Software Engineering II

OO Software, Continuing Class Diagrams, & Package Diagrams

Dr. Amr S. Ghoneim

What is OO Software?

- ***Object-oriented software*** means that we organize software as a collection of discrete objects that incorporate both data structure and behavior.
- The fundamental unit is the ***Object***
- An object has ***state (data)*** and ***behavior (operations)***
- In ***Structured programming***, data and operations on the data were separated or loosely related.

1. OO Characteristics

- Identity
- Classification
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Generics
- Cohesion
- Coupling

Identity

- An object can be ***concrete*** like a ***car***, a ***file***, …
- An object can be ***conceptual*** like a ***feeling***, a ***plan***,…
- Each object has its own identity even if two objects have exactly the same ***state.***

Omar's car

Rana's car

Identity

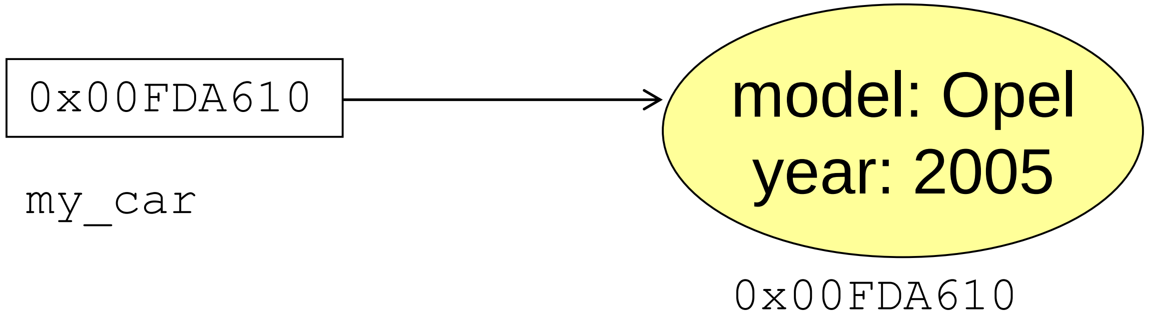
- ***Object identity*** is the property by which each object can be identified and treated as a distinct software entity.

- Each object has unique identity which distinguishes it from all its fellow objects. It is its memory address (or ***handle***) that is referred to by one or more ***object identifiers (variables)***.

Identity

```
car  my_car = new car(“Opel”, 2005);
```

- The object handle is 0x00FDA610 is referenced by an object identifier (object variable) **`my_car`**



Classes and Objects

- Each object is an ***instance*** of a class
- Each object has a reference to its class (knows which class it belongs to)

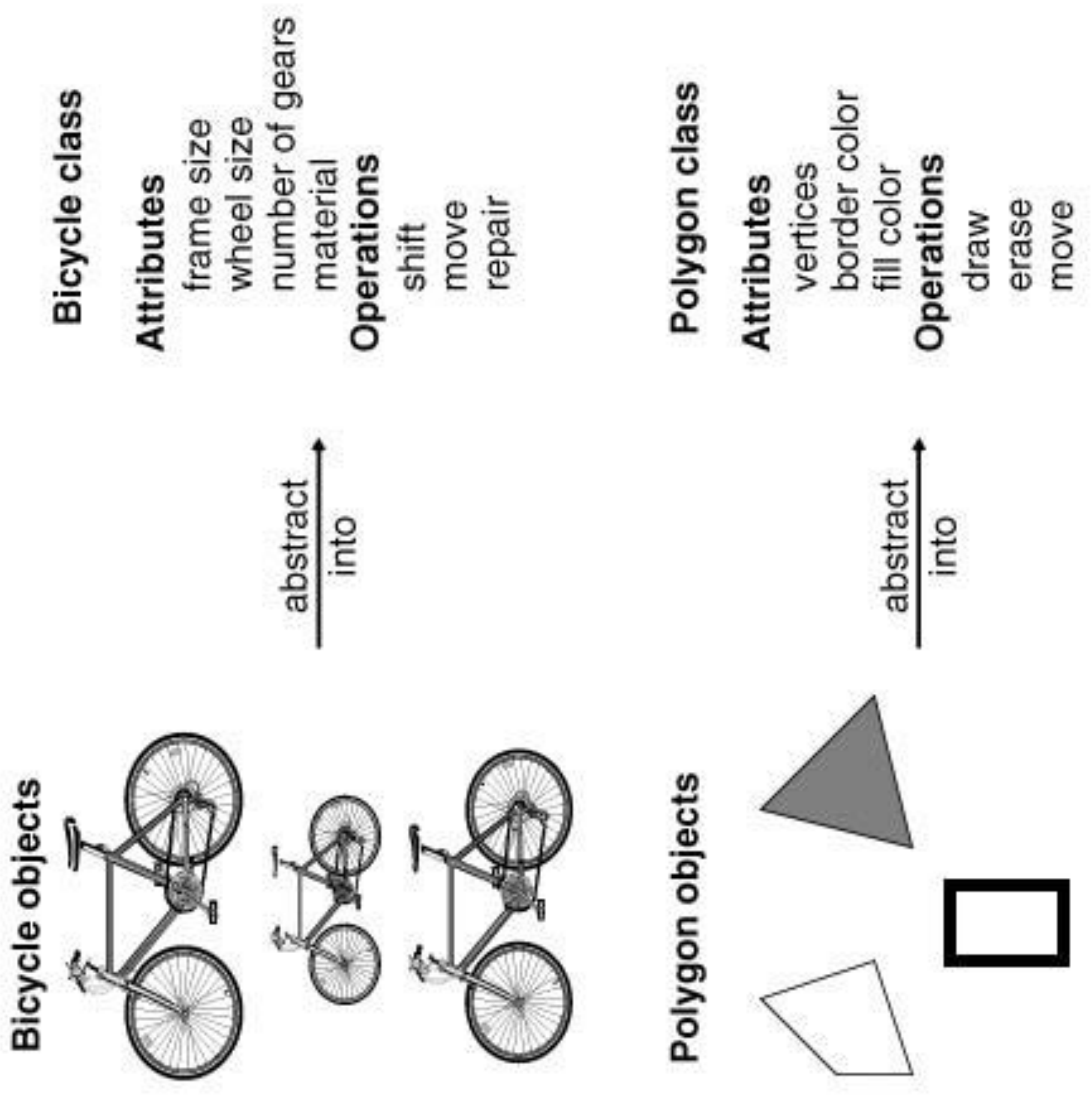
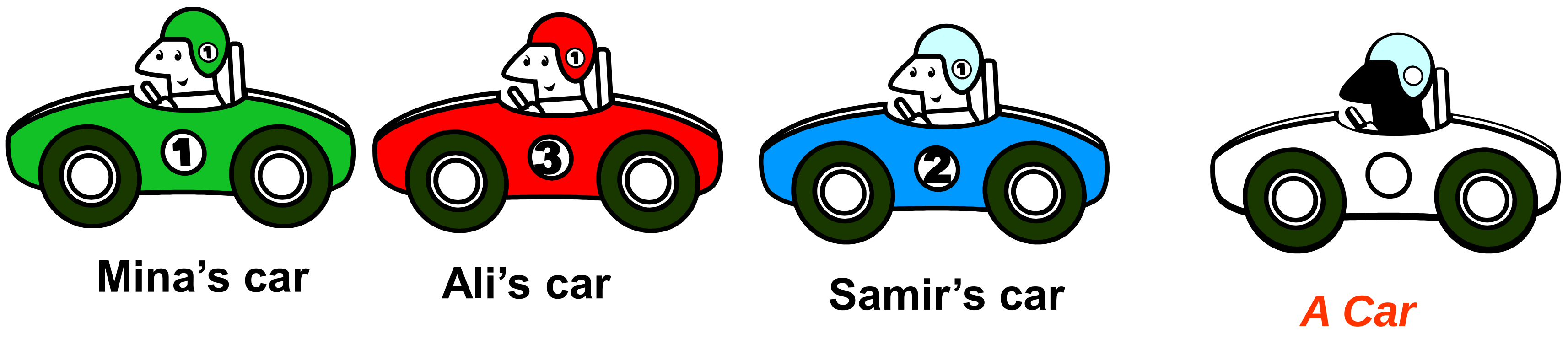


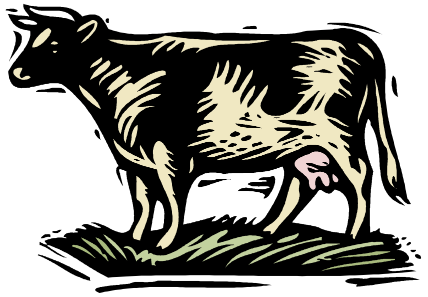
Figure 1.2 Objects and classes. Each class describes a possibly infinite set of individual objects.

Classification

- ***Classification*** means that objects with the same data structure (***attributes***) and behavior (***operations***) belong to the same ***class***
- A class is an ***abstraction*** that describes the properties important for an application
- The choice of classes is arbitrary and application-dependent.



Classification



- Objects in a class share a common ***semantic*** purpose in the system model
- Both car and cow have *price* and *age*
- If both were modeled as pure *financial assets*, they both can belong to the same class.
- If the application needs to consider that:
  - Cow eats and produces milk
  - Car has speed, make, manufacturer, etc.
- Then model them using separate classes.
- ***So the semantics depends on the application***

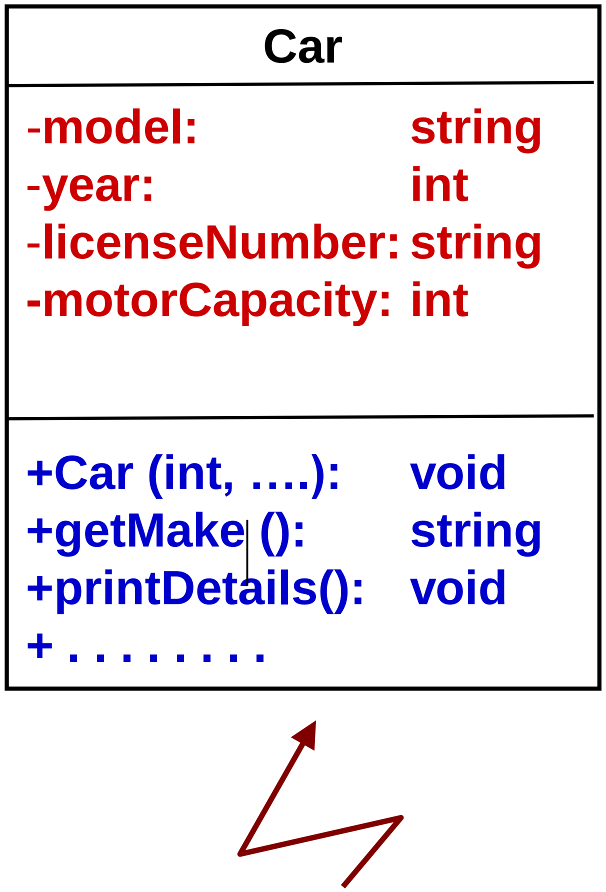
Abstraction

- ***Abstraction*** is the selective examination of certain aspects of a problem.

- Abstraction aims to isolate the aspects that are important for some purpose and suppress the unimportant aspects.

- The purpose of abstraction determines what is important and what is not.

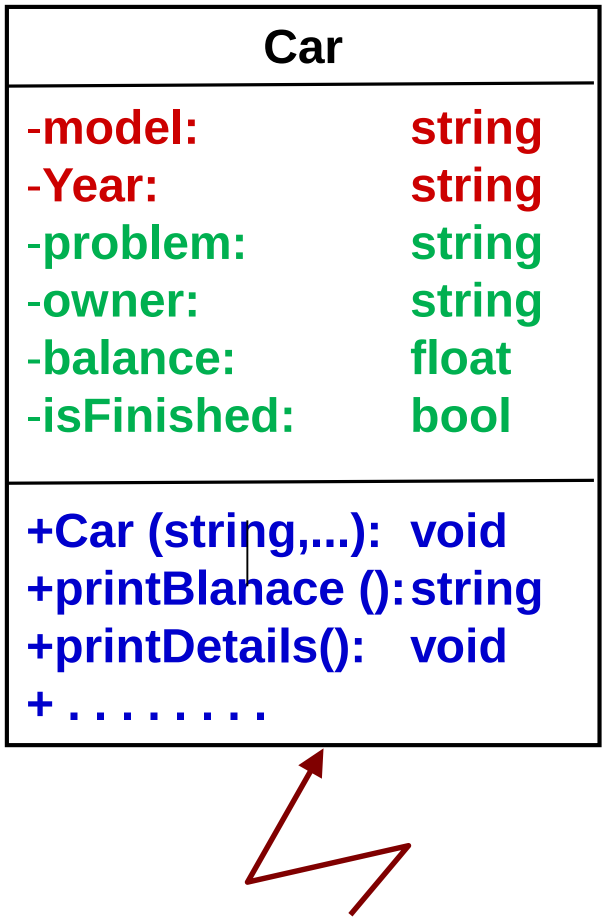
Abstraction



```
Car
-model:            string
-year:             int
-licenseNumber:    string
-motorCapacity:    int

+Car (int, ….):    void
+getMake ():       string
+printDetails():   void
+ . . . . . . . .
```

In Department of Motor Vehicles



```
Car
-model:               string
-Year:                string
-problem:             string
-owner:               string
-balance:             float
-isFinished:          bool

+Car (string,...):    void
+printBlanace ():     string
+printDetails():      void
+ . . . . . . . .
```

At the mechanic

Encapsulation

- ***Encapsulation*** separates the external aspects of an object, that are accessible to other objects, from the internal implementation details that are hidden from other objects.
- Encapsulation reduces ***interdependency*** between different parts of the program.
- You can *change the implementation* of a class (to enhance performance, fix bugs, etc) *without affecting* the applications that use objects of this class.

Encapsulation

- ***Data hiding.*** information from within the object cannot be seen outside the object.
- ***Implementation hiding.*** implementation details within the object cannot be seen from the outside.

Encapsulation

List	
- **items:**	**int [ ]**
- **length:**	**int**
+ List (array):	**void**
+ search (int):	**bool**
+ getMax ():	**int**
+ sort():	**void**

```
void sort () {   // Bubble Sort
    int i, j;
    for (i = length - 1; i > 0; i-) {
        for (j = 0; j < i; j++) {
            if (items [j] > items [j + 1]) {
                int temp = items [j];
                items [j] = items [j + 1];
                items [j + 1] = temp;
            }
        }
    }
}`
```

```
void sort () {   // Quick Sort
    ……….
}`
```

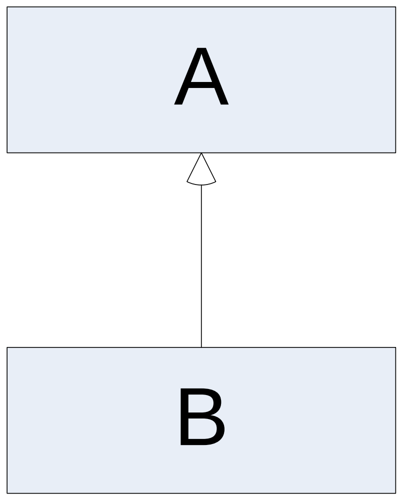
Inheritance

- ***Inheritance*** is the sharing of ***features*** (attributes and operations) among classes based on a hierarchical relationship.
- A ***superclass*** (also ***parent*** or ***base***) has general features that ***sublcasses*** (***child*** or ***derived***) inherit. and may refine some of them.

- Inheritance is one of the strongest features of OO technology.

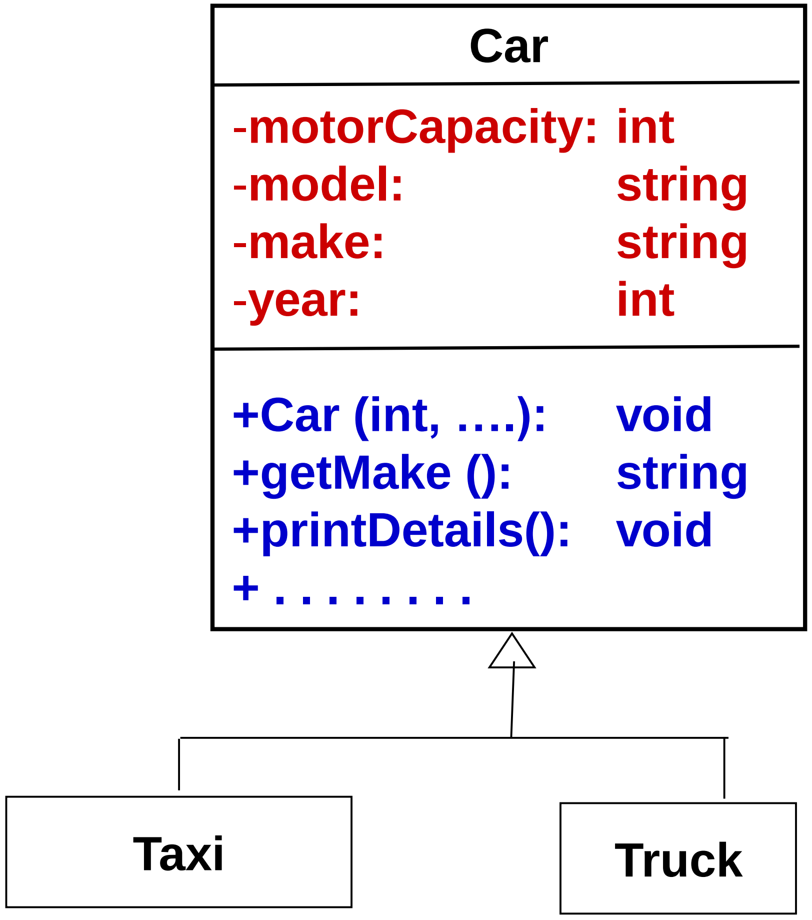
Inheritance

- Inheritance is the facility by which objects of a class (say B) may use the methods and variables that are defined only to objects of another class (say A), as if these methods and variables have been defined in class B
- Inheritance is represented as shown in ***UML*** notation.



How to use Inheritance?

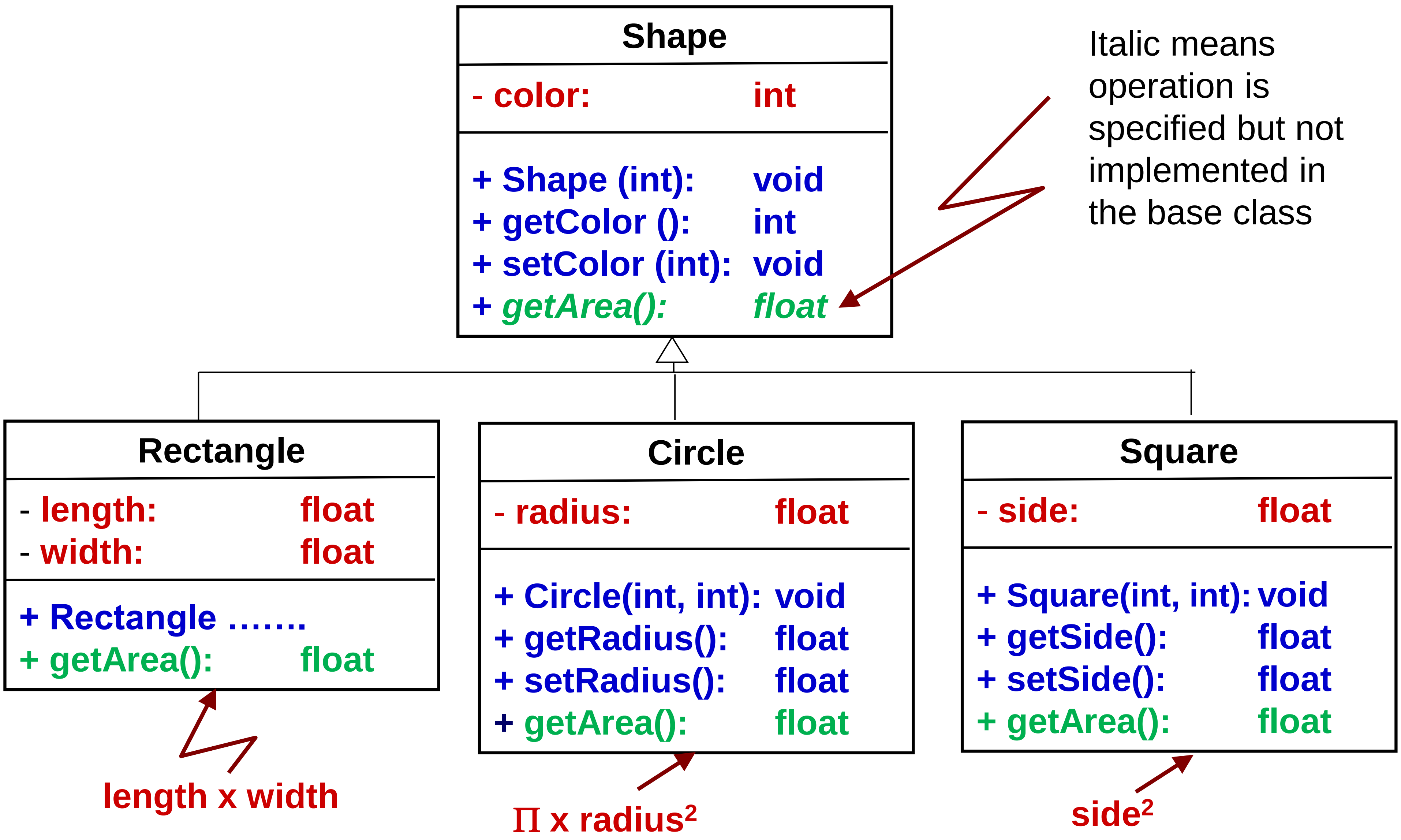
- Inheritance helps building software incrementally:
- *First*; build classes to cope with the most straightforward (or general) case,
- *Second*; build the special cases that inherit from the general base class. These new classes will have the same features of the base class plus their own.



Polymorphism

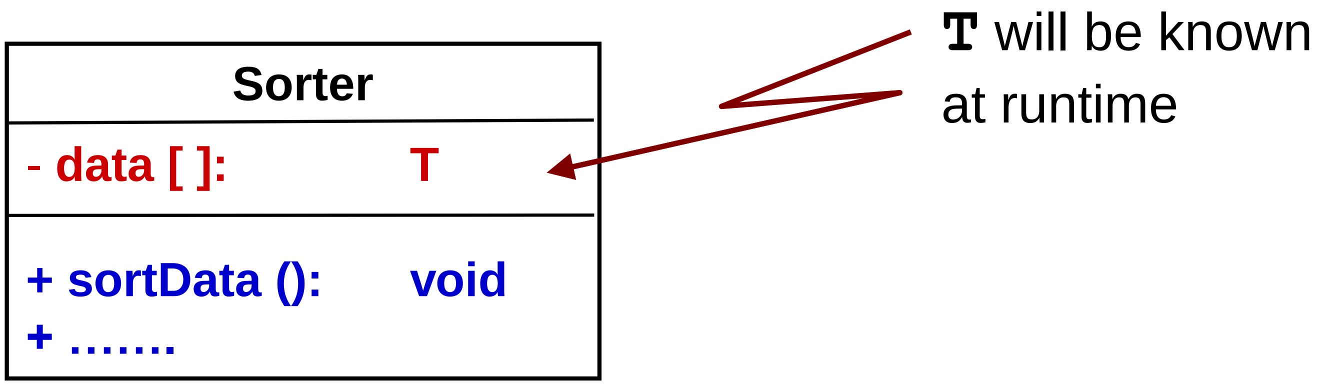
- ***Polymorphism*** means that the same operation may behave differently for different classes.
- An ***operation*** is a procedure or a transformation that the object performs or is subject to.
- An implementation of an operation by a specific class is called a ***method***.
- Because an OO operation is polymorphic, it may have more than one method for implementing it, each for a different class.

Polymorphism



Generics

- ***Generic Class*** : the data types of one or more of its attributes are supplied at run-time (at the time that an object of the class is instantiated)
- This means that the class is ***parameterized***, i.e., the class gets a parameter which is the name of another type.



Continuing Class Diagrams

Association Classes

- An ***Association class*** is an association that is also a class (has attributes and operations)

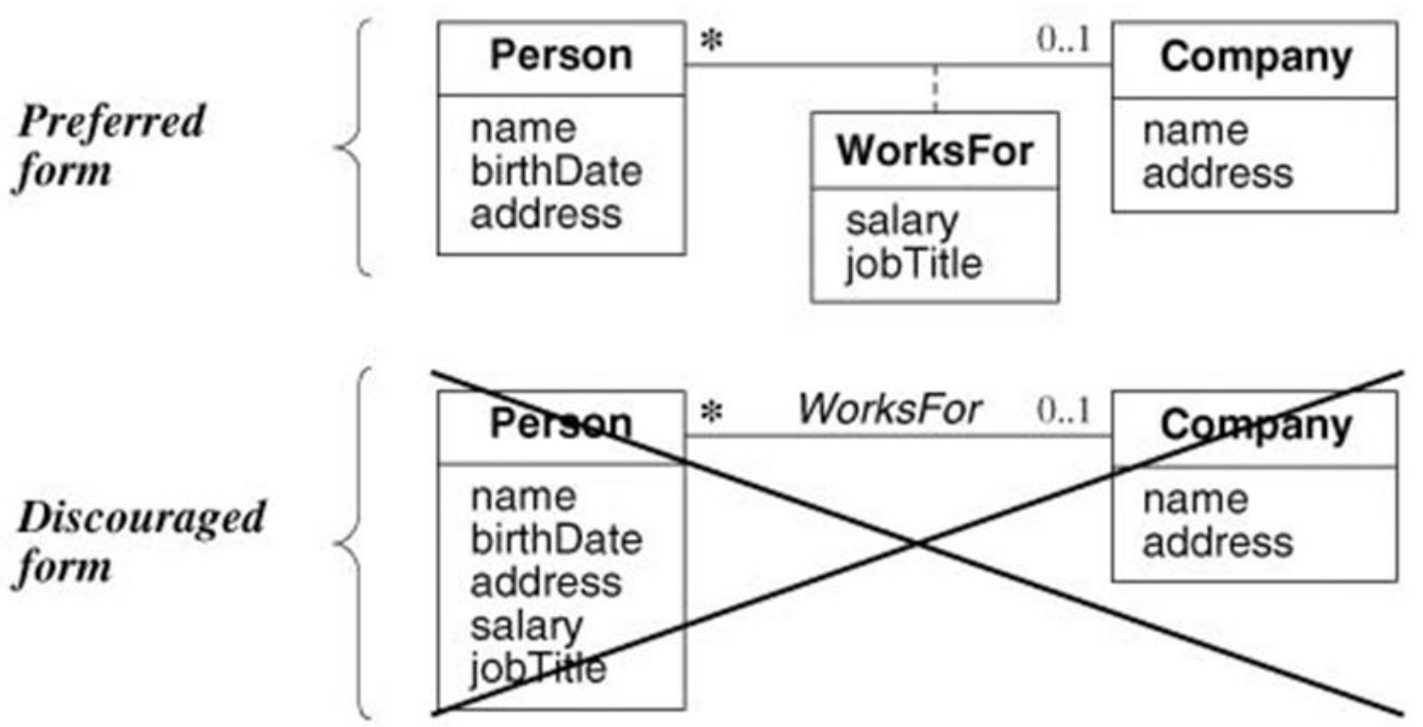
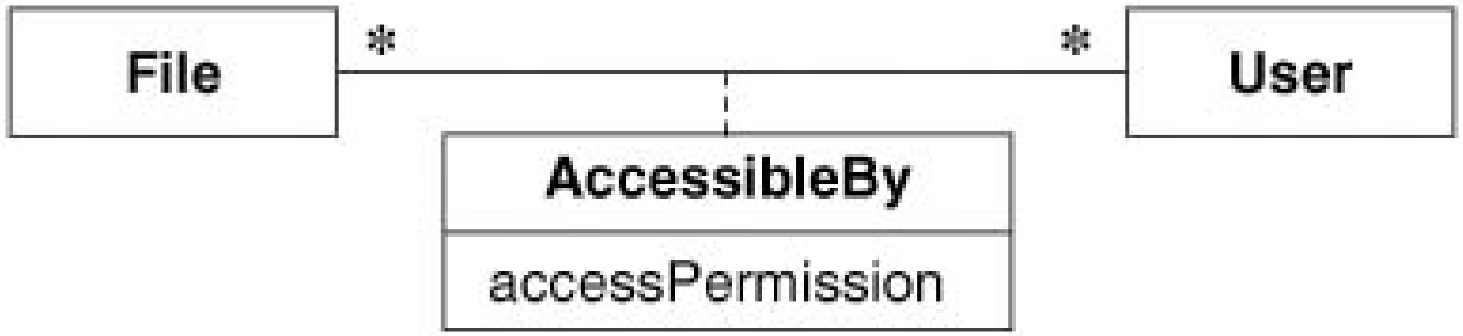


Figure 3.19 Proper use of association classes. Do not fold attributes of an association into a class.

Association Classes

- Many-to-may associations provide compelling rationale for association classes..
- *accessPermission* is a joint property of File and User and cannot placed in one of them only.



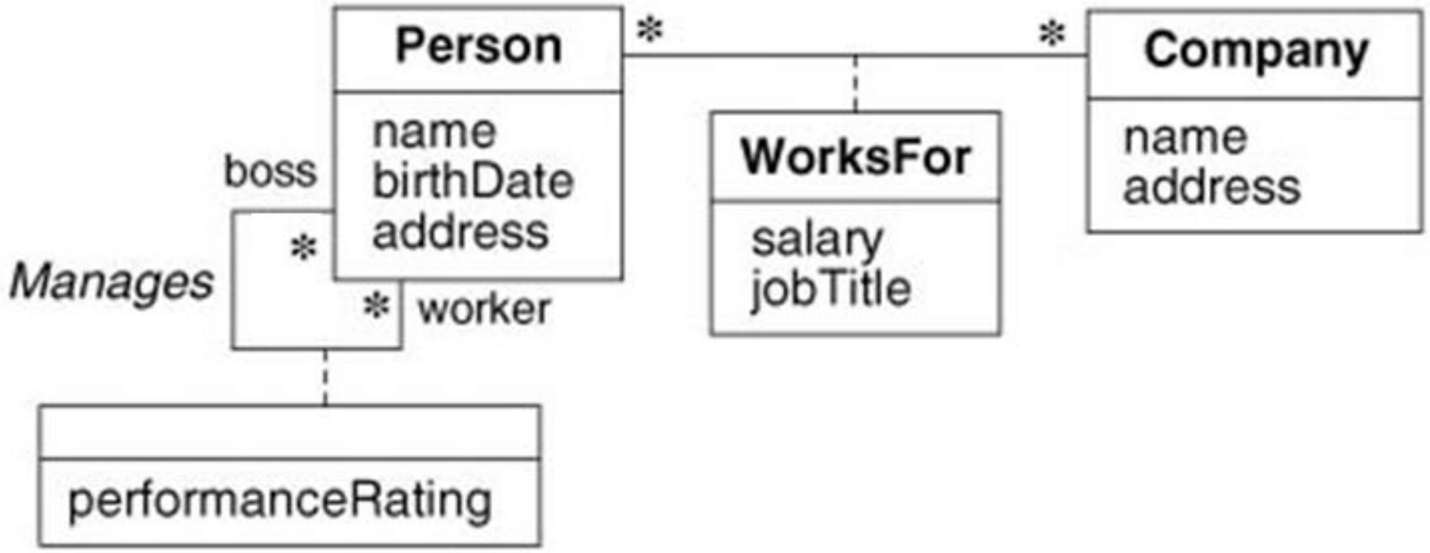
/etc/termcap	read	John Doe
/etc/termcap	read-write	Mary Brown
/usr/doe/.login	read-write	John Doe

Figure 3.17 An association class. The links of an association can have

Exercises

- Sketch a class diagram for the following:

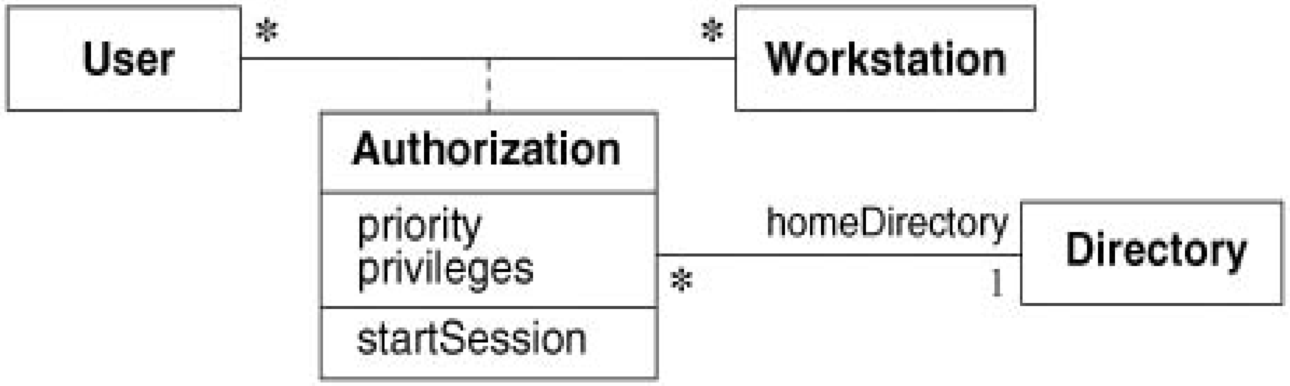
Each person works for a company receives a salary and has a job title. **The person can work for more than one company**. The boss evaluates the performance of each worker.



Exercises

- 2. Sketch a class diagram for the following:

A user may be authorized on many workstations. Each authorization has a priority and access privileges. A user has a home directory for each authorized workstation, but several workstations and users can share the same home directory.



Association class vs. Ordinary class

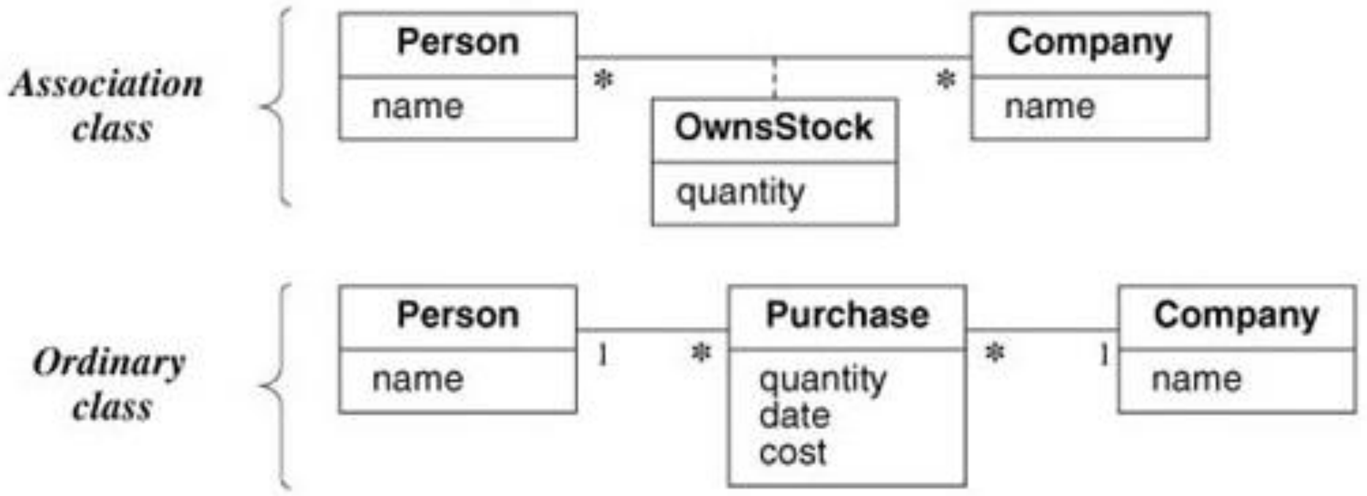


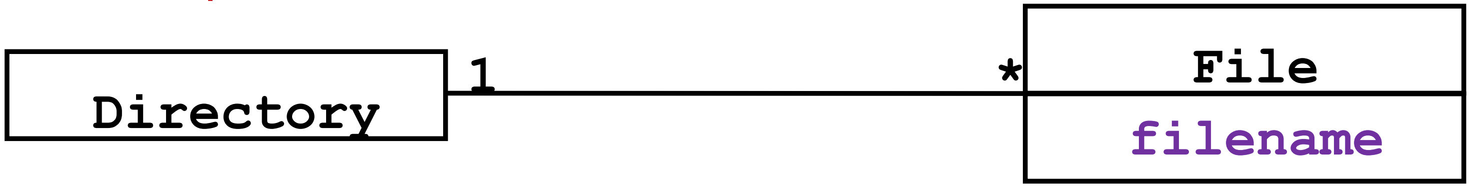
Figure 3.21 Association class vs. ordinary class. An association class is much different than an ordinary class.

- Association class *OwnsStock* has only one occurrence for each pair of *person* and *company* (i.e. only one link exists between pair of person and company objects).
- In contrast there can be any number of *purchase* objects for each pair of *person* and *company* objects.

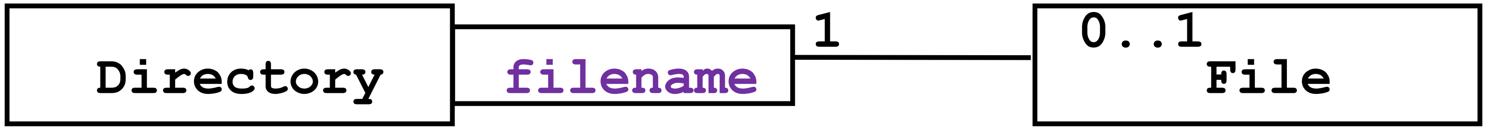
Qualified Association

- A ***qualified association*** is an association in which an attribute called the ***qualifier*** is used to reduce a *many* relation to *one* relation on the other end.

Not qualified



qualified



Qualified Association

- A bank has many accounts [0..*]
- But a bank + account number gives only one account
- Account Number is attribute of account class.

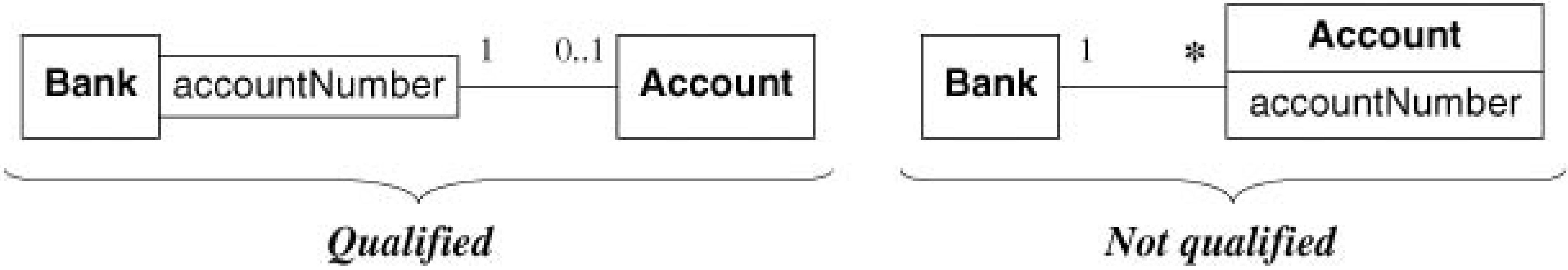


Figure 3.22 Qualified association. Qualification increases the precision of a model.

A sample class model

- Model the classes in a system that represents flights. Each city has at least an airport. Airlines operate flights from and to various airports. A flight has a list of passengers, each with a designated seat. Also a flight uses one of the planes owned by the operating airline. Finally a flight is run by a pilot and a co-pilot.

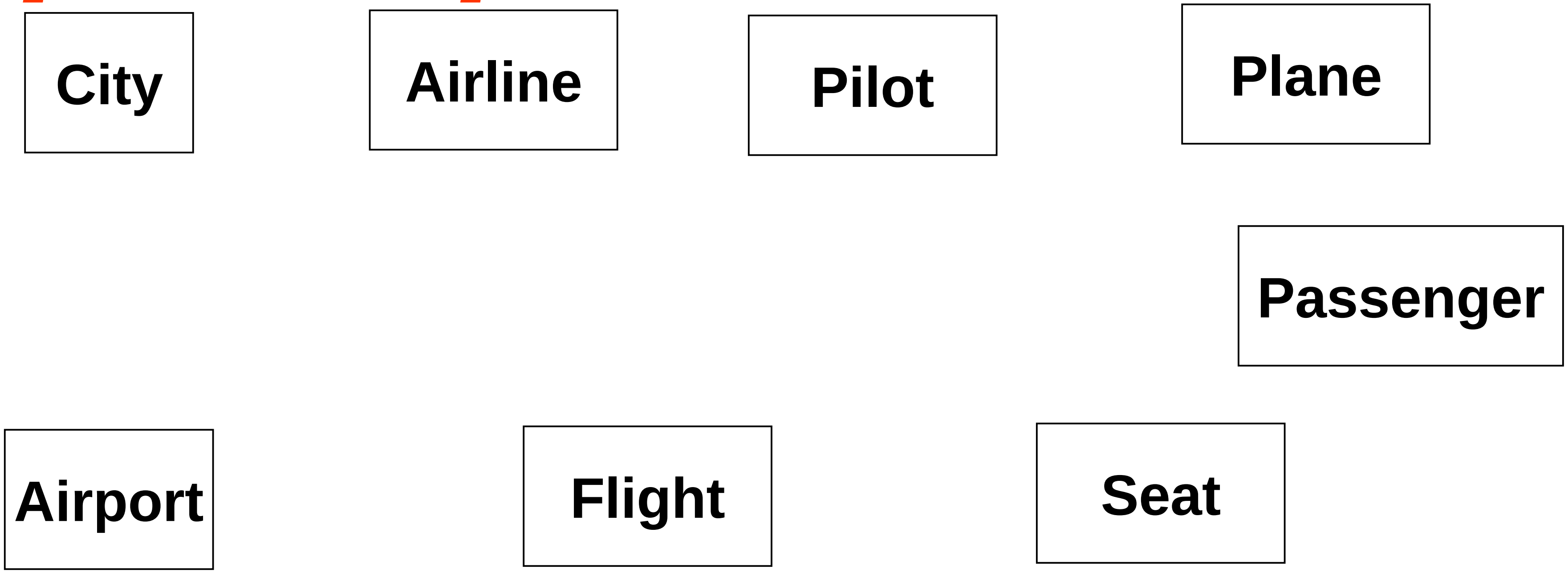
- The order of modeling is:
  - Define classes
  - Define associations
  - Define multiplicity

A sample class model

- Model the classes in a system that represents ***flights***. Each ***city*** has at least an ***airport***. ***Airlines*** operate flights from and to various airports. A flight has a ***list*** of ***passengers***, each with a designated ***seat***. Also a flight uses one of the ***planes*** owned by the operating airline. Finally a flight is run by a ***pilot*** and a ***co-pilot***.

A sample class model

- ***Flights***
- ***city***
- ***Airlines***
- ***list*** of ***passengers***
- ***Seat***
- ***planes***
- ***pilot*** and a ***co-pilot***.



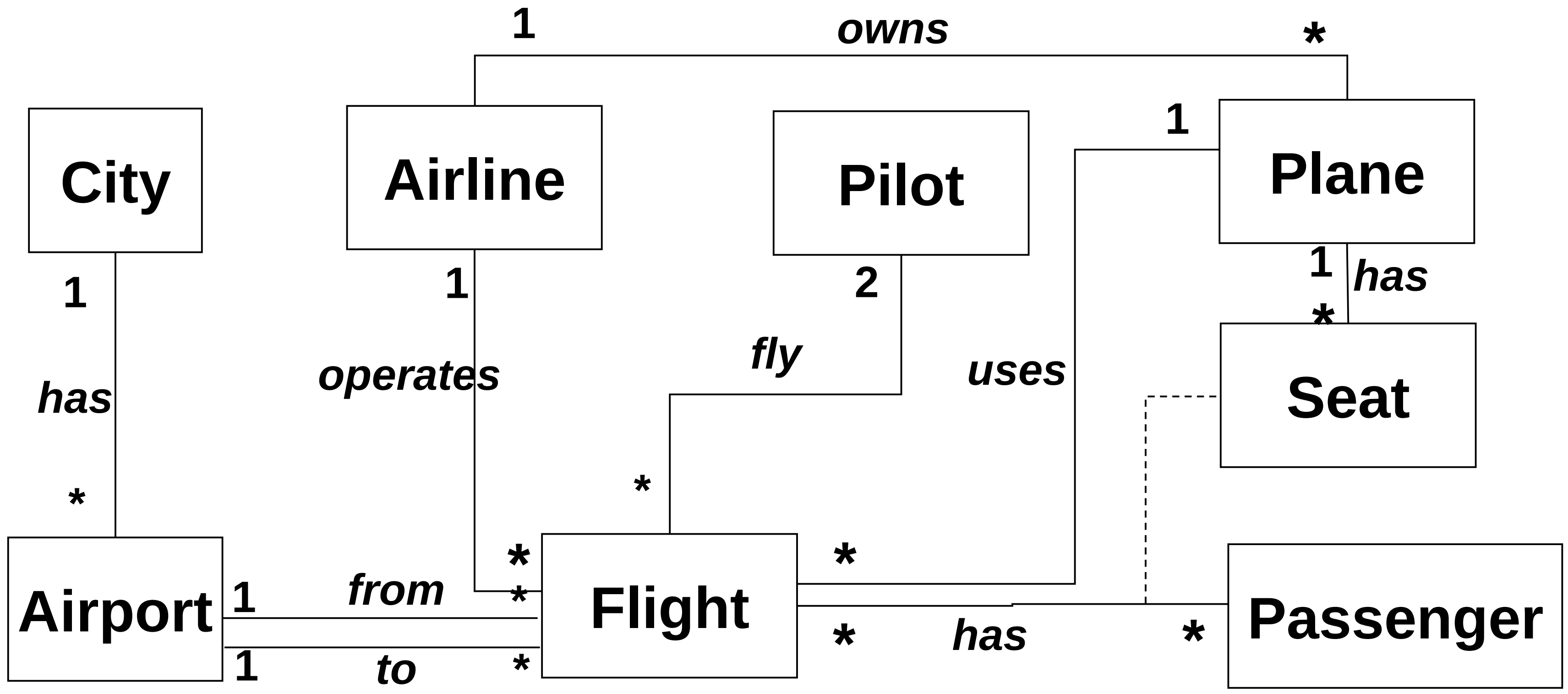
A sample class model

- Model the classes in a system that represents flights. Each city ***has*** at <u>least an</u> airport. Airlines ***operate*** flights ***from*** and ***to*** various airports. A flight ***has*** a list of passengers, each with a designated seat. Also a flight ***uses*** <u>one</u> of the planes ***owned*** by the operating airline. Finally a flight is ***run*** by a pilot and a co-pilot.

A sample class model

City ***has*** airport(s)
Flight ***from*** airport
Flight ***has*** passengers
Flight ***uses*** airplane
Flight ***run by*** pilot and co-pilot

Airline ***operates*** flights
Flight ***to*** airport
Passenger ***has*** seat
Airplane ***owned by*** airline

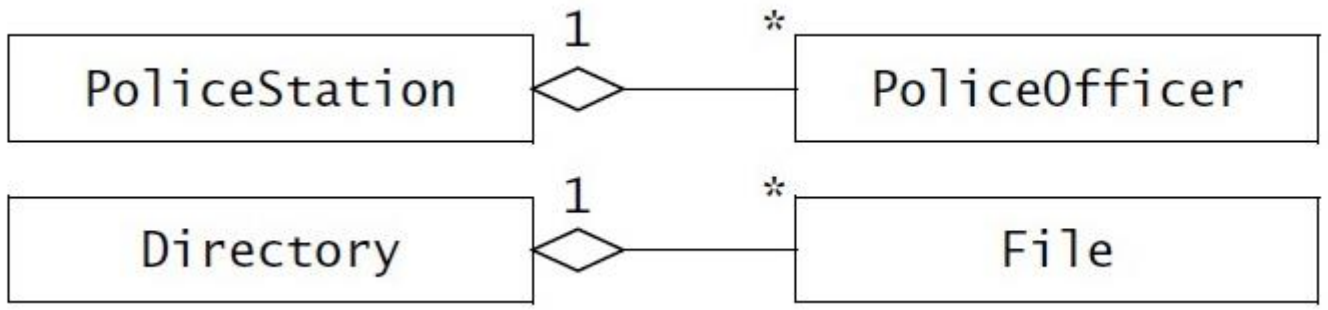


A sample class model

- Add some suitable attributes and operations to the classes of flight system.

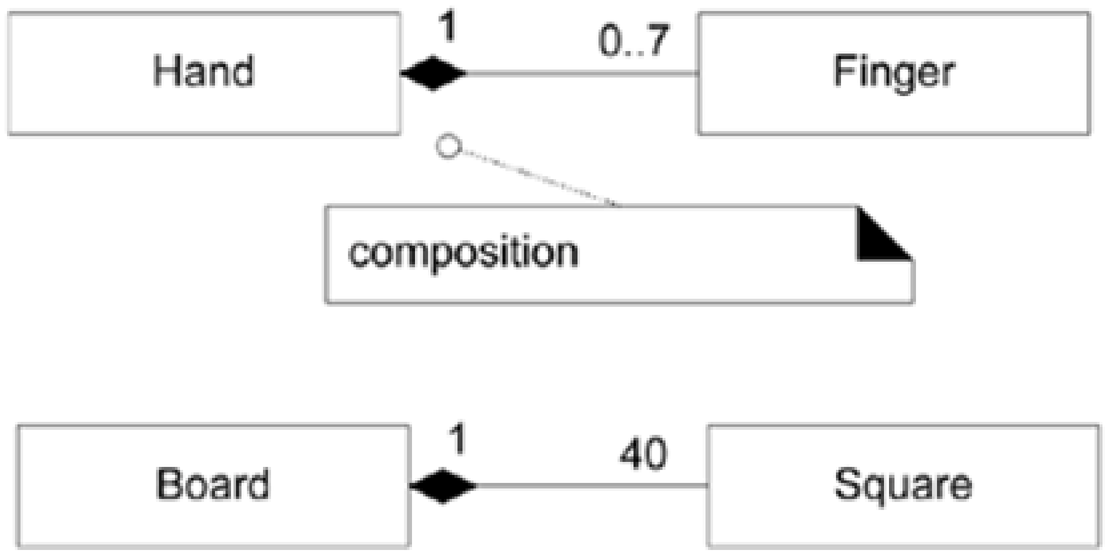
Further Class Concepts
Aggregation Relation

- Aggregation is a special kind of association where an aggregate object contains constituent parts. It's *has-a* relationship

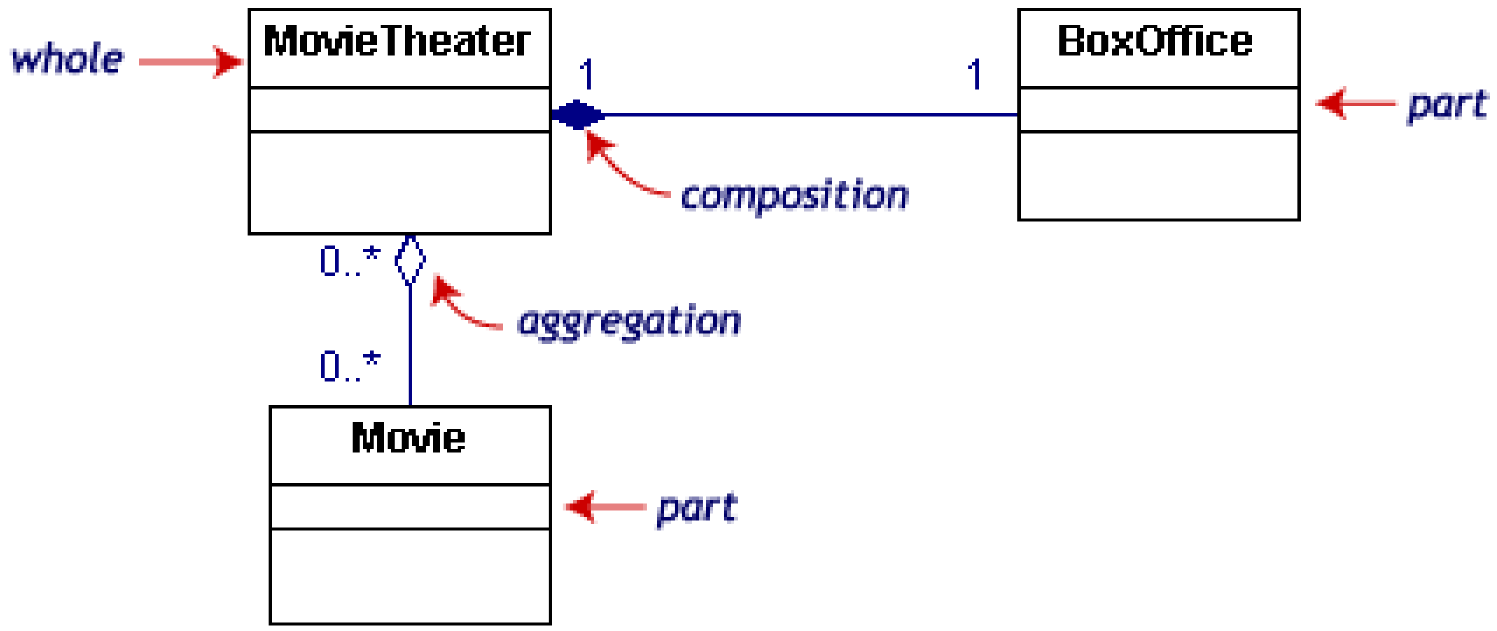


Further Class Concepts
Composition Relation

- Composition is a special kind of association in which a constituent part belongs at most to one assembly and exists only if the assembly exists.



Further Class Concepts Aggregation & Composition Relation



Further Class Concepts Generalization-Specialization and Inheritance

- ***Generalization*** is the relationship between class (the superclass) and one or more variations of the class (the subclass).
- The superclass holds common attributes, operations and associations.
- The subclasses add specific attributes, operations and associations
- Each subclass is said to ***inherit*** the features of the its superclass.
- Inheritance is called "***is-a***" relationship.

Generalization-Specialization and Inheritance

- The terms ***generalization***, ***specialization*** and ***inheritance*** refer to aspects of the same idea, they are used in place of one another .
- ***Generalization*** refers to the super-class generalizing its subclasses
- ***Specialization*** refers to the fact that the subclass specializes or refines the super-class,
- ***Inheritance*** refers to the mechanism for implementing *generalization* / specialization.

Generalization, Specialization and Inheritance

- Generalization is ***transitive*** across an arbitrary number of levels.
- An ***ancestor*** is parent or grandparent class of a class
- A ***descendant*** is a child or grandchild of a class

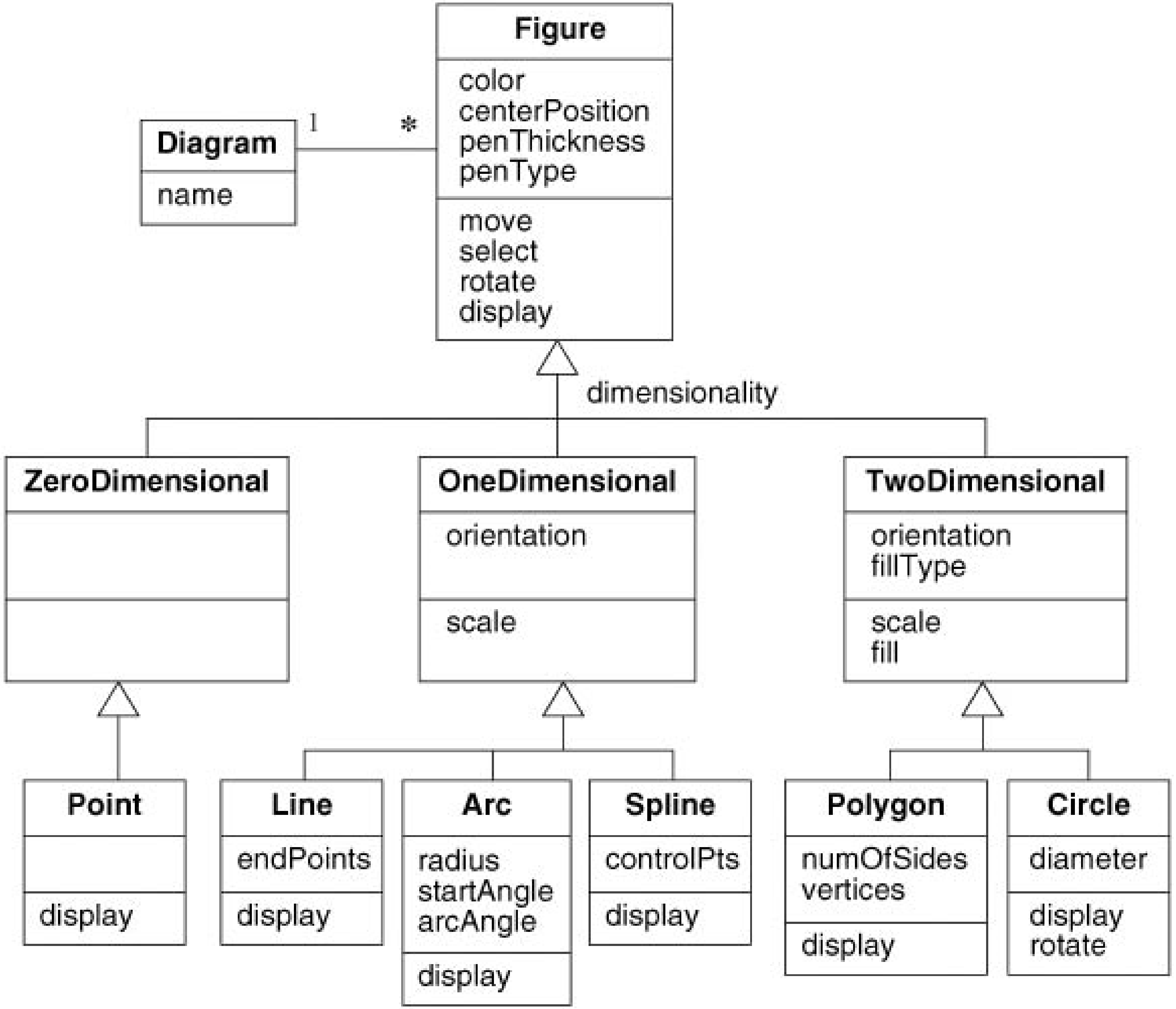


Figure
color
centerPosition
penThickness
penType
move
select
rotate
display
Diagram
name
1
*
dimensionality
ZeroDimensional
OneDimensional
orientation
scale
TwoDimensional
orientation
fillType
scale
fill
Point
display
Line
endPoints
display
Arc
radius
startAngle
arcAngle
display
Spline
controlPts
display
Polygon
numOfSides
vertices
display
Circle
diameter
display
rotate

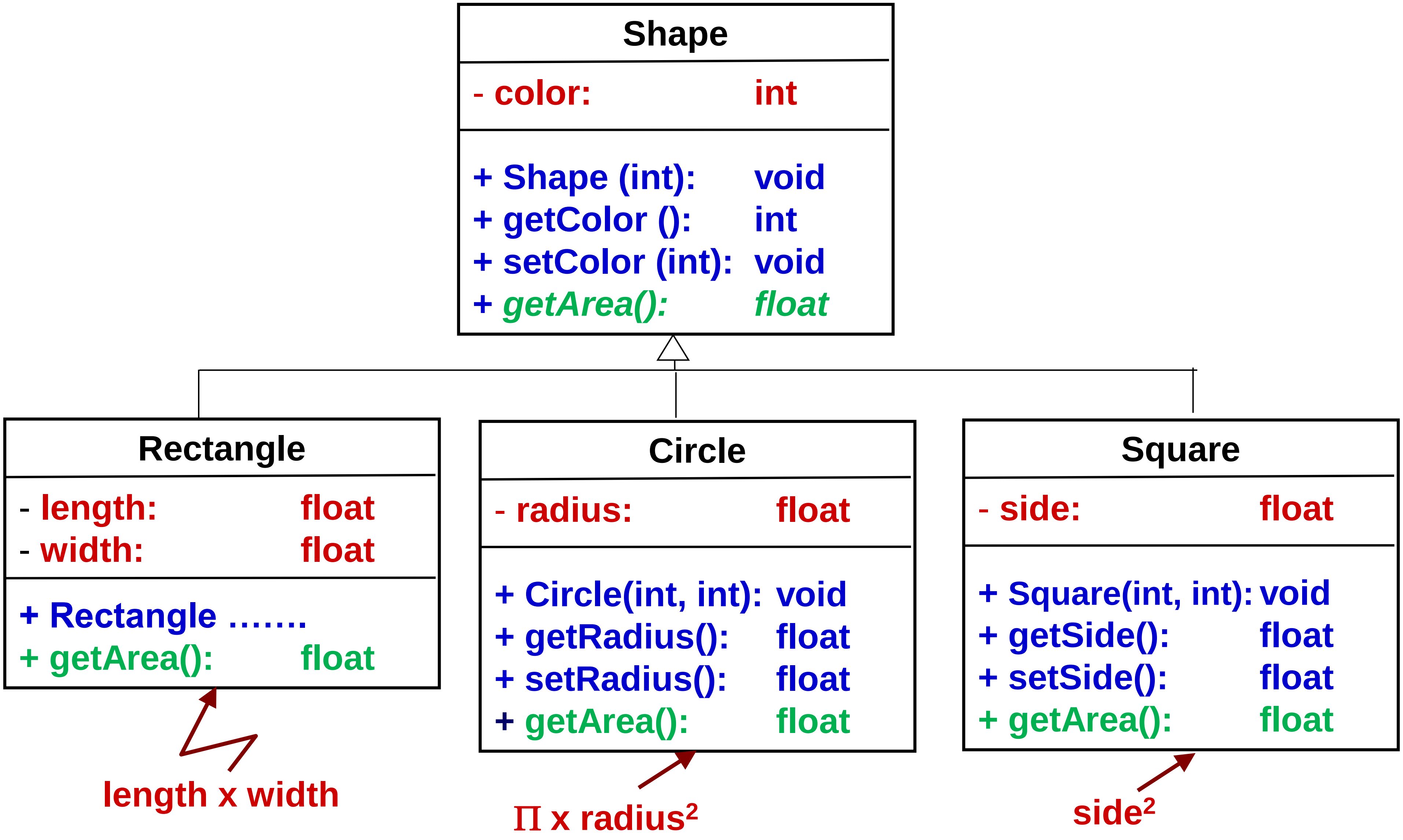
Use of Generalization

- Generalization serves three purposes.
  1. *Supporting polymorphism*.
  2. *Structuring the description of objects* and their relation to each other based on their similarity and differences.
  3. *Reusing code*. You inherit code from super-classes or from a class library automatically with inheritance. Reuse is more productive than repeatedly writing code form scratch. When reusing code, you can adjust the code if necessary to get the exact desired behavior.

Overriding Features

- A subclass may ***override*** a superclass feature by defining a feature with the same name.
- The overriding feature (in the sub-class) replaces the overridden feature (in the super-class)
- Never override a feature so that it is inconsistent with the original inherited feature.
- Overriding should preserve the attribute type, number and type of arguments of an operation and its return type.

Generalization, Specialization and Inheritance



Further Class Concepts
Abstract Class

- An incomplete class that cannot be instantiated.

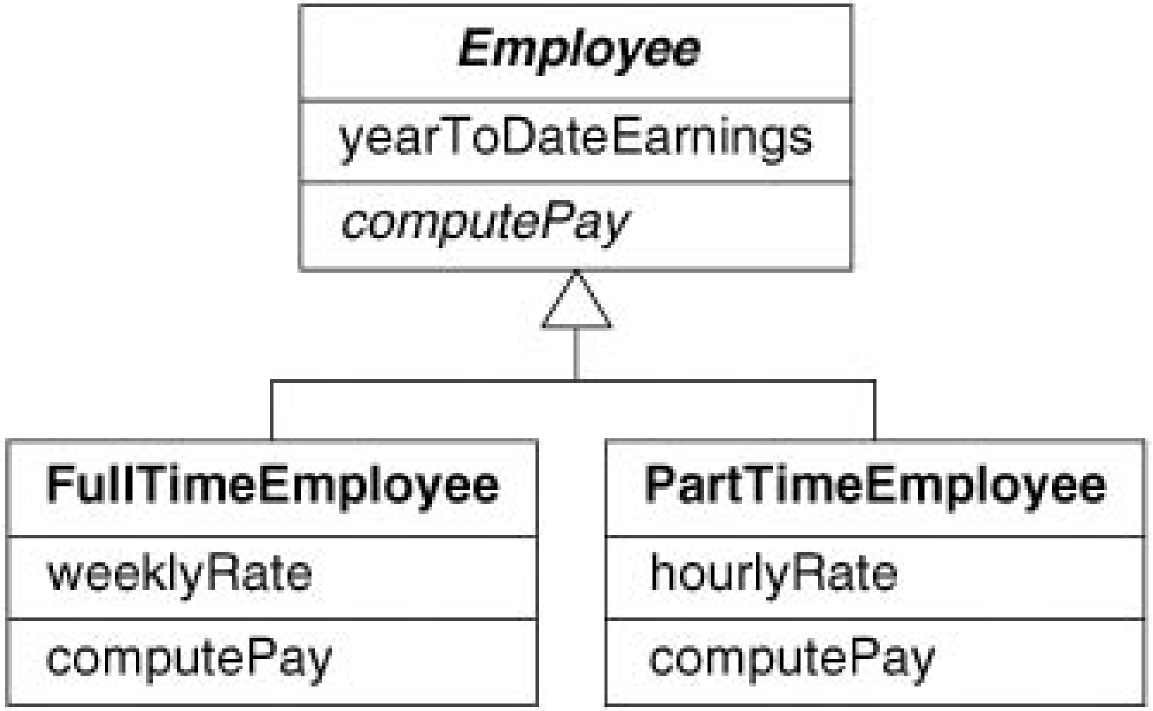


Figure 4.13 Abstract class and abstract operation. An abstract class is a class that has no direct instances.

Class Modeling Tips

- Scope: A model represents the relevant aspects of the problem. *Exercise judgment in deciding which objects to show and which ones to ignore.*
- Simplicity: *Make the model as simple as possible.* Simpler models are easier to understand and develop.
- Diagram Layout: *Draw your diagram in a way that is easy to understand*. Avoid cross lines. Position important classes where they are visually prominent.
- *Names: Carefully choose class names*. Choose descriptive names and use singular nouns to name classes.

Package Diagram

- Packages let you organize large models so that they are more understandable.

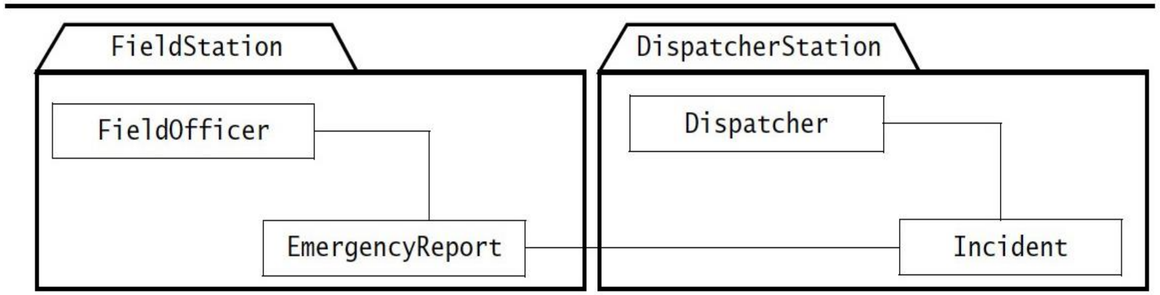
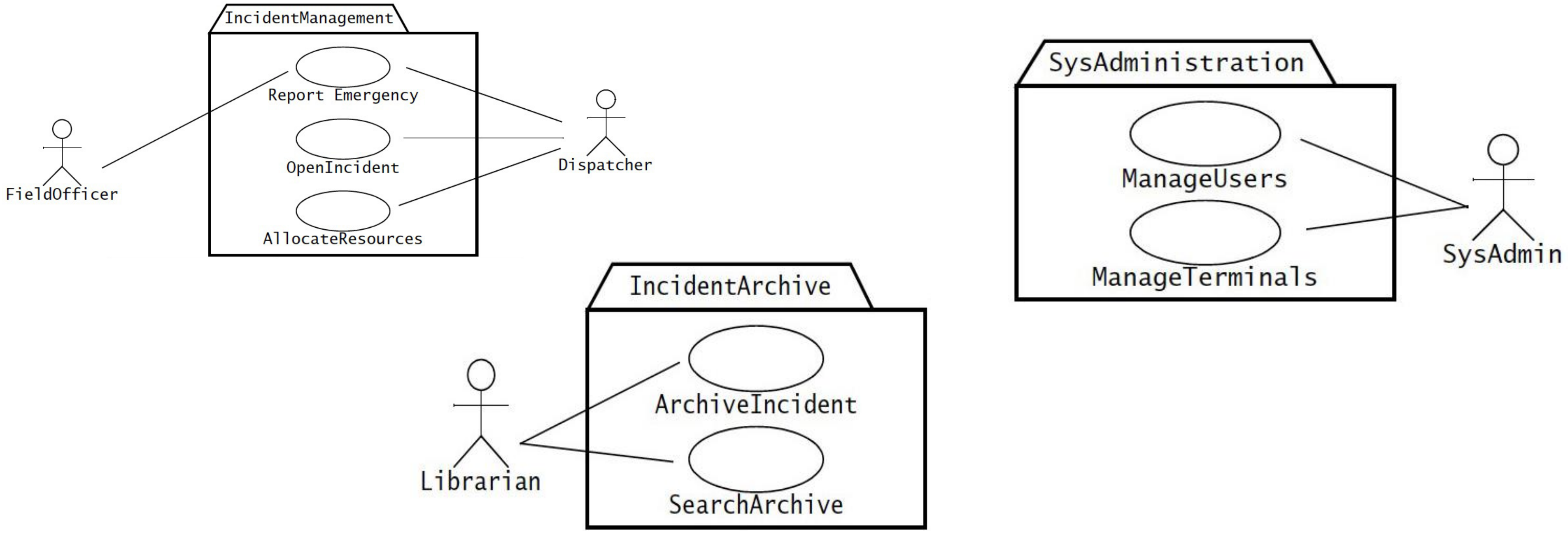


Figure 2-47 Example of packages. The FieldOfficer and EmergencyReport classes are located in the FieldStation package, and the Dispatcher and Incident classes are located on the DispatcherStation package.

Package Diagram



Example of packages: UseCase are organized by actors

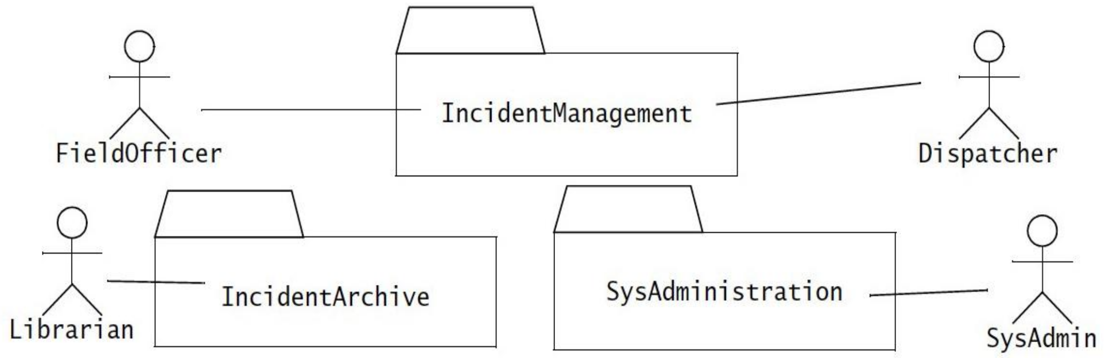


Figure 2-46 Example of packages. This figure displays the same packages as Figure 2-45 except that the