9. Generalization and Specialization

- 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.

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

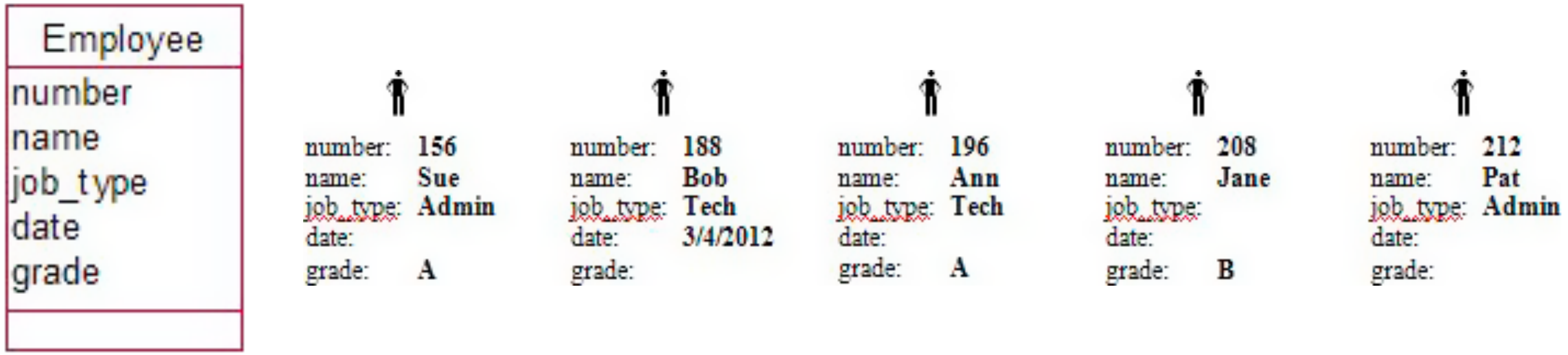
Are some of the objects in a given class different enough from other objects to warrant reconsidering how they are defined?

Today's Lecture

- Classes or Objects with Much in Common
- Specialization
- Generalization
- Inheritance in Summary
- When Inheritance Is Not a Good Idea
- When Is Inheritance Worth Considering?
- Should the Superclass Have Objects?
- Objects That Belong to More Than One Subclass
- Composites and Aggregates
- It Isn't Easy
- Exercises
- Summary

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.

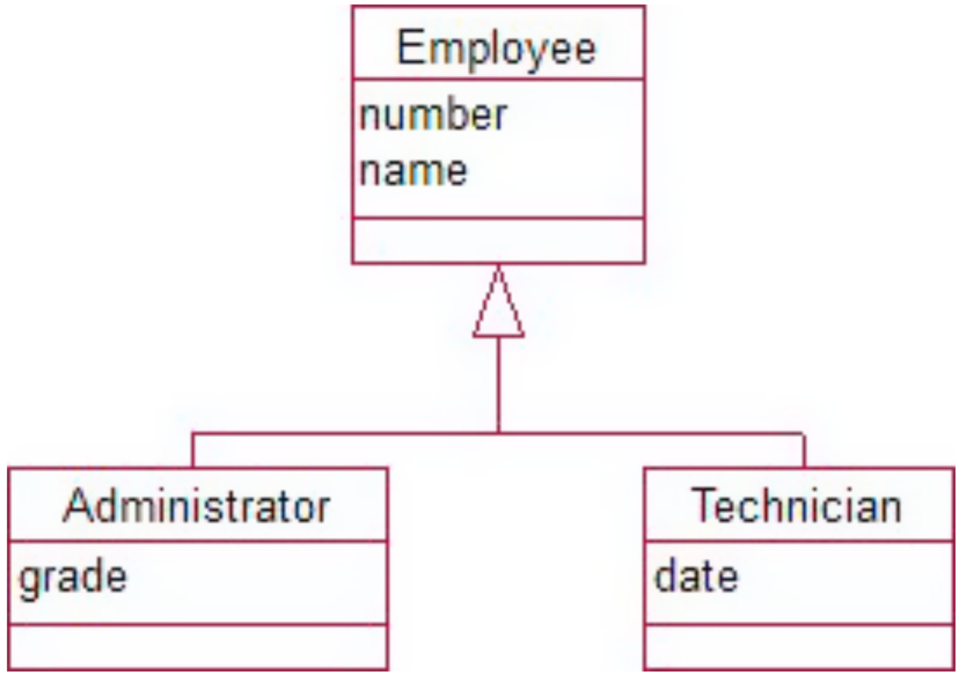


number:	**156**	**188**	**196**	**208**	**212**
name:	**Sue**	**Bob**	**Ann**	**Jane**	**Pat**
job_type:	**Admin**	**Tech**	**Tech**		**Admin**
date:		**3/4/2012**			
grade:	**A**		**A**	**B**	

- 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.
- What can we deduce about Pat?
- “Does it really matter whether we can enter inconsistent or incomplete data?”
- 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 is not very practical.

Specialization

- 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*.
- A class, Employee, with two subclasses, sometimes called *inherited classes*, named Administrator and Technician.

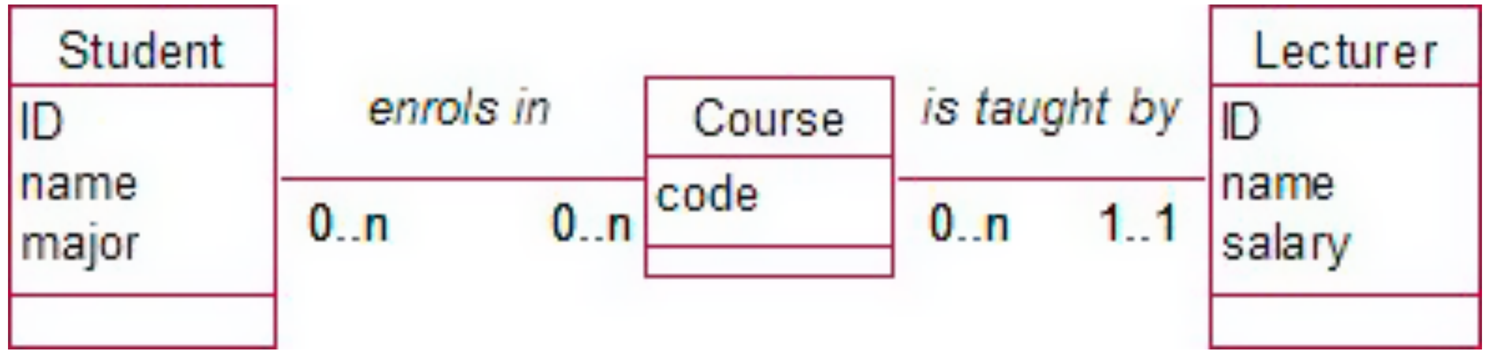


Specialization ···>

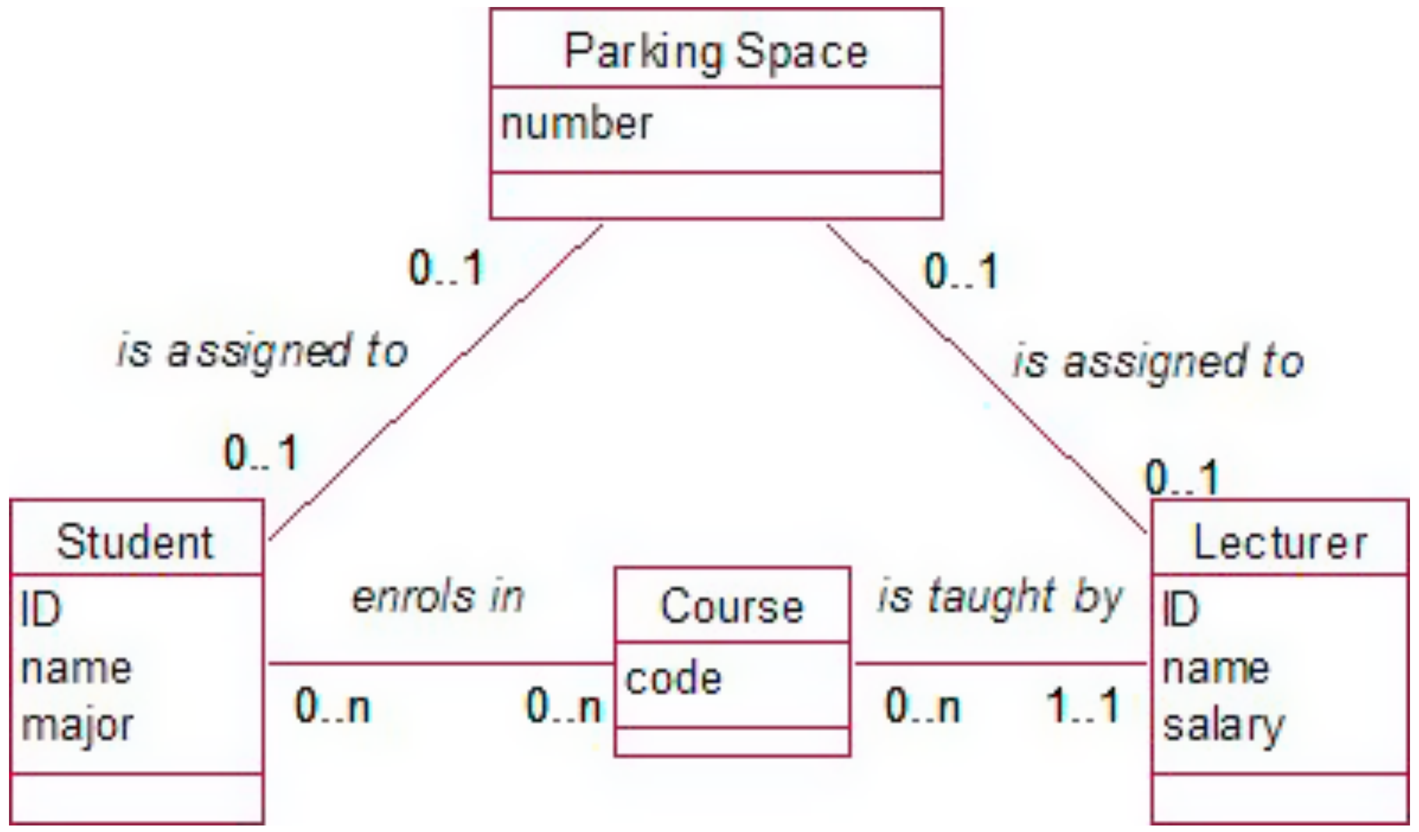
- Each object is of one of the three classes: Employee, Administrator, or Technician. There is now no possibility of having a technician with a grade.
- It is, however, possible to have an employee such as Pat who is an administrator with an unknown grade.
- We also have an employee who is neither an administrator nor a technician
- If we need to keep information about other types of employees, we can simply add more subclasses.
- 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*.
- 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. 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 with information about lecturers and students and the courses they teach or enroll in.

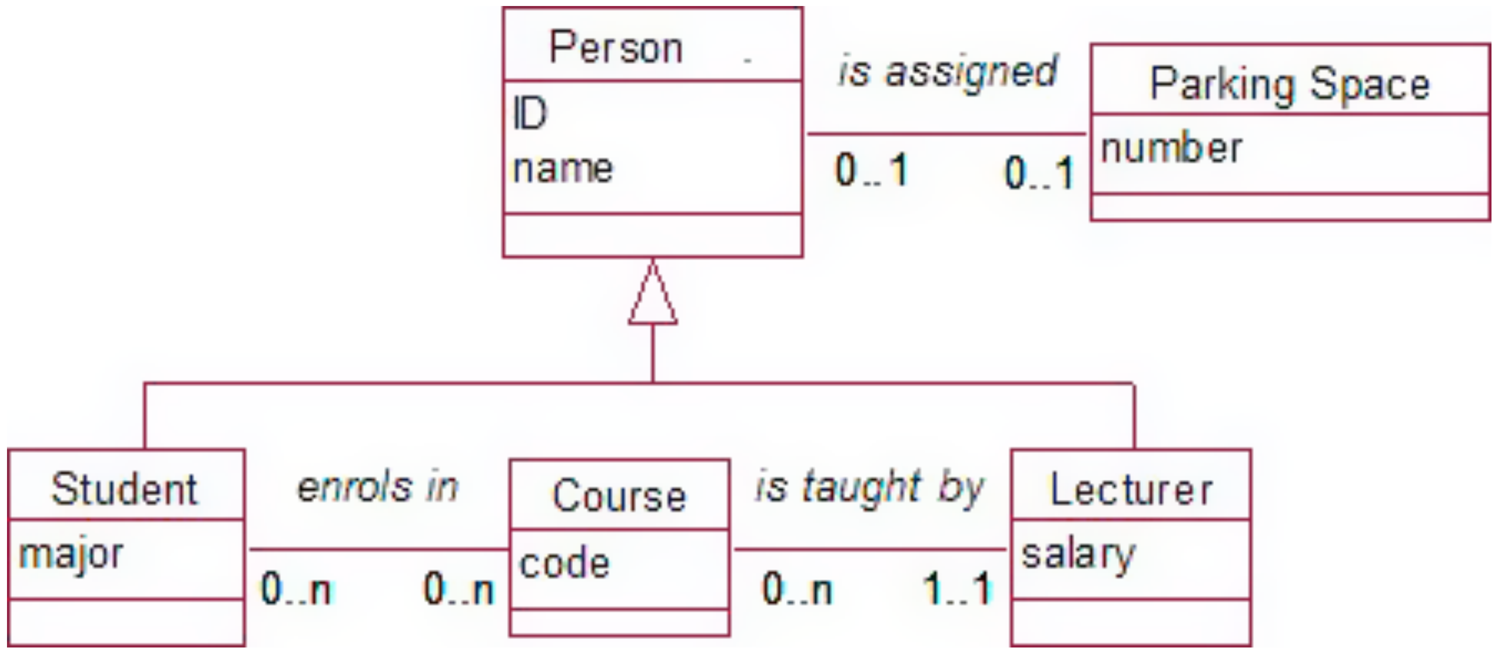


- The university may have a parking problem and decide that each person is allowed one and only one designated parking space.
- If we wish to include this information in our model, we may try a solution.



- We run into a real problem pretty soon. Can you see what it is?

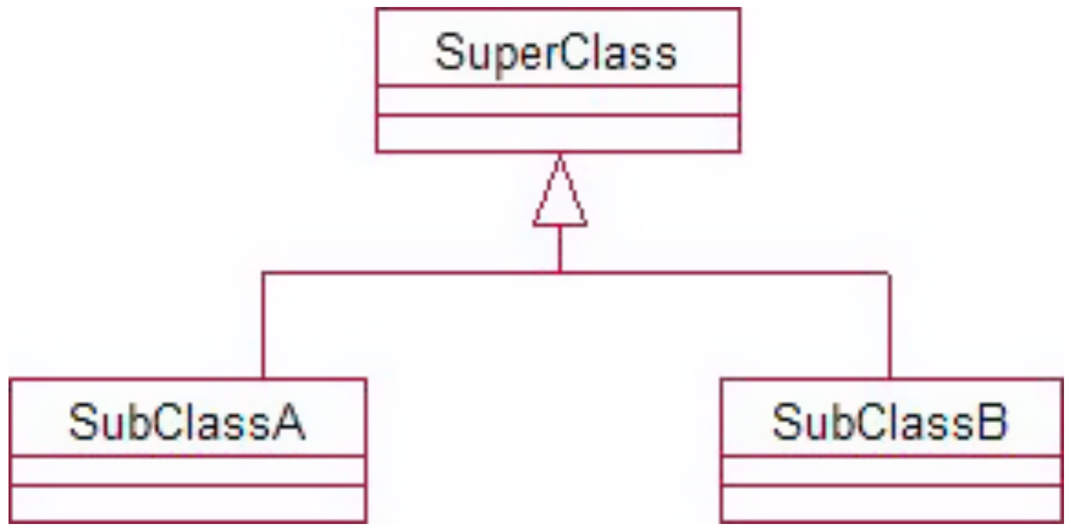
- Reading the model from the bottom to the top, we have that students and lecturers can each have at most one parking space.
- However, from the top to bottom, we have that a parking space could be assigned to a student, and the same space could be assigned to a lecturer.
- What the model doesn't show is that a single parking space cannot be assigned to both a student and lecturer simultaneously.
- 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 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 are just two sides of the same coin.
- They both lead to the type of generic data model.



- **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.
- “A parking space could be associated with a lecturer OR a student.”

- 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.

  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.

  Is an administrator a type of employee? (always)

  Is an employee a type of administrator? (sometimes)

- 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.

  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.

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

- Consider a Salesperson class as a subclass of Agent.

When Inheritance Is Not a Good Idea

- 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.
- It is 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.

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?

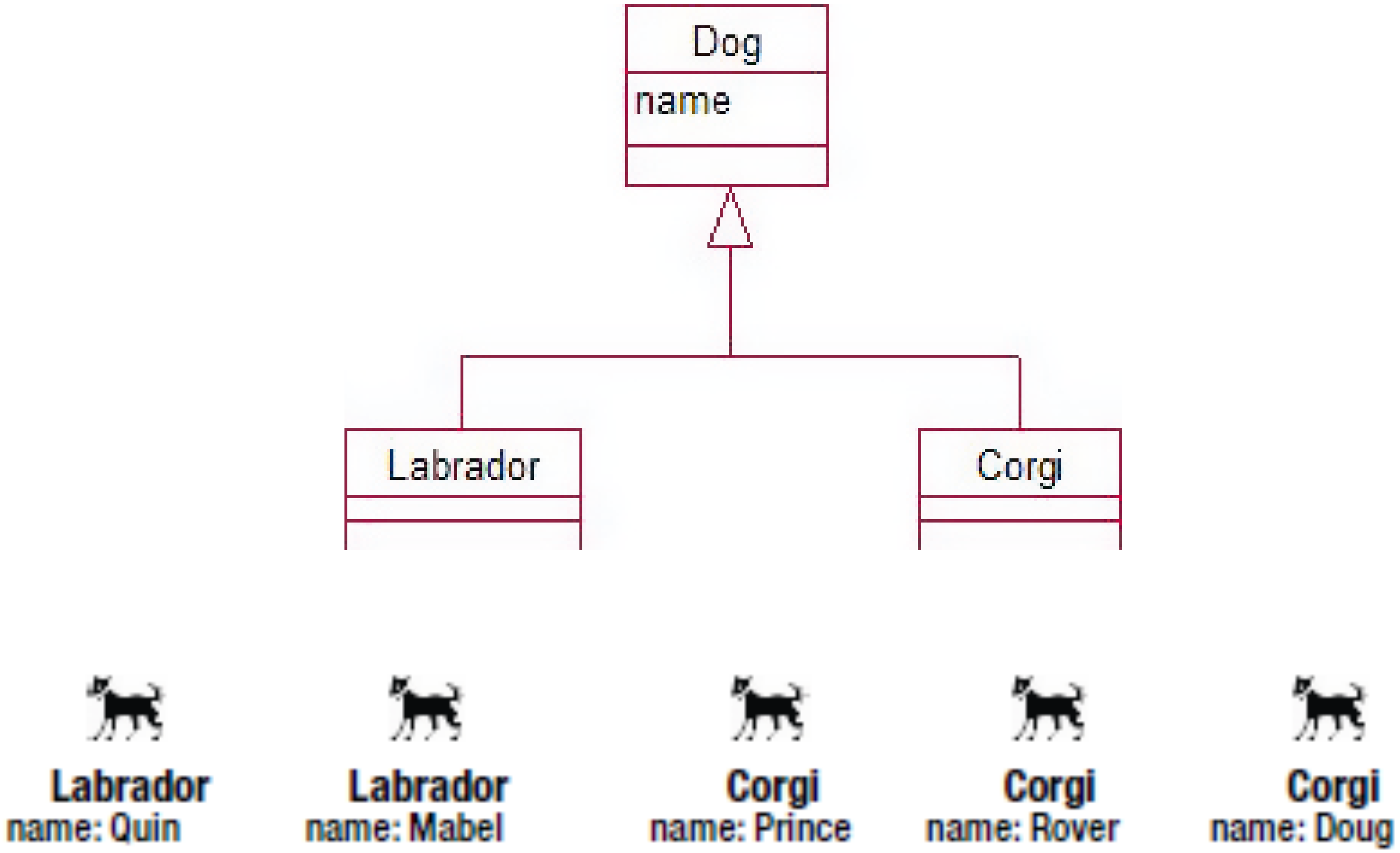
- A quick way to help decide whether something is a class or an object is to ask a question such as, "Am I likely to have several of <whatever> and am I interested in them as a group?"

  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.

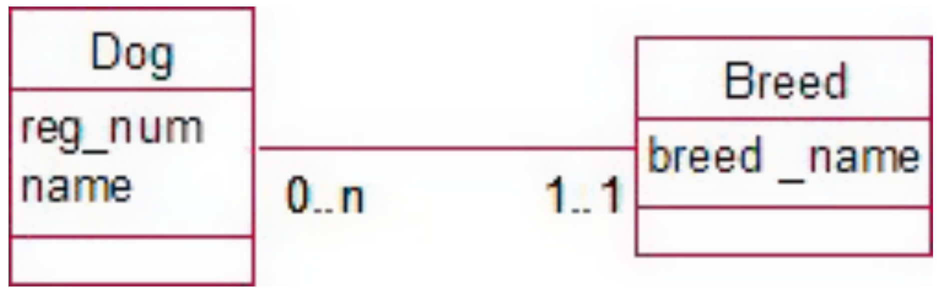
- Corgis and Labradors are potential classes, whereas Quin and Rover are more likely to be objects of one of our dog classes.

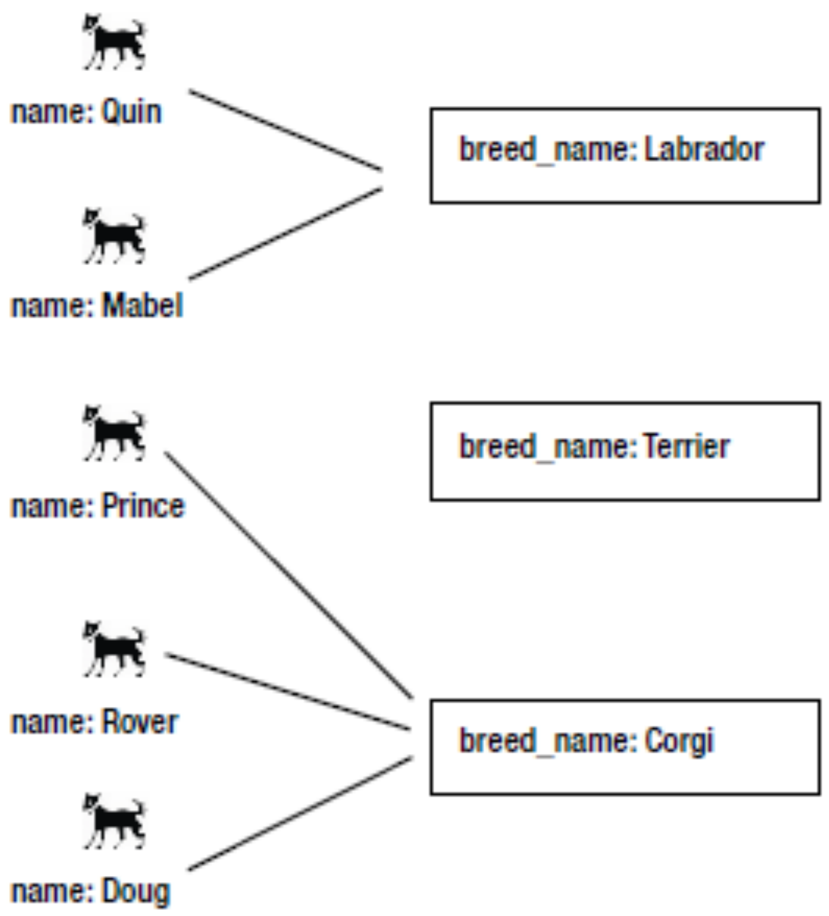
When Inheritance …



Confusing an Association with a Subclass

- The model 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.
- This can be done with a simple association between our Dog objects and objects of another class called Breed (assuming pure-bred dogs for now).
- The model is a much simpler way of representing our problem than the last model. The resulting database will be much easier to maintain also.

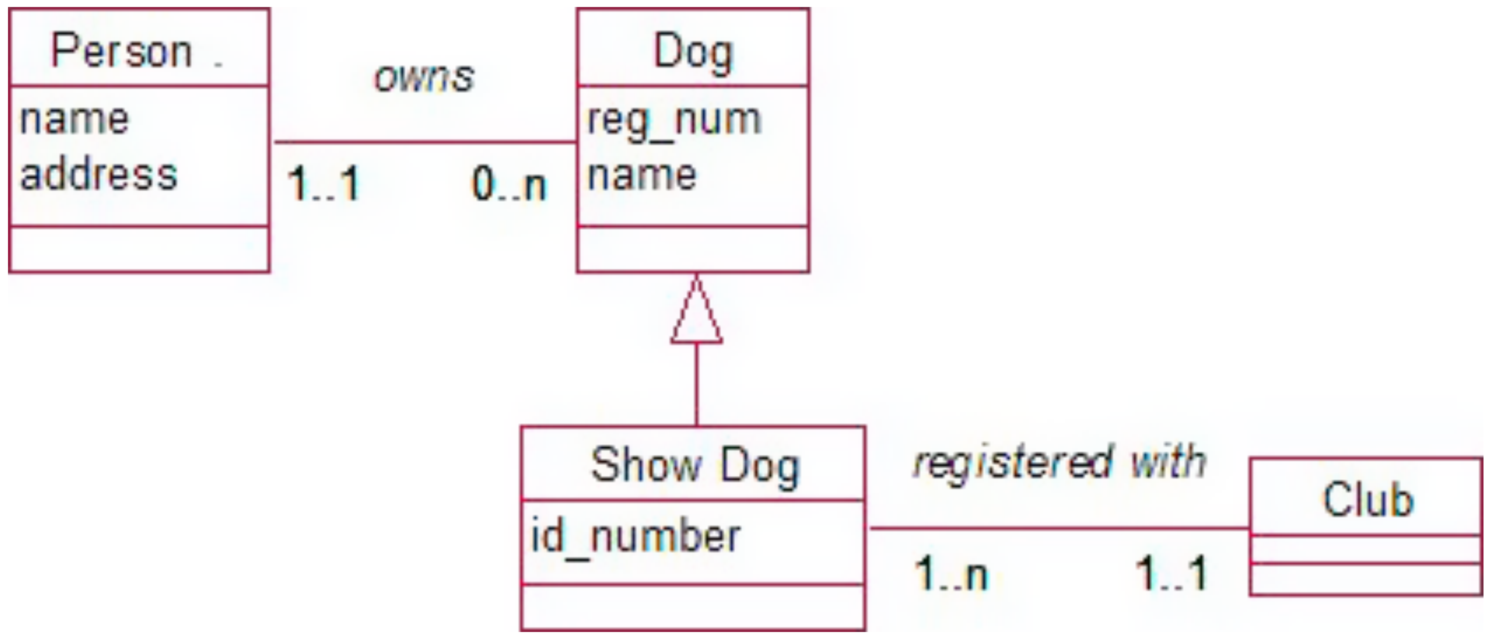




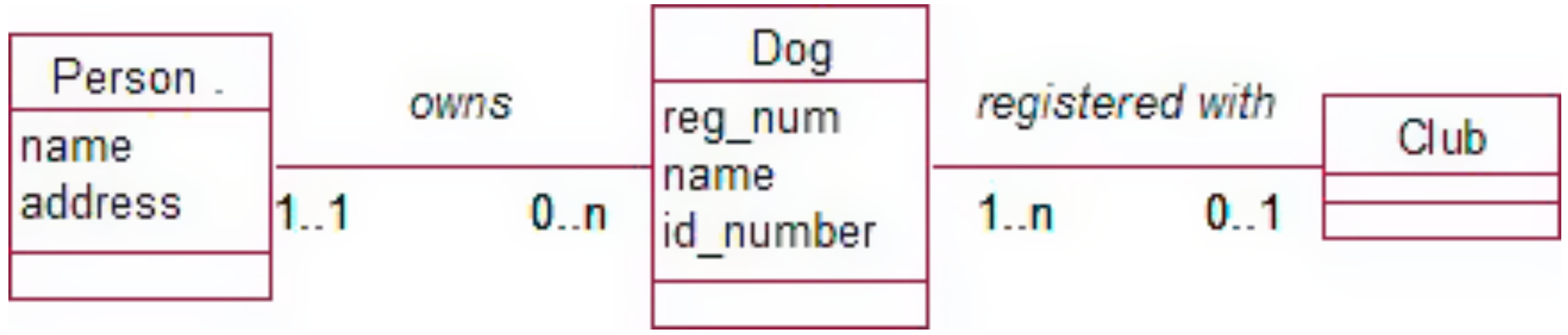
- What if the problem changed to say that we want to keep the fees payable to the kennel club and that these are different for the different breeds (e.g., a Labrador will cost $100, a Corgi $80, and a Terrier $85)? Now that we have some different information about the breeds, should we reconsider specialized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

- 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 the council wanted to keep this information, a model might be worth considering.



- 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.
- Can you see any drawbacks to this model?

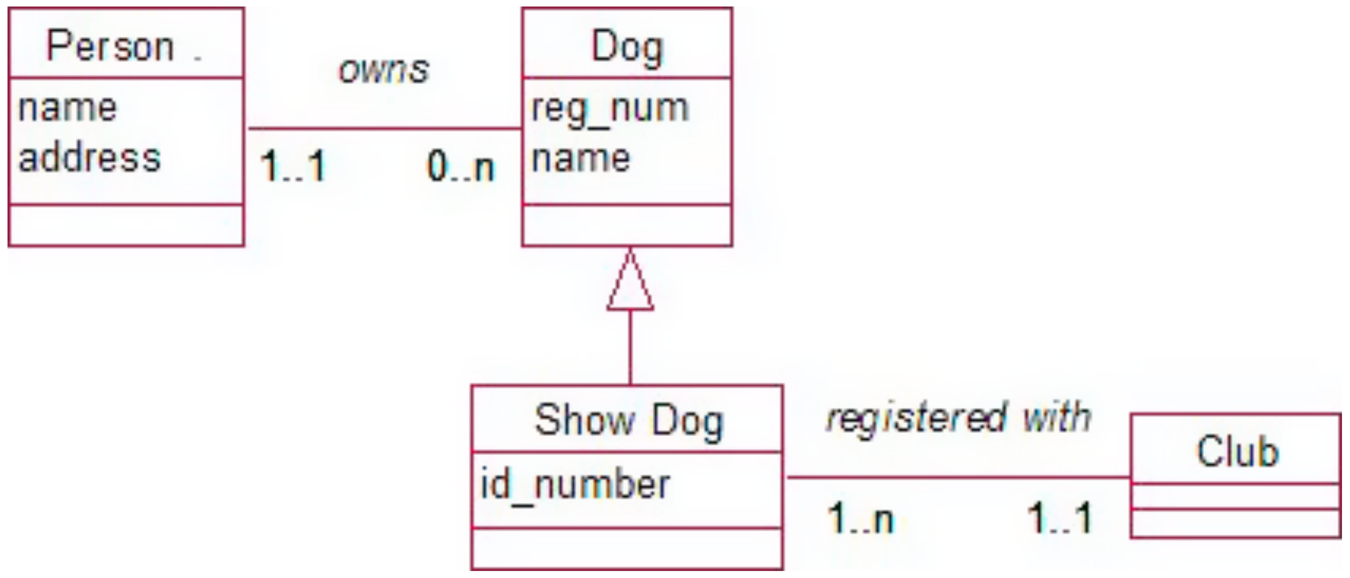


When is Inheritance Worth Considering?

- It is possible to capture all the required information with the model,
- but it is not so easy to keep the data accurate.
- 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.
- What is the main objective? What is the scope? How important is the accuracy of this data?
- 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?

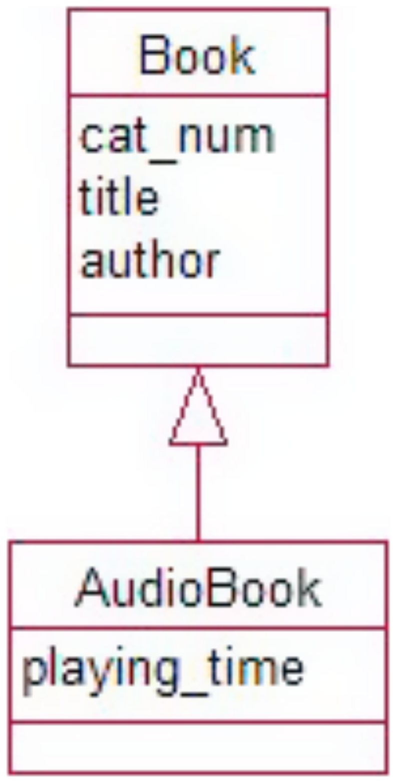
- We had a class of dogs with show dogs as a subclass. The implication here is that your ordinary old pet will be an object of the superclass Dog, while show dogs will be objects of the subclass.



- This can lead to a few problems as the project evolves.
- 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.

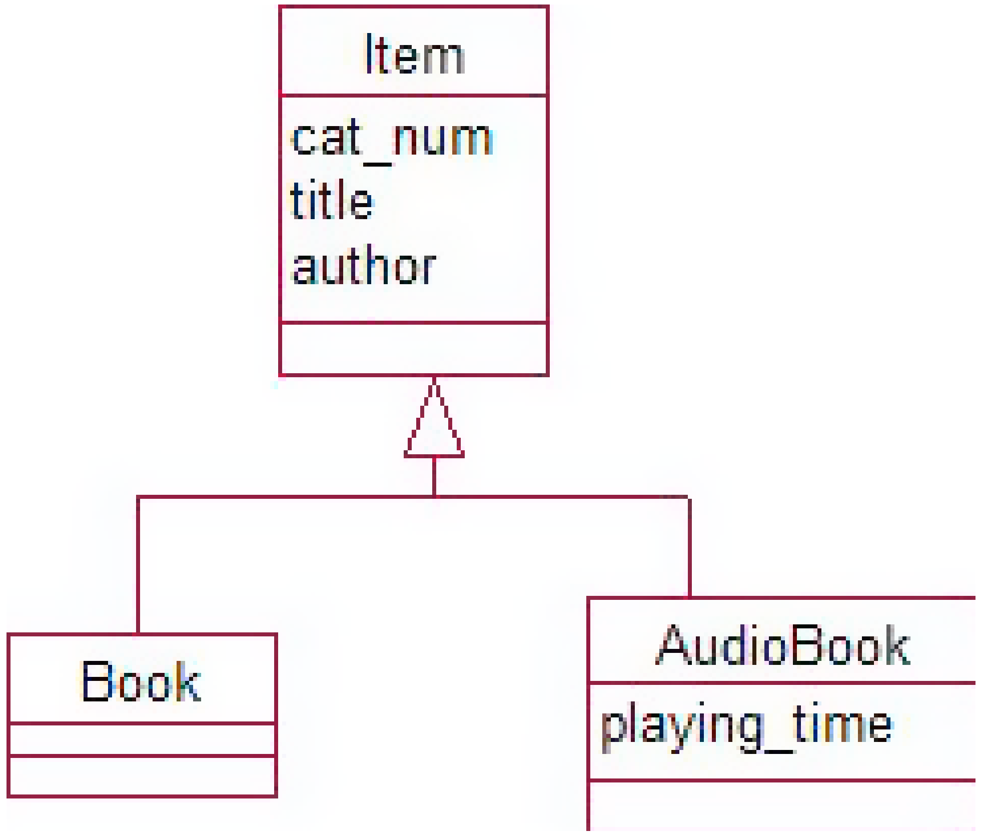
Should the Superclass Have Objects?

- 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,



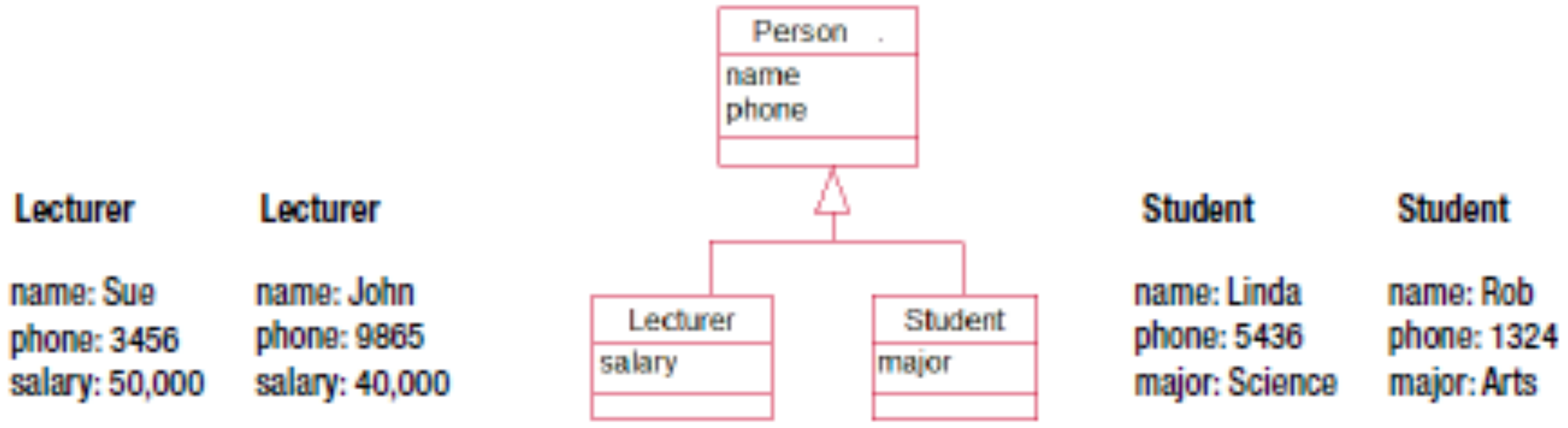
Should the Superclass Have Objects?

- 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.
- 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.



Objects That Belong to More Than One Subclass

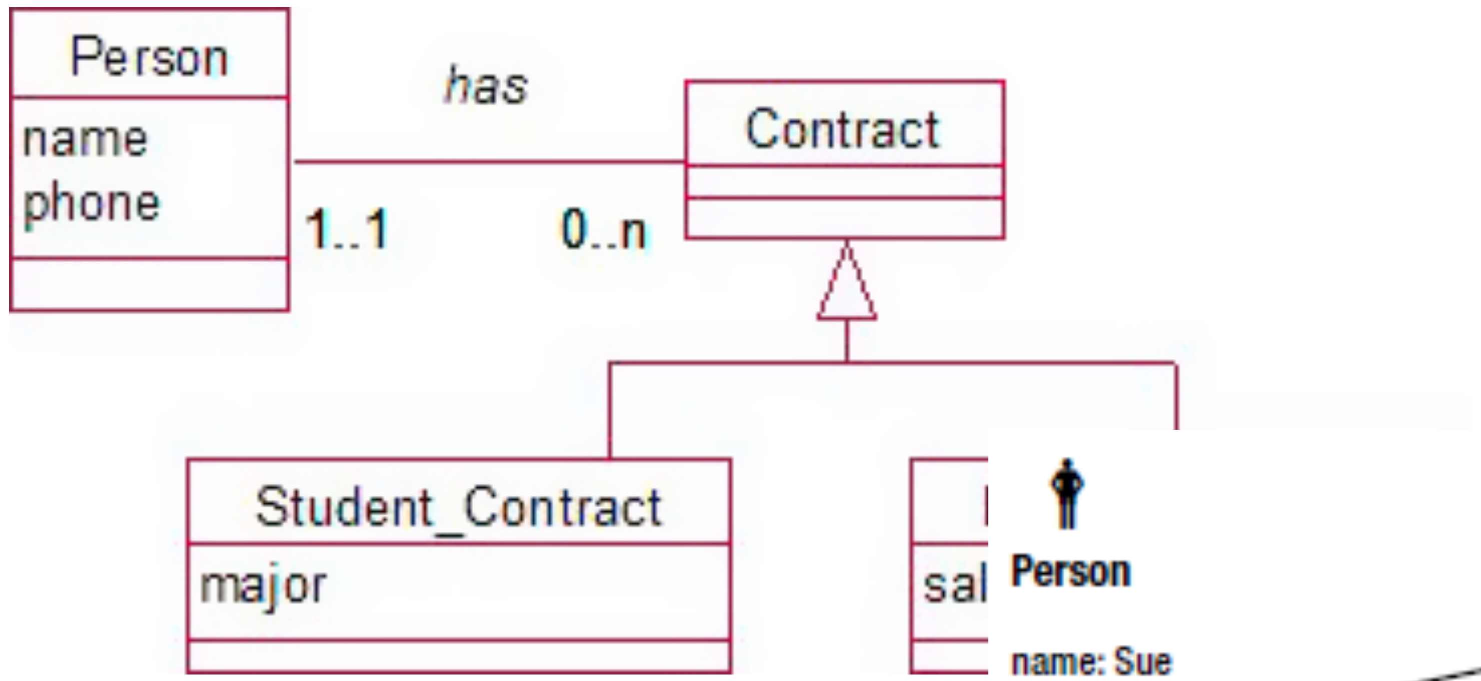
- We have a model with Lecturer and Student represented as subclasses of a Person class, along with some objects of the two subclasses. We see that for this case our objects have to be either a lecturer or a student.

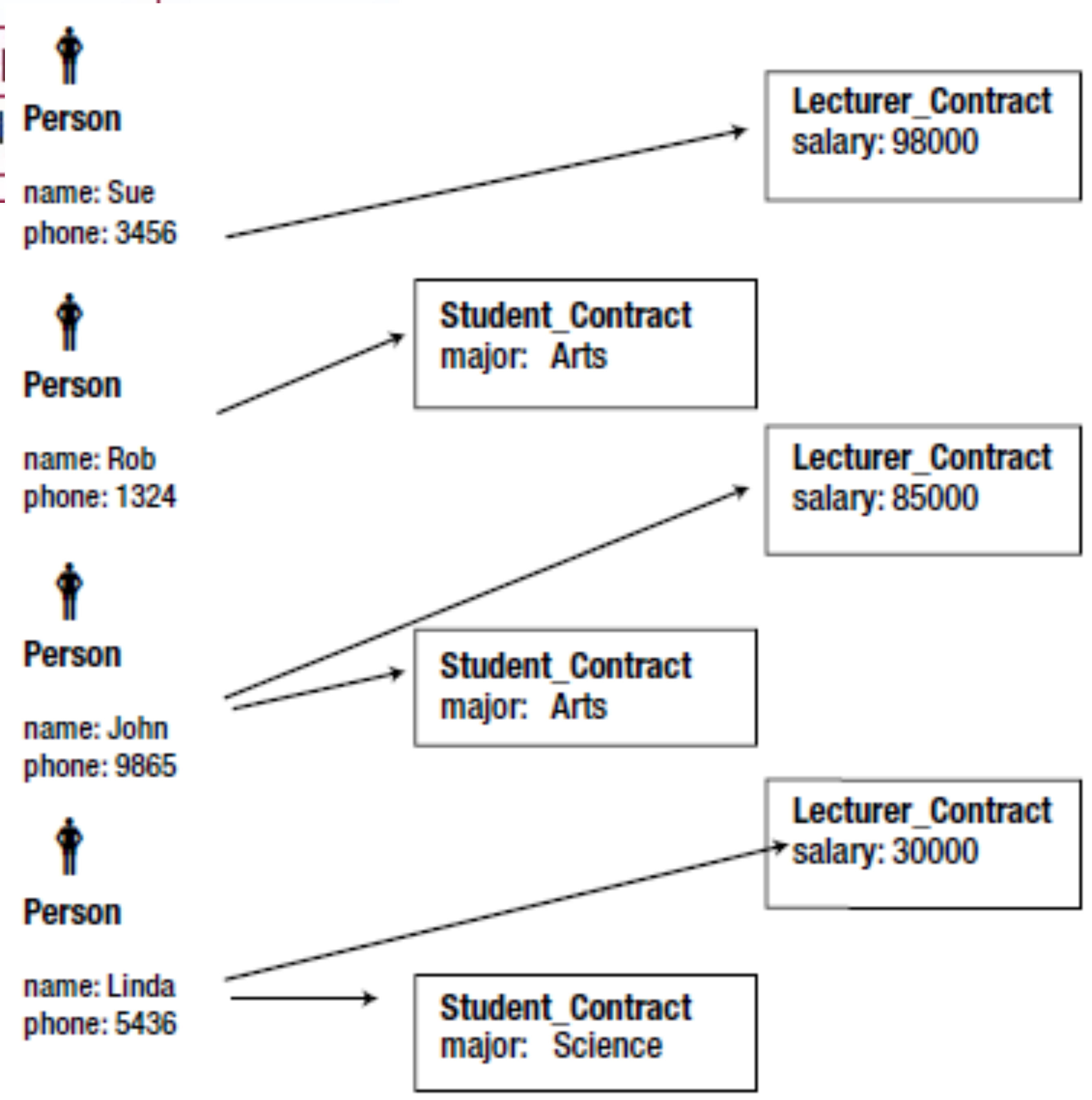


- What is more likely is that there is some overlap between the two. What if lecturer John is also doing some part-time study for an arts degree and student Linda is doing some part-time teaching to fund her fees? Where do we store John's major and Linda's salary?

- There are difficulties with the approach.
- 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.
- 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.

Objects That Belong …

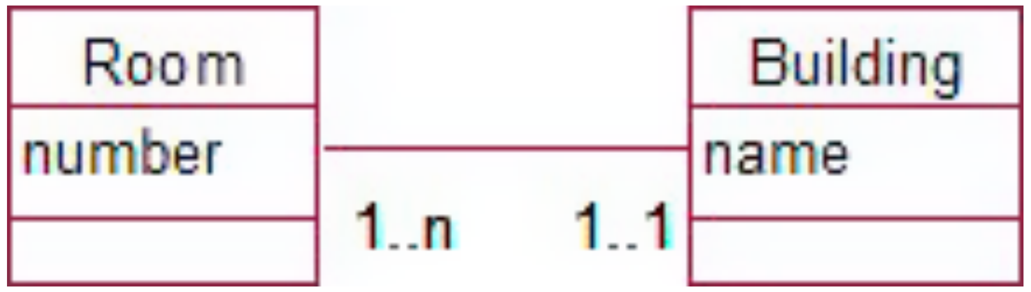




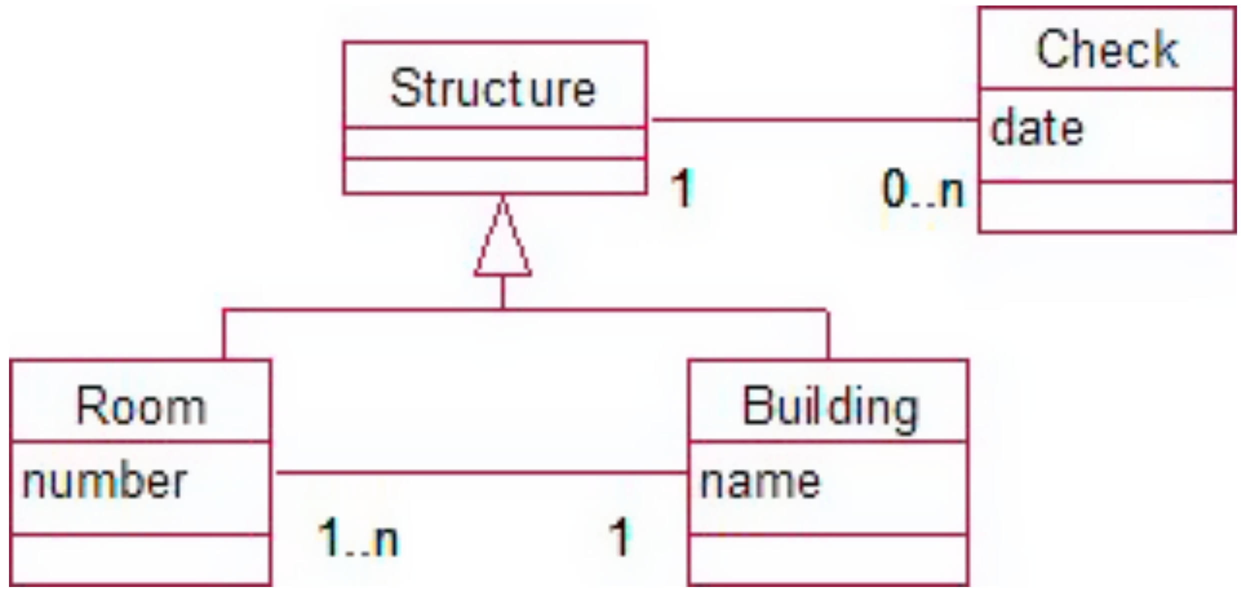
- We very clearly have four people who are undertaking a number of different roles.
- If we wish to include administrators with information specific to their contracts, we just add another subclass, Administrator_Contract, to our Contract class.
- There is a slight problem, however. we have that a person can be related to many contracts, but we don't have any constraints as to which type of contracts.
- 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

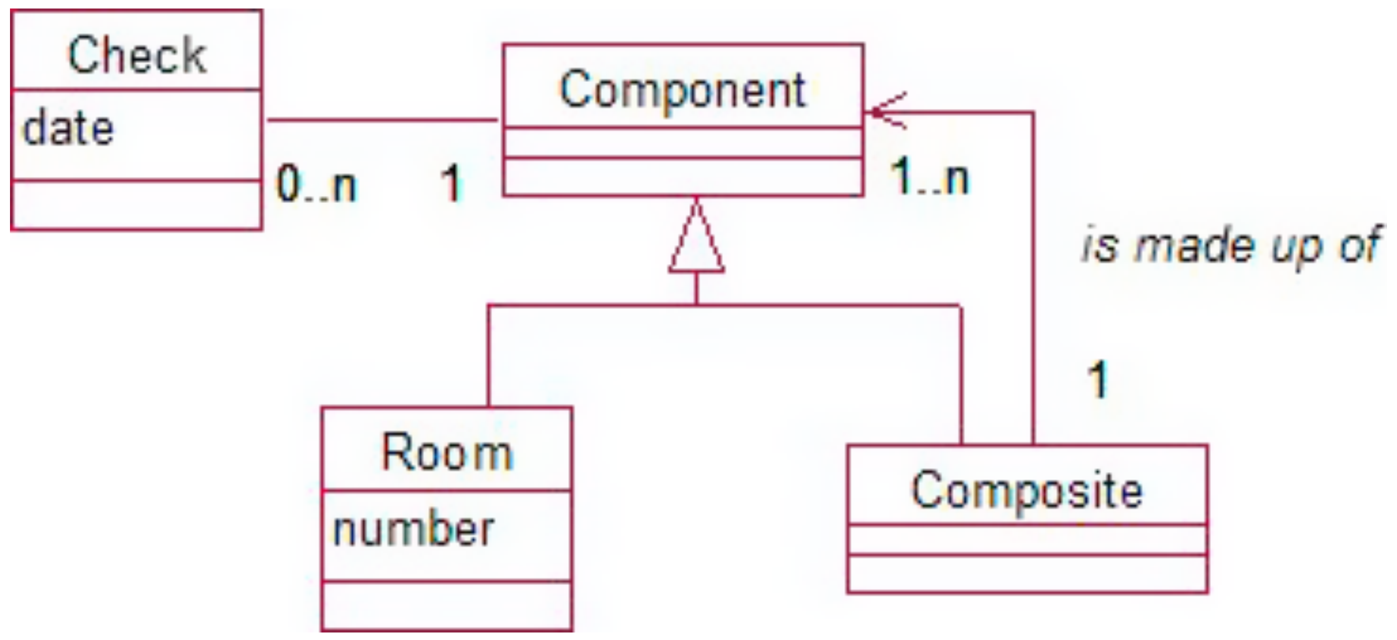
- We have objects that are made up of other objects: a forest is made up of trees or a building is made up of rooms.
- When dealing with situations in which we have this type of composition or aggregation of objects, inheritance can become useful.
- Consider a building that has a number of rooms.
- Every 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?



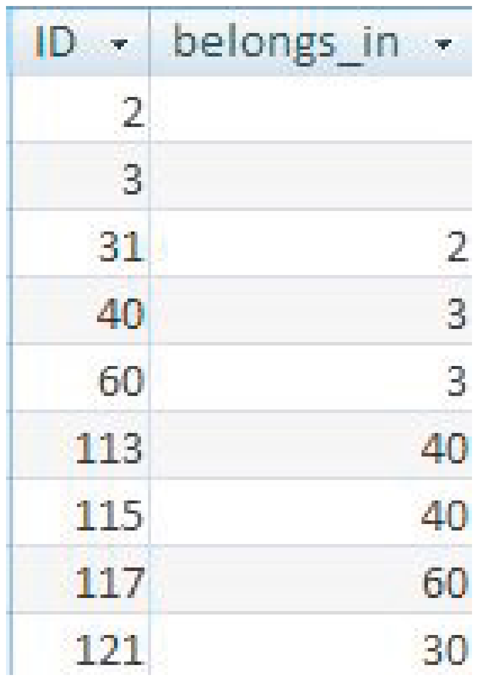
- A Room and a Building have something in common. They both can be associated with safety checks. This therefore becomes a candidate for an inheritance solution.



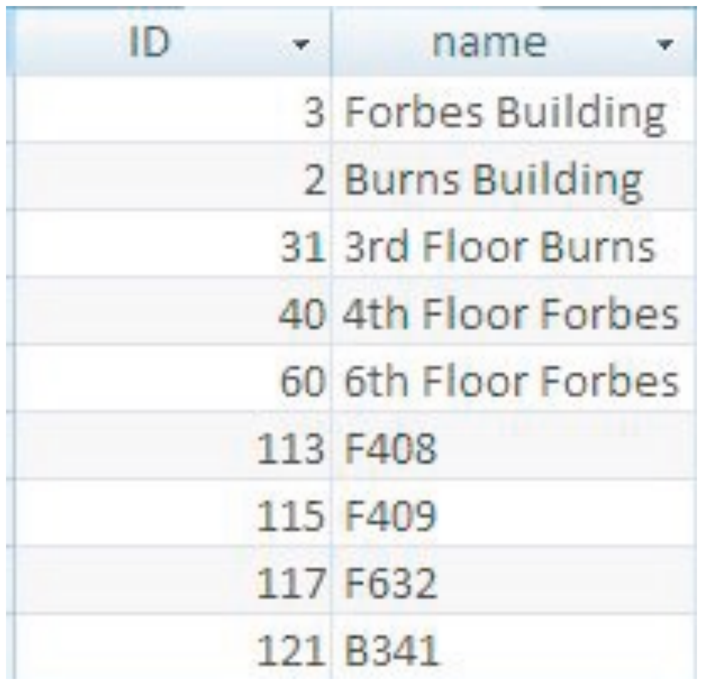
- Now we can store data about a single check that incorporates a building (and we know what rooms were included).
- If necessary the check can be associated with a single room. While this solution is fine, it can be made much more general by using a software pattern.
- They mostly deal with behavior whereas our focus is on data. The solution is based on the Composite pattern.



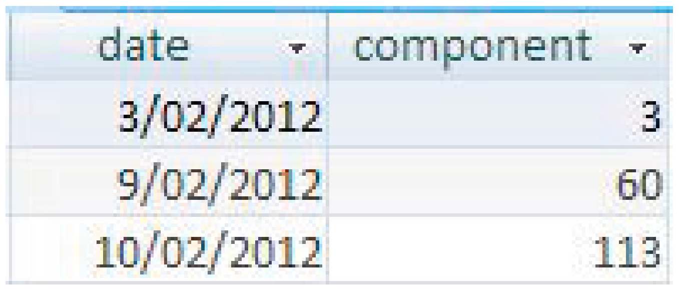
- We have something (a component) on which a safety check can be made. That component may be an individual component (in this case a room) or a collection of other components.
- The new model is more general because a Composite can be made up of other composites.
- With the Composite pattern, we can record safety checks at any of these levels (room, floor, building, etc.).
- The tables show how the information might be recorded.



ID	belongs_in
2	
3	
31	2
40	3
60	3
113	40
115	40
117	60
121	30



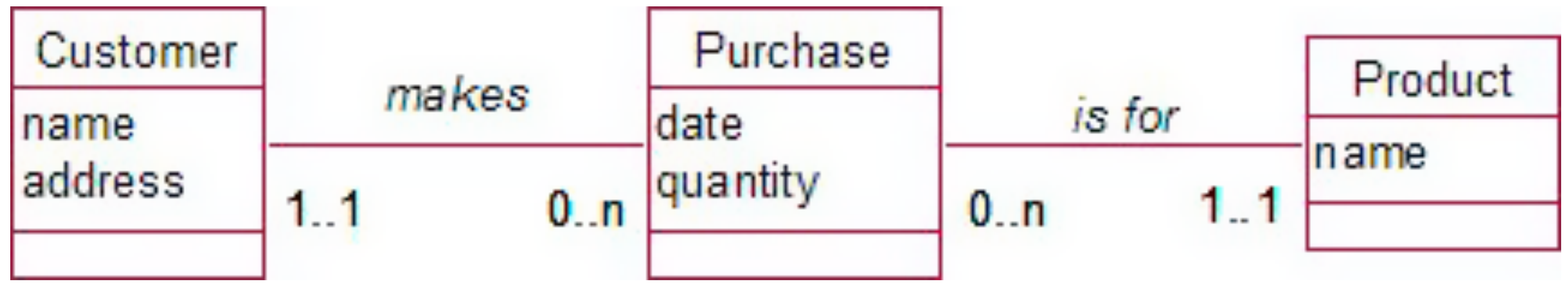
ID	name
3	Forbes Building
2	Burns Building
31	3rd Floor Burns
40	4th Floor Forbes
60	6th Floor Forbes
113	F408
115	F409
117	F632
121	B341



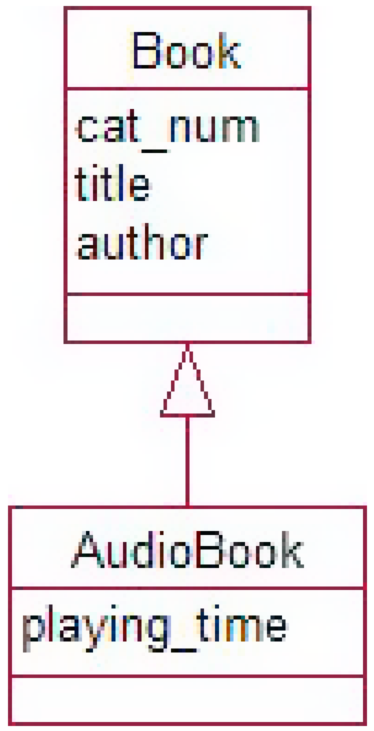
date	component
3/02/2012	3
9/02/2012	60
10/02/2012	113

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.
- Dealing with inherited classes becomes considerably more difficult if we need to add *behavior* (or methods) to our classes.
- Even for just static data, we can still run into problems when we try to design an inheritance hierarchy.

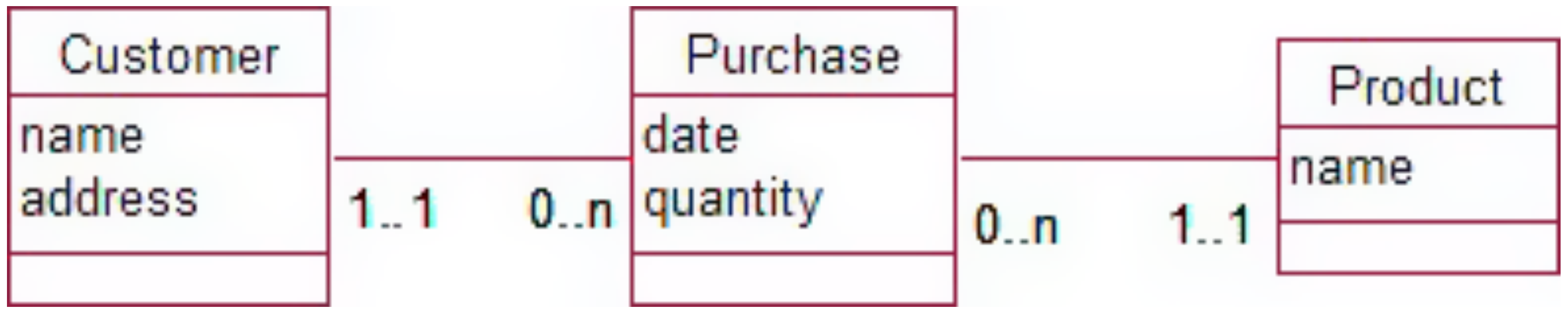


- Dividing animals up into fish, mammals, birds, and so on may do quite well as we enter data about bears, dogs, sharks, and sparrows.
- But what happens when we come to whales? Whale doesn’t fit at all into the model.
- A whale is a mammal, but it also needs to be shown as living in sea water, similar 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.



Exercise 9-1

Consider the model in the Figure 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.

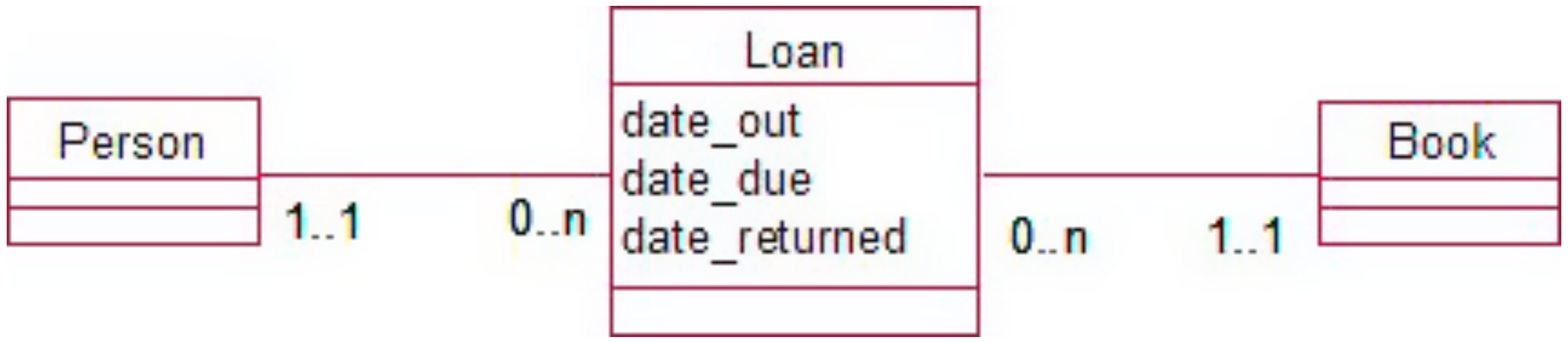


The company changes the way it does business to allow customers to walk in off the street and pay cash. No customer needs to be associated with a cash purchase. Discuss how effective the following changes to the data would be.

- *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 Cash Customer.*
- *Create subclasses of Customer: Cash_Customer and Account_Customer.*
- *Create subclasses of Purchase: Cash_Purchase and Account_Purchase.*

Exercise 9-2

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 the figure.



Summary

Situations when inheritance is a possibility include the following:

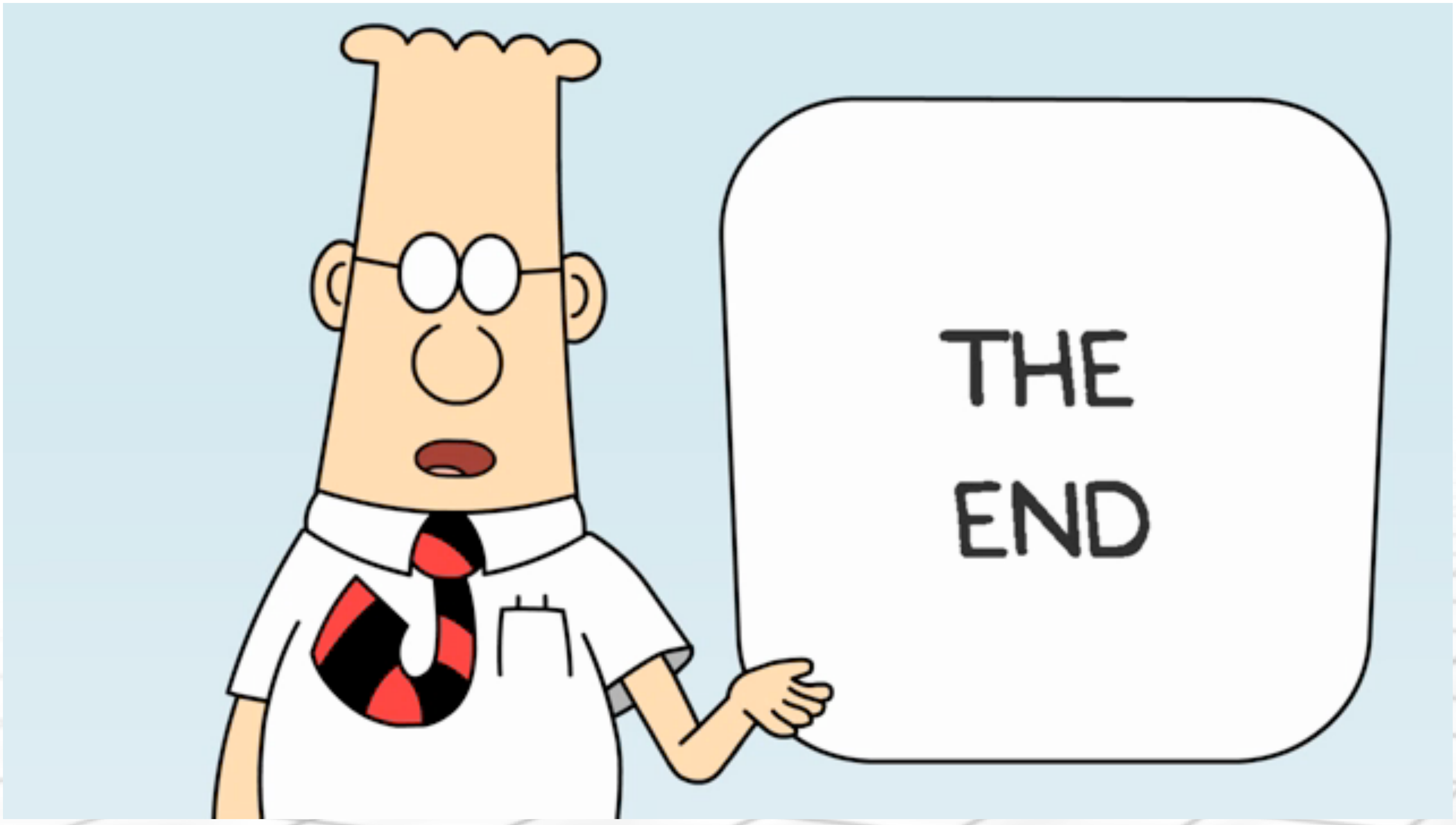
- If different objects have mutually exclusive values for some attributes (e.g., administrators have grades but technicians have dates), consider specialized subclasses.
- When you think this is like that except for… consider subclasses.
- When two classes have a similar relationship with another class, 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.



출처: metachannels.com

Thank you!