we can also identify roles[16] that should be played by the same component—because they access the same information or their functions are closely related. The Date Checker, for example, might turn out to be a role of the same component that helps Sales staff set up courses[17] (see Figure 1.25).

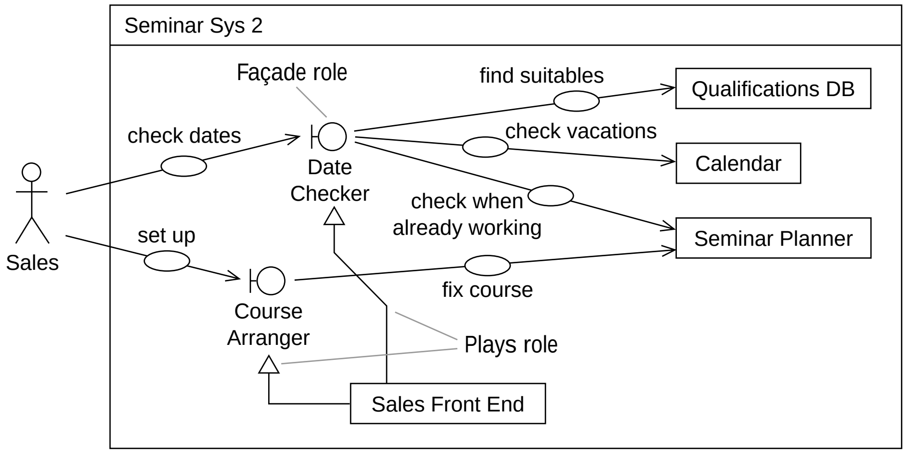


Figure 1.25 Combining roles in the design into Sales FrontEnd.

The façade symbol denotes an object type that we could draw with a plain box. The symbol highlights the nature of this object's role as an interface between the central components of the system and the users.

1.8.3 Partitioning the Model between Components

Each of the components[18] performs only some of the system's functions and includes only part of its state, which we can see by drawing the static models (see Figure 1.26).

16. Façades and other interface issues are discussed in Section 6.6.4.

17. Chapter 8 explains composition of roles (and other descriptions).

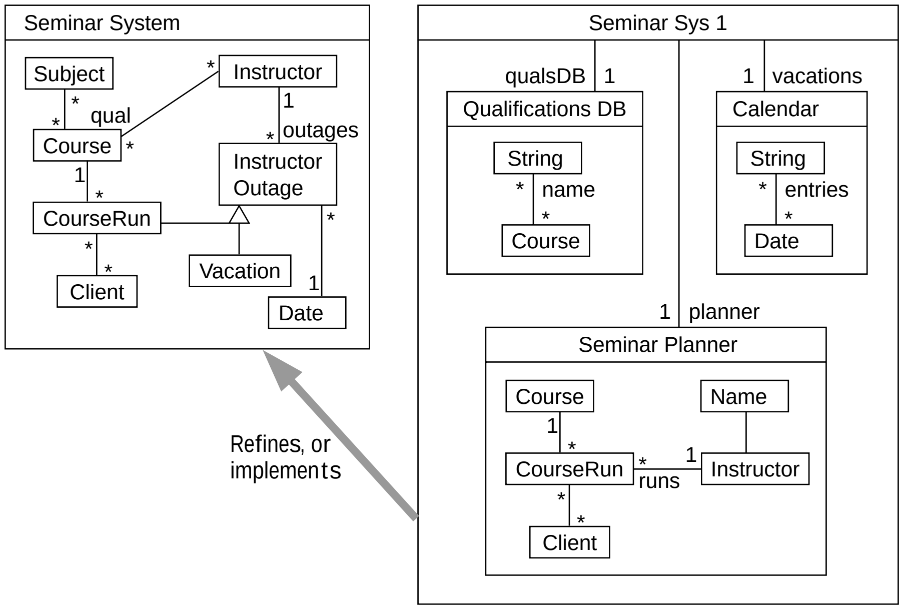


Figure 1.26 Refinement and state mapping: implementation to spec.

Each component has its own model. Because some of them are more general than required for this system—for example, the Calendar associates any Strings with dates and is not specific to Instructors and CourseRuns—not all of them use the same vocabulary. But we can *retrieve* or map the separate components' models back to the system model. For example, each SeminarSystem::Instructor is primarily represented in Seminar Sys 1 components by a String, which is the Instructor's name. To obtain the associations of a SeminarSystem::Instructor given a String n, use these definitions:

```
qual = qualsDB.courses [name=n]
    -- all the courses in the qualsDB linked to name n
outages = vacations.dates [n:entries]
    -- all the dates in the vacations component whose entries contain the name n
            + planner.instructors [name=n].runs
    -- all runs associated with those instructors in the planner that have name n
```

In this manner, we can reconstruct all the attributes and associations we used in the requirements model from the component design.

1.8.4 Collaborations

Now let's compare the functional requirements summaries for the spec and implementation 1 (see Figure 1.27). The dashed boxes are collaborations.[19] A *collaboration* is a collection

18. A full discussion of such partitioning from business to code is in Section 10.11.

of actions and the types of objects that participate in them. Note that it is the collaboration that is being refined rather than only the software systems, because the different versions require different user behaviors to achieve the requirements. The same thing generally applies to collaborations between components or objects at any scale.

Catalysis treats collaborations as first-class units of design work. This is because we take seriously the maxim that decisions about the interactions between objects are the key to good decoupled design. Collaborations can be generalized and applied in many contexts.

1.8.5 Postcondition Retrieval

Each action can be documented with a postcondition in the terms of its participating component. We can check that, given the mappings between the components' models and the overall specification, the various operations in Seminar Sys 1 achieve what was set out for them in the requirements spec for Seminar System.

For example, Calendar::make_entry is supposed to implement SeminarSystem:: set_vacation. Let's presume that the postcondition of make_entry is to associate a String to a Date. In the preceding section, we said that what the spec calls Outages include the dates in which the instructor's name appears in the vacations diary. So make_entry of the Instructor's name will indeed add an outage, as required by the spec's set_vacation. With a bit more work, we can document how some sequence of find suitables, check vacation, and check working together constitute a correct implementation of the abstract action check schedule.

A practical advantage of postconditions is that they can be executed as part of a test harness. As we've just seen, this is true even when they are written in terms of an abstract model: the retrievals can be used to translate from the implementation to the specification's terms.

1.9 *Assigning Responsibilities*

Until now we have used actions to represent something that happens between a set of participants. We can write exactly what the outcome is but still abstract away from the exact dialog: who takes responsibility for what and how the outcome is achieved. So we could draw, for example, Figure 1.28.

This is part of the Catalysis philosophy of being able to write down the important decisions separately from the detail. At some stage in a completed design, we must have rendered all actions down to a dialog, in which each action is a message with a definite sender (with responsibility for initiating the action), a receiver (with responsibility for achieving the required outcome), and parameters (that do what the receiver asks of them). These are

19. The refinement of collaborations is covered in Chapter 6.

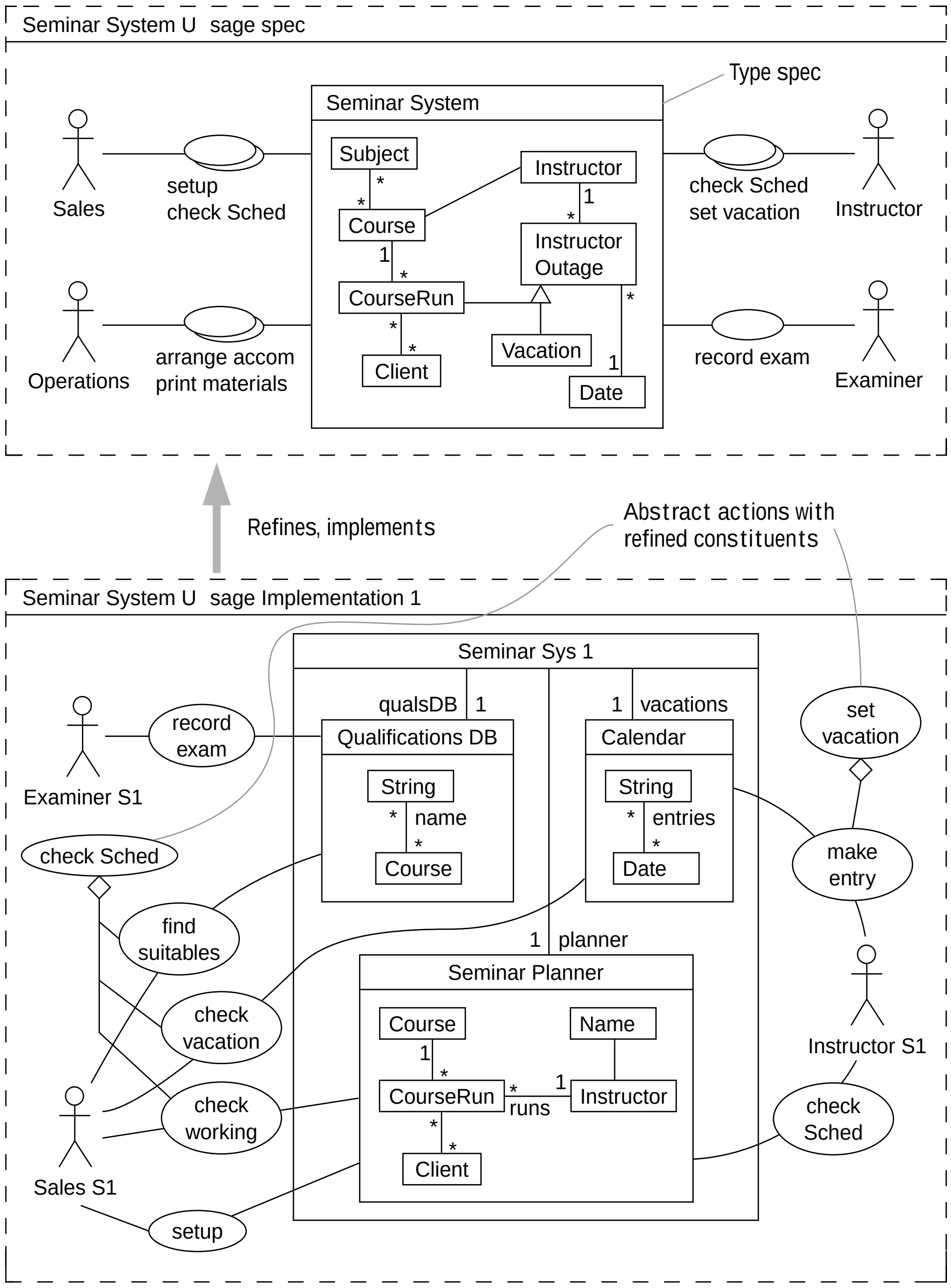


Figure 1.27 Action refinement.

called *directed* actions; when viewed strictly from the side of the receiver, they are called *localized* actions.

Directed actions may be implemented in CORBA, in COM, as method calls in an OO programming language, or as a set of calling conventions in some other style; the directed actions are mapped based on technical architecture choices.[20] There is still a dialog at some level (such as call and return), but this is set by the architecture and local conventions rather than being specific to the participants.

A general strategy for assigning responsibilities is to begin with the holder of the responsibility as a separate role, such as VacationScheduler (see Figure 1.29). Then you decide whether and how to combine the roles (see Figure 1.30).

So when you describe roles in a collaboration using types, you can still defer decisions about how those roles are packaged into objects or components. Appropriate combinations are those that place responsibilities together whose nature and implementation may be changed together.

1.9.1 Flexibility and Decoupling

As a designer, you have a number of concerns. Your job is to put together some objects and make a bigger one that meets a requirement. You must be clear what that requirement

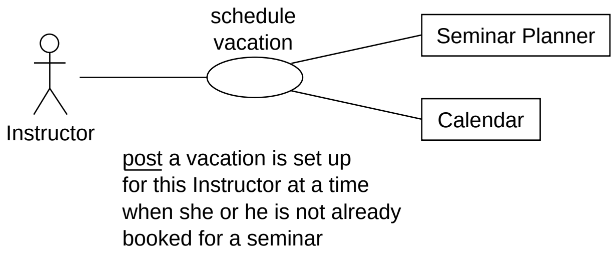

schedule vacation
Seminar Planner
Calendar
Instructor
post a vacation is set up for this Instructor at a time when she or he is not already booked for a seminar

Figure 1.28 An action postcondition can abstract a detailed dialog.

is; you must ensure that the design meets the requirement (perhaps renegotiating the requirement after the design!), and you must bring in the finished result on time with the available resources. You need considerable skill to balance these constraints.

A further constraint, traditionally not so high on the list, is the primary thrust of object-oriented and component-based design. This is to ensure that the finished system not only works but also can be changed easily to meet changing business requirements.

We do this (on any scale) by *decoupling:* separating concerns.[21] The object that deals with vacations need not be the same as the one that deals with exams. The two concerns should be separated so that (1) it is easy to change the way that vacations are scheduled without disturbing how exams are set and (2) it would be possible to create a different

20. Technical architecture is discussed in Section 10.7 and Section 12.6.

21. Section 7.4 discusses package-level decoupling. Design patterns (such as Pattern 16.15, Role Decoupling, or Pattern 16.17, Observer) provide specific techniques for decoupling.

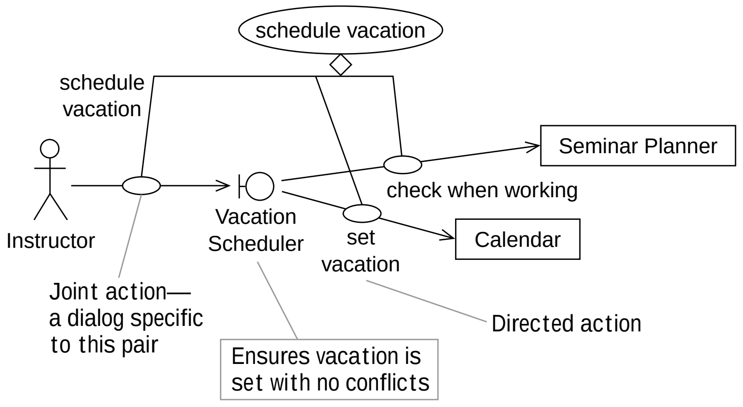


Figure 1.29 Role and interaction design.

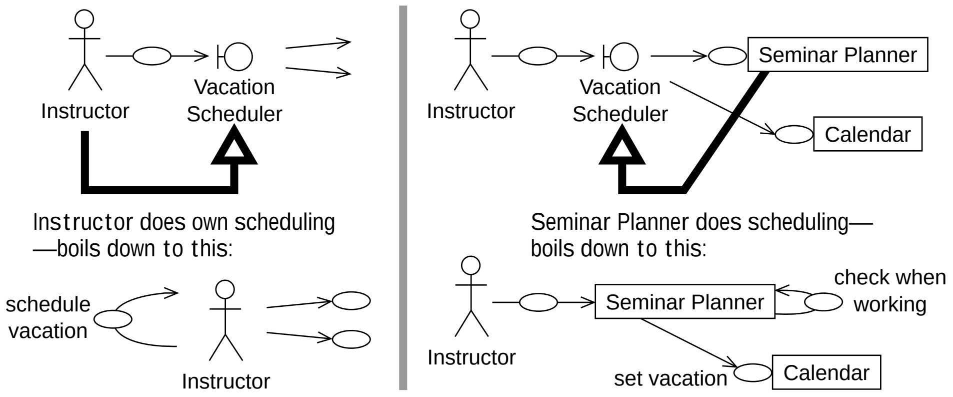


Figure 1.30 Composing roles.

configuration of the same kit of objects in which there were vacations but no exams or vice versa.

The design may also have to provide for a family of related products rather than only one end product. For example, our seminar company's branches in various countries may have to comply with various local regulations. One solution is to make several copies of the code and make small modifications to the code wherever the differences apply. As other modifications are made through time, the national versions will become separate and will need separate teams of programmers to maintain them. A better solution is to move all the national differences into one object so that we need substitute only that one to set up for a different country.

This skill of decoupling, or separation of concerns, distinguishes good designs from those that merely work. Decoupling (at any scale) means a careful distribution of responsibilities among the objects and careful design of how they collaborate to achieve the overall goals of the larger object of which they form a part.

Because this skill is crucial, in Catalysis we provide the means to separate different layers of design decisions:

- The behavior that is required (postconditions)
- Assignment of responsibilities (roles and collaborations)
- The way each object and action is implemented (successive refinement)

1.9.2 Component Frameworks

Let's look at an overall view of the second implementation of the Seminar System, the one with the front end. Indeed it could have different front ends for the different user roles. This view (see Figure 1.31) combines the various façade roles we've discussed previously.

The Seminar System Implementations may be complete in the sense that every required action is dealt with, but what we see in the diagram are specifications of the components. We can't really take our money and go home until we've procured or made an implementation for each one.

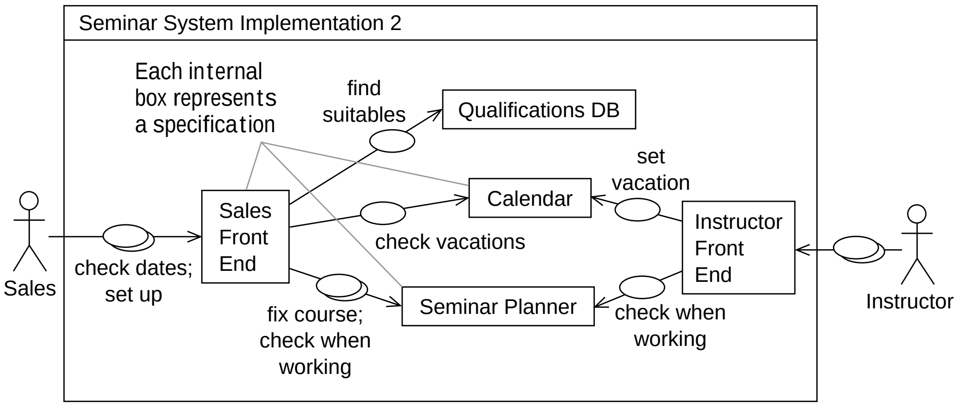


Figure 1.31 A specification of the design.

In fact, there may be several implementations for each component, but the functional success of the overall scheme is dependent only on their specifications. The choice of implementation to *plug in* to each component socket is independent of the others (at least from a functional point of view—there may be performance or other couplings).

In general, a component framework is a collaboration in which all the components are specified with type models; some of them may come with their own implementations.[22] To use the framework, you plug in components that fulfill the specifications. The plug-in implementations may do far more than the spec requires; that doesn't matter, provided that

22. Chapter 11 discusses such framework techniques. Specific techniques for "plugging in" to a larger implementation are discussed in Section 11.5.

we can retrieve the models and postconditions as we illustrated before. For illustration, let's draw implemented objects in bold (see Figure 1.32).

Preferably, the relationship between the sockets in the framework and the components that plug in to them is a pure implementation relationship not involving inheritance of program code. Inheritance in that sense tends to introduce coupling between the superclasses and the subclasses. It restricts the plugs and sockets to being written in the same language, and it means that the framework designers must provide some source code, which they may not want to give away.

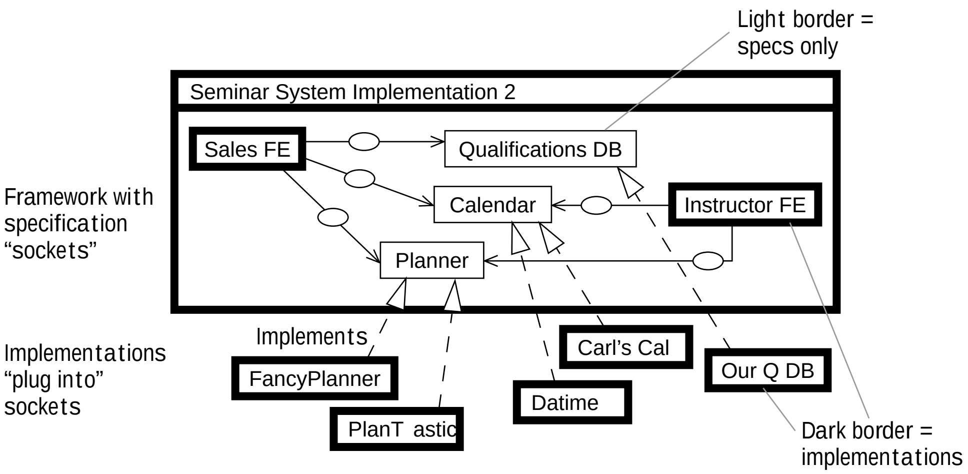

Light border = specs only
Seminar System Implementation 2
Sales FE
Qualifications DB
Calendar
Instructor FE
Planner
Framework with specification "sockets"
Implements
Implementations "plug into" sockets
FancyPlanner
PlanTastic
Datime
Carl's Cal
Our Q DB
Dark border = implementations

Figure 1.32 A framework is a partial implementation with specs of missing parts.

1.9.3 Component Kits

Car designers don't usually design a car entirely from scratch. At least the nuts and bolts are usually borrowed from previous designs. Designers usually design a whole family of products, which are built from a kit of components.[23] The components are built to a common architecture—that is, a set of design conventions that allow interoperability between components in many different configurations. Steering wheels might have different shapes, but all of them have the same attachment to the central shaft. Components from different kits are hard to couple together, because they don't share architecture.

Our seminar system might one day be expanded to support scheduling equipment, invoicing the clients (as soon as possible) and paying the instructors (as late as possible). Several such systems may be federated in the different branches of the company worldwide (so that they can borrow resources from one another). Different branches may want

23. The need for standardization across components is discussed in Section 10.2.2.

different configurations of the system to support their own working practices (as determined by analysts and written down as a business model).

This family of software systems is best built with pluggable components. Some of this partitioning will allow behavior to be changed easily by plugging in alternative components; other partitioning will reflect the fact that operations and accounting departments are distributed to different rooms and use different computers.

The component architecture covers three principal areas:

- The choice of technology (CORBA, function calls, and so on) for connecting components.
- The interchange models—how Clients, Instructors, and so on are represented. This is done with static models.
- Definition of abstract *connectors*[24] between components and their realization down to localized actions (see Figure 1.33). This is done with template collaborations, showing a scheme of interaction that can be mapped into specific types for any pair of components.

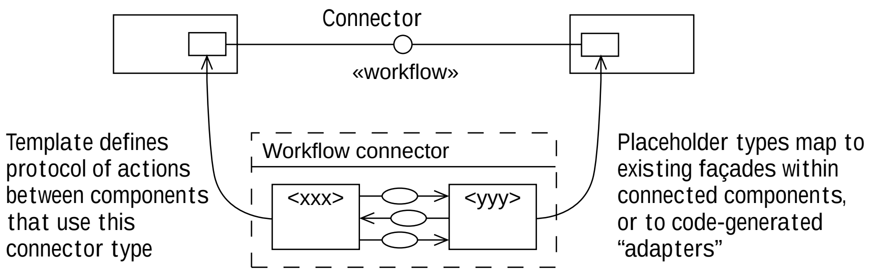

Connector
«workflow»
Template defines protocol of actions between components that use this connector type
Workflow connector
<xxx>
<yyy>
Placeholder types map to existing façades within connected components, or to code-generated "adapters"

Figure 1.33 Defining higher-level component connectors.

1.10 *Object-Oriented Design*

Components can be treated as robustly packaged objects.[25] This might mean that a component comes with a test kit, that it is designed to be fairly defensive against its interlocutors that do not observe the documented preconditions, that it executes in its own space, and that it can cope with intermittent failure of its neighbors. All the principles we have discussed hitherto are therefore just as applicable within a single programming space as they are between objects that are distributed all over the planet.

Nevertheless, it is often useful to make a distinction between a component layer of design and an object layer. There are factors that in practice impose differences in style.

24. See Section 10.8.3 for specifics on how to define abstract connectors.

25. More-detailed discussion of object-oriented design can be found in Chapter 16. Classes and types in OO languages are discussed in Section 3.13.

One is that there may be significant replication of information between the components in a distributed system, for both performance and reliability reasons. A component generally works with others (including people) to support a particular business-level action (or *use case*); an object generally represents a business concept. These two process-biased and object-biased views give rise to separate tiers in many designs.

The vanilla process of object-oriented design begins with the types used for a component model and turns many of them into classes (see Figure 1.34). Hence, Instructor, Course Run, and Course now become classes. Collaborations are worked out and roles are assigned to the classes, as we did for components in Section 1.8. The actions at this level are finally standard OO messages. The associations become pointers, decisions are made about their directionality, and object cleanup is designed (if garbage collection is not built into the language).

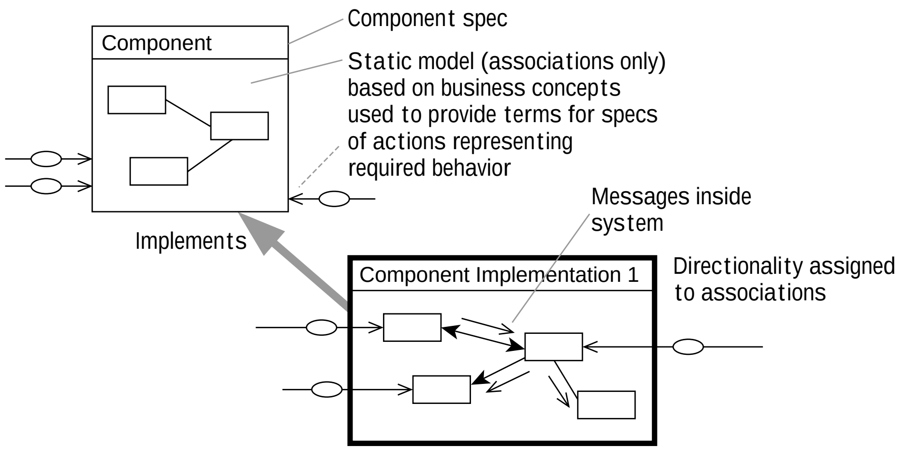

Component spec
Component
Static model (associations only) based on business concepts used to provide terms for specs of actions representing required behavior
Messages inside system
Implements
Component Implementation 1
Directionality assigned to associations

Figure 1.34 Package imports and structured documents.

Design patterns are used to guide these decisions (as they can be used throughout the development process). The end result of OO design is a collection of

- Classes that encapsulate program variables and code.
- Types that define the behavior expected at the interfaces to classes. Classes implement types. In some design styles, all parameters and variables are declared with types; classes are referred to in the code only to instantiate new objects, and even that is encapsulated within *factory* objects.

1.11 *The Development Process*

There is no single process[26] that fits every project: each one has different starting points, goals, and constraints. For this reason, we provide *process patterns* that help you plan a project appropriately to your situation. However, there are some general features.

• *Component-based development:* The main emphasis of component-based development (CBD) is on building families of products from kits of interoperable components. CBD separates design into three major areas:

- *Kit architecture:* The definition of common interconnection standards, in which great skill and care are required
- *Component development:* Careful specification and design and subsequent enhancement of reusable assets
- *Product assembly:* Rapid development of end products from components

• *Short cycles:* The principles of rapid application development (RAD) are recommended. In particular, short development cycles with a well-defined goal at the end of each cycle are good for morale and for moving a project forward. Also, we follow the maxim “Don’t wait until it’s 100% done” for any one phase.

• *Phased development:* This approach is sometimes known as the Empire State Building: take a small vertical slice as far as you can as early as possible in order to get early feedback. Build the rest gradually around it. Phased development is possible if the design is well decoupled.

• *Variable degree of rigor:* The extent to which postconditions are written in a formal style or in natural language is optional. We prefer more rather than less rigor because we have found it helps find problems early.

The same variability applies to the number of separate layers of design you maintain. Clearly, more layers require more maintenance work as well as suitable support tools.

• *Robust analysis phase:* The construction of Catalysis business and requirements models covers more than in the more conventional style. In Catalysis, more of the important decisions are pinned down. As a result, there is less work later in the design stage and less work over the maintenance part of the life cycle, the part that accounts for most of a software system’s cost. (To cope with any uncomfortable feeling of risk that this approach may generate, see the remarks in Sections 1.11.2 and 1.11.3.)

• *Organizational maturity:* Depending on where your team is in its organizational maturity, there are different ways to approach and adopt Catalysis. If your team is used to a repeatable process with defined deliverables and time scales, fuller adoption would be advised; otherwise, start with a “Catalysis lite” process. The team should be prepared to learn the same notations and techniques so as to be able to communicate effectively, and management should sign up to invest in component design and to provide resources for migration.

26. For a process overview, see Chapter 13. The entire process is detailed in Part V.

1.12 *Three Constructs Plus Frameworks*

Catalysis is based on three modeling concepts—type, collaboration, and refinement—and frameworks are used to describe recurring patterns of these three (see Figure 1.35). With these concepts we build a great variety of patterns of models and designs. Types and refinement are familiar to people who are accustomed to precise modeling. Collaborations and frameworks are perhaps more novel, and they add an important degree of expressive power.

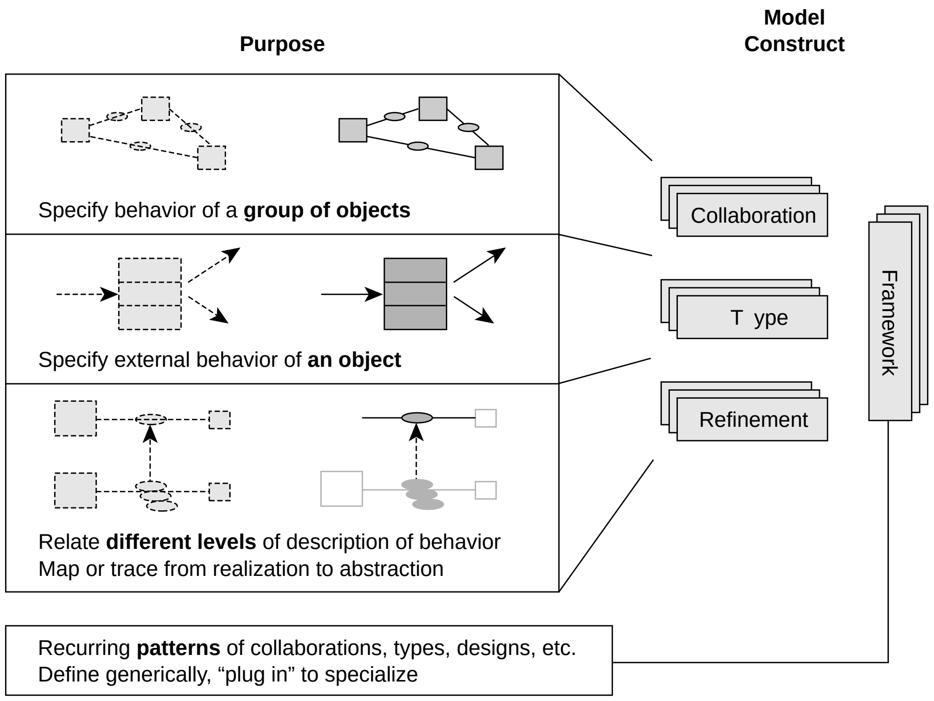
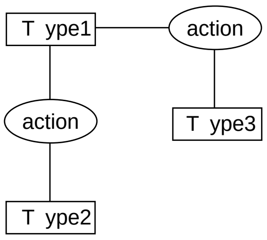

Purpose
Model Construct
Specify behavior of a **group of objects**
Specify external behavior of **an object**
Relate **different levels** of description of behavior
Map or trace from realization to abstraction
Recurring **patterns** of collaborations, types, designs, etc.
Define generically, “plug in” to specialize
Collaboration
T ype
Refinement
Framework

Figure 1.35 Three modeling constructs, with patterns as frameworks.

1.12.1 Collaboration: Interactions among a Group of Objects

The most interesting aspects of design involve partial descriptions of a group of objects and their interactions. For example, a trading system might involve a buyer, a seller, and a broker. Their behavior can be described in terms of their detailed interaction protocols or, more abstractly, in terms of a single high-level action, trade.

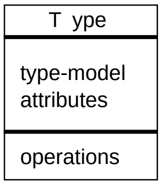


A collaboration defines a set of actions between objects playing roles relative to others in the collaboration. It provides a unit of scoping—constraints and rules that apply within versus outside the group of collaborators—and of refinement: more-detailed realizations of joint behavior. Each action abstracts details of multiparty interactions and of detailed dialogs between participants.

Chapter 4, Interaction Models: Use Cases, Actions, and Collaborations, describes modeling of interactions among a group of objects.

1.12.2 Type: External Behavior of One Object

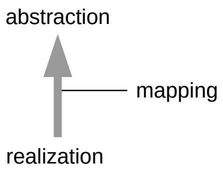
A type defines an object by specifying its externally visible behavior. Whereas a class describes one implementation of an object, a type does not prescribe implementation; you can have many implementations of the same type specification.



Precise description of behavior needs an abstract model of the state of any correct implementation and of input or output parameters. Catalysis uses a *type model* for this. Types specify behavior in terms of the effect of operations on conceptual attributes. For a simple type, these attributes and their types are listed textually; more-complex types may have a type model drawn graphically and even factored into separate drawings.

Chapter 2, Static Models: Object Attributes and Invariants, describes how attributes abstract variations in the implementation of object state. Chapter 3, Behavior Models: Object Types and Operations, describes how operation specifications describe externally visible behavior of an object, independently of algorithmic and representation decisions.

1.12.3 Refinement: Layers of Abstraction

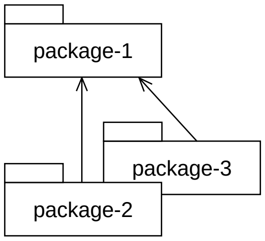


A *refinement* is a relationship between two descriptions of the same thing at two levels of detail, wherein one—the *realization—conforms* to the other—the *abstraction*. A refinement is accompanied by a mapping that justifies this claim and shows how the abstraction is met by the realization.

There are several kinds of refinement. A component design—a realization—refines the component specification—its abstraction. A class implements its behaviors in terms of a particular representation that conforms to a type spec. A particular sequence of fine-grained actions may realize a single, more abstract action. Refinement in Catalysis is more general than the standard ideas of subclassing and subtyping.

A significant part of a Catalysis development process consists of refining or abstracting a description, creating a series of refactorings, extensions, and transformations that ultimately shows the implementing code to conform to the highest-level requirements abstraction (although not necessarily produced in top-down order!). Reengineering, whether business or code, consists of abstracting the existing design to a more general requirement and then refining it to a new design having better performance and so on. In Catalysis, a *design review* is largely concerned with refinement: what did you set out to build, and how did you build it?

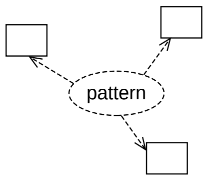
Catalysis uses *packages* to separate design units that will be managed separately, such as different levels of abstraction, permitting reuse of abstract models by multiple independent realizations. A package groups a set of definitions—including types, actions, and collaborations—that can then be *imported* into other packages, making its definitions visible in the importing package.



Chapter 6, Abstraction, Refinement, and Testing, discusses refinement in detail; basic forms of refinement are introduced in Part II, Modeling with Objects.

1.12.4 Frameworks: Generic, Reusable Models and Designs

Specifications, models, and designs built with the three preceding constructs all show recurring patterns. The collaborations for processing an order for a book at an on-line bookstore and for accepting a request to schedule a seminar are also similar in structure—a generic collaboration.



The key to such patterns is the relationships between elements, as opposed to individual types or classes. An application of such a pattern specializes all the elements in parallel and mutually compatible ways. Catalysis provides a fourth construct to capture the essence of such patterns: frameworks. A *framework* is described as a generic package; it is applied by importing its package and substituting problem-specific elements for the generic model elements as appropriate.

Chapter 9, Model Frameworks and Template Packages, describes how frameworks are defined in Catalysis and shows how frameworks provide an enormous degree of extensibility to a modeling language.

1.13 *Three Levels of Modeling*

As shown in Figure 1.36, Catalysis addresses three levels of modeling: the problem domain or business, the component or system specification (externally visible behavior), and the internal design of the component or system (internal structure and behavior).

1.13.1 Problem Domain or Business: The "Outside"

The term *domain* or *business* covers all concepts of relevance to your clients and their problems—that is, the environment in which any target software will be deployed. If you are designing a multiplexor in a telecommunication system, your users are the designers of the other switching components, and the business model will be about things such as packets, addresses, and so on. If you are redesigning the ordering process of a company, the business model is about orders, suppliers, people's roles, and so on.

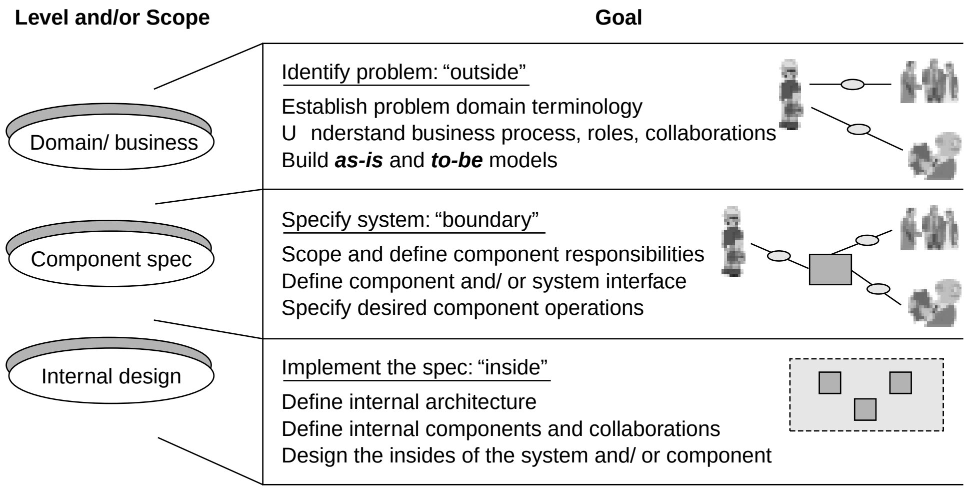

Level and/or Scope
Goal
Domain/ business
Component spec
Internal design
Identify problem: "outside"
Establish problem domain terminology
U nderstand business process, roles, collaborations
Build as-is and to-be models
Specify system: "boundary"
Scope and define component responsibilities
Define component and/ or system interface
Specify desired component operations
Implement the spec: "inside"
Define internal architecture
Define internal components and collaborations
Design the insides of the system and/ or component

Figure 1.36 Three recursive levels of description.

There may 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 one of them than the other. The modeling constructs support separating and joining of such views. Chapter 14, How to Build a Business Model, describes how to go about building a business model.

1.13.2 Component Specification: The "Boundary"

A component specification describes the external behavior required of the component. Catalysis uses a type specification to describe behavior that is visible at the boundary between the component and its environment. A type specification defines the actions in its environment that a component or object participates in.

Chapter 10, Components and Connectors, discusses more-general component models, in which the kinds of the *connectors* between components can themselves be extended to include new forms of component interaction, such as *properties* and *events*. Chapter 15, How to Specify a Component, describes how to go about writing a component specification.

1.13.3 Component Implementation: The "Insides"

The internal design of a component describes how it is assembled from smaller parts that interact to provide the required overall behavior. The design is described as a collaboration, and it must conform to the specification of the component. Note that the "outside" for this design is the type model in the component specification.

At some point during internal design, you must consider the implementation technology and make trade-offs on performance, maintainability, reliability, and so on. Hardware choices (solitary or distributed) and software choices (database, user interface, programming language, tiered architectures) affect how the system is implemented.

Chapter 16, How to Implement a Component, describes how to do the internal design of a component. Chapter 10, Components and Connectors, discusses how to define component designs abstractly and precisely; and Chapter 11, Reuse and Pluggable Design: Frameworks in Code, discusses the design of pluggable class and component frameworks.

1.14 *Three Principles*

Catalysis is founded on three principles: abstraction, precision, and pluggable parts (see Figure 1.37).

1.14.1 Abstraction

To *abstract* means to describe only those issues that are important for a purpose, deferring details that are not relevant.

The word *abstract* often has connotations such as esoteric, academic, and even impractical. In our context, however, it means to separate the most important aspects of a problem from the details, enabling us to tackle first things first. Abstraction is essential in dealing with complexity.

© ***abstraction*** A process of hiding details that provides the following benefits:
- The ability to deal with far-reaching requirements and architecture decisions uncluttered by detail
- Layered models—from business rules and processes to code
- Methodical refinement and composition of components

Think of a software development project as a stream of decisions. Some of them depend on others. There would be no point in trying to design database tables before you establish what the system is going to do. In other words, some decisions are more important than others; making them is a prerequisite to getting the others right.

The important abstractions include the following.

- *Business model and rules:* The context our design is operating in
- *Requirements:* What must be done, as opposed to how it is to be achieved
- *Overall schemes of interaction:* General descriptions without detailed protocols
- *Architecture:* The big decisions about major patterns and components
- *Concurrency:* Which functions can be performed simultaneously and how they will avoid interference while working in coordination

The important choices often don't get made, or even noticed, until way down the line, often in coding. And almost as often, people worry about trivial problems to avoid tackling the big issues.

We need a language to describe the important decisions separately from the clutter of performance and platform issues involved in full implementations. Typical programming

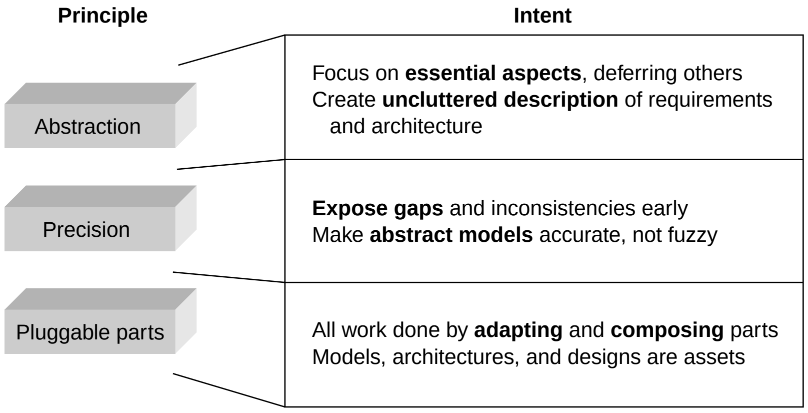

Principle
Intent
Abstraction
Precision
Pluggable parts
Focus on essential aspects, deferring others
Create uncluttered description of requirements and architecture
Expose gaps and inconsistencies early
Make abstract models accurate, not fuzzy
All work done by adapting and composing parts
Models, architectures, and designs are assets

Figure 1.37 Three principles of Catalysis.

languages are better suited for expressing solutions than problems. For this reason, requirements and other high-level descriptions are usually written in a mixture of prose and ad hoc diagrams.

1.14.2 Precision

Whereas code is precise, natural language and ad hoc diagrams are not. How often do groups of analysts or designers discuss requirements around a whiteboard and leave with different interpretations of the problem to be solved? or produce reams of documents ridden with latent bugs and inconsistencies? Documentation that is concise and accurate is far more likely to be useful.

© ***precision*** A characteristic of accuracy that allows you to do the following:
- Expose gaps and inconsistencies early by being precise enough to be refutable
- Trace requirements explicitly through models
- Support tools at a semantic level well beyond diagrams and databases

During implementation, the unforgiving precision of the programming language forces any gaps and inconsistencies to the surface. For this reason, many of us feel confident about a design only when the code has been written. Unfortunately, code also makes us deal with many detailed language and platform-specific issues.

Abstract descriptions are not necessarily ambiguous. If I say “I am quite old, really,” that’s ambiguous. You might think I am geriatric or perhaps that I am a teenager pleased at nearing the age of 18. But if I say “I’m over the age of 21,” that is abstract but perfectly precise. There is no question about what I am prepared to tell you, nor about what I am not prepared to give away.

Abstract high-level descriptions that are not clearly defined are often impossible either to refute or to defend convincingly. Although being precise takes effort, when appropriately used it enhances testability and confidence at all levels. We place a high value on refutable abstractions.

Given a precise notation for abstractions, you can determine whether a given design conforms to the abstraction and can trace how each piece of an implementation realizes each requirement. Tools can help keep track of the propagation of changes in either requirements or implementations.

1.14.3 Pluggable Parts

Building adaptable software is about designing components and plugging them together. Each component is a cohesive piece of design or implementation.

© ***pluggable parts*** The portions of a software effort that are designed to let you do the following:
- Get the most from each piece of design work
- Gain fast, reliable development through reuse
- Reuse not only classes but also frameworks, patterns, and specifications

Software built without using well-defined components will be inflexible: difficult to change in response to changes in requirements. If you don't use previously built components in your designs, you're doomed to repeatedly cover the same ground and make many of the same mistakes. And changes will be much more difficult to incorporate.

A good component is one that can be made to work with a wide variety of others, and that is the key idea behind polymorphism. Such a design makes sense only if you can express accurately what you expect of the other components to which it may be coupled. Plug-in compatibility relies on unambiguously specified interfaces.

This idea of adapting and using components to produce other components should apply at all levels of development, from business models to components that encapsulate generic problem specifications to assembling binary components to produce a running system.

1.15 *Summary*

The three sections that follow briefly recap this chapter's overview of the Catalysis method.

1.15.1 Process Overview

We have seen an example taken through various stages in design:

- Business process modeling
- System context design and requirements specification
- Component design and component specification
- Object-oriented design and implementation of components

The rest of the book elaborates these techniques.

1.15.2 Features Overview

The tour has taken us through a number of features of Catalysis.

- The most important decisions can be separated from the more-detailed ones. What happens, who does it, and how it is done are all separable issues (see Section 1.9.1).
- The states of objects are modeled with associations and attributes (see Section 1.4.1).
- Actions are described in terms of their effects on objects. They can be defined with postconditions (or state charts, as we'll see later) and illustrated with snapshots (see Section 1.1.1).
- Abstract specifications can be made very precise, avoiding ambiguities (see Section 1.1.2).
- Actions and objects can be abstracted and refined—that is, described at different levels of detail. The relationship can be traced, or retrieved, all the way from business goals to program code (see Section 1.2).
- Development is separated into a number of layers, dealing with business analysis, requirements specification, components, and object design (see Section 1.3).
- Templates abstract similar models. We have seen them used to simplify a static model and to define component connectors (see Section 1.5).
- Collaborations—schemes of interaction—are first-class units of design (see Section 1.8.4).
- Components and objects are designed similarly, although with different emphasis on the way they are chosen and responsibilities assigned (see Sections 1.8 and 1.9).
- Components with different views and representations of a business concept can be related to the common business model with retrievals (see Sections 1.8.3 and 1.8.5).
- Components can be designed to plug in to each other and in to frameworks. The plugpoints are defined with action specifications (see Sections 1.9.2 and 1.9.3).
- A component architecture defines a kit by establishing the conventions of interoperation, which are represented by connectors (see Section 1.9.3).

1.15.3 Benefits Overview

• *Enterprise-level design:* The separability of different layers of decision makes Catalysis particularly suitable for design in very substantial projects.

• *High-integrity design:* The precision of Catalysis specifications makes them suitable for the design of mission-critical systems and embedded software, where reliable design is an important issue. The rigor can be used in a variable manner.

• *Traceability:* Catalysis refinement lets you separate abstract models from many possible realizations. The abstract models are still precise enough to be traced to, and even refuted or defended against, concrete realizations; the refinement also enables change propagation management.

• *Pattern reuse and full extensibility:* Catalysis frameworks can be used to define domain-specific patterns of models, collaboration protocols, and component/connector architectures. In fact, primitive types and even the modeling constructs themselves are defined in Catalysis.

- *Component-based development:* Components can be specified by one party, implemented by a second, and used by a third. All these parties must understand the specifications they are working to. To be truly configurable, the components in a kit must work with a wide variety of other members of the kit. Therefore, each designer cannot know exactly what other components he or she is dealing with. For this approach to work, component kit architecture and component development must be seen as a high-integrity design. Catalysis extends clear component specification with the connector abstraction, simplifying the design of component-based products.
- *A behavior-centric and data-centric approach:* Previous methods, such as OMT, have been criticized by “behaviorists” for what is perceived as a data-centric approach. In Catalysis the two views support each other simply. You describe the behavior of a component in terms of attributes that relate to the clients’ concerns rather than any implementation.
- *Tool support:* Catalysis enables a high level of tool support far beyond drawings on a database with some document generation. The standard notations and the clear relationship between artifacts also mean that you can use popular object-modeling UML tools on a Catalysis process by following simple usage guidelines.