MACIASZEK, L.A. (2007):
***Requirements Analysis and System Design*, 3rd ed.**
Addison Wesley, Harlow England
ISBN 978-0-321-44036-5

Chapter 4
Moving from Analysis to Design

Topics

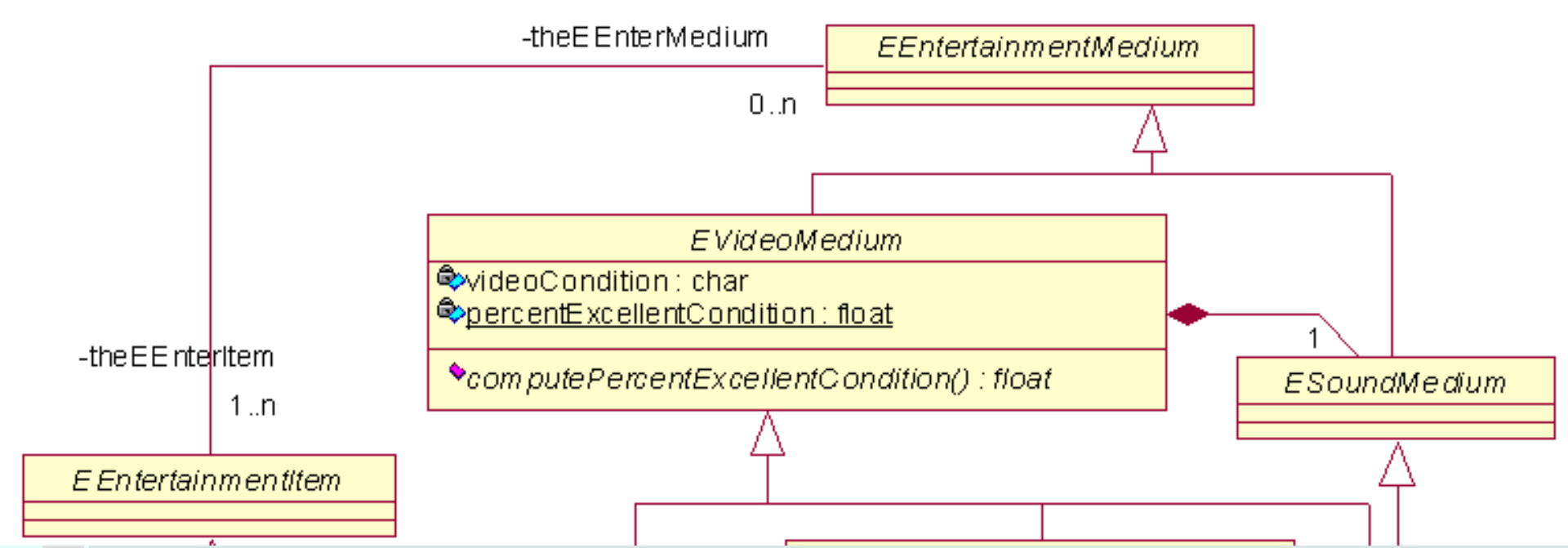
- Advanced class modeling
- Advanced generalization and inheritance modeling
- Advanced aggregation and delegation modeling
- Advanced interaction modeling

2. Advanced generalization and inheritance modeling

- *Generalization* is a useful and powerful concept, but it can also create many problems
- Intricate mechanisms of *inheritance*

Generalization and substitutability

- Generalization introduces new classes, it can reduce the overall number of *association* and *aggregation* relationships in the model
- The benefits of generalization arise from the *substitutability* principle – a subclass object can be used in place of a superclass object



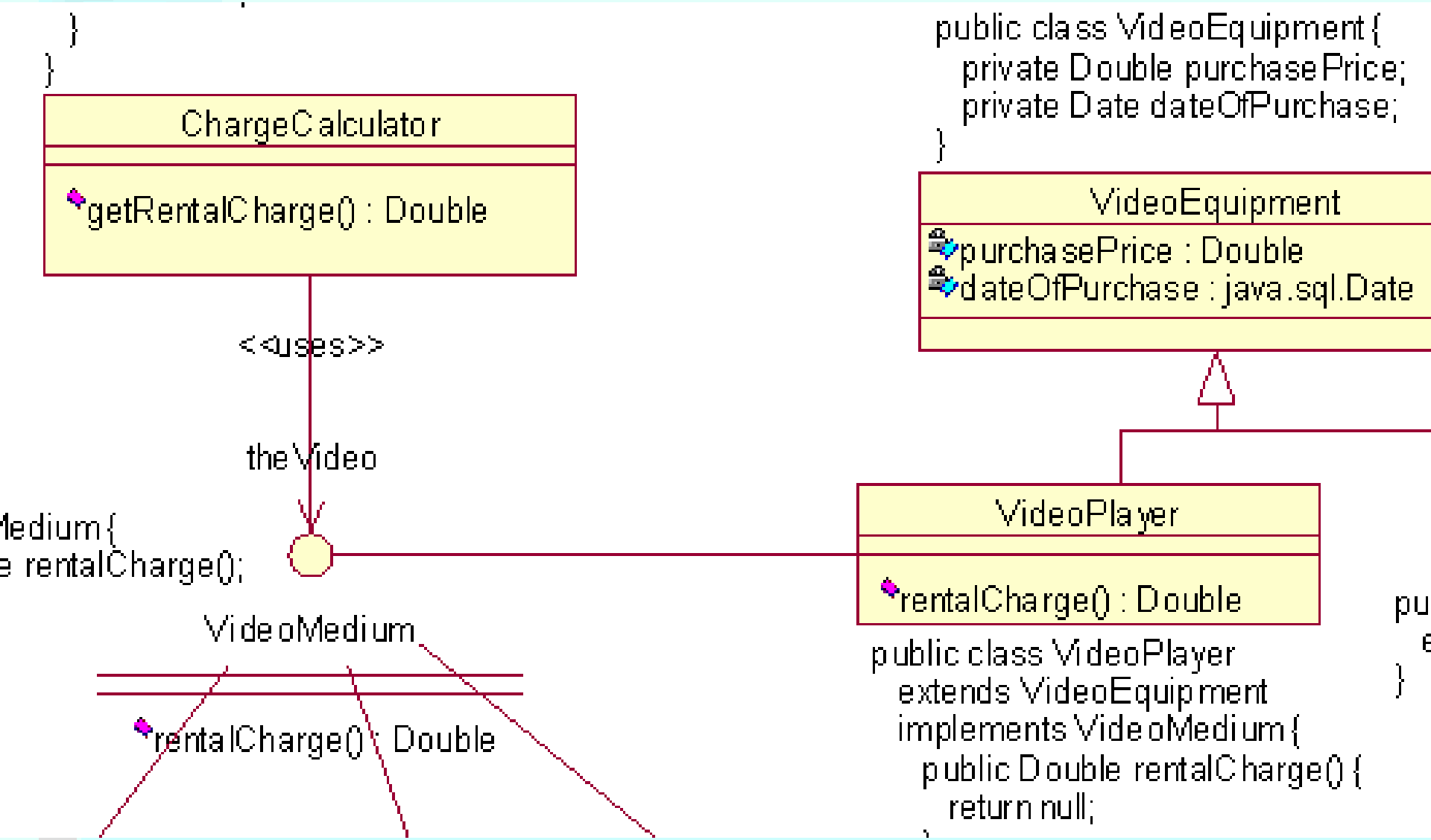
Inheritance versus encapsulation

- *Encapsulation* demands that an object's state (attribute values) be accessible only through the operations in the object's interface.
- Encapsulation is orthogonal to inheritance and query capabilities and has to be traded off against these two features.
- Encapsulation refers to the notion of the class, not the object – in most object programming environments (with the exception of Smalltalk) an object cannot hide anything from another object of the same class.

Interface inheritance

- When generalization is used with the aim of substitutability, then it may be synonymous with the notion of *interface inheritance* (*subtyping*, *type inheritance*).
- Interface inheritance provides a means of achieving multiple implementation inheritance in languages that do not support such inheritance (like in Java).
- There is a difference between the notions of *interface* and *abstract class*.

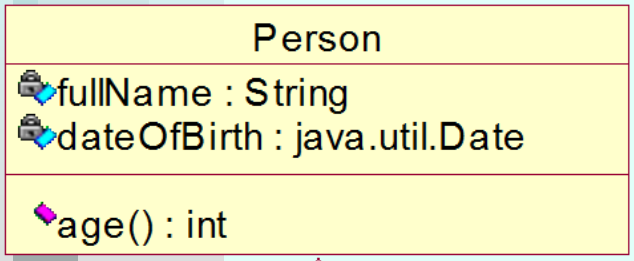
Interface and implementation inheritance

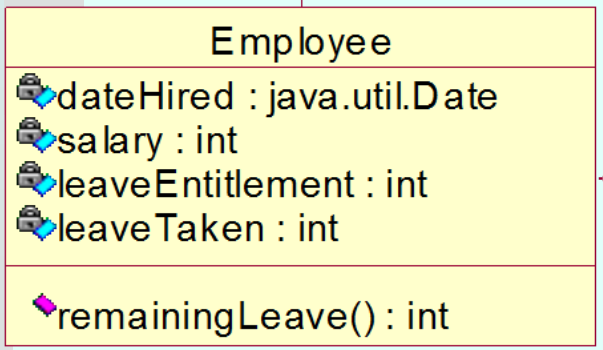


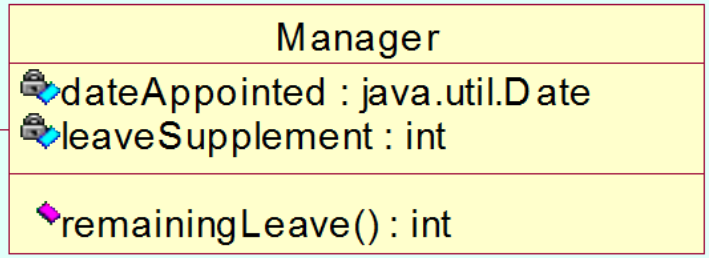
Implementation inheritance

- Generalization can be used to imply *substitutability*, and it can then be realized by an *interface inheritance*
  - it involves only the inheritance of contract fragments – operation signatures
- However, generalization can also be used (deliberately or not) to imply *code reuse*, and it is then realized by an *implementation inheritance*
  - it involves the inheritance of code – the inheritance of implementation fragments
- *Implementation inheritance* – also called *subclassing*, *code inheritance*, or *class inheritance* – combines the superclass properties in the subclasses and allows them to be *overridden* with new implementations when necessary.

Extension inheritance is OK



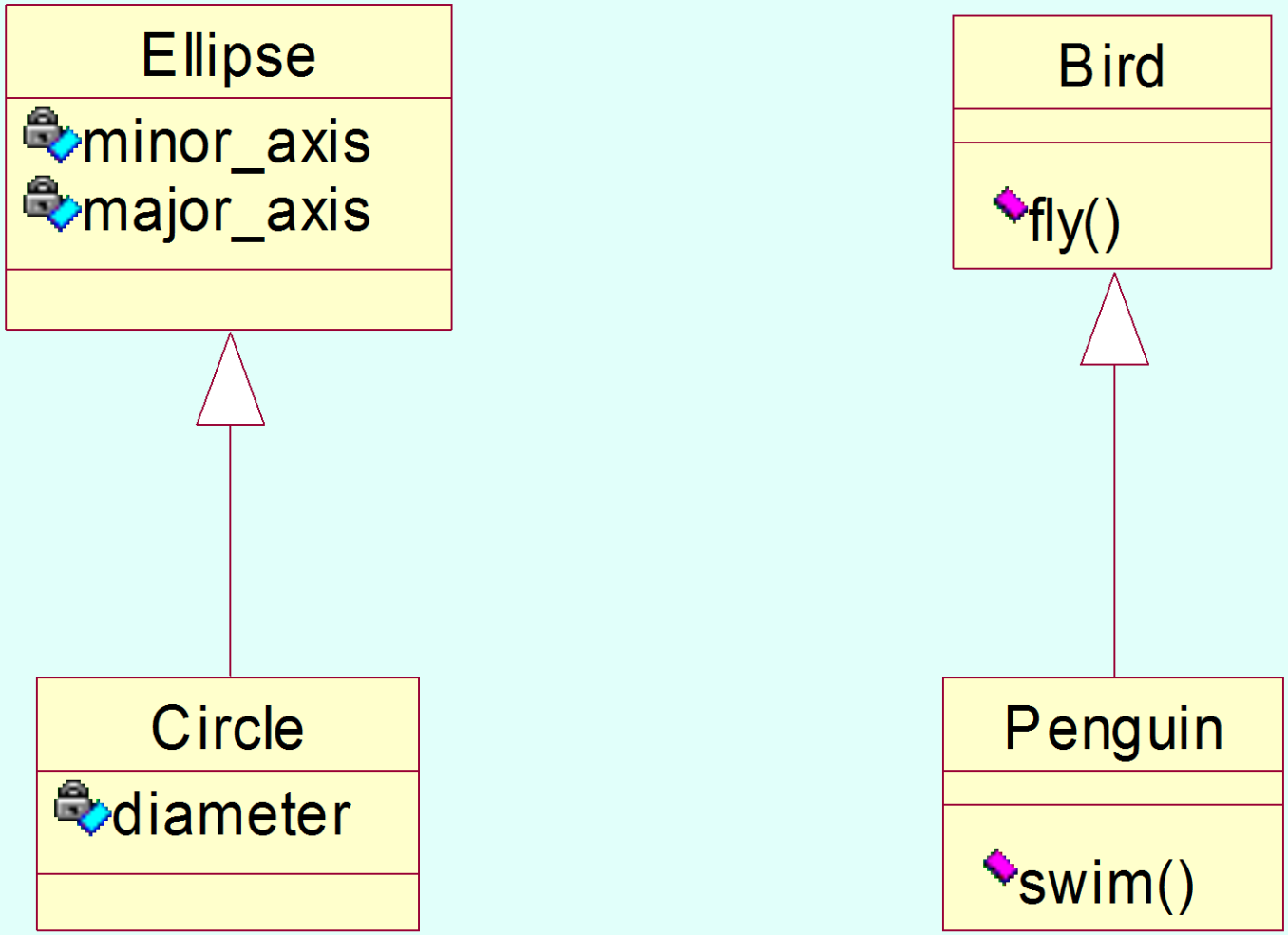




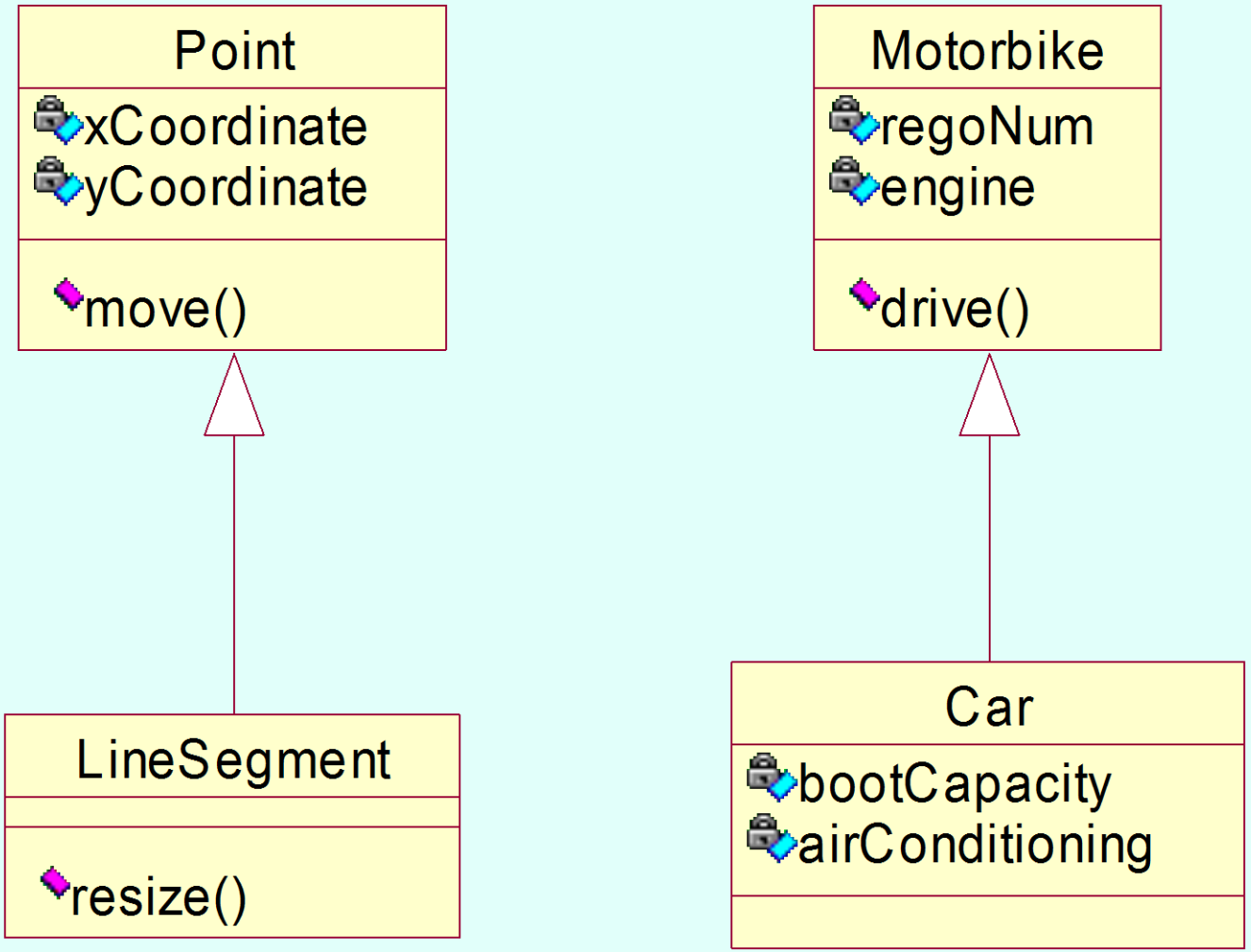

```
public class Manager extends Employee
{
    private Date dateAppointed;
    private int leaveSupplement;

    public int remainingLeave()
    {
     int mrl;
     mrl = super.remainingLeave() + leaveSupplement;
     return mrl;
    }
}
```

Restriction inheritance is problematic



Convenience inheritance is not OK



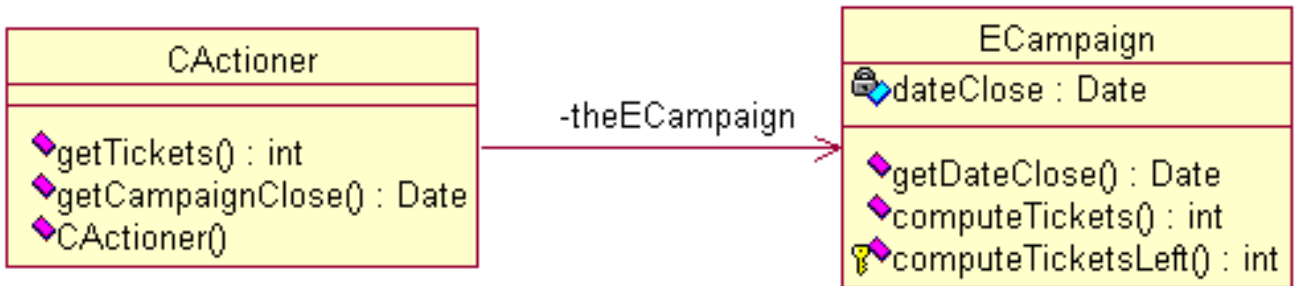
The evils of implementation inheritance

- fragile base class
- overriding and callbacks
- multiple implementation inheritance

Fragile base class

- Change in a superclass automatically affects the existing subclasses
- Changes on superclass' public interfaces are particularly troublesome
  - Changing the signature of a method.
  - Splitting the method into two or more new methods.
  - Joining existing methods into a larger method.
- 'madness is inherited, you get it from your children'

Overriding and down-calls

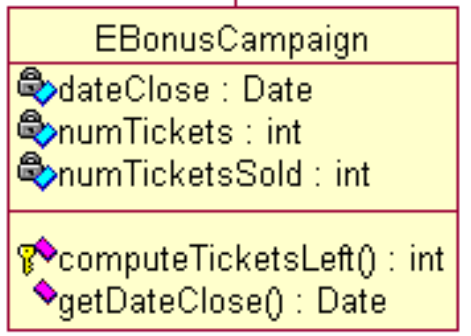



```
public CActioner()
{
  theECampaign = new EBonusCampaign();
}

public void getTickets()
{
  int tnum;
  tnum = theECampaign.computeTickets();
  return tnum;
}
```

```
public int computeTickets()
{
  int tnum;
  tnum = computeTicketsLeft();
  return tnum;
}
```

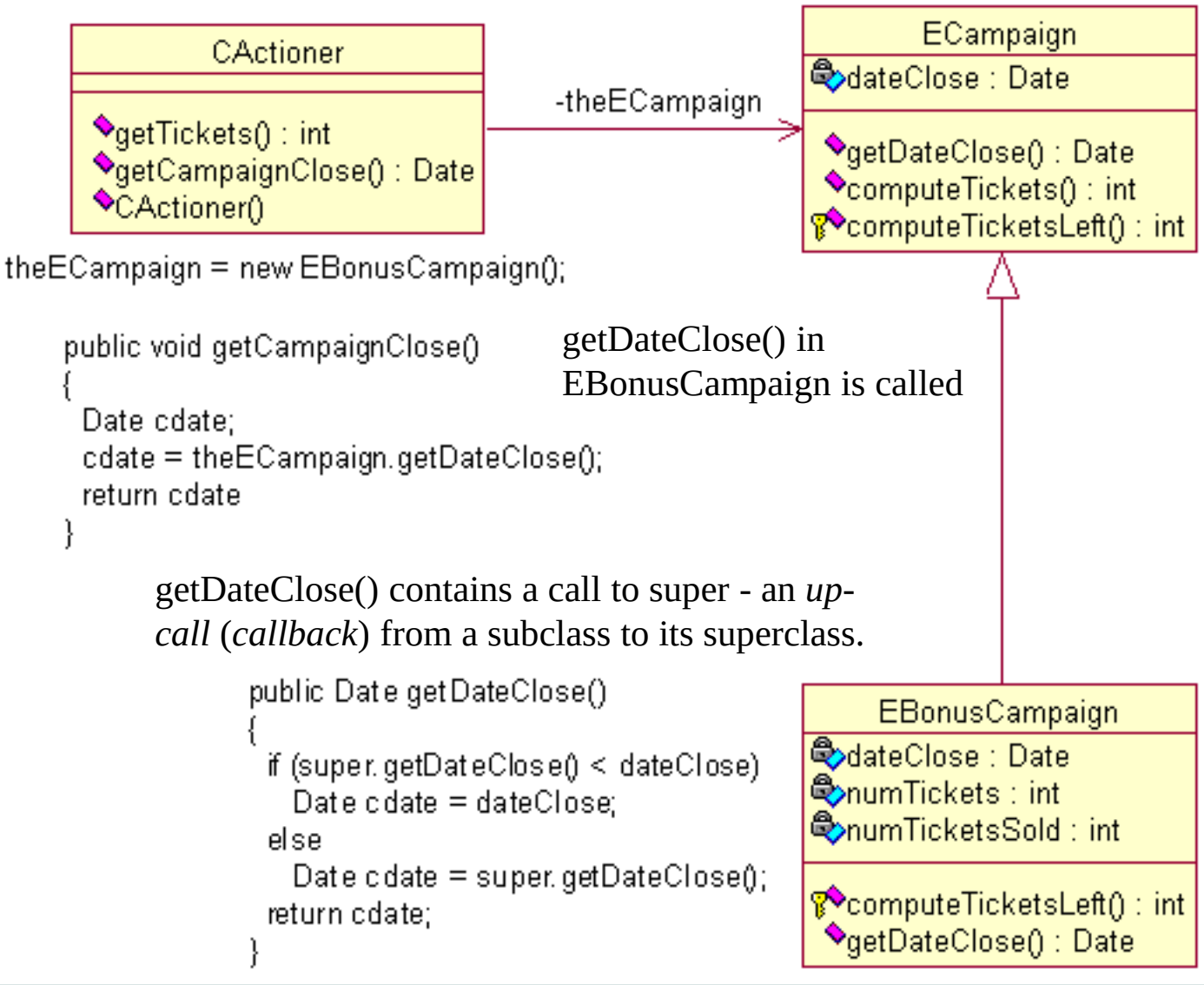
computeTicketsLeft() is overridden in EBonusCampaign and this overridden method is called




```
protected int computeTicketsLeft()
{
  int tnum;
  tnum = numTickets - numTicketsSold;
  return tnum;
}
```

computeTickets() uses ECampaign's method because EBonusCampaign (referenced in theECampaign) does not have its own, overridden, version of computeTickets ()).

Overriding and up-calls




```
theECampaign = new EBonusCampaign();
```

```
public void getCampaignClose()
{
  Date cdate;
  cdate = theECampaign.getDateClose();
  return cdate
}
```

getDateClose() in EBonusCampaign is called

getDateClose() contains a call to super - an *up-call* (*callback*) from a subclass to its superclass.

```
public Date getDateClose()
{
  if (super.getDateClose() < dateClose)
    Date cdate = dateClose;
  else
    Date cdate = super.getDateClose();
  return cdate;
}
```

Multiple inheritance

- ***Multiple interface inheritance***
  - multiple supertyping
  - merging of interface contracts)
- ***Multiple implementation inheritance***
  - multiple superclassing
  - merging of implementation fragments
  - exaggerates the problems due to the fragile base class, overriding, and callbacks
  - some problems result from the lack of support in object systems for *multiple classification*
- Java recommends using *multiple interface inheritance* to provide solutions that otherwise would require *multiple implementation inheritance*

Review Quiz 4.2

1. What related principle makes generalization useful?
2. How does inheritance compromise encapsulation?
3. What concept can be used in lieu of multiple implementation inheritance?