Airline Reservation System

Ootumlia Airlines runs sightseeing flights from Java Valley, the capital of Ootumlia. The reservation system keeps track of passengers who will be flying in specific seats on various flights, as well as people who will form the crew. For the crew, the system needs to track what everyone does, and who supervises whom. Ootumlia Airlines runs several daily numbered flights on a regular schedule. Ootumlia Airlines expects to expand in the future, therefore the system needs to be flexible; in particular, it will be adding a frequent-flier plan.

Example 5.2 *Using the description of the Airline system from Appendix C, list the nouns and noun phrases that might end up being classes in a system domain model. For those nouns that should not become classes, explain why not.*

Nouns that are put on an initial list of classes include: `Flight`, `Passenger`, `Employee`.

Other nouns or noun phrases that we choose not to include in the initial list of classes:

- 'Ootumlia Airlines', 'Java Valley', 'Ootumlia'. These are instances.
- 'Reservation system'. This is not a class because it is part of the system, not the domain information represented by the system.
- 'Sightseeing Flight'. Rejected in favor of `Flight`, since the latter is more general and therefore makes the system more flexible.
- 'Seat'. Appears to be an attribute of `Flight`.
- 'Crew'. This word implies the entire crew, when we really want to store information about individual members of that crew. We could have created a class `CrewMember`, but `Employee` seemed more flexible – allowing us to use the class in future for people who are not actually crew members.
- 'Schedule'. This is a word that describes a complex bundle of information that would be better represented by classes such as `Flight`, and the attributes and associations of those classes.
- 'Future'. This is a noun, but it is not part of the system to be developed.
- 'Frequent Flier Plan'. This is not part of the current scope.

Example 5.3 *Continuing from Example 5.2, add an initial set of associations and attributes to the classes you identified. Add and delete classes as necessary.*

A central class at which to start identifying associations and attributes is `Flight`. The problem description informs us that there is a schedule of numbered flights; from this we can infer that a flight has a `date`, a `time` and a `flightNumber`. However, we notice that every day the flights that leave at the same time have the same flight number. Therefore it makes sense to split `Flight` into two classes we will call `RegularFlight` (containing the time and flight number for flights that regularly depart at the same time of day) and `SpecificFlight` (that departs on a particular day, and on which passengers are booked). The association between these classes is one-to-many. We do not need to name the association explicitly since the default 'has' appears adequate. (This approach does not allow us to deal well with charter flights, but we leave that as an exercise.)

We now move on to understand how passengers are booked. There is clearly a many-to-many relationship between `Passenger` and `SpecificFlight` but, as discussed earlier in Section 5.3, we need to add a `Booking` association class; this will contain the `seatNumber`.

Each `Passenger` has a name, but we will also assign him or her a number in case names are identical, and to allow for the anticipated frequent-flier plan.

The phrase 'who supervises whom' implies that `Employee` needs a reflexive association with a `supervisor` role. The phrase 'what everyone does' implies an attribute `jobFunction`.

The class diagram so far is shown in Figure 5.29.

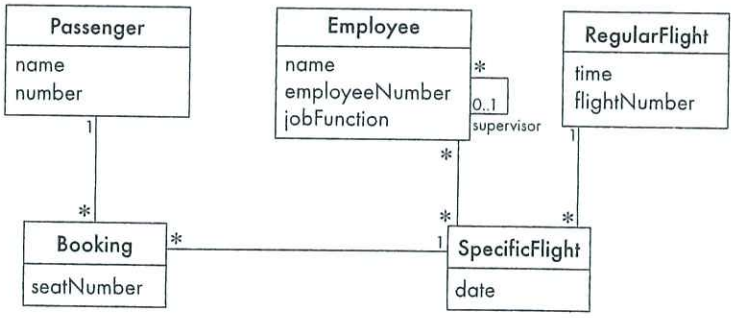

Figure 5.29 **First attempt at a class diagram for the airline system**

Example 5.5 *Create a list of responsibilities for the Airline system discussed in Examples 5.2 to 5.4. Allocate each responsibility to a class and discuss your reasoning for the allocation. Finally, update the class diagram as necessary.*

The following are a few of the responsibilities derived from the system description, with an indication of in which classes they might be put. It is clear when creating this list of responsibilities that the airline system description is rather too brief – forcing us to deduce the presence of unstated requirements.

- Creating a new `RegularFlight`. This could be a class (static) responsibility of the `RegularFlight` class. But we prefer, as much as possible, to give responsibilities to instances. We will therefore introduce a new class called `Airline` (shown in Figure 5.32) that will have this responsibility. There will probably be only one instance of `Airline`.
- Searching for a particular `RegularFlight`. In order to do this, we need a class that maintains a collection of all the instances of `RegularFlight`. That class will be `Airline`, which will therefore have this responsibility.
- Modifying the attributes of a `RegularFlight`. Each class should normally modify its own attributes; this responsibility should therefore go in `RegularFlight`.
- Creating a `SpecificFlight`. We choose to put `RegularFlight` in charge of this, since the new `SpecificFlight` will be an occurrence of a particular `RegularFlight` that already exists at the time this responsibility is initiated.

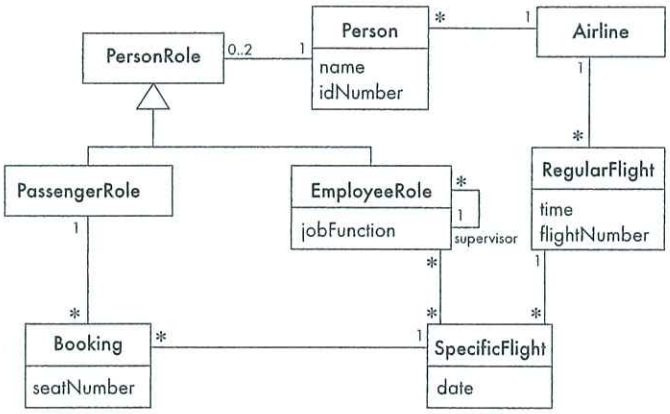

Figure 5.32 Airline system after adding the `Airline` class

- Canceling a `SpecificFlight`. This can be put in `SpecificFlight`.
- Booking a `PassengerRole` on a `SpecificFlight`. This could go in either `PassengerRole` or `SpecificFlight`. It could also go in `Booking`, but we prefer not to put it there since the appropriate Booking is not yet created by the time this responsibility is initiated. We choose to give this responsibility to `PassengerRole`, since in the real world it is the passengers who initiate bookings.
- Canceling a `Booking`. This should be put in `Booking`.

5.11 Difficulties and risks when creating class diagrams

The following is the key difficulty to anticipate when creating class diagrams in an industrial context:

- **Modeling is a particularly difficult skill.** Many people who are excellent programmers nevertheless have considerable difficulty thinking at the level of abstraction needed to create effective models. Also, since education programs have not traditionally focused on it, software developers often have significantly less knowledge about modeling than about design and programming. Taken together, these mean that software projects are at risk from models that are incomplete, incorrect or insufficiently flexible.
  Resolution. Ensure that members of the team have adequate training in modeling. Have an experienced modeler as part of every team. Review all models thoroughly.

5.12 Summary

In this chapter we introduced UML, the Unified Modeling Language. Then we showed you how to create class diagrams, one of the most important types of diagrams in UML. Class diagrams model classes and how they are related.

Very careful analysis is required to create good class diagrams. A good starting point is to take a description of the problem, or a statement of requirements, and look for the nouns in it. Once you have created a basic list of classes, it is best to start arranging your model starting with the classes that are the most central or important to the system.

Class modeling can proceed by creating an initial set of associations among the classes, and adding attributes and generalizations. You then assign responsibilities to the classes and derive operations from these responsibilities. This whole process should be performed iteratively.

Any class diagram should be subjected to detailed review. A key thing to ensure is that all generalizations are valid – that is, they follow the *isa* rule and everything in the superclass makes sense in subclasses. Other common types of error are poor naming of elements of the diagrams and incorrect multiplicity.

Creating good class diagrams is a central skill in modern software engineering, but it takes time to become an expert. We suggest practicing with many examples, and implementing your models. The process of implementation – actually getting a system to run – will help you heighten your awareness of potential flaws.