

Chapter 10

Case study

OBJECTIVES

In this chapter you will learn:

- ☐ how to integrate the previous work;
- ☐ how to produce a specification of a small system;
- ☐ how to embed Z in an English framework.

This chapter draws together the material covered in previous chapters. This is done by taking a case study and using it to specify a system of medium size. This will involve specifying the problem in both English and Z. Additionally, in the hope that it will help readers in writing their own specifications, the process by which the specification is produced will be explained, and indications given of how and why the route taken has been chosen.

The case

A university owns a large building which is used as a student residence. All rooms offer the same type of single accommodation. If a student applies for a room, and one is available, he or she is offered it. Otherwise, they join a waiting list. Students who are offered rooms may refuse them, but doing so means that they are put to the back of the waiting list. Sometimes a new room becomes available, either because someone leaves or because a room which has not been in use is put into the stock. When this happens, the room is immediately offered to the person at the head of the waiting list. To maximize the profit from the residence, usable rooms are not kept empty for any longer than is necessary.

A system is required to process students' applications for accommodation and their departures from it, and allows questions about occupancy to be answered. In particular, the following operations should be supported:

- adding a room to the residence's stock;
- processing an application for a room;
- accepting an offer of a room;

STATE SPECIFICATION

105

- identifying the room of a student;
- rejecting an offer of a room;
- leaving the waiting list;
- leaving a room;
- identifying the student housed in a room;
- withdrawing a room from possible occupancy.

At this stage, readers might gain most by trying to specify the system for themselves before reading on. It will be interesting to compare such work with what follows, but bear in mind that there is no one right answer.

State specification

In designing the state model, the software engineer is looking for the minimum number of state variables that describe, conveniently and understandably, the states of the system and the operations on it. Finding such a model is often difficult. Indeed, in fairness to readers intending to write Z specifications, it should be stated that the work shown below is the result of a number of iterations. Various possible state models were tried and found wanting; sometimes they looked promising, but, although coping well with some operations, would fail adequately to cope with others. Software engineers are advised to persevere and try out as many different versions as are necessary to get a good model. Otherwise, any problems encountered here will be magnified many times over later in the development process.

The choice of state model depends on the functions the system is expected to perform, i.e. the system's scope. The smaller the scope, the simpler the model should be.

It is for users to decide a system's requirements. Ideally, they know what they want and the specification is left to describe the 'what' of the system, not the 'how'. But sometimes the situation is more fluid, with users not too sure what they want. In such situations, it is possible that they will be influenced by the system model, and that this, in turn, affects what the requirements become. In other words, the tail wags the dog. This is because a model both constrains the way that a system can be thought about – thereby affecting the questions that we ask about it and our expectations of it – and affects the relative ease with which operations can be implemented – thereby changing our priorities as to what to implement. Thus, having chosen a system model, decisions about a project's scope often slip in, occurring without either the knowledge of users or the explicit awareness of the software team. For this reason, it is important to state, as far as is possible, both the explicit design decisions made by the project team and those implicit within the model itself. However, the difficulty of doing this should not be under-estimated; we are all affected by our cultural conditioning, so many assumptions will be made subconsciously and not be available for description.

Two examples will be given of how these issues affect the case study. First, if the empty rooms are modelled as a set and if the only data stored about them is an identifier, then in the absence of any additional data they must all be treated as equal,

Important point. What follows is the result of some experimentation. In your own work, do not expect to get it correct first time

Readers should note that in a commercial environment this would be considered a *simple* specification

since there is no information to differentiate between them in terms of quality or the order in which they became vacant, or whatever. Once such a design decision has been made, a user requirement that applicants be offered, say, the largest empty room or the room most recently vacated becomes difficult to implement, since it would entail changing the model to include the extra data required.

Second, if it is decided that STUDENT should be a given set, it is possible to ignore what it is that qualifies a person to be a resident. This makes the model simpler to understand, but at a cost that the scope of the project is reduced in that, apart from type checking, the system will not attempt to determine if a person applying for a room is a valid applicant or not.

Preliminary assumptions and design decisions

At this stage, the following assumptions and consequent design decisions are made:

- The system's scope is limited to processing just those operations described above. This implies that only details concerned with rooms and students need to be stored.
- Apart from their status, being empty, under offer, or occupied, all rooms are considered as equal. Therefore, no other information about them needs to be kept. No details will be kept of rooms that do not fall in any of these categories.
- Apart from their status, waiting for an offer, being offered a room, or occupying one, all students are considered equal. Therefore, no other information about them needs to be kept. No details will be kept of students that do not fall in any of these categories.
- It is assumed that checks on whether or not a person is a student who is entitled to accommodation are carried out elsewhere. The system will not make these checks.
- It is assumed that checks on whether or not a room is one valid for occupancy by a student are carried out elsewhere. The system will not make these checks.
- It is assumed that offers, once made, stand indefinitely. The system does not cancel offers if a response has not been made within a certain time. Therefore, time is ignored. Clearly, since it has been known for students not to reply quickly to an offer, a useful addition to the system would be an operation to cancel an offer after a certain time. But this would make the system more complex.
- There is no need for the system to deal with its own security, by, for instance, making unauthorized access difficult. In other words, the security of the system is dealt with elsewhere.
- Although this is not always the case in specifications, detailed concerns about the user-friendliness of the system are considered outside the scope of this work. Indeed, getting involved with such details can make the specification harder to understand. Consequently, the policy adopted here is to use a brief message to describe the output from operations. Clearly, the implementation should provide something more meaningful.

These assumptions mean that the information needed by the system is limited to: who lives where; who has been offered what; what rooms are sitting empty; and who

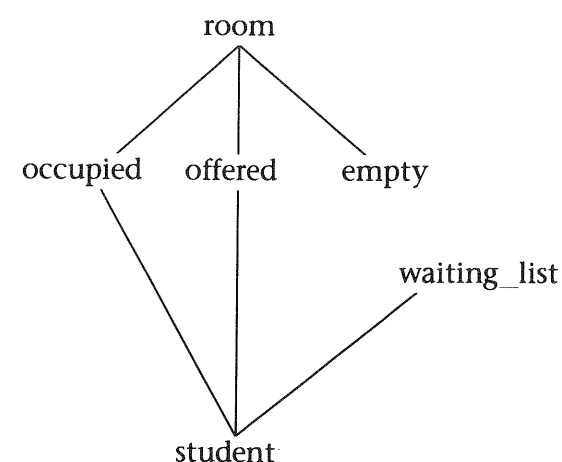


Figure 10.1 Objects in the system and their connections.

is waiting and in what order. This is shown schematically in Figure 10.1, in which the lines show where a connection needs to be made.

Types

It is necessary to describe the types of the variables to be used in the state specification and the operations on it.

In the comments above, it was explained that the only information to be kept will be unique identifiers of rooms and students in the system. Therefore, the given sets are [STUDENT, ROOM] where

STUDENT – the set of people who might be housed;

ROOM – the set of rooms in which they can be housed;

A message type is also needed. With this, it is often easiest to adopt a bottom-up approach and list the possible values *after* they have been used in the schemas. In this case, the following messages have been used:

MESSAGE ::= FAIL | OK | OK_EMPTY | OK_FILLED | OK_NEW |
OK_OFFER | OK_STAND | OK_WAIT | REFUSE

State variables

To hold the data needed to model the system, the following state variables are used:

occupants – shows for each occupied room who lives there;

offers – shows for each room under offer who it has been offered to;

empty – shows those rooms which are available to be offered for occupation, i.e. those rooms which are empty and not under offer;

waiting_list – shows the order in which students are placed on the waiting list.

It is worth noting that the variable name 'empty' has been found confusing by a number of people who have worked through this case study. This suggests that it should be changed. One change suggested has been 'empty_rooms'. Since this chapter is interested as much in the process of specification as the final result, it will be left as it is, for the reader to make his or her own opinion about it. It is a good example of the importance of choosing meaningful variable names.

State declaration

Each one of the state variables must be modelled, and restrictions both on and between variables must be listed. Together these make up the declaration and state invariant parts of the state schema.

As described above, a good way to progress with this sort of system is to think of how a paper-based system to do the same job would work. For the residence, the paper model shown in Figure 10.2 would be useful.

It must be repeated that implicit within the decision to hold just the amount of data shown in the diagram is the assumption that all students and rooms are equal.

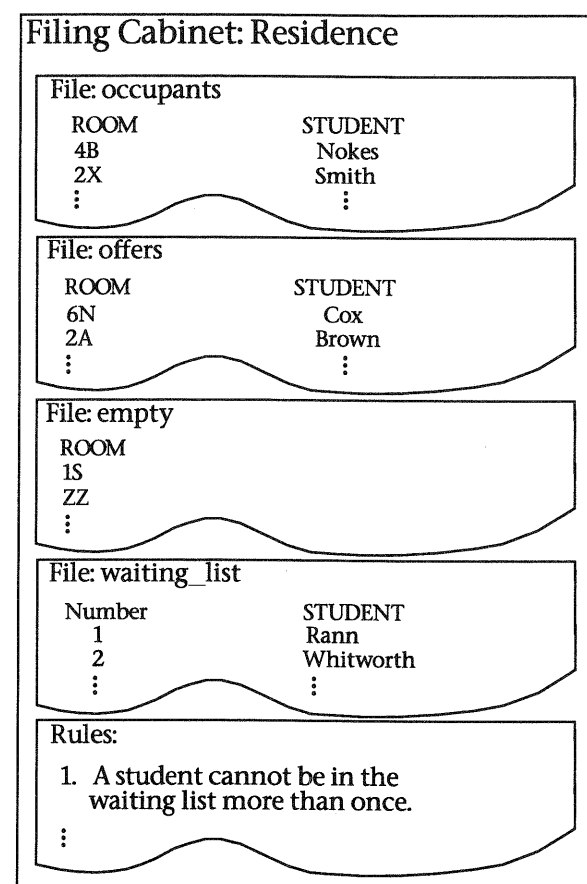


Figure 10.2 Paper model of residence.

Clearly, if either of these assumptions is not accepted by users, additional data will need to be stored.

Once this model has been created, it is relatively easy to write the rest of the specification. However, it is a common mistake for people to concentrate on getting the Z right; leaving the English description to lag along behind. This is bad practice, not only because users get a specification which they cannot understand, but also because software engineers lose a valuable check on their own work. It is much better to work at the same time with both the English and the Z; using one to spot errors in the other.

The state variables are, therefore, modelled as follows:

occupants

It is assumed that:

- only students can rent rooms;
- each room can only be rented by one student;
- a student can only rent one room.

Consequently, *occupants* can be modelled as a partial injective function from ROOM to STUDENT (so the declaration is *occupants*: ROOM $\rightarrow$ STUDENT). (It is partial because it is not necessary that all rooms are occupied; it is a function because each room is occupied by a single student; it is injective because a student can only occupy one room.)

offers

It is assumed that:

- students are given an offer for a specific room;
- a room is offered to only one student;
- only one room is offered to a student.

Consequently, *offers* can be modelled as a partial injective function from ROOM to STUDENT (*offers*: ROOM $\rightarrow$ STUDENT). (It is partial because it is not necessary that all rooms are under offer; it is a function because each room is offered to a single student; it is injective because a student cannot have an offer to more than one room.)

empty

As stated above, it has been assumed that all empty rooms are equal in that there is no order placed on how they are allocated. Consequently, *empty* can be modelled as a set of ROOM (*empty*: $\mathbb{P}$ ROOM).

waiting_list

It is assumed that:

- the waiting list is ordered according to the precedence of the students' room applications;
- a student cannot be on the waiting list more than once.

To satisfy the first assumption, the waiting list is modelled as a sequence (*waiting_list*: seq STUDENT).

Even if the residence is always fully occupied, *occupants* would still be partial. This is because ROOMS is the set of possible rooms, not just those which are available at any one time. This is better than 'hard-wiring' ROOMS, because it allows the system to expand as new rooms are built

To satisfy the second assumption, a state invariant is needed to ensure that every value in the sequence must be different. This can be written as

$$\# \text{waiting_list} = \# \text{ran}(\text{waiting_list})$$

The reason why this works was dealt with in Chapter 7. To recap, it is necessary to realize that a sequence is just a special function; one whose domain is the set of integers from 1 to the size of the sequence, and whose range is formed by the sequence's values. Therefore, if a value appears in the sequence more than once, the size of the sequence's range is less than the size of its domain. Conversely, if the sequence's range is as large as its domain, all the values must be different.

Invariants between variables

The declarations have implicitly given invariants on the state variables. For instance, the underlying type of *occupants* is a relation. The fact that it has been declared as a partial injection, provides constraints, invariants, on it. These are implicit; not being found 'below the line'.

It is also necessary to give the invariants between the variables.

It is assumed that a student with a room cannot reapply. So, no students have more than one of: a room, an offer, or a place on the waiting list. That is, the intersection between the following pairs of sets is empty:

- students in rooms and students offered rooms (this is written: $\text{ran}(\text{occupants}) \cap \text{ran}(\text{offers}) = \{\}$);
- students in rooms and students on the waiting list ($\text{ran}(\text{occupants}) \cap \text{ran}(\text{waiting_list}) = \{\}$);
- students on the waiting list and students offered rooms ($\text{ran}(\text{offers}) \cap \text{ran}(\text{waiting_list}) = \{\}$).

It is assumed that a room can only be in one of three states: occupied, offered, empty. Since at any moment of time it cannot be in more than one of these states, the intersection between the following pairs of sets is empty

- occupied rooms and rooms under offer ($\text{dom}(\text{occupants}) \cap \text{dom}(\text{offers}) = \{\}$);
- occupied rooms and empty rooms ($\text{dom}(\text{occupants}) \cap \text{empty} = \{\}$);
- offered rooms and empty rooms ($\text{dom}(\text{offers}) \cap \text{empty} = \{\}$).

The university's policy of not keeping rooms empty when students need housing implies that there can only be a waiting list if all the rooms are already occupied or offered. This is written

$$\text{waiting_list} \neq \langle \rangle \Rightarrow \text{empty} = \{\}$$

State specification

Putting together what we have seen already, the state of the system is specified by

Residence

occupants : ROOM $\rightarrow$ STUDENT
offers : ROOM $\rightarrow$ STUDENT
empty : $\mathbb{P}$ ROOM
waiting_list: seq STUDENT

$\text{ran}(\text{occupants}) \cap \text{ran}(\text{offers}) = \{\}$
 $\text{ran}(\text{occupants}) \cap \text{ran}(\text{waiting_list}) = \{\}$
 $\text{ran}(\text{offers}) \cap \text{ran}(\text{waiting_list}) = \{\}$
 $\text{dom}(\text{occupants}) \cap \text{dom}(\text{offers}) = \{\}$
 $\text{dom}(\text{occupants}) \cap \text{empty} = \{\}$
 $\text{dom}(\text{offers}) \cap \text{empty} = \{\}$
 $\text{waiting_list} \neq \langle \rangle \Rightarrow \text{empty} = \{\}$

Operations

Each operation on the state has to be specified. Once again, if there is any uncertainty as to what is required, it is necessary to state clearly any assumptions that have been made.

Initial state

In the initial state there are no occupants, no offers, no empty rooms and no waiting list

InitResidence

Residence'

occupants' = $\{\}$
offers' = $\{\}$
empty' = $\{\}$
waiting_list' = $\langle \rangle$

At first sight, this may appear to be a strange choice for an initial state, especially if the residence is already open and running. But, if the present situation was explicitly described in the 'Init' schema, problems arise with reusability. This is because the system could not be used for other residences until unwanted data were deleted.

Add a room

This operation takes the identification of a room which has just become available for occupation (perhaps because it has just been built) and either allocates it immediately, if necessary, or else adds it to the set of empty rooms.

The room's identification is input. The status of the room, being newly available (that is, it is unoccupied, it has not been offered, and is not presently available but empty) or not, and of the waiting list, being empty or not, gives rise to three cases:

Why is the following insufficient?

$$\text{ran}(\text{occupants}) \cap \text{ran}(\text{offers}) \cap \text{ran}(\text{waiting_list}) = \{\}$$

Why is the following wrong?

$$\text{empty} = \{\} \Rightarrow \text{waiting_list} \neq \langle \rangle$$

The table is not part of Z. It is a documentation technique to make the English explanation easier to understand. Remember, the English description should stand alone. We are aiming for a belt and braces job

Waiting list empty	Room newly available	
	Yes	No
Yes	Case 1	Error
No	Case 2	Case 3

Case 1

The room is newly available and there is no waiting list. It joins the set of empty rooms. All other variables remain unchanged. The message 'OK_EMPTY' is returned:

<i>AddRoomEmpty</i>
Δ Residence room? : ROOM student!: STUDENT reply! : MESSAGE
$room? \notin dom(occupants) \cup dom(offers) \cup empty$ $waiting_list = \langle \rangle$ $empty' = empty \cup \{room?\}$ $occupants' = occupants$ $offers' = offers$ $waiting_list' = waiting_list$ $reply! = OK_EMPTY$

Case 2

The room is newly available and there is a waiting list. It is offered to the person at the head of the waiting list. This person is taken from the waiting list. All other variables remain unchanged. The name of the student offered the room and the message 'OK_OFFER' are returned:

<i>AddRoomOffer</i>
Δ Residence room? : ROOM student!: STUDENT reply! : MESSAGE
$room? \notin dom(occupants) \cup dom(offers) \cup empty$ $waiting_list \neq \langle \rangle$ $offers' = offers \cup \{(room?, head(waiting_list))\}$ $waiting_list' = tail(waiting_list)$ $occupants' = occupants$ $empty' = empty$ $student! = head(waiting_list)$ $reply! = OK_OFFER$

his is equivalent to:
 $room \notin dom(occupants) \wedge$
 $room \notin dom(offers) \wedge$
 $room \notin empty$

Case 3

The room is not newly available. The state remains unchanged. The message 'FAIL' is returned:

<i>AddRoomFail</i>
$\exists$ Residence room?: ROOM reply!: MESSAGE
$room? \in dom(occupants) \cup dom(offers) \cup empty$ $reply! = FAIL$

This error case could be split into further sub-cases, if required. A balance needs to be struck between preciseness (which in this situation means providing useful error messages) and overall clarity, requiring being able to see the wood for the trees

It is worth noticing that implicit within this schema is the design decision to provide a single error message for the three possible error cases given by the room already: occupied, offered or empty. (The possibility that the room is not an eligible one is taken care of by the type checking.) In certain situations, this could be unacceptable. The final decision rests with the user.

The whole *AddRoom* operation is specified by

$AddRoom \hat{=} AddRoomEmpty \vee AddRoomOffer \vee AddRoomFail$

Application for a room

This operation processes the application by a student for a room.

The name of the person applying is input. The status of the applicant, being known (that is, already has a room, an offer, or is on the waiting list) or not, and of empty rooms, being available or not, gives rise to three cases:

Empty rooms available	Applicant known	
	No	Yes
No	Case 1	Error
Yes	Case 2	Case 3

Case 1

The applicant is not known and there are no empty rooms available. The application is accepted with the student put at the end of the waiting list. All other variables remain unchanged. The message 'OK_WAIT' is returned:

<i>ApplyOKWait</i>
Δ <i>Residence</i> <i>student?</i> : STUDENT <i>room!</i> : ROOM <i>reply!</i> : MESSAGE
$student? \notin ran(occupants) \cup ran(offers) \cup ran(waiting_list)$ $empty = \{\}$ $waiting_list' = waiting_list \setminus \langle student? \rangle$ $occupants' = occupants$ $offers' = offers$ $empty' = empty$ $reply! = OK_WAIT$

Case 2

The applicant is not on the books. There is an empty room available. The application is accepted with the student being offered (arbitrarily) one of the empty rooms. All other variables remain unchanged. The room's identification and the message 'OK_OFFER' are returned.

<i>ApplyOKOffer</i>
Δ <i>Residence</i> <i>student?</i> : STUDENT <i>room!</i> : ROOM <i>reply!</i> : MESSAGE
$student? \notin ran(occupants) \cup ran(offers) \cup ran(waiting_list)$ $empty \neq \{\}$ $(\exists r: ROOM \cdot r \in empty \wedge$ $offers' = offers \cup \{(r, student?)\} \wedge$ $empty' = empty \setminus \{r\} \wedge$ $room! = r$ $)$ $occupants' = occupants$ $waiting_list' = waiting_list$ $reply! = OK_OFFER$

Case 3

The applicant is known. The application is rejected and the state remains unchanged. The message 'REFUSE' is returned:

<i>ApplyRefuse</i>
Ξ <i>Residence</i> <i>Student?</i> : STUDENT <i>reply!</i> : MESSAGE
$student? \in ran(occupants) \cup ran(offers) \cup ran(waiting_list)$ $reply! = REFUSE$

The whole *Apply* operation can be specified by
 $Apply \hat{=} ApplyOKWait \vee ApplyOKOffer \vee ApplyRefuse$

Acceptance of an offer

This operation processes the acceptance of an offer of a room.
The name of the student is input. (The design decision is made not to try to validate this input by also inputting the room offered and checking that the two match. In other words, no attempt is made to check that the student is agreeing to the particular room offered. This is only possible because students do not have a choice as to what room to accept.) According to the status of the person, having an offer or not, there are two cases:

Student has an offer	
Yes	No
Case 1	Error Case 2

Case 1

The student whose name has been input has been made an offer of a room. The person becomes an occupant of that room. The offer is withdrawn. All other variables remain unchanged. The message 'OK_FILLED' is returned:

<i>AcceptOK</i>
Δ <i>Residence</i> <i>student?</i> : STUDENT <i>reply!</i> : MESSAGE
$student? \in ran(offers)$ $occupants' = occupants \cup \{(offers^{-1}(student?), student?)\}$ $offers' = offers \setminus \{student?\}$ $empty' = empty$ $waiting_list' = waiting_list$ $reply! = OK_FILLED$

To find the new value of occupants, it is necessary to know the room offered to *student?*. In the schema, this is found by applying the inverse of the function *offers* to *student?*. This is possible here because, since *offers* is an injection, its inverse is also a function. It would not be possible otherwise.

Case 2

The student has not been made an offer of a room. The state remains unchanged. The message 'FAIL' is returned:

<i>AcceptFail</i>
$\exists$ <i>Residence</i> <i>Student?:</i> STUDENT <i>reply!</i> : MESSAGE
<i>student?</i> $\notin$ <i>ran(offers)</i> <i>reply!</i> = FAIL

The *Accept* operation is specified by

$$Accept \triangleq AcceptOK \vee AcceptFail$$

These are only some of the operations in the system. More follow in the exercises.

Readers may wish at this stage to jump to Chapter 14, which does not rely on any of the intervening material, to test their knowledge of Z. If there is any poorly understood material, it will be useful to revise this before proceeding to the more technical chapters which follow.

Exercises

1. Specify the query operation *Address* which, when given the name of a student, returns a message identifying that student's room.
2. Specify the operation *Reject* which, when given the name of a student, deals with their rejection of an offer of a room.
3. Specify the operation *LeaveWaiting* which, when given the name of a student, withdraws them from the waiting list.
4. Specify the operation *Leave* which, when given the name of a student, deals with their departure from a room.

Chapter 11

Schema operators

OBJECTIVES

In this chapter you will learn about the following:

- ☐ inclusion;
- ☐ disjunction;
- ☐ conjunction;
- ☐ decoration;
- ☐ change of state, Δ ;
- ☐ no change of state, Ξ ;
- ☐ renaming;
- ☐ hiding.

One of the nice things about Z is that the schema represents a powerful structuring mechanism. This makes it possible to encapsulate larger amounts of data than can conveniently be held in a single state variable, and more complex operations than could be described by a simple function. Particularly when larger systems are being specified, it is necessary to maintain the level of abstraction offered by schemas and to work with these directly, rather than with their contents. To do this, operators, which act on schemas are needed. Some of these schema operators, such as Δ and Ξ , have already been met in an informal way; here they will be looked at more formally. Also, other schema operators which will be useful later in the book will be introduced. However, a full description of all of Z's schema operators is outside the scope of this book.

Inclusion

An informal way to think about a schema, *S*, say, is as a block of text. Then, whenever *S* is seen it can be replaced with the text that it represents. However, it is not quite as easy as that. First, it has to be remembered to keep declarations above the 'line' and predicates below. Second, it is necessary to make sure that a variable is not declared with different types in different places.