

As the data model begins to develop, situations will sometimes arise where we find that a class may not describe our possible objects as neatly as we might like. We might find that we have some objects for which some of the attributes do not really apply. For example, if we have a class to record information about all the people associated with a company, we might find that some have hourly pay rates while others have annual salaries. In many respects, much of the information about each of the employees is similar, but there are differences. We may also come across the case where we have started with two separate classes, for example **Lecturers** and **Students**, and then begin to realize that there is a great deal of information in common or that they are involved in the same relationships (who has parking permits, say). How do we handle these “same only different” cases in a pragmatic way?

Some questions that are useful to keep in mind are:

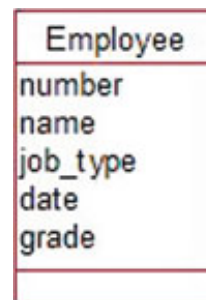
Do the two classes have enough in common to reconsider how they are defined?

Are some of the objects in a given class different enough from other objects to warrant reconsidering how they are defined?

Classes or Objects with Much in Common

Consider a company wishing to keep information about its employees. For all employees it needs to keep employee numbers, names, contact addresses, and job type, but depending on the type of job, the rest of the information might be different. Administrators might have a grade and technicians might have a date their certification needs to be renewed. Some workers might have a yearly salary, while others might have an hourly rate.

Let’s just take a simple case of an outsourcing company keeping information about administrators and technicians. We could, as a start, consider having just one class, **Employee**, as shown in Figure 6-1. In Figure 6-2, we show some possible objects of that class.








				
number: 156	number: 188	number: 196	number: 208	number: 212
name: Sue	name: Bob	name: Ann	name: Jane	name: Pat
job_type: Admin	job_type: Tech	job_type: Tech	job_type:	job_type: Admin
date:	date: 3/4/2012	date:	date:	date:
grade: A	grade:	grade: A	grade: B	grade:

Figure 6-2. Some possible objects of the class *Employee*

What are we supposed to make of Ann? She is recorded as a technician, but instead of an expiry date, she has a grade. There is a bit of confusion here now. Is she both a technician and an administrator? If she is a technician, why doesn't she have an expiry date? Or (as is most likely) has there been some sort of data entry mess-up? A database that allows for obviously inconsistent or incomplete data to be entered is not going to deliver accurate or reliable information. We could have added some constraints to our use case description on maintaining the **Employee** data (e.g., if **job_type** = Tech, then **grade** must be empty and **date** must have a value), but this is quite messy and can only become more and more complicated as other job types are added. We could contemplate removing **job_type** altogether on the grounds that we can infer the type of job from the presence or absence of an expiration date or grade. We can deduce that Jane is an administrator even though the **job_type** field is empty. However, if we remove the **job_type** field, what can we deduce about Pat? At the moment, we know that she is an administrator whose grade is currently unknown or not required. Without the **job_type** field, we would know nothing.

The real question, of course, is, "Does it really matter whether we can enter inconsistent or incomplete data?" For some applications, it may not. However, if one of the objectives of the project is to be able to produce reliable statistics about the types of job and abilities of employees, clearly the simple class in Figure 6-1 is not very practical.

Specialization

The situation in the previous section is an example of *specialization*. In general, we have employees who share many characteristics, but depending on each person's job type, we may wish to keep different specialized data. Data modeling provides a mechanism for this idea through *sub-* and *superclasses*, an idea known as *inheritance*. Figure 6-3 shows a class, **Employee**, with two subclasses, sometimes called *inherited classes*, named **Administrator** and **Technician**.

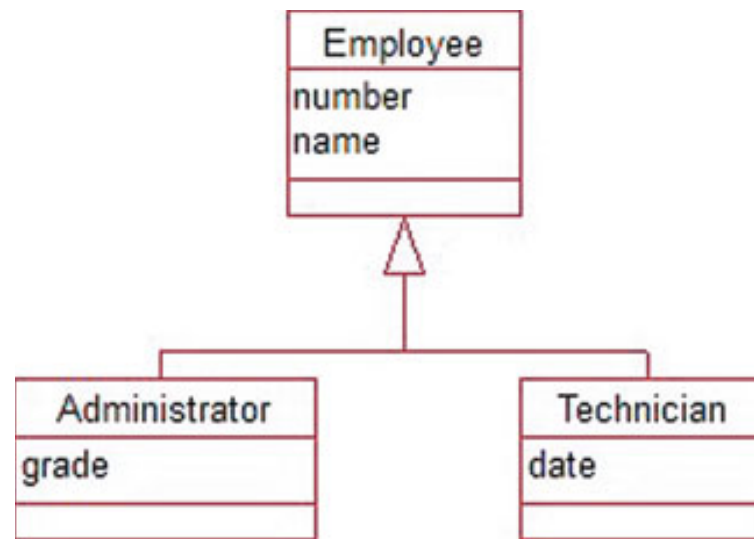


Figure 6-3. Subclasses to contain specialized information

The two classes beneath the arrow are derived from **Employee**, which means that in addition to any of their own attributes, they will also have all the attributes from the **Employee** class.

We now have three classes: objects of **Employee** will have a **number** and a **name** (and in general any other information that is relevant to all employees); objects of **Administrator** will have a **number**, **name**, and **grade**; and objects of **Technician** will have a **number**, **name**, and a **date**. Some possible objects are shown in Figure 6-4.

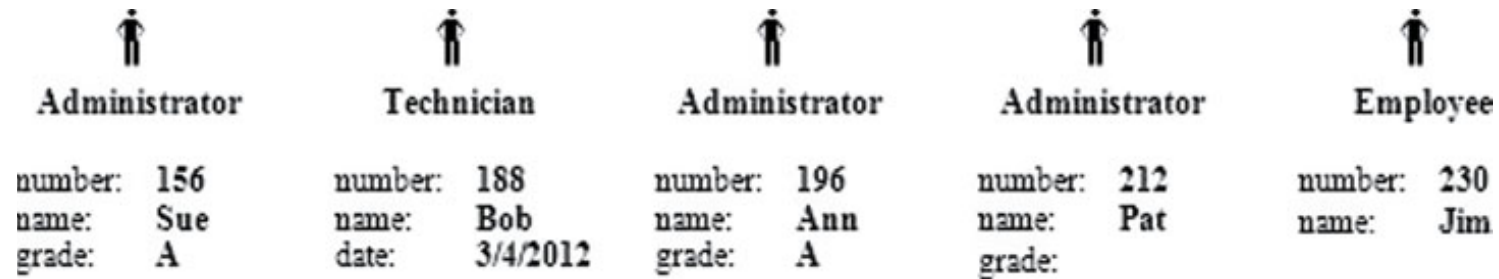


Figure 6-4. Some objects consistent with the model in Figure 6-3

Each object is of one of the three classes: **Employee**, **Administrator**, or **Technician**. There is now no possibility of having a technician with a grade. It is, however, possible to have an employee such as Pat who is an administrator with an unknown grade. We also have an employee who is neither an administrator nor a technician (although we will see later in the chapter that this is not recommended).

With this model, we are able to keep accurate information about the different types of employees and the specialist data associated with their jobs. If we need to keep information about other types of employees, we can simply add more subclasses. For example, we might find we need to add another class to keep information about electricians and their registration numbers.

The ability to add new classes without disrupting the data kept about existing classes is very important for creating software that can evolve as situations change. In software engineering it is known as the Open Closed Principle: *Software entities (e.g. classes) should be open for extension and closed for modification*. Looking at Figure 6-3, this means that we should not alter the top class **Employee** once we start storing data. Any changes to that class will affect all the existing subclasses (**Technician** and **Administrator**) and that may have an impact on any applications that use them. However, we can extend the problem by adding additional subclasses (such as **Electrician**). This means that the top or parent class must be designed very carefully at the start; it should be as general as possible.

Generalization

A model using classes and subclasses is also useful when we start with two distinct classes and find that they have some behavior in common. Let's consider a database such as in Figure 6-5, with information about lecturers and students and the courses they teach or enroll in.

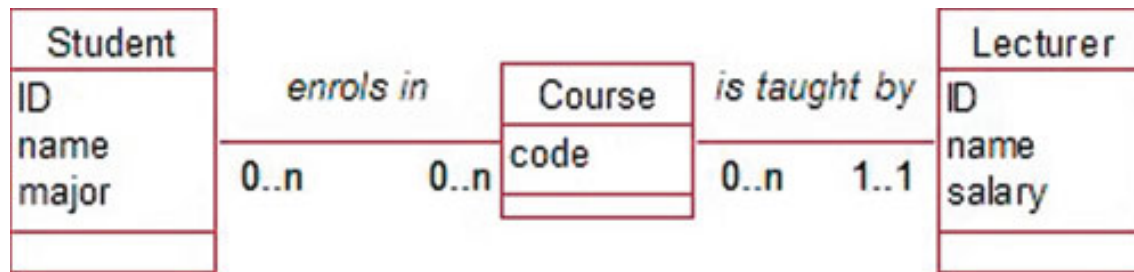


Figure 6-5. Lecturers and students as independent classes

The university may, as all universities do, have a parking problem and decide that each person is allowed one and only one designated parking space. If we wish to include this information in our model, we may try a solution as in Figure 6-6. However, we run into a real problem pretty soon. Can you see what it is?

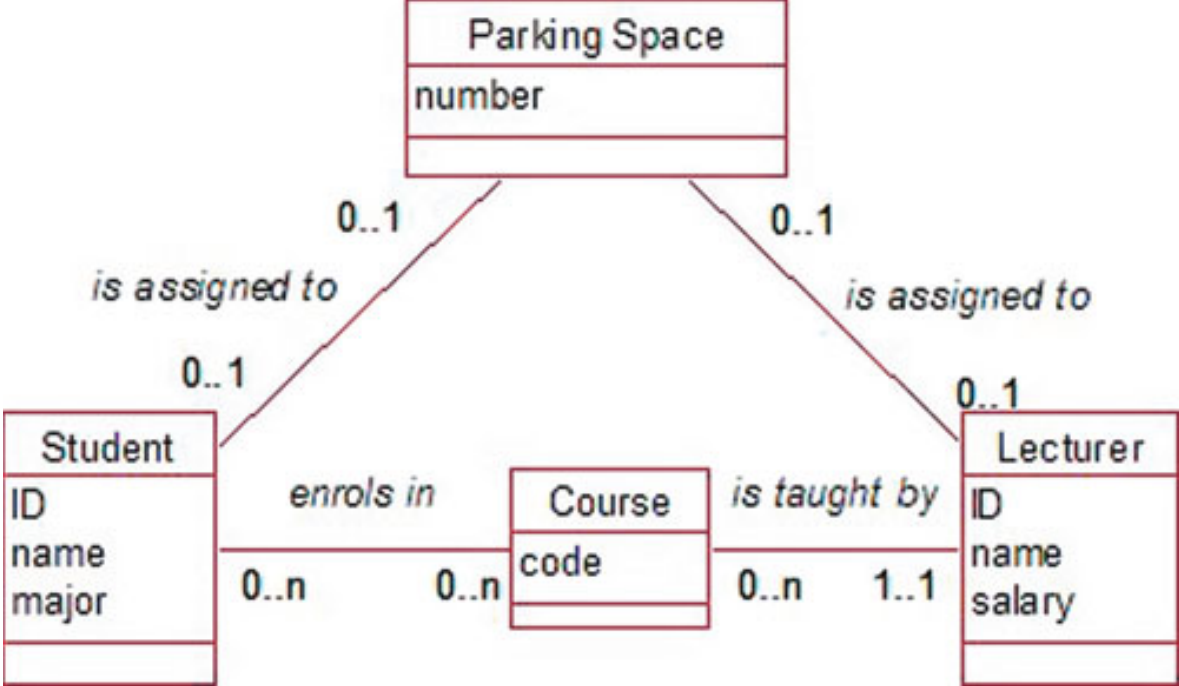


Figure 6-6. Possible model for maintaining car parking information

Reading the model from the bottom to the top, we have that students and lecturers can each have at most one parking space. That is fine. However, from the top to bottom, we have that a parking space could be assigned to a student, and the same space could be assigned to a lecturer. What the model doesn't show is that a single parking space cannot be assigned to both a student and lecturer simultaneously.

We have come across constraints on particular objects before, and we could specify these in the use cases for maintaining the data. However, we have a more elegant solution here. In many respects, our **Lecturer** and **Student** objects have the same behavior—they are assigned parking spaces. They also have attributes in common—they each have a name and an ID. We can capture this common data and behavior by creating a superclass, as in Figure 6-7.

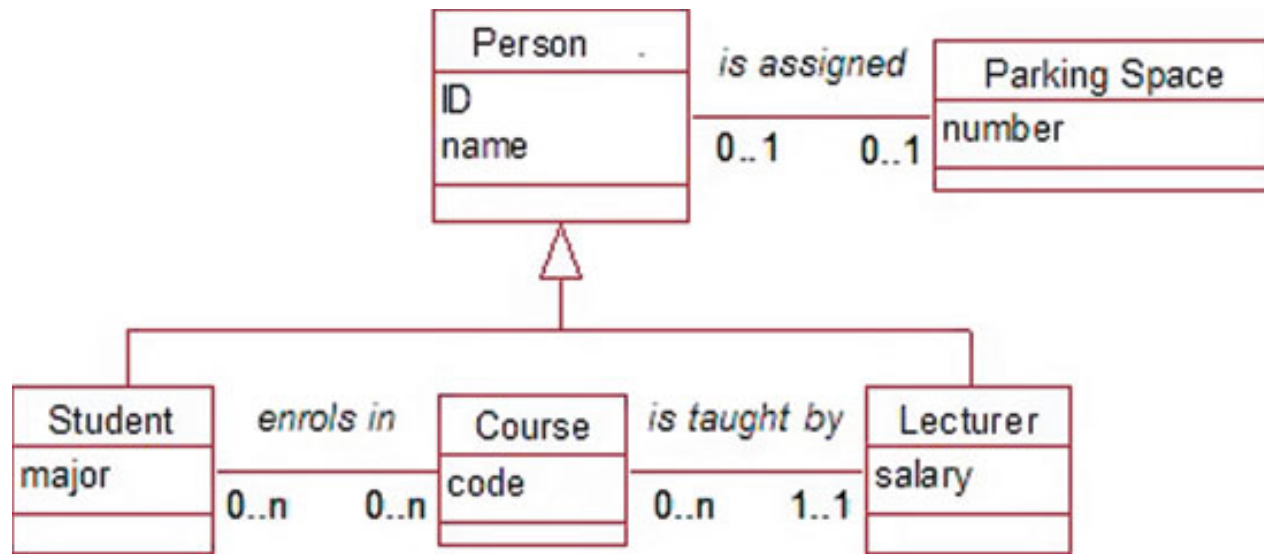


Figure 6-7. Common behavior captured in a superclass

In this model we have people, with IDs and names (and other common attributes), who can each be assigned a parking space. Students are people with a major who enroll in courses, whereas lecturers are people with a salary who teach courses. We do not have the problem now of extra tricky constraints because a parking space cannot be assigned to both a lecturer *and* a student; we just have that a parking space is assigned to one person.

Inheritance in Summary

Specialization and generalization in the examples we have looked at so far in this chapter are just two sides of the same coin. They both lead to the type of generic data model shown in Figure 6-8.

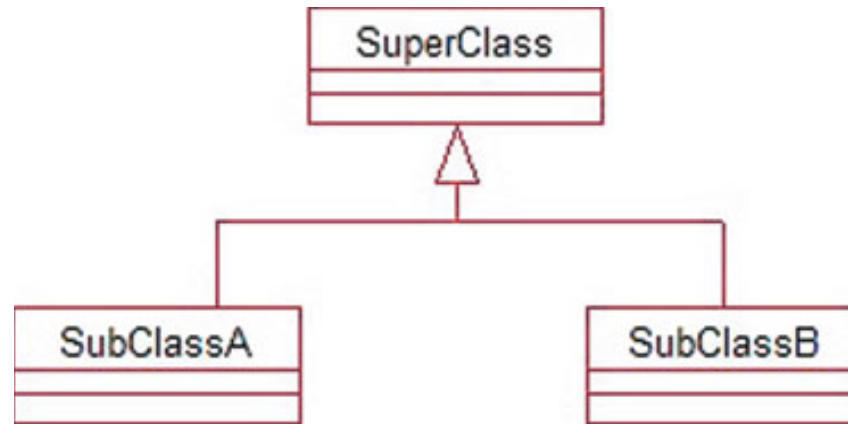


Figure 6-8. *A data model showing inheritance*

SubClassA and **SubClassB** are both specialized types of the class **SuperClass**. They will have all the properties of the **SuperClass** and in addition have their own specialized attributes and/or relationships with other classes.

Whenever you find yourself thinking things like, “But a parking space could be associated with a lecturer OR a student,” or, “A booking could be for an individual OR a company,” and so on, consider a superclass to capture the common behavior.

When you find yourself thinking, “Some objects will have a value for this attribute but not that one,” or, “Only some objects of this class will have a relationship with an object of that class,” you should consider creating some subclasses to capture that specialist behavior.

To check whether inheritance (subclasses and superclasses) is applicable to a given problem, you should ask the following questions. For example, to check whether **SubClassA** is really a subclass of **SuperClass** in Figure 6-8, ask:

*Is an object of **SubClassA** a type of **SuperClass**? (always/sometimes/never)*

*Is an object of **SuperClass** a type of **SubClassA**? (always/sometimes/never)*

If the answer to the first question is “always” and the answer to the second is “sometimes,” the problem is a good candidate for an inheritance model. For example, we can check the validity of Figure 6-3 by asking:

Is an administrator a type of employee? (always)

Is an employee a type of administrator? (sometimes)

These answers mean that making **Administrator** a subclass of **Employee** is possible.

Asking the always/sometimes/never questions can help make sense of complicated problem descriptions. Say we have a complex employee hierarchy with administrators, agents, salespeople, and so on. Those two always/sometimes/never questions can sort things out. Let’s say we discover that:

An agent is always a salesperson, and a salesperson is always an agent.

We know that for this particular situation “salesperson” and “agent” are two different words for the same thing. We should have one class called either **Salesperson** or **Agent**.

However, say we have that:

A salesperson is always an agent, and an agent is sometimes a salesperson.

Here we have good grounds for considering a **Salesperson** class as a subclass of **Agent**.

When Inheritance Is Not a Good Idea

Inheritance in a data model is not as common as you might think at first. Humans are very good at categorizing things into hierarchies, and once people get a hold of the idea of inheritance in data modeling, there can be a temptation to use it everywhere. In the last section, I was careful to say that an affirmative answer to the question “Is A a type of B?” only meant that using inheritance *might* be a possible way of making sense of a problem. In this section, we will look at a couple of examples where inheritance is definitely not a good way to think about a problem.

Confusing Objects with Subclasses

Consider a database of dogs of different breeds. We may have a hierarchy of breeds and might at first sight think that inheritance is a possibility. Consider the following statements:

- A Corgi is a dog.
- Rover is a Corgi.
- Quin is a Labrador.
- A Labrador is a dog.

While the four statements are similar, they do not all suggest subclasses. Rover and Quin are not classes; they are *objects* of some class of dog. Corgi and Labrador, on the other hand, could possibly be subclasses of some super Dog class, but then again maybe not. Let's consider how we know whether something is an object or a class. Why is Rover probably an object and Corgi possibly a class?

A quick way to help decide whether something is a class or an object is to ask a question such as, "Am I likely to have several of <whatever> and am I interested in them as a group?" For example:

Am I likely to have several Corgis and am I interested in them as a group? Probably. Therefore Corgi is a potential class.

Might I have several Rovers and am I interested in them as a group? There might well be several dogs called Rover, but it is hard to think of why we would be interested in them as a group just because of their common name.

Corgis and Labradors are potential classes, whereas Quin and Rover are more likely to be objects of one of our dog classes. A possible hierarchy and some objects that are consistent with the preceding statements are shown in Figure 6-9.

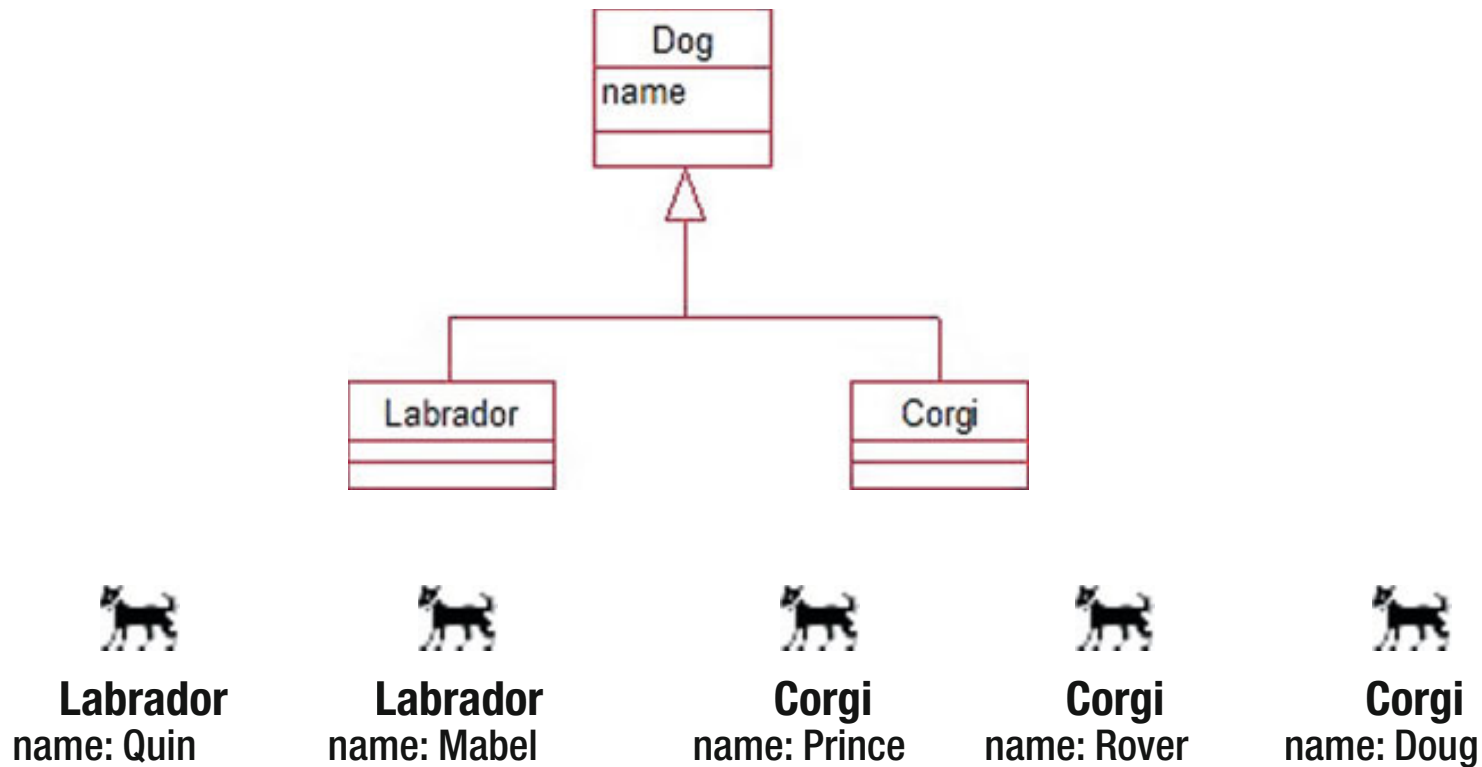


Figure 6-9. Some possible classes, subclasses, and objects of a dog data model

Confusing an Association with a Subclass

The model in Figure 6-9 may look fine for a start, but in fact we don't need inheritance to maintain simple information about the different breeds of our dogs. We are not keeping any different information about Labradors than we are about Corgis or any other breed (so far). We are merely noting that some of our dogs are Corgis and some are Labradors, and this can be done with a simple association between our **Dog** objects and objects of another class called **Breed** as shown in Figure 6-10 (assuming pure-bred dogs for now).

The model in Figure 6-10 is a much simpler way of representing our problem than the model in Figure 6-9. The resulting database will be much easier to maintain also. In Figure 6-9, if we add a new breed, we need to add a new *subclass*. For the model in Figure 6-10, we just need to add another *object* of our **Breed** class.

What if the problem changed to say that we want to keep the fees payable to the kennel club and that these are different for the different breeds (e.g., a Labrador will cost \$100, a Corgi \$80, and a Terrier \$85)? Now that we have some different information about the breeds, should we reconsider specialized classes?

No. What we have here are just different *values* for an attribute, **fee**, which can easily be accommodated in the **Breed** class. We only need to consider specialized classes if we have different attributes or relationships (not just different values for an attribute).

When Is Inheritance Worth Considering?

We have seen that what looks like inheritance can often be represented more simply (and effectively) by simple relationships. At what point is it worth considering inheritance? Let's think of another scenario for our dog model.

Let's say the town council keeps a register of dogs. Some of these dogs are just your plain old family pet, while others might be show dogs with affiliations to kennel clubs. If (big if) the council wanted to keep this information, a model such as the one in Figure 6-11 might be worth considering.

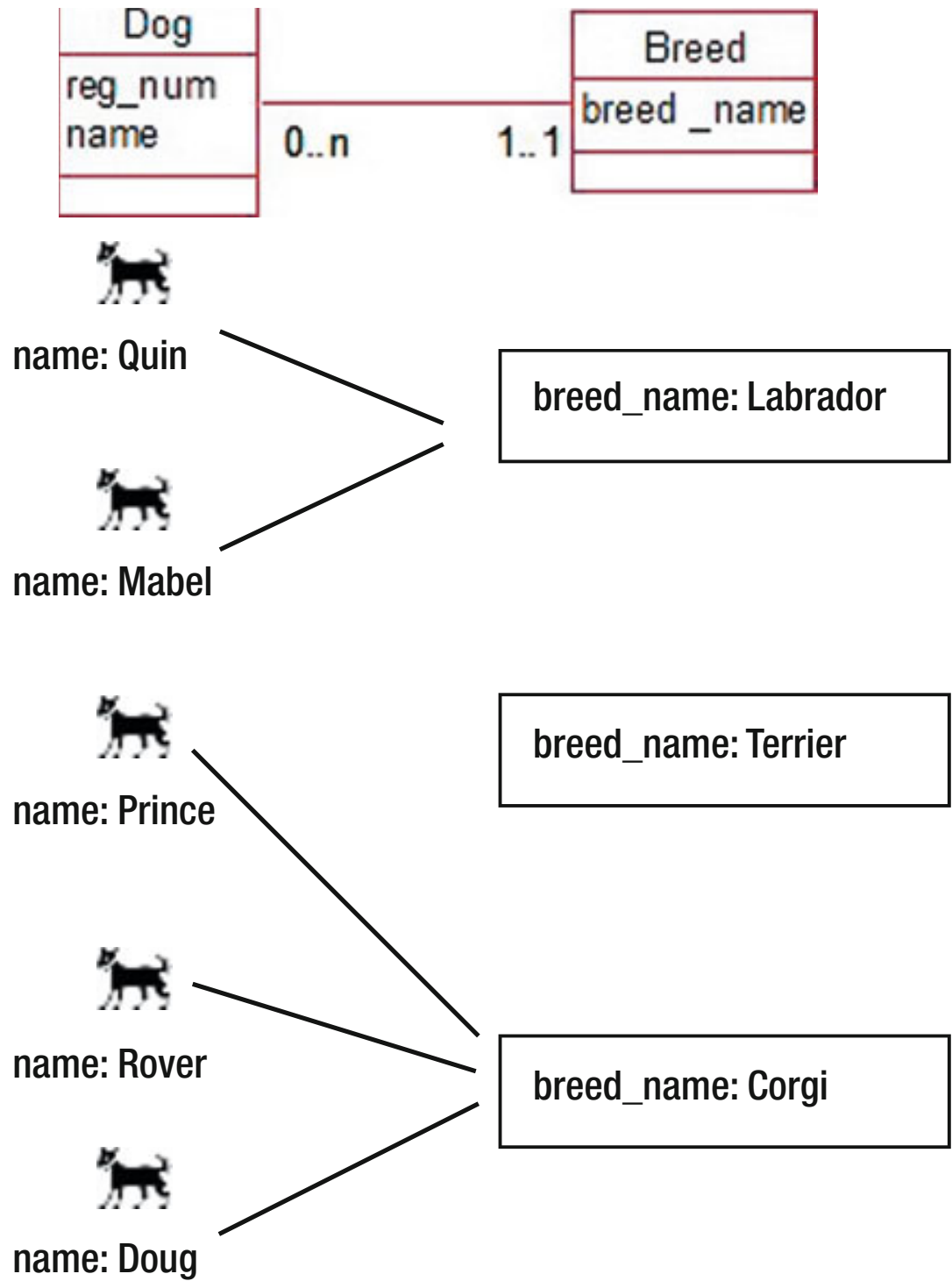


Figure 6-10. Each Dog object is associated with a breed.

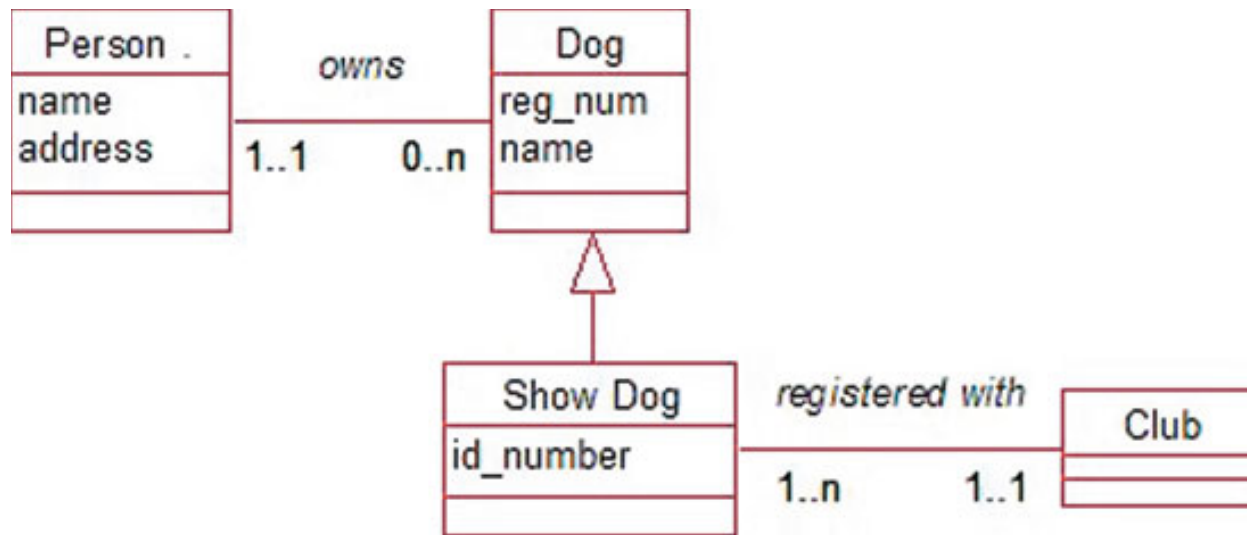


Figure 6-11. Possible model using inheritance to show different behavior

In Figure 6-11, we see that show dogs have not only additional attributes (e.g., an **id_number** to perhaps point to records of their genealogy), but also different associations (i.e., a show dog will be registered with a kennel club whereas an ordinary pet will not).

How else could we have modeled this? Well, we could have given all dogs an **id_number** attribute (that could be left unspecified for ordinary pets) and let all dogs have an optional relationship with a club as shown in Figure 6-12. Can you see any drawbacks to this model?

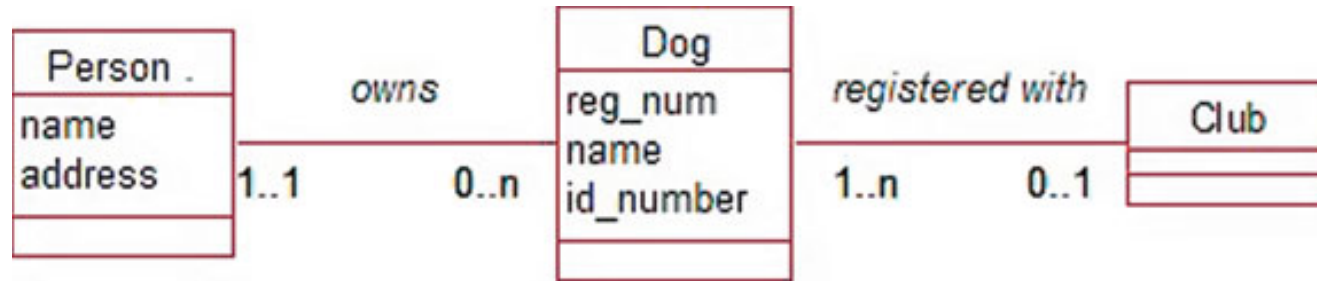


Figure 6-12. Possible model without using inheritance

It is possible to capture all the required information with the model in Figure 6-12, but it is not so easy to keep the data accurate. We run into much the same problems as we had with our model of **Employees** in Figures 6-1 and 6-2. What about dogs with no **id_number** that are associated with a club and vice versa? Are these show dogs, or has there just been a data entry mishap?

The decision of whether to use inheritance or not depends on how important the completeness and accuracy of the data are to the objective of the project. We get right back to the questions we considered in Chapter 3. What is the main objective? What is the scope? How important is the accuracy of this data? On the whole, when you are starting out on a problem, it is best to keep your solution as simple as possible. Inheritance provides an elegant solution to many problems involving specialization and generalization, but you should only use it when it is necessary.

Should the Superclass Have Objects?

In Figure 6-11, we had a class of dogs with show dogs as a subclass. The implication here is that your ordinary old pet will be an object of the superclass **Dog**, while show dogs will be objects of the subclass. This can lead to a few problems as the project evolves.

As we have seen, we should only be considering inheritance when we have objects with specialized data that needs to be accurately maintained. We need to make sure that the model we develop will be able to cope with changes or additions to the scope in the future.

Consider a library that sets up a small database to maintain information about books (catalog number, title, author). After some time it includes audio books in its collection. An audio book has all the same attributes as an ordinary book, but in addition it has a playing time. This seems like a reasonable candidate for setting up a specialized subclass, and we may arrive at a model as in Figure 6-13.

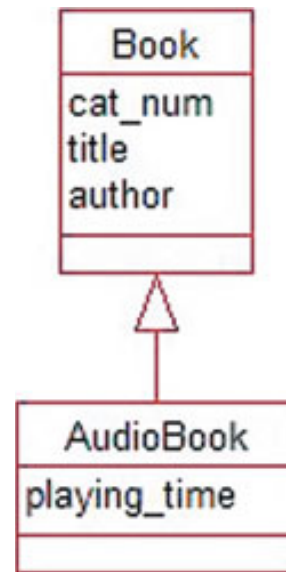


Figure 6-13. A specialized class for audio books

If we set up a database based on this model, we will probably have some objects of type **Book** (with a value for number, title, and author) and some objects of type **AudioBook** (with a value for number, title, author, and playing time). Some time later, the problem may change and we may want to keep some additional information about our regular books—say, number of pages. We now have a problem. Our audio books are objects of the superclass: if we add an attribute **page_num** to the parent **Book** class, it will be inherited by our **AudioBook** subclass objects, which is not what we want at all.

This problem can occur when a parent class of a data model has objects. It is generally advisable that any class which has subclasses should be an *abstract* class, meaning that it cannot have objects. The idea of an abstract class is to keep information that is common to its subclasses and which the objects of those classes can inherit. Following this advice we would have had an abstract class (say **Item**) at the top of our hierarchy with two subclasses, **Book** and **AudioBook**, as in Figure 6-14. This way we can make changes to each of our subclasses without affecting the other.

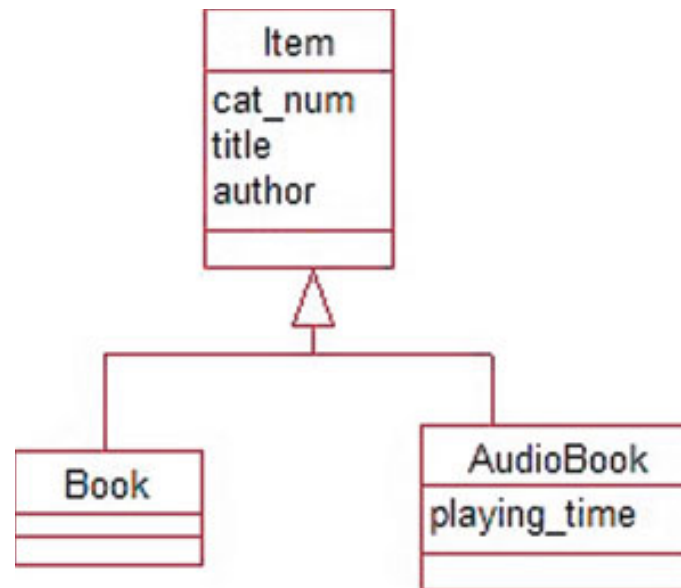


Figure 6-14. *Item as an abstract class*

Figure 6-14. Item as an abstract class

It does not matter that **Book** might have no additional attributes originally. The class is there, ready for when we want to make a change. We can now add the number of pages to the **Book** class without it affecting the **AudioBook** class.

Objects That Belong to More Than One Subclass

In most of the examples in this chapter, the problems have been very simplified. In Figure 6-15, we have a model with **Lecturer** and **Student** represented as subclasses of a **Person** class, along with some objects of the two subclasses. We see that for this case our objects have to be either a lecturer or a student.

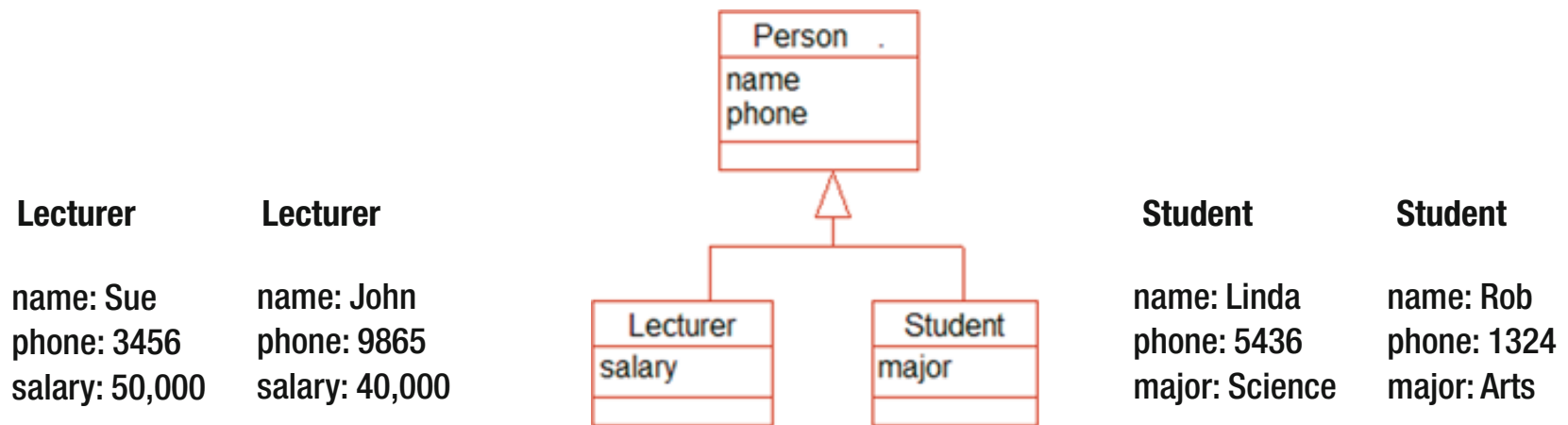


Figure 6-15. Students and lecturers are distinct.

Figure 6-15 copes with the simple case of lecturers and students being distinct. What is more likely, however, is that there is some overlap between the two. What if lecturer John is also doing some part-time study for an arts degree and student Linda is doing some part-time teaching to fund her fees? Where do we store John's major and Linda's salary?

We could make another two objects, an additional **Student** object for John and a **Lecturer** object for Linda, but there are problems with this approach. We would now have six **Person** objects in total when in reality there are only four people. Any counts or summaries of numbers of people will be inaccurate. An additional problem is that we would have two objects for Linda, and they will both have values for **name** and **phone**, causing problems when Linda's contact details change. They will have to be updated in two places.

One solution is to consider another class that inherits from both **Student** and **Lecturer**. Inheriting from two different parents is sometimes referred to as *multiple inheritance*. Objects of our new **Lecturer/Student** class will have attributes **name**, **phone**, **salary**, and **major** as in Figure 6-16.

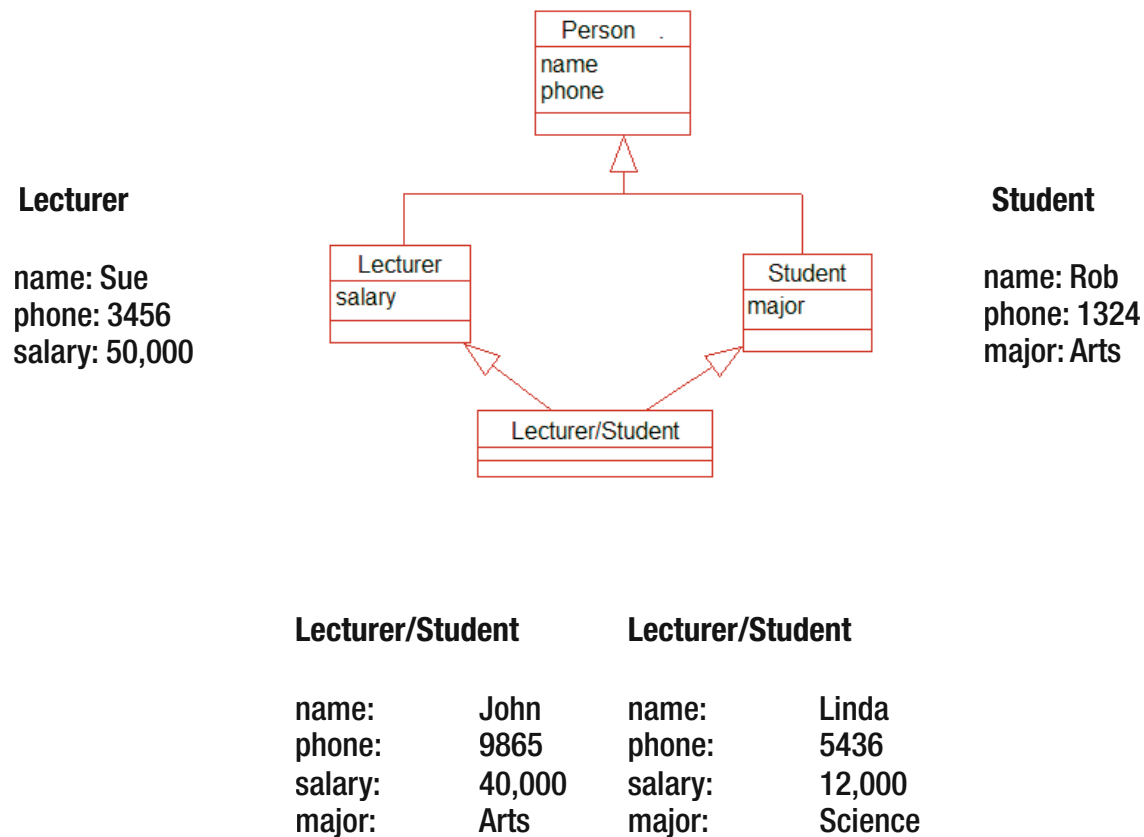


Figure 6-16. Multiple inheritance to capture objects of two classes (not recommended)

Figure 6-16. *Multiple inheritance to capture objects of two classes (not recommended)*

There are difficulties with the approach in Figure 6-16. The obvious problem, from a purely pragmatic design point of view, is that when more classes are added at the middle level, we will be in trouble. If we add new classes (e.g., **Administrator** and **Cleaner**) as additional subclasses of **Person**, we will need to add a whole slew of subclasses at the bottom level to cope with all the possible combinations. For example, administrators who do some lecturing, cleaners who do some study, poor students who do a whole raft of extra jobs to fund their studies, and so on. This approach very soon gets out of hand.

Our problem is that we have been thinking of students and lecturers as different types of people when in fact they are all just people doing different things. A better way to think of this type of scenario is not so much that there are different types of *people*, but that there are people who play many different *roles*. We can model these jobs or roles as a class with many different subclasses for the jobs about which we need to store different information. Rather than have subclasses of **People**, we can have another class (let's call it **Contract**) that has subclasses for the different roles we need to describe. Each person can then have many contracts as shown in the model in Figure 6-17.

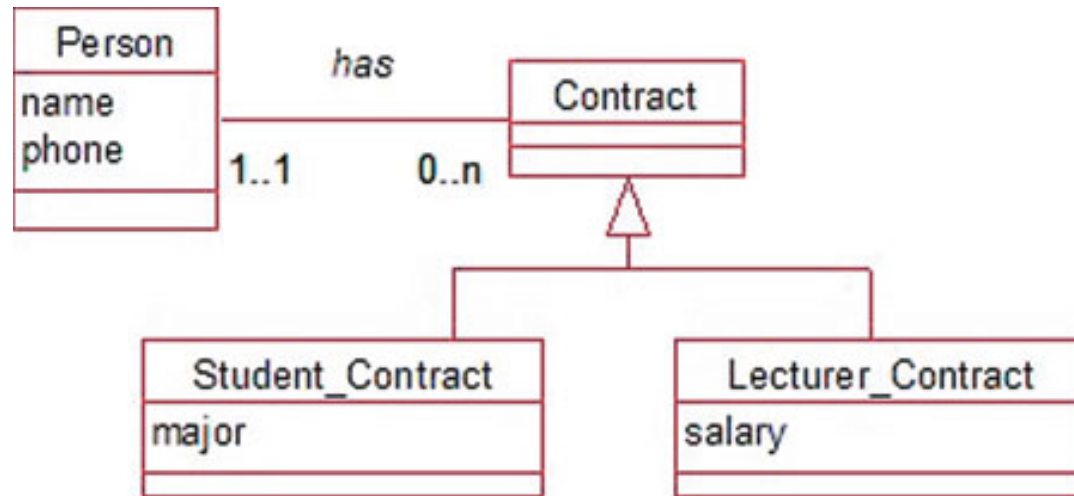


Figure 6-17. *People can have many roles (contracts)*

Some objects of a data model like Figure 6-17 and consistent with the scenario in Figure 6-16 are shown in Figure 6-18. We see that we very clearly have four people who are undertaking a number of different roles. A person can have more than one contract, and each contract is associated with a single person.

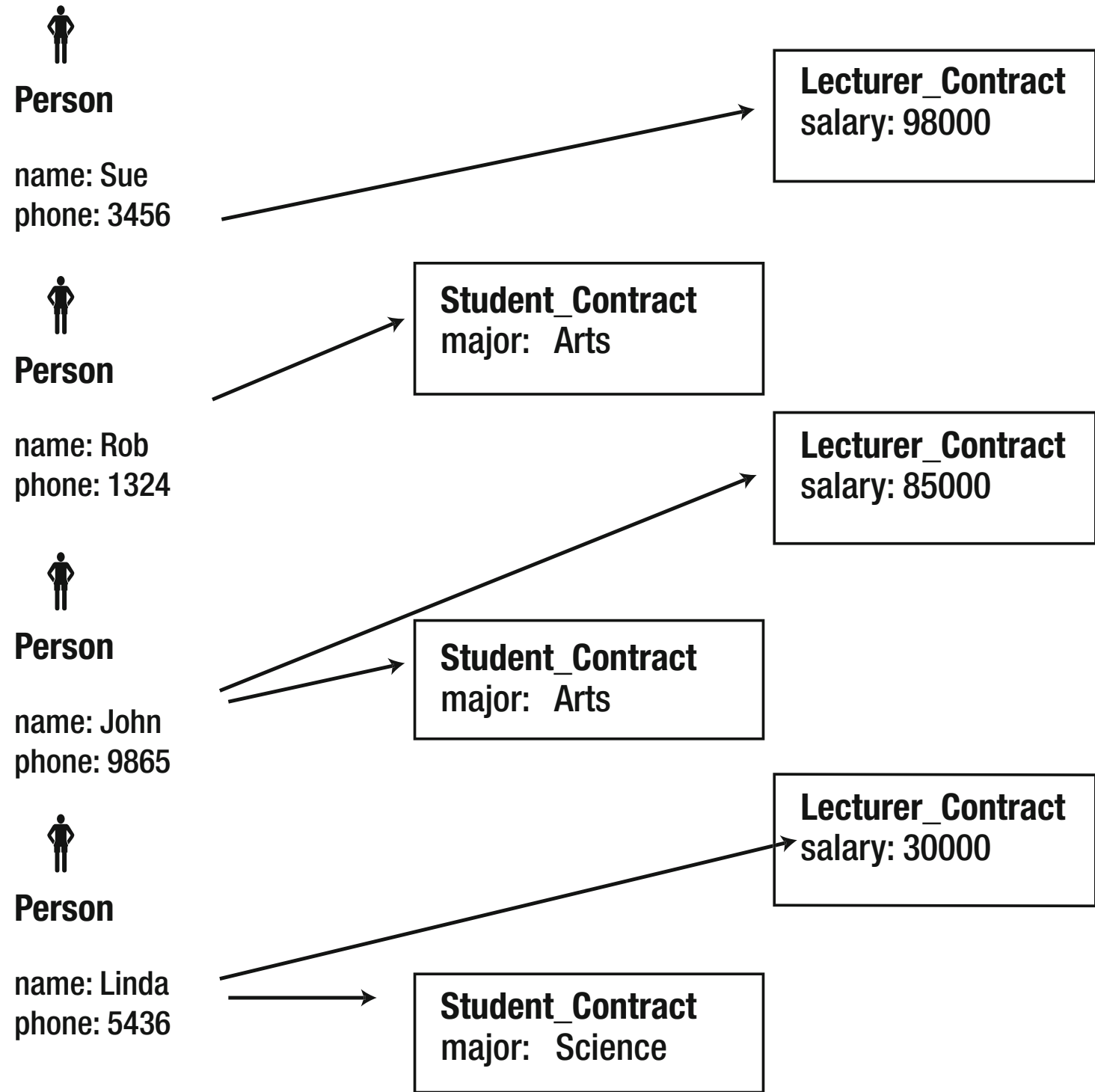


Figure 6-18. Using roles (contracts) as an alternative to multiple inheritance

The approach in Figure 6-17 is easy to adapt when new roles are added (e.g., administrator, cleaner). If we wish to include administrators with information specific to their contracts, we just add another subclass, **Administrator_Contract**, to our **Contract** class.

There is a slight problem, however. In Figure 6-17, we have that a person can be related to many contracts, but we don't have any constraints as to which type of contracts. The model does not prevent Linda being associated with several **Lecturer** contracts and/or several **Student** contracts. In reality, this may be what the problem actually requires. Linda may follow her arts degree with a science degree. John may be promoted and take out a new contract for \$120,000. We can add some date attributes to the parent **Contract** class so the contracts can be recognized as being in succession or overlapping. Overall, the model in Figure 6-17 is very flexible and allows us to address many complications in a transparent manner.

However, we need to be sure the objectives of our problem require this sort of accuracy about people and the roles they undertake. If the objective is to keep reliable statistics about different types of employees, their pay, and their qualifications, then this sort of model is necessary to help us understand what is going on. If that information is of only secondary importance (i.e., our main objective is keeping student enrollments and results), then maybe we do not need to introduce subclasses to keep the specialized data about the other more minor roles that people might play.

Composites and Aggregates

Often we have objects that are made up of other objects: a forest is made up of trees or a building is made up of rooms. Some people use a special notation for this type of relationship, but I've never found it particularly helpful to do so.

When dealing with situations in which we have this type of composition or aggregation of objects, inheritance can become useful. Consider a building that has a number of rooms, as in Figure 6-19. Every year the building must undergo a safety check. Occasionally an individual room will need to be checked. We can have a **Check** class which records the date. Should the **Check** class be associated with the **Building** or the **Room** class, or both?

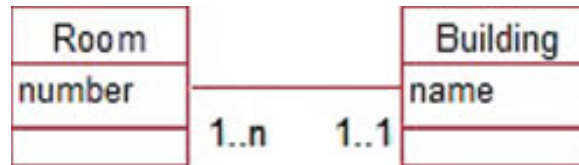


Figure 6-19. A building has many rooms.

For the purposes of this problem a **Room** and a **Building** have something in common. They both can be associated with safety checks. This therefore becomes a candidate for an inheritance solution. A first attempt is shown in Figure 6-20.

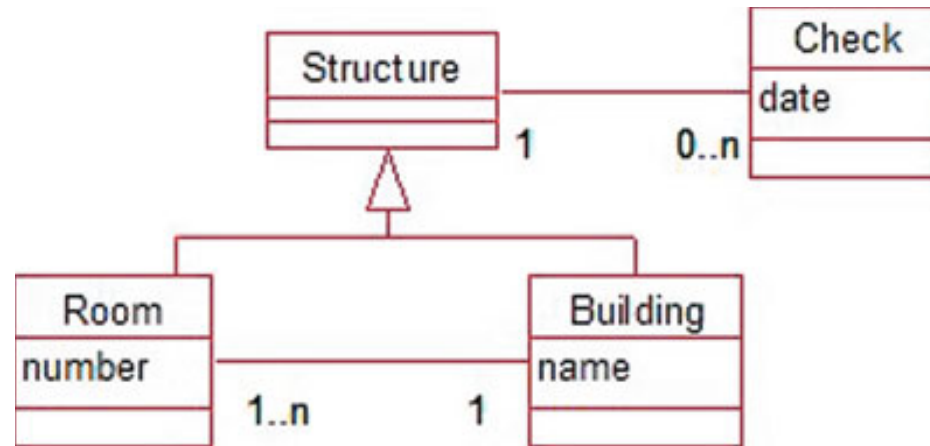


Figure 6-20. Buildings and rooms are both types of structures that may have safety checks.

Now we can store data about a single check that incorporates a building (and we know what rooms were included). However, if necessary the check can be associated with a single room. While this solution is fine, it can be made much more general by using a software pattern. There are a number of design patterns¹ that are useful for software developers. They mostly deal with behavior whereas our focus is on data. The solution in Figure 6-21 is based on the **Composite** pattern.

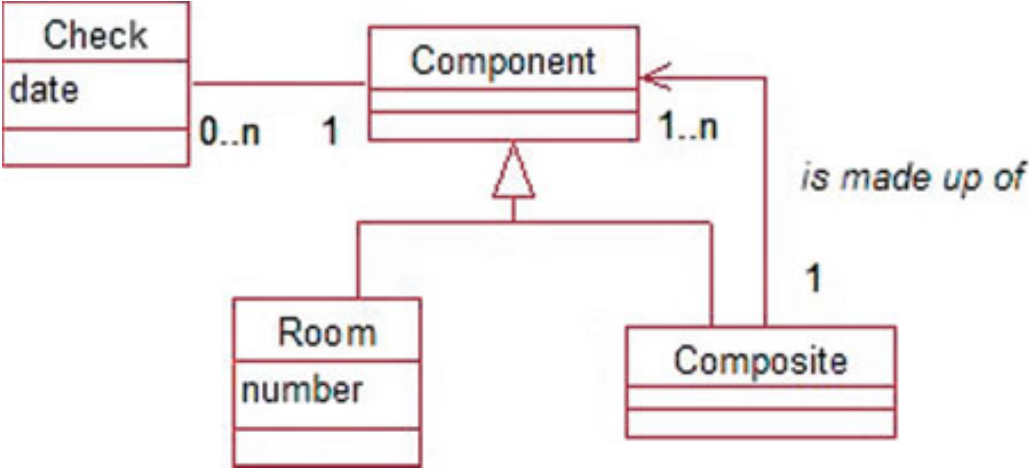


Figure 6-21. Using the Composite pattern. Checks are made on components, which may be a single room or a composite of other components.

The model in Figure 6-21 says that we have something (a component) on which a safety check can be made. That component may be an individual component (in this case a room) or a collection of other components. For instance, we can have a **Composite** object, which is a building that is made up of rooms. This would be analagous to Figure 6-20. However, our new model is more general because a **Composite** can be made up of other composites. For example: an industrial estate may be made up of buildings, each being made up of floors, which each contain rooms. With the **Composite** pattern in Figure 6-21, we can record safety checks at any of these levels (room, floor, building, etc.).

We will look at how to represent inheritance in a relational database in Chapter 7. In the meantime, the tables in Figure 6-22 show how the information might be recorded. The **Check** table shows that Component 3 was checked on 3/02/2012. From the **Composite** table we can deduce (by looking down the right hand column) that **Components** 40 and 60 belong in 3 and that **Components** 113, 115, and 117 belong in 40 or 60. The **Component** table records what all these entities are, so we know that the check on 3/02/2012 was on the Forbes Building, which contains a 4th and 6th floor, which in turn contain rooms F403, F409, and F632. On 10/02/2012 there was a check for 113 (Room F408) which has no subparts (does not appear in the right hand column of the **Composite** table).

ID	belongs_in	ID	name	date	component
2		3	Forbes Building	3/02/2012	
3		2	Burns Building	9/02/2012	
31	2	31	3rd Floor Burns	10/02/2012	
40	3	40	4th Floor Forbes		
60	3	60	6th Floor Forbes		
113	40	113	F408		
115	40	115	F409		
117	60	117	F632		
121	30	121	B341		

Figure 6-22. Database tables representing the Composite pattern

¹ Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Indianapolis, IN: Addison and Wesley, 1994.)

It Isn't Easy

Inheritance offers some wonderfully elegant ways to model very complicated problems. However, getting a hierarchy of classes and subclasses that will cope with all the eventual data is very difficult. We have only touched on the *data* aspects of inheritance here in this chapter. Dealing with inherited classes becomes considerably more difficult if we need to add *behavior* (or methods) to our classes. Adding behavior, however, is outside the scope of the data-based problems we are considering in this book.

Even for just static data, we can still run into problems when we try to design an inheritance hierarchy. Consider the model in Figure 6-23.

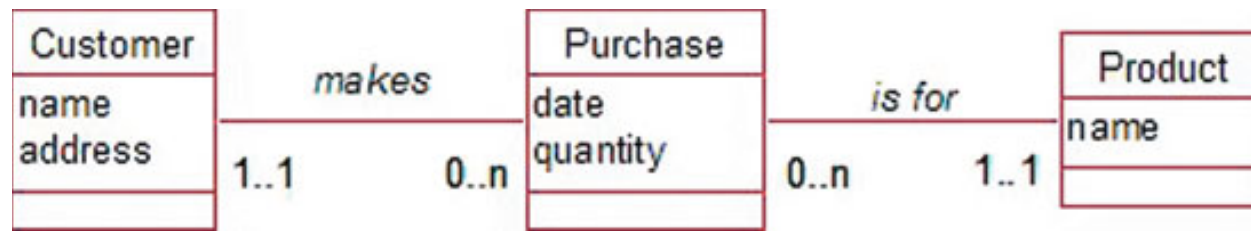


Figure 6-23. An amateur biologist's model for keeping data about animals

Dividing animals up into fish, mammals, birds, and so on may do quite well as we enter data about bears, dogs, sharks, and sparrows. But what happens when we come to whales? Whale doesn't fit at all into Figure 6-13's model. A whale is a mammal, but it also needs to be shown as living in sea water, similar to a fish. If a database has been implemented and lots of data entered, it can be very difficult to insert layers or move subclasses between layers. Getting it correct at the start is very important. Poorly thought-out inheritance can cause more problems than it solves, so use it very sparingly in database problems.

In Chapter 7, we will see how to capture the most important parts of these data models in a relational database.

Summary

Situations when inheritance is a possibility include the following:

- If different objects have mutually exclusive values for some attributes (e.g., administrators have grades but technicians have dates), consider specialized subclasses.
- When you think *this is like that except for...* consider subclasses.
- When two classes have a similar relationship with another class, consider a new generalized superclass (e.g., if both students and staff are assigned parking spaces, consider a generalized class for people).

Before you use inheritance, make sure that

- you have not confused objects with subclasses (e.g., Rover is probably an object, Dog or Cat could be classes).
- you have considered whether an association with a category class would be sufficient (e.g., Labrador and Corgi could be objects of a Breed class, and each dog could be associated with a breed).
- it is not just the value of an attribute that is different (e.g., don't consider inheritance because the fees for Labrador and Collie are different).

Other considerations:

- Classes that have subclasses should be abstract, which means they will never have any objects. This allows the problem to be more readily extended.
- Consider associations with roles when you come across the *my object is a member of both these classes* dilemma.
- Don't introduce the complexity of inheritance unless the specialized data in the subclasses is important to the main objectives of the project.

TESTING YOUR UNDERSTANDING

Exercise 6-1.

Consider the model in Figure 6-24 which describes purchases of a product by customers of a small mail order company selling toys. For simplicity, each purchase is for one or more of a single toy. Each transaction must have a customer so that he or she can be invoiced and the product delivered. The data will be used to prepare statistics about the different products sold, values of purchases, and the spending habits of customers.

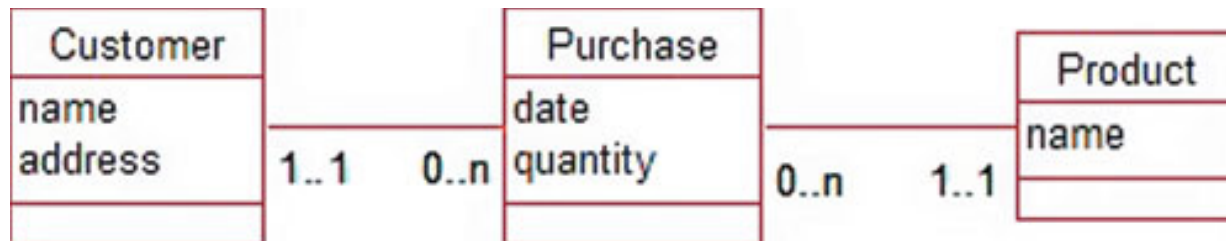


Figure 6-24. Customers purchase products.

The company changes the way it does business to allow customers to walk in off the street and pay cash. No customer needs to be associated with a cash purchase. Discuss how effective the following changes to the data would be.

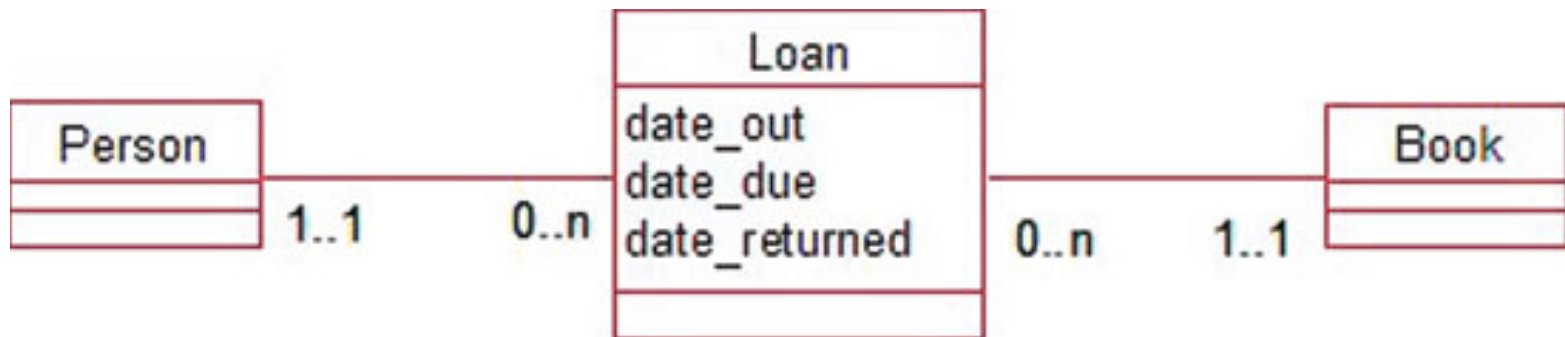
- Change the optionality at the customer end of the relationship to 0 so not all purchases need a customer.
- Leave the optionality as 1 but include a dummy customer object, with name CashCustomer.
- Create subclasses of Customer: Cash_Customer and Account_Customer.
- Create subclasses of Purchase: Cash_Purchase and Account_Purchase.

Exercise 6-2.

1. A farmer keeps information about the application of fertilizer (e.g. amount, date) to his crops. His farm is made up of large sections which are divided into fields. Usually an application of fertilizer is applied to an entire section, but occasionally it is to an individual field. How would you model this?

Exercise 6-3.

2. A volunteer library has staff, members, and books. It wants to know which books are on loan to whom, know how to contact the borrower, and charge fees for overdue books. Reference books cannot be borrowed. Members are fined \$5 per day for overdue books, but staff do not receive fines. How might you model this situation? Some initial classes are shown in Figure 6-25.



6-25. *People can borrow books.*
