6.1 Introduction to Patterns

The recurring aspects of designs are called *design patterns*.

- A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context
- Many of them have been systematically documented for all software developers to use
- A good pattern should
  - —Be as general as possible
  - —Contain a solution that has been proven to effectively solve the problem in the indicated context.

Studying patterns is an effective way to learn from the experience of others

Pattern description

Context:

- The general situation in which the pattern applies

Problem:

- A short sentence or two raising the main difficulty.

Forces:

- The issues or concerns to consider when solving the problem

Solution:

- The recommended way to solve the problem in the given context.
  - ‘to balance the forces’

Antipatterns: (Optional)

- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)

- Patterns that are similar to this pattern.

References:

- Who developed or inspired the pattern.

6.2 The Abstraction-Occurrence Pattern

- ***Context***:
  - Often in a domain model you find a set of related objects (*occurrences)*.
  - The members of such a set share common information
    - but also differ from each other in important ways.
- ***Problem***:
  - What is the best way to represent such sets of occurrences in a class diagram?
- ***Forces***:
  - You want to represent the members of each set of occurrences without duplicating the common information

Examples

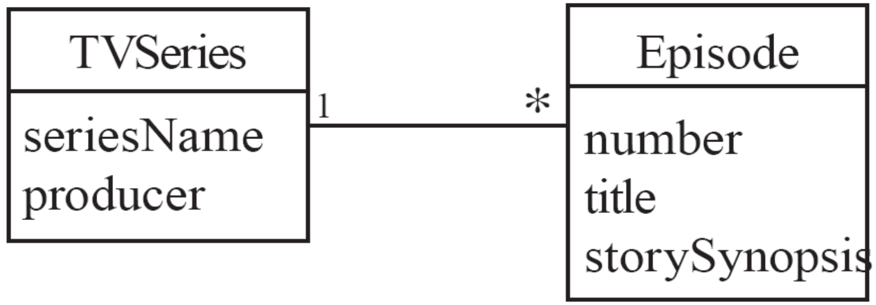
- ***All the episodes of a television series***
  - ***They have the same producer and the same title, but different episode numbers and story-line***

- ***The flights that leave at the same time everyday for the same destination***
  - ***They have the same flight number, but occur on different days and carry different passengers and crew***

- ***All the copies of the same book in a library***
  - ***They have the same title and author, but different barcode identifiers and potentially different borrowers***

Abstraction-Occurrence: Solution

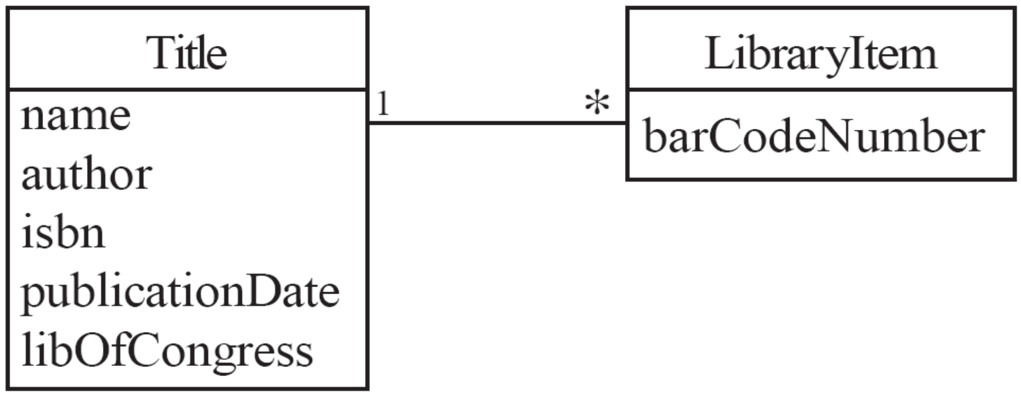
Abstraction-Occurrence: Example



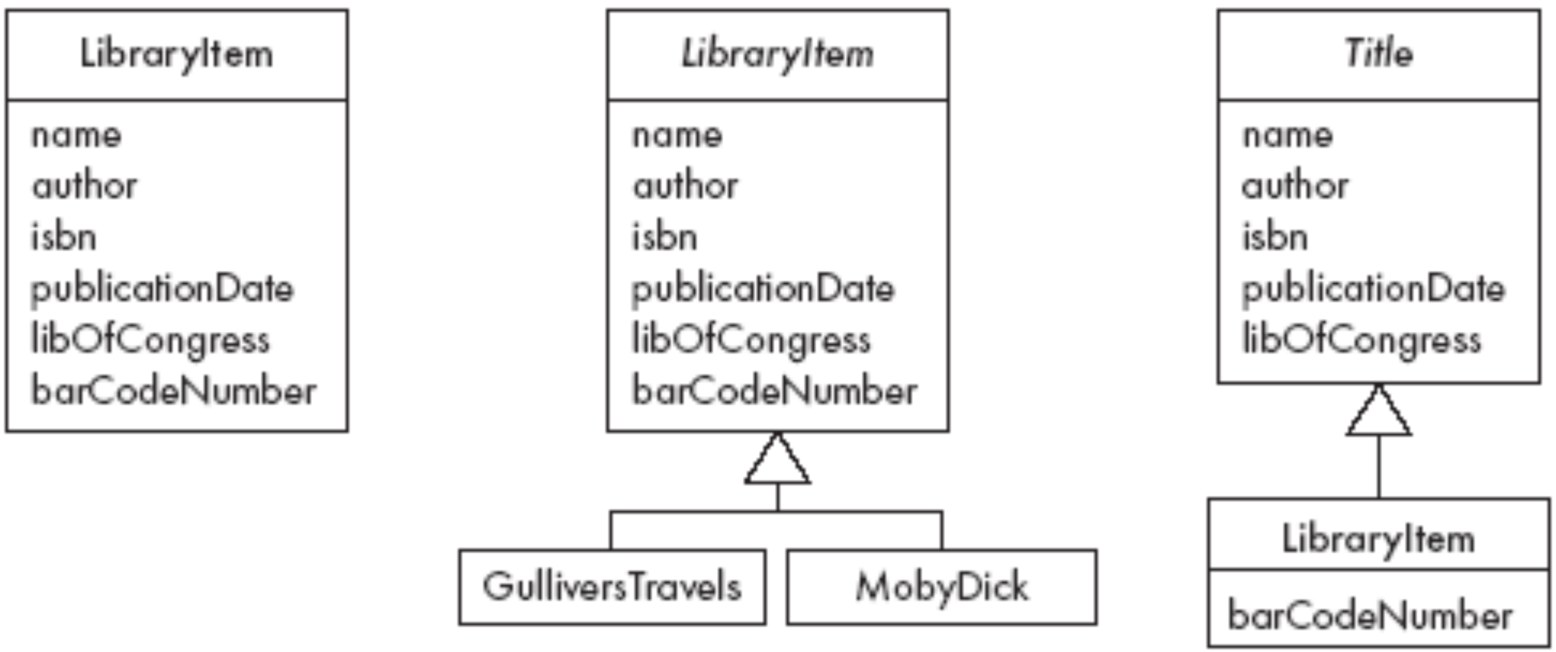

```
public class TVSeries{
   private String seriesName;
   private String producer;
   private Vector<Episode> episode;

   void addEpisode(Episode episode){
      this.episode.add(episode);
   }
}
```

```
public class Episode{
   private int number;
   private String title;
   private String storySynopsis;
   private TVSeries series;

   public Episode(TVSeries series,
                  int number,
               String title,
                  String storySynopsis){
      this.series = series;
      this.number = number;
      this.title = title;
      this.storySynopsis = storySynopsis;
      series.addEpisode(this);
 } }
```

Abstraction-Occurrence: Example

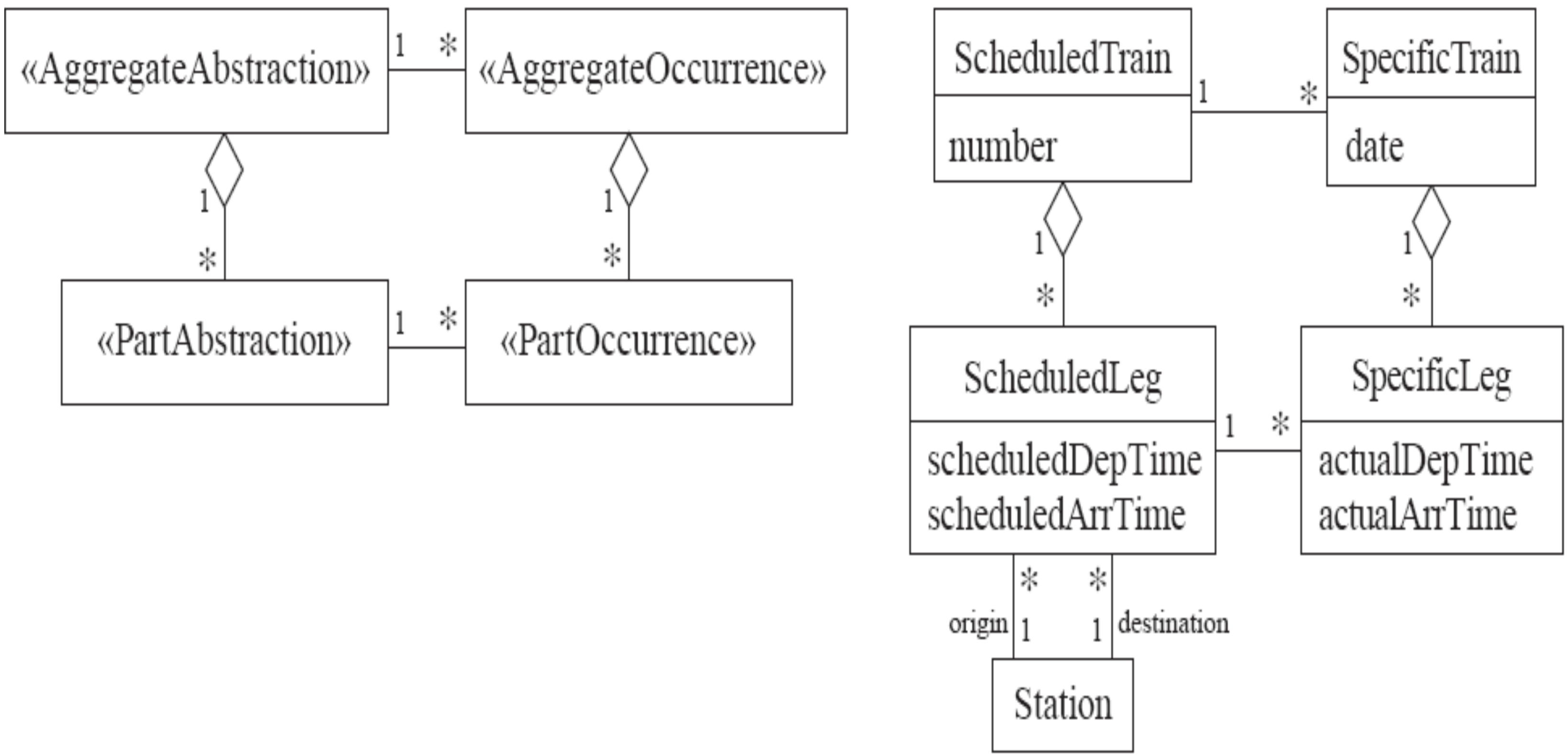


Abstraction-Occurrence: Antipatterns



Abstraction-Occurrence Square Pattern

When the abstraction is an aggregate, the occurences are typically aggregates

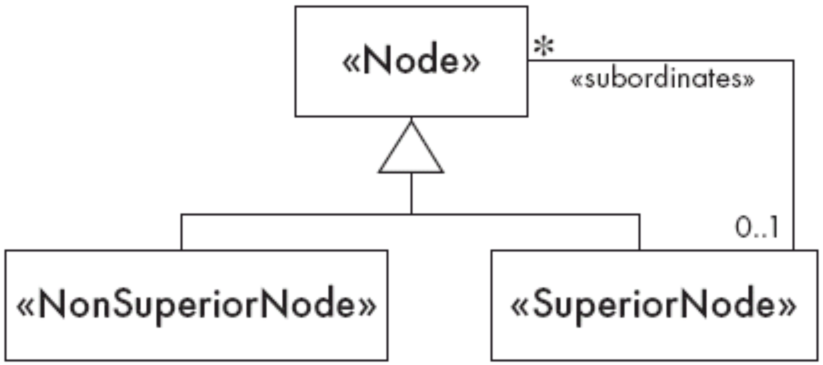


6.3 The General Hierarchy Pattern

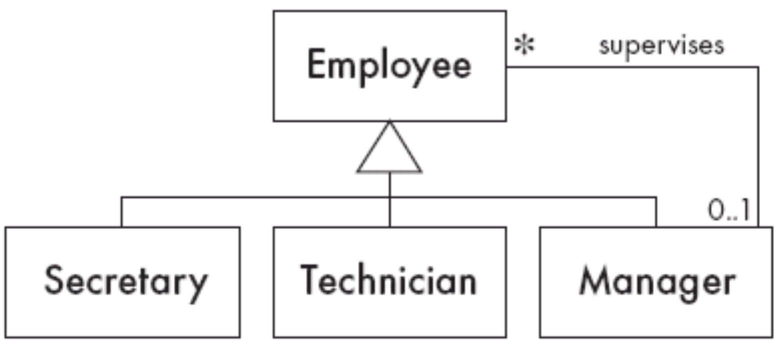
- ***Context***:
  - —Objects in a hierarchy can have one or more objects above them (superiors),
    - and one or more objects below them (subordinates).
  - —Some objects cannot have any subordinates
- ***Problem***:
  - —How do you represent a hierarchy of objects, in which some objects cannot have subordinates?
- ***Forces***:
  - —You want a flexible way of representing the hierarchy
    - that prevents certain objects from having subordinates
  - —All the objects have many common properties and operations

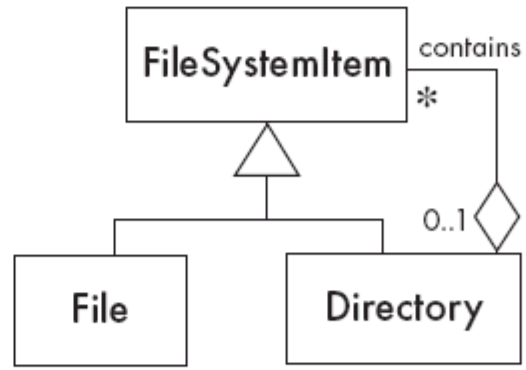
General Hierarchy

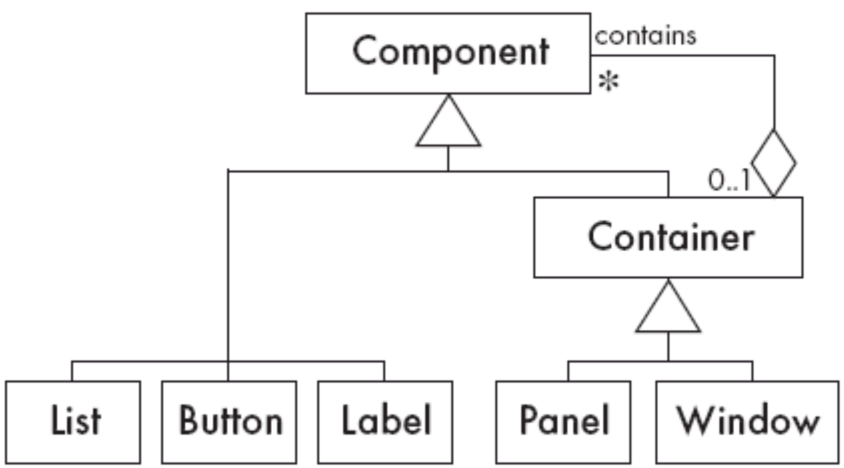
- ***Solution:***

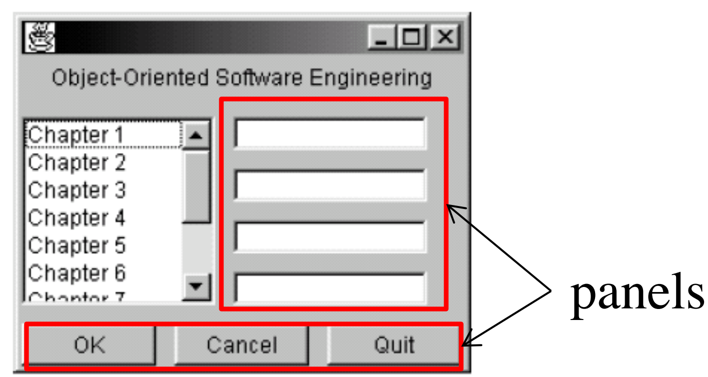


General Hierarchy: Examples

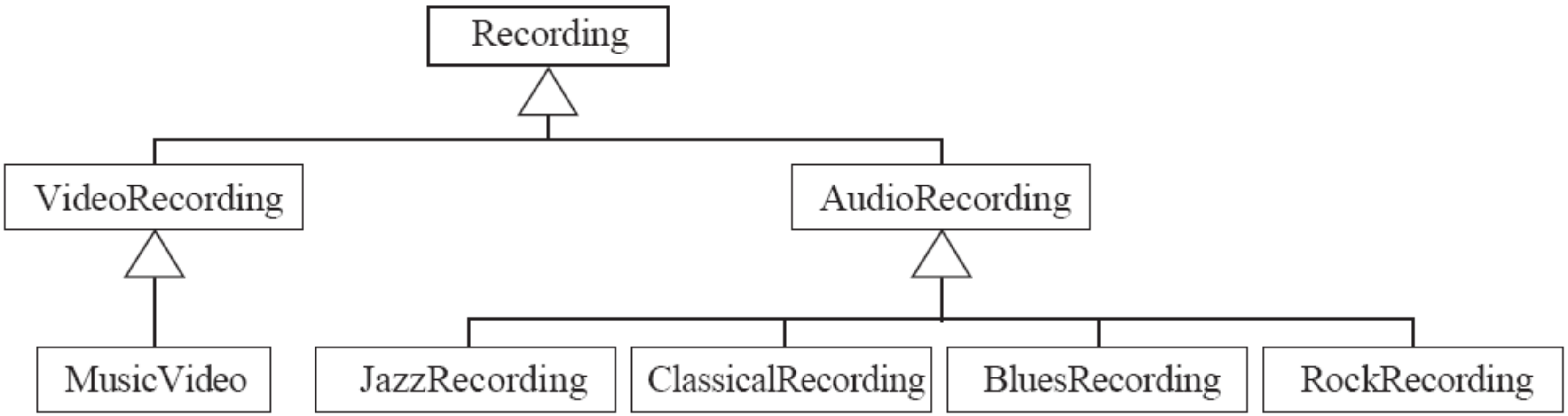








General Hierarchy: Antipattern



6.4 The Player-Role Pattern

- ***Context***:
  - A *role* is a particular set of properties associated with an object in a particular context.
  - An object may *play* different roles in different contexts.
- ***Problem***:
  - How do you best model players and roles so that a player can change roles or possess multiple roles?

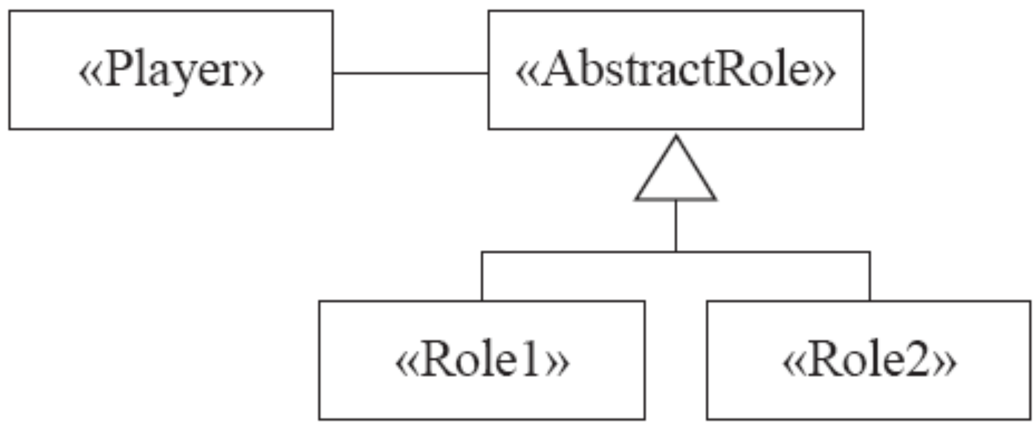
Player-Role

- ***Forces***:
  - —It is desirable to improve encapsulation by capturing the information associated with each separate role in a class.
  - —You want to avoid multiple inheritance.
  - —You cannot allow an instance to change class

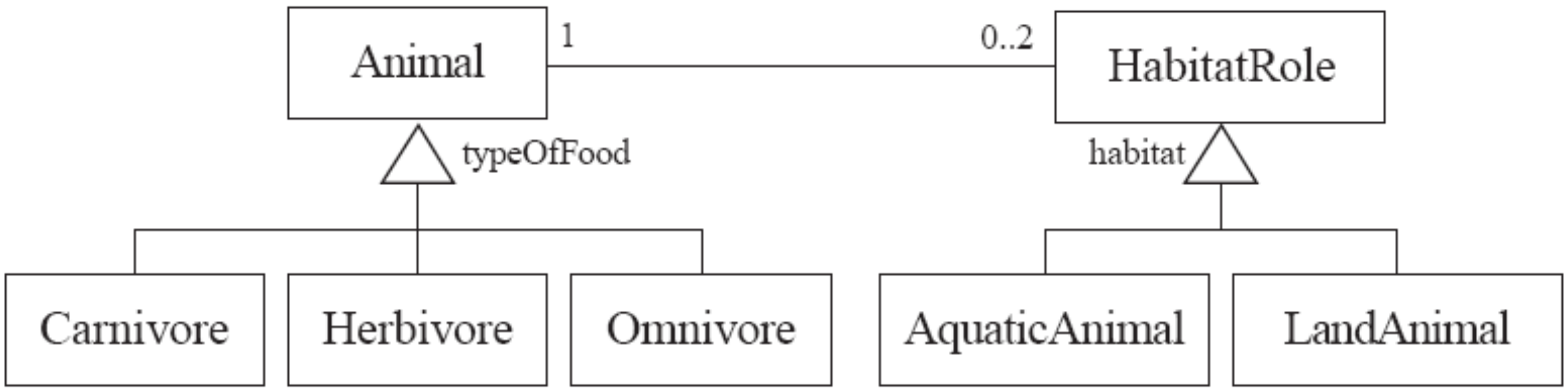
- ***Example:***
  - — A student can be either an undergraduate student or a graduate student at any given point in time – and it is likely to need to change from one of these roles to another
  - — A student can also be registered as a full-time student or as part-time student – in this case, a student may change roles several times

Player-Role

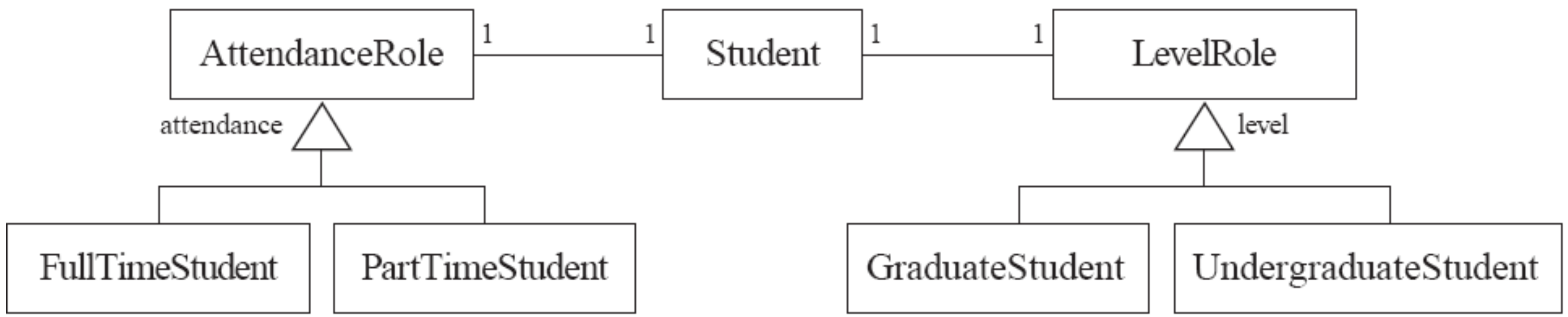
An abstract class
or an interface



Player-Role: Example



Player-Role




```
public class Student {
   private AttendanceRole attendance;
   private LevelRole level;

   …
}
```

Player-Role

```
public class Student {
    …
    public void setAttendanceRole(AttendanceRole a){attendance = a; …}

    public void setLevelRole(LevelRole l){level = l; …}

    public char getPassingGrade(){level.getPassingGrade();}
}
```

```
public abstract class  LevelRole{
    abstract char getPassingGrade();
    …
}
```

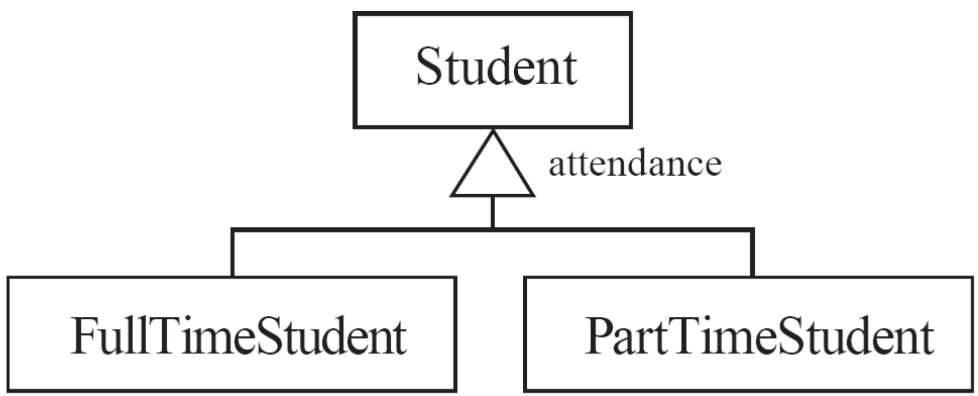
```
public class GraduateStudent
 extends LevelRole {
   char getPassingGrade(){return 'C';}
}
```

```
public class UndergraduateStudent
extends LevelRole {
   char getPassingGrade(){return 'D';}
}
```

Player-Role

Antipatterns:

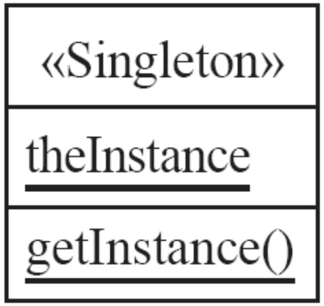
- Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.
  - This, however, creates an overly complex class – and much of the power of object orientation is lost
- Create roles as subclasses of the «Player» class.



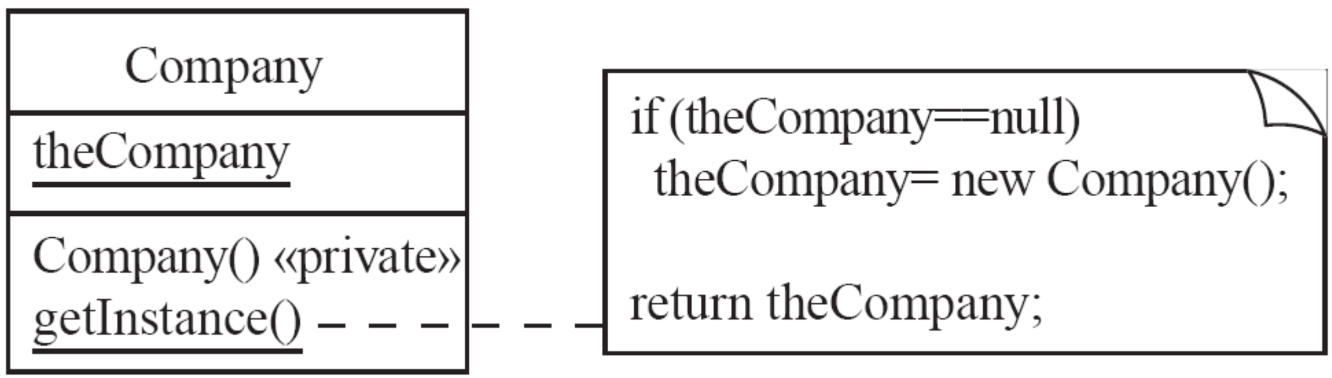
6.5 The Singleton Pattern

- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it

Singleton: Solution



Singleton: Example



Singleton: Lazy vs. Eager Initialization

Lazy initialization:

```
public static Company getInstance() {
  if (theCompany == null)
    theCompany = new Company();
  return theCompany;
}
```

Eager initialization:

```
public class Company {
  private static Company theCompany = new Company();

  public static Company getInstance() {
     return theCompany;
  }
}
```

Singleton Issues

Lazy vs. eager initialization. Which one is better?

- Laziness, of course!
  - Creation work (possibly holding on to expensive resources) avoided if the instance never needed

Singleton Issues

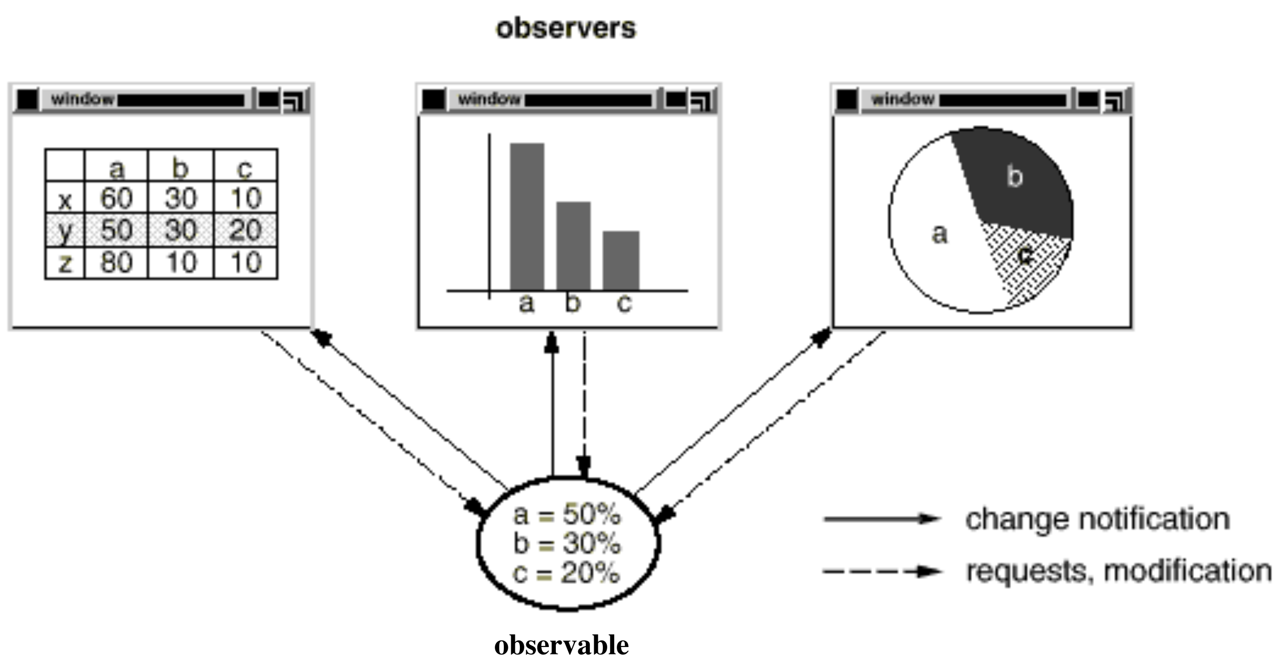
Why not make all the service methods static methods of the class itself?

- To permit subclassing: Static methods are not polymorphic, don’t permit overriding.
- Object-oriented remote communication mechanisms (e.g. Java RMI) only work with instance methods
  - Static methods are not remote-enabled.
- More flexibility: Maybe we’ll change our minds and won’t want a singleton any more.

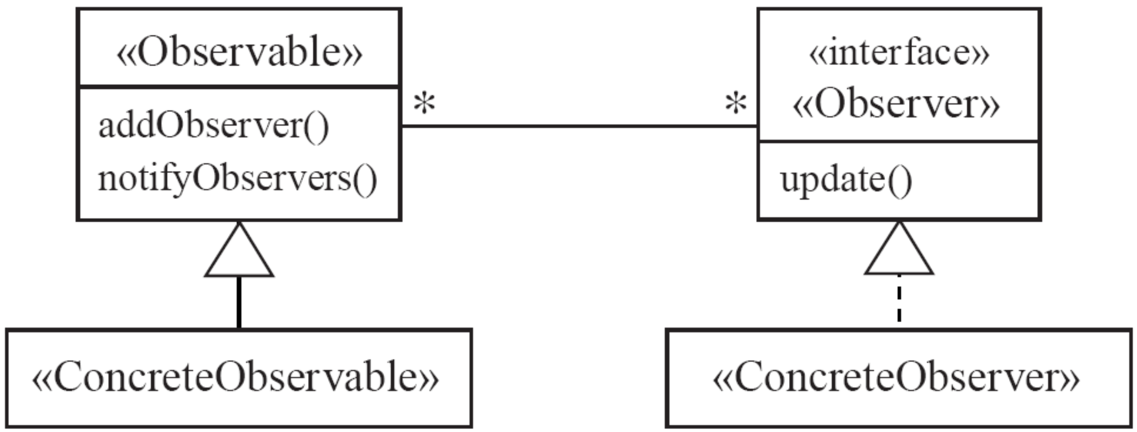
6.6 The Observer Pattern

- ***Context***:
  - When a two-way association is created between two classes, the code for the classes becomes inseparable.
  - If you want to reuse one class, then you also have to reuse the other.
- ***Problem***:
  - How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
  - How do you ensure that an object can communicate with other objects without knowing which class they belong to?
- ***Forces***:
  - You want to maximize the flexibility of the system to the greatest extent possible

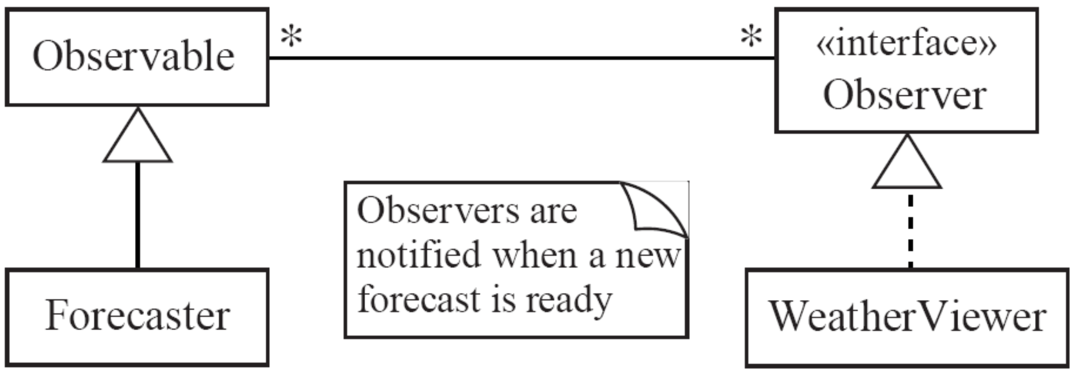
The Observer Pattern



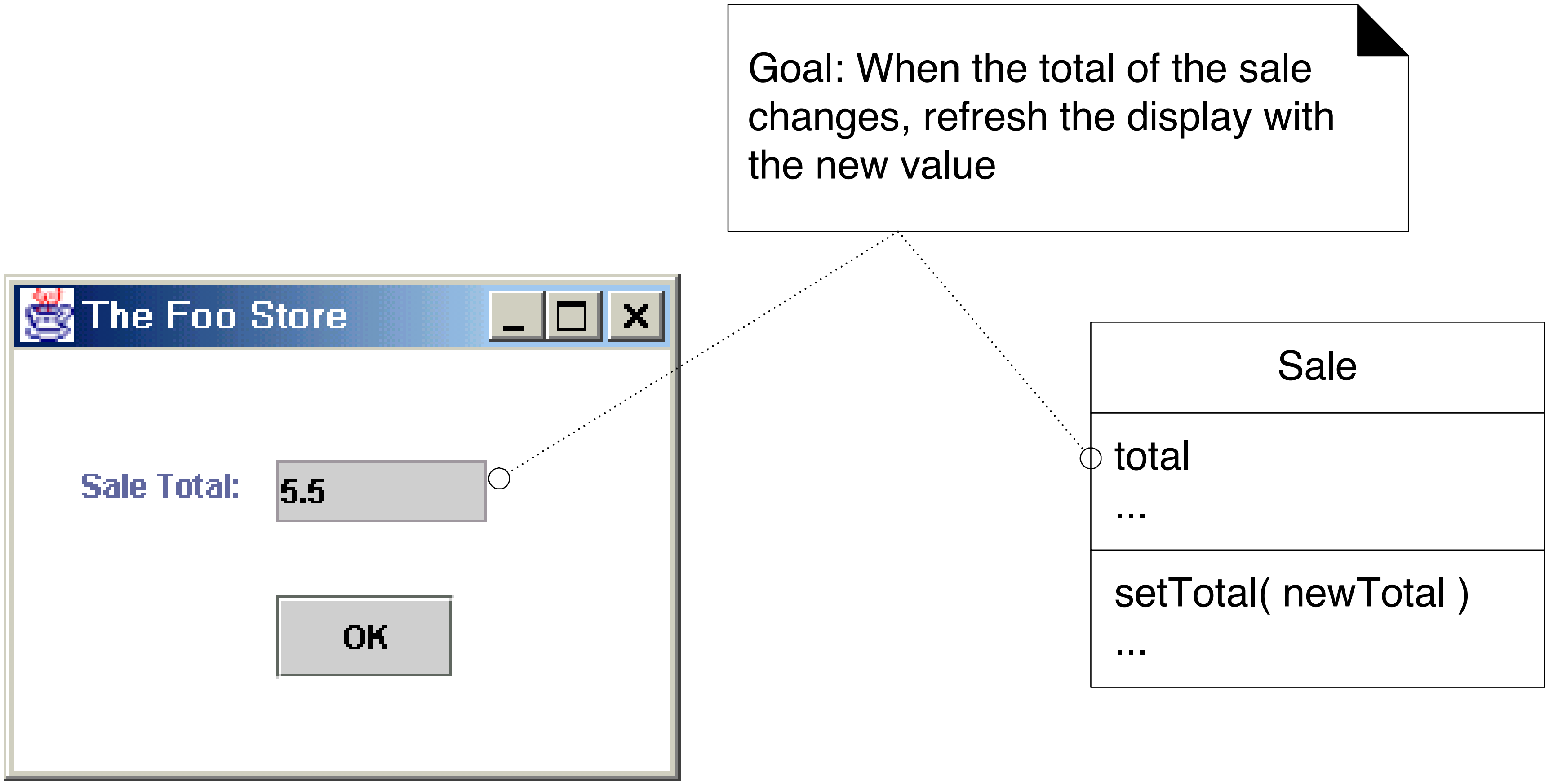
Observer: Solution



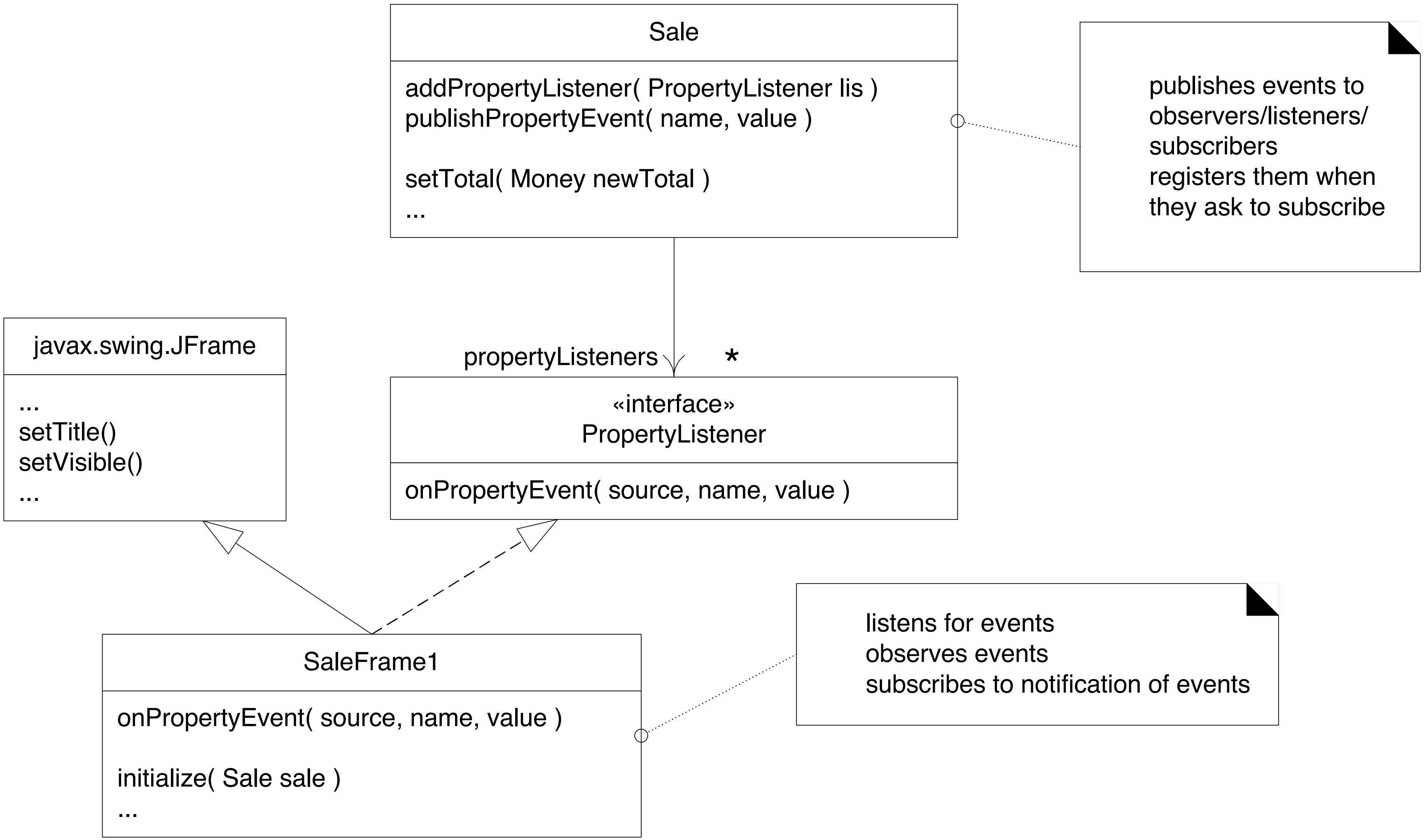
Observer: Example

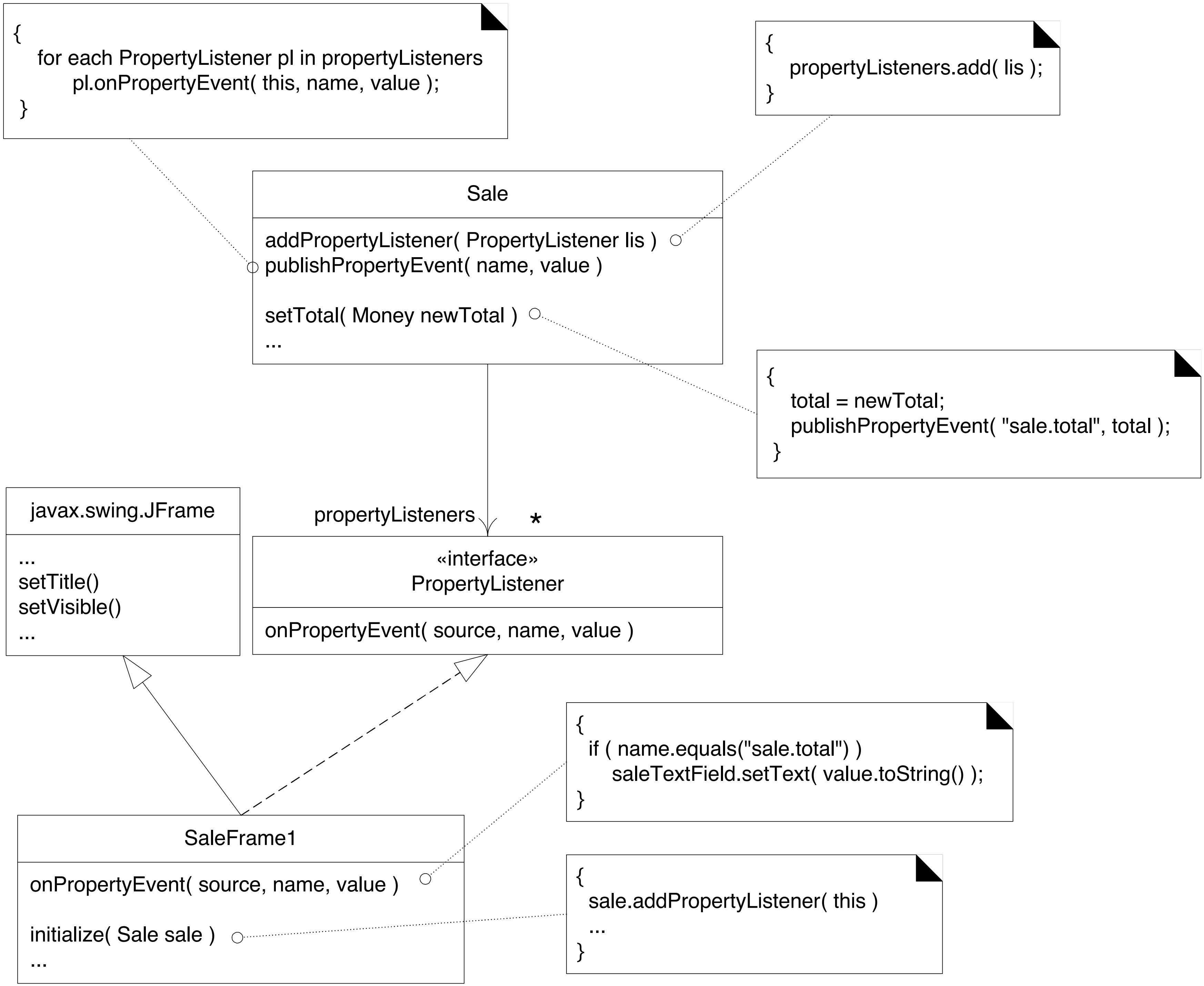


Observer: Example

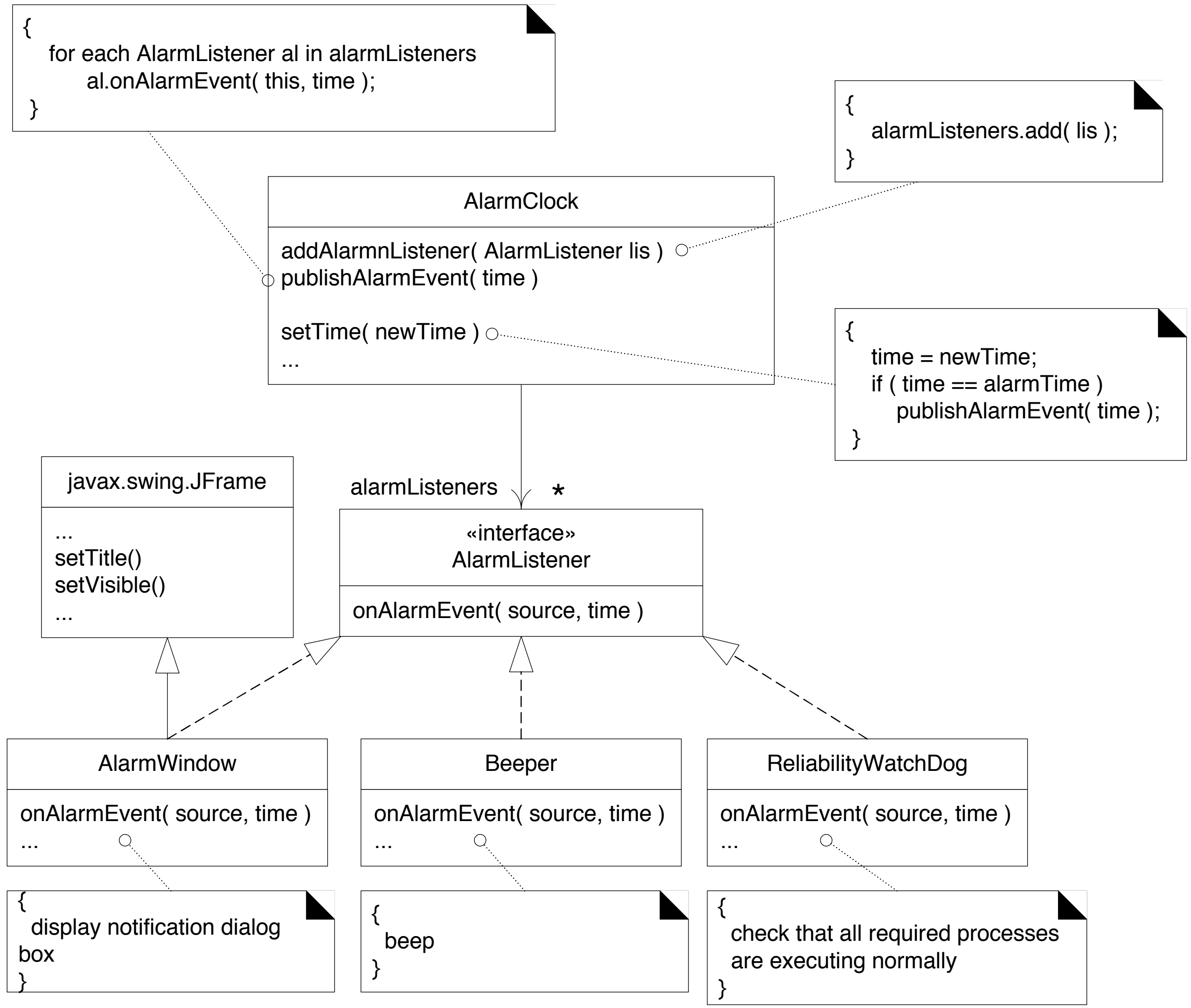


Observer: Example




{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

Observer: Example



Observer: Antipatterns

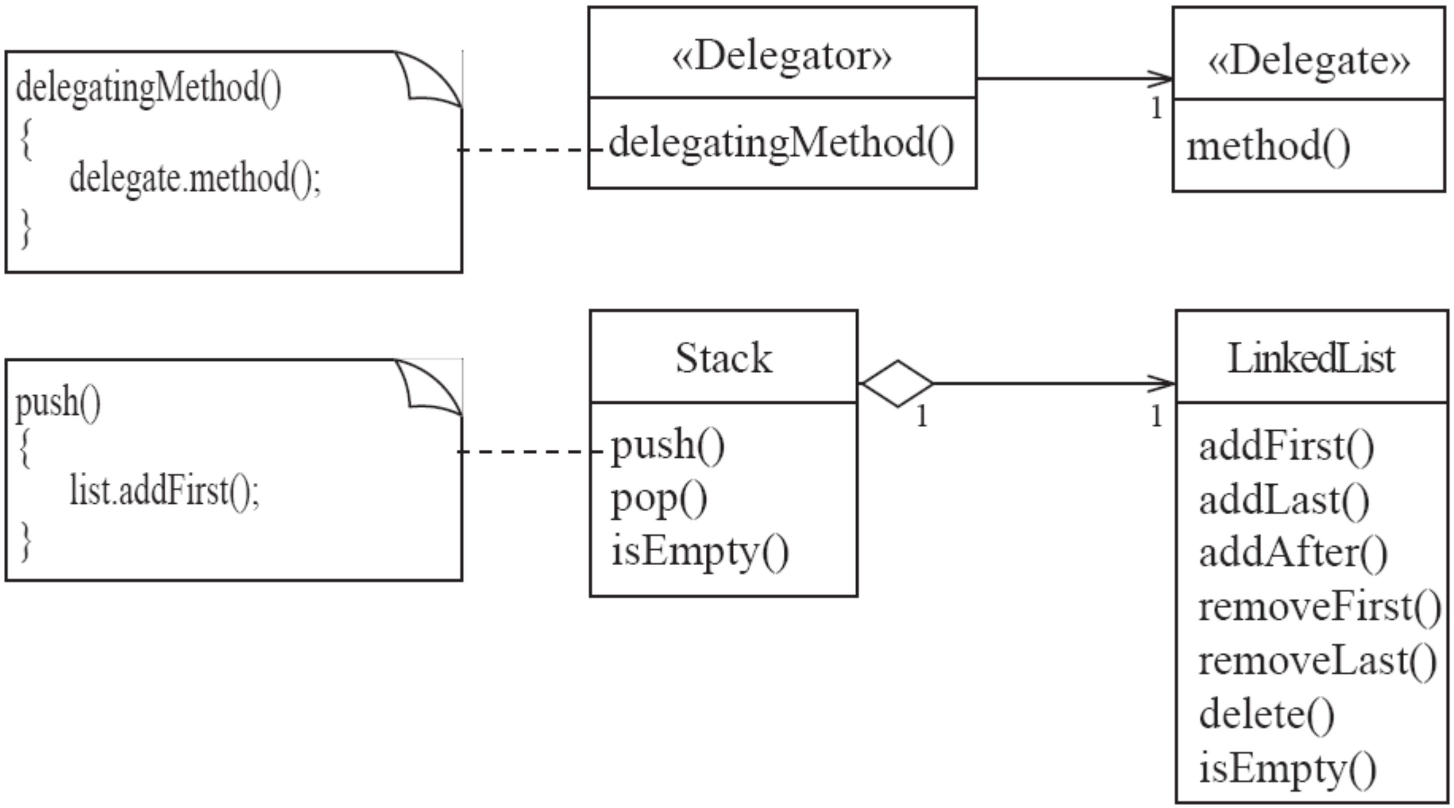
- Connect an observable directly to an observer so that they both have references to each other.
  - This means that you cannot plug in a different observer

- Make the observers *subclasses* of the observable.
  - This will not work because then each observer is at the same time an observable
  - It is not therefore possible to have more than one observer for an observable

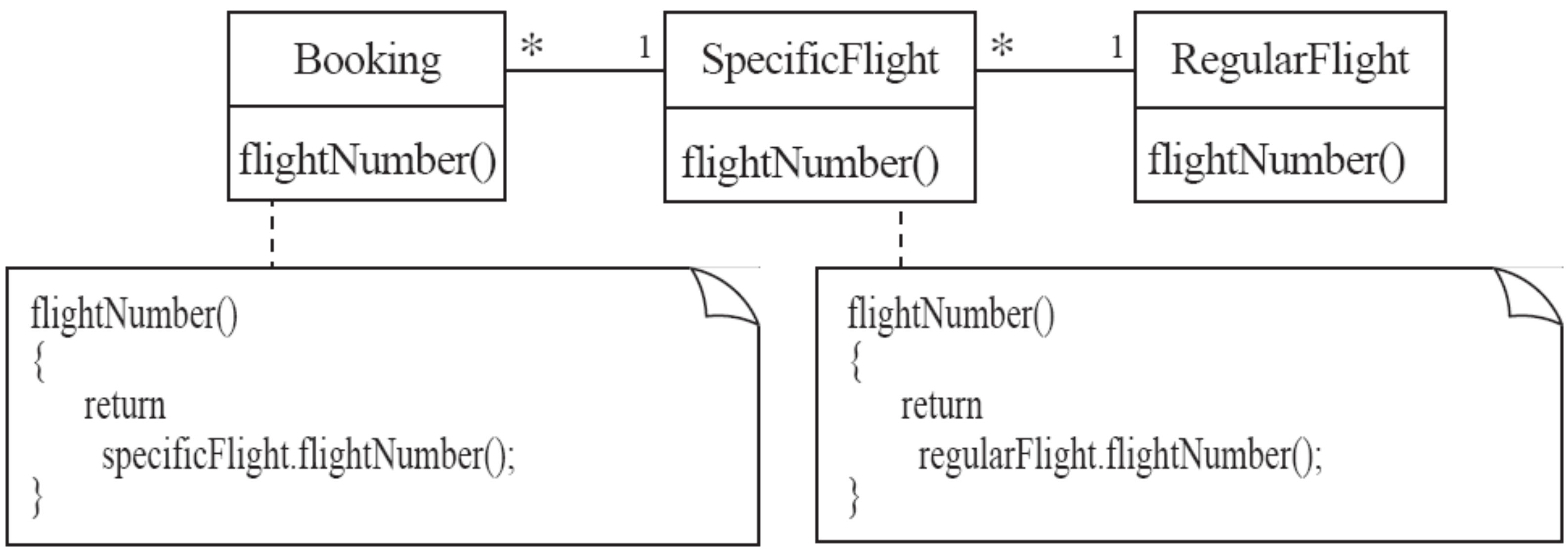
6.7 The Delegation Pattern

- ***Context***:
  - You are designing a method in a class
  - You realize that another class has a method which provides the required service
  - Inheritance is not appropriate
    - E.g. because the isa rule does not apply
- ***Problem***:
  - How can you most effectively make use of a method that already exists in the other class?
- ***Forces***:
  - You want to minimize development cost by reusing methods

Delegation: Solution



Delegation: Example of two levels of delegation



Delegation

Antipatterns

Overuse generalization and *inherit* the method that is to be reused

- For example, making Stack a subclass of LinkedList
- Some of the methods of LinkedList, such as addAfter, do not make sense in a Stack, yet they would be available

Delegation

Antipatterns

Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate», you might consider having many different methods in the «Delegator» call the delegate’s method

— This would create many more linkages in the system

Delegation

Antipatterns

- Accessing non-neighboring classes in delagation
  - Bad, because the further a method has to reach to get its data, the more sensitive it becomes to changes in the system
  - For example, it would not be good for Booking’s flightNumber method to be written as:

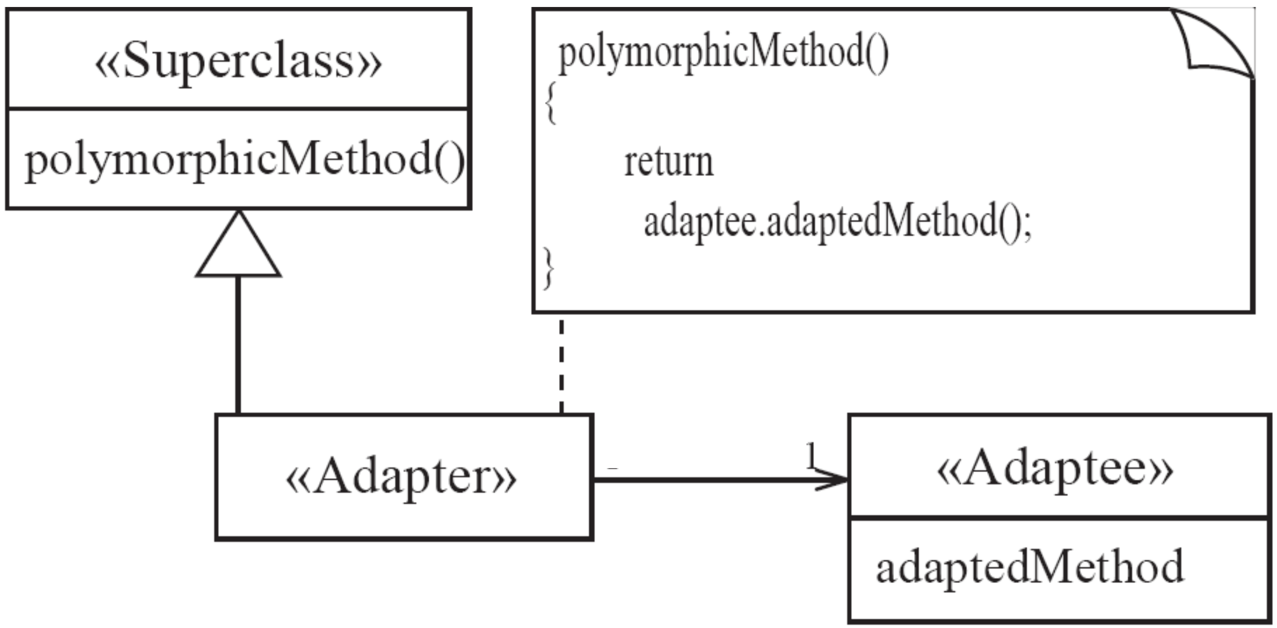
```
return specificFlight.regularFlight.flightNumber();
```

6.8 The Adapter Pattern

- ***Context***:
  - You are building an inheritance hierarchy and want to incorporate it into an existing class.
  - The reused class is also often already part of its own inheritance hierarchy.
- ***Problem***:
  - How to obtain the power of polymorphism when reusing a class whose methods
    - have the same function
    - but *not* the same signature

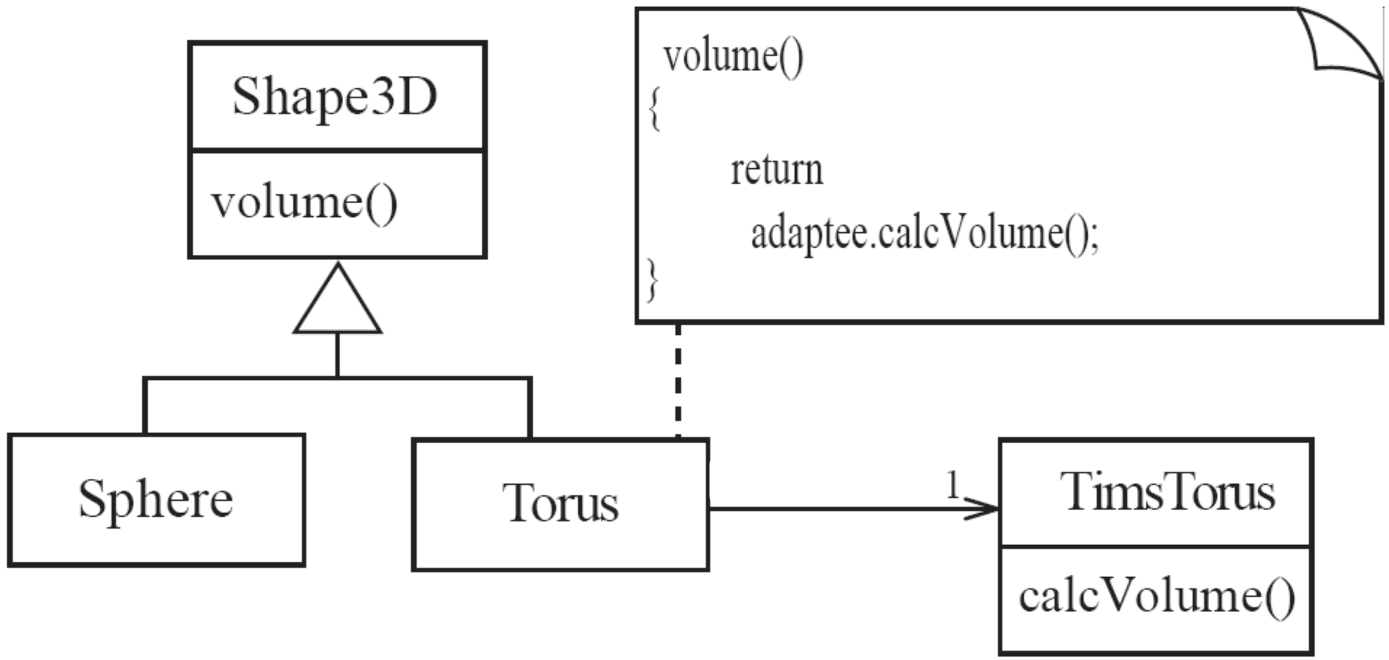
    as the other methods in the hierarchy?
- ***Forces***:
  - You do not have access to multiple inheritance or you do not want to use it.

Adapter: Solution

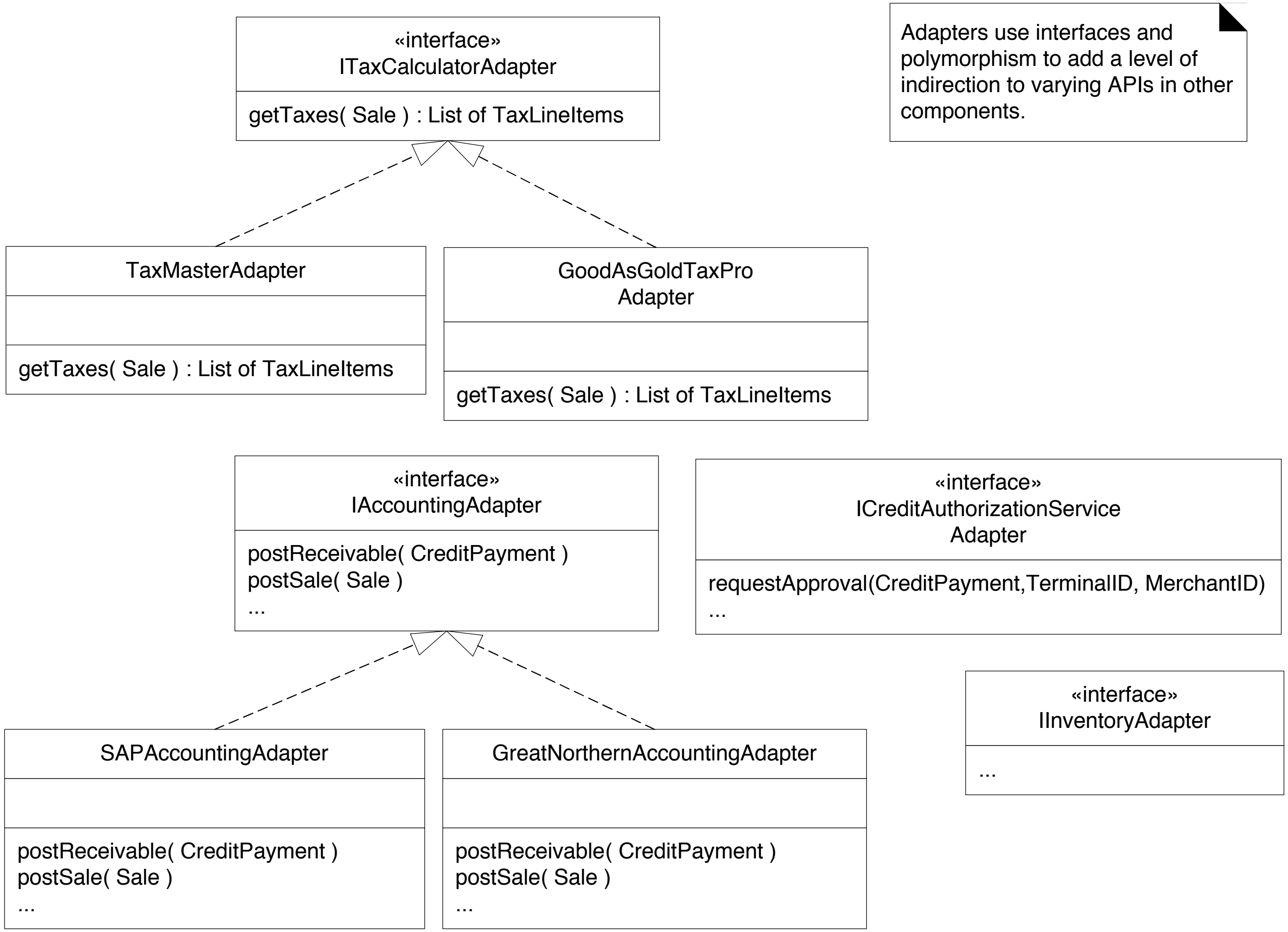


Adapter

Example:



A Variation of Adapter Pattern: Example



Assume that we are developing a system that should support several kinds of external third-party components

6.9 The Façade Pattern

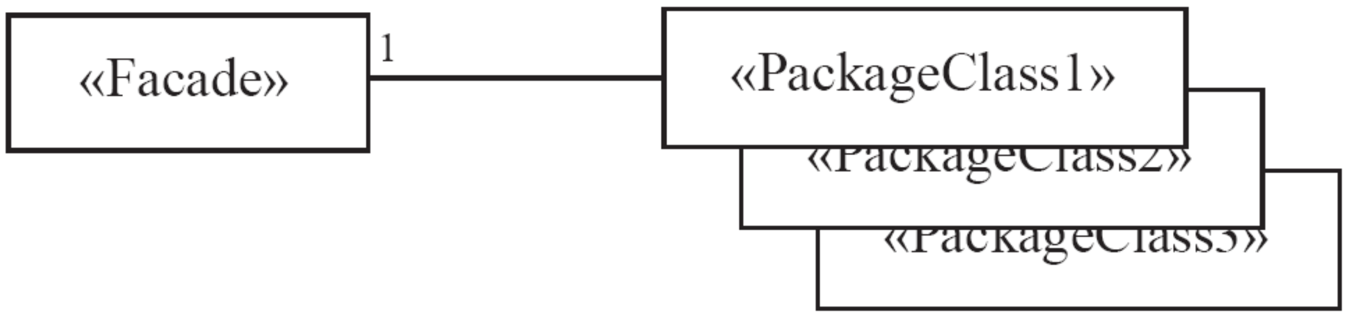
- ***Context***:
  - —Often, an application contains several complex packages.
  - —A programmer working with such packages has to manipulate many different classes
- ***Problem***:
  - —How do you simplify the view that programmers have of a complex package?
- ***Forces***:
  - —It is hard for a programmer to understand and use an entire subsystem
  - —If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

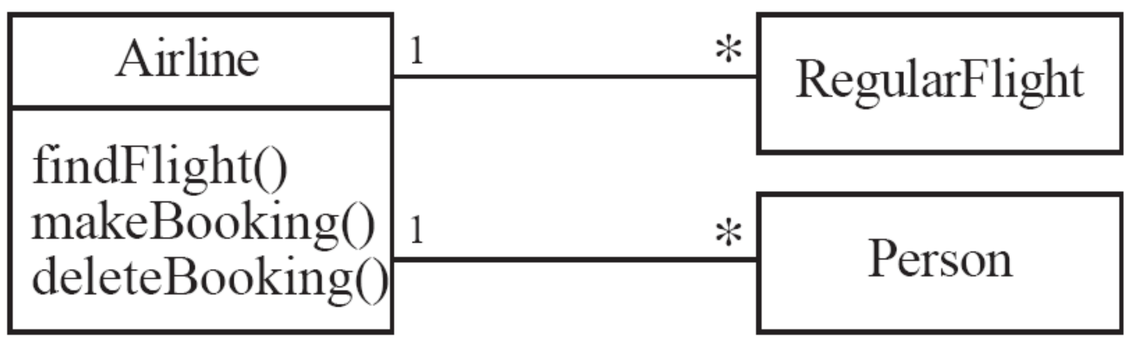
Example Scenario

Suppose you are developing a software system for a travel agency and recently bought a third-party airline ticket reservation system.

- **An airline ticket reservation system may have complex subsystems**
  - A finance subsystem to handle credit card operations
  - An itinerary subsystem to compute an itinerary
  - A booking subsystem to make reservations
  - A printing subsystem to print out tickets/reservations
- **You know that you want to perform a fixed set of operations, e.g., search itinerary, make reservation, delete reservation, etc.**
- **You don’t want to deal with the complexity of the airline ticket reservation system every time you need to perform these operations in the code**
  - For example, making a reservation may require
    - Using the itinerary subsystem to compute an itinerary
    - Using the finance subsystem to validate credit card information
    - Using the booking subsystem to complete the reservation
    - Using the printing subsystem to print out the reservation ifno

Façade: Solution





Façade Pattern: Example (complex parts)

```
// complex parts
public class CPU {
   public void freeze() { ... }
   public void jump(long position) { ... }
   public void execute() { ... } }
public class Memory { public void load(long position, byte[] data) { ... } }
public class HardDrive { public byte[] read(long lba, int size) { ... } }

//Façade
public class Computer {
   public void startComputer() {
    cpu.freeze();
    memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR,
SECTOR_SIZE));
    cpu.jump(BOOT_ADDRESS);
    cpu.execute(); } }
```

6.10 The Immutable Pattern

- ***Context***:
  - An immutable object is an object that has a state that never changes after creation
- ***Problem***:
  - How do you create a class whose instances are immutable?
- ***Forces***:
  - There must be no loopholes that would allow ‘illegal’ modification of an immutable object
- ***Solution***:
  - Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
  - Instance methods which access properties must not have side effects.
  - If a method that would otherwise modify an instance variable is required, then it has to return a *new* instance of the class.

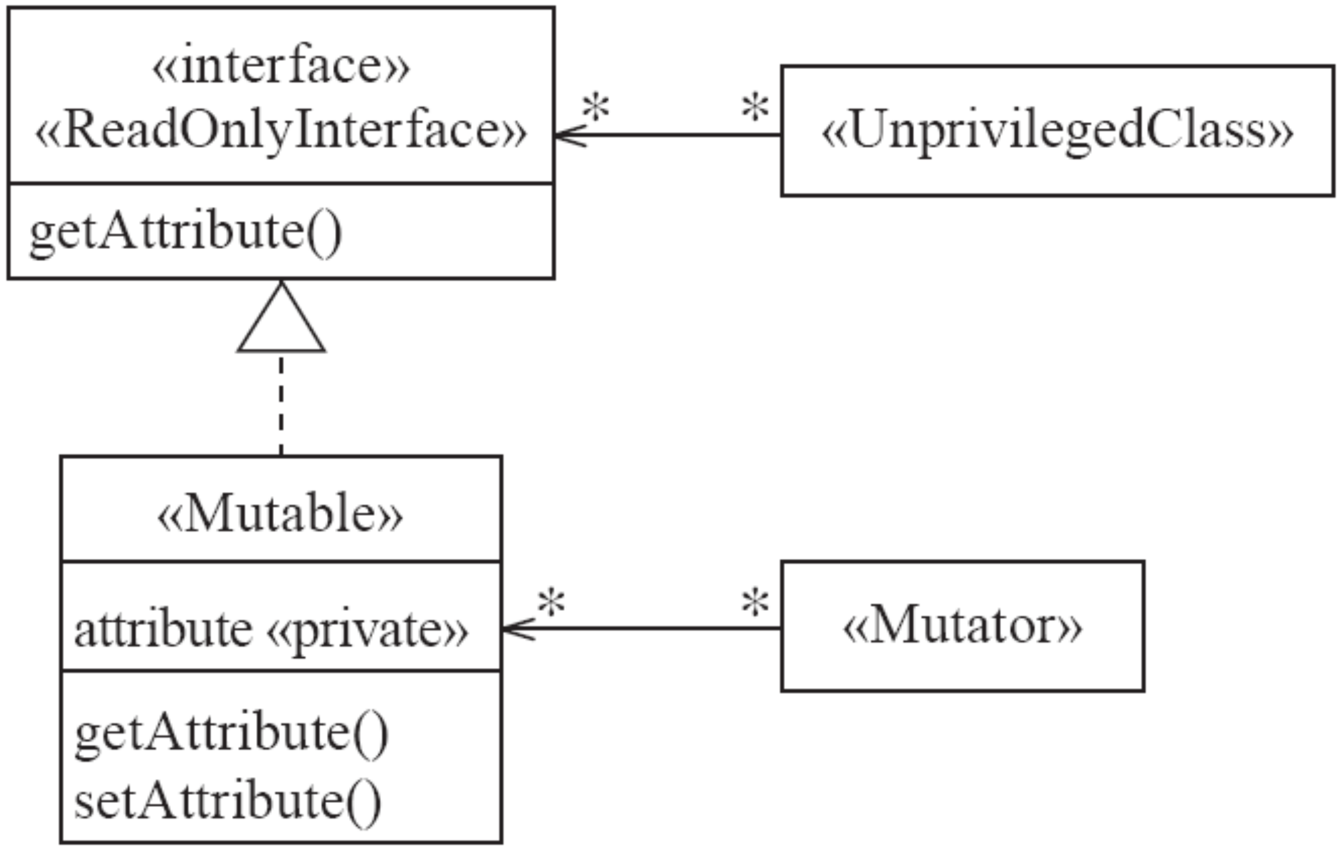
Immutable: Example

```
public class Point{
  private int x;
  private int y;

  public Point(int x, int y){
      this.x = x;
      this.y = y;
   }

   public int getX(){ return x; }
   public int getY(){ return y; }

   public Point translate(int xAmount, int yAmount){
       return new Point(x + xAmount, y + yAmount);
   }
}
```

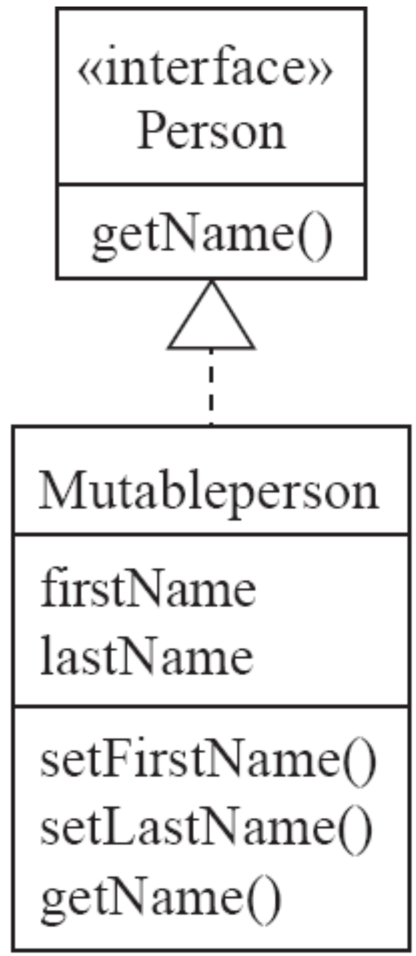
6.11 The Read-only Interface Pattern

- ***Context***:
  - You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable
- ***Problem***:
  - How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?
- ***Forces***:
  - Restricting access by using the **`public`**, **`protected`** and **`private`** keywords is not adequately selective.
  - Making access **`public`** makes it public for both reading and writing

Read-only Interface: Solution



Read-only Interface: Example



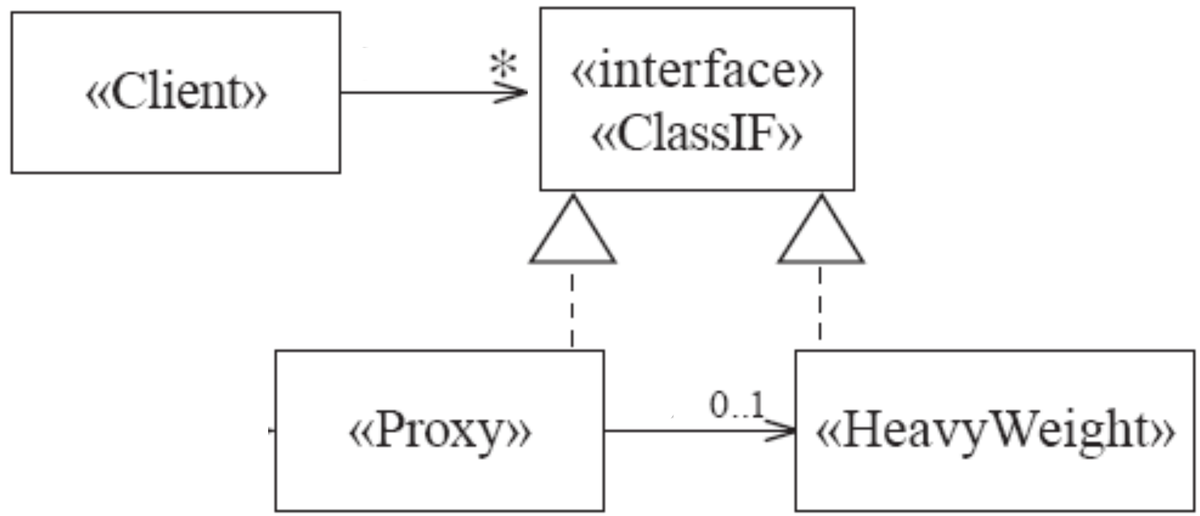
Read-only Interface: Antipatterns

- Make the read-only class a *subclass* of the «Mutable» class
- Override all methods that modify properties
  - such that they throw an exception

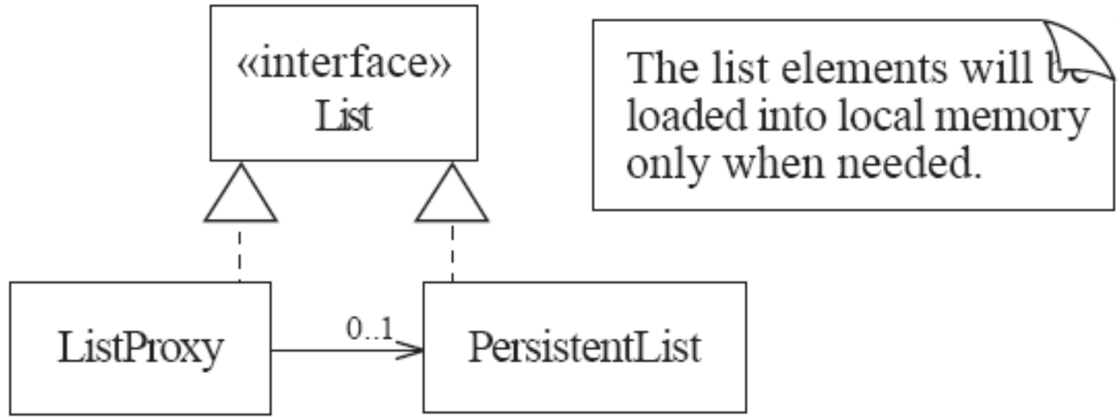
6.12 The Proxy Pattern

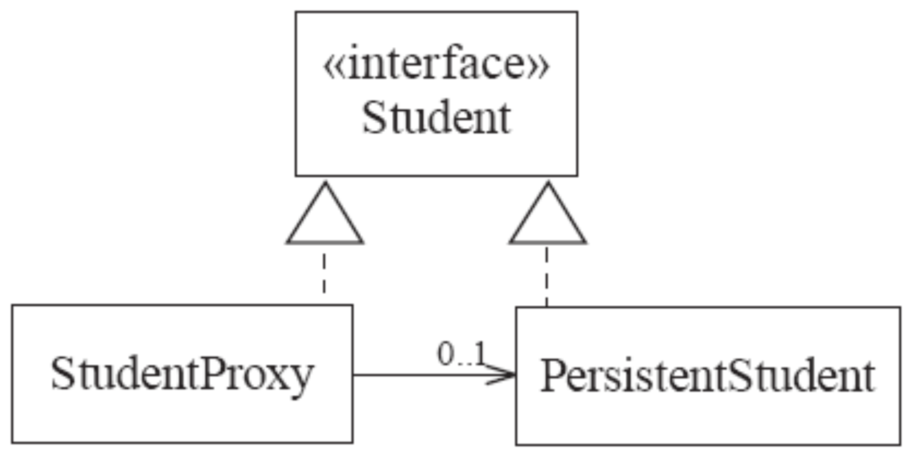
- ***Context***:
  - —Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).
  - —There is a time delay and a complex mechanism involved in creating the object in memory
- ***Problem***:
  - —How to reduce the need to create instances of a heavyweight class?
  - — A related problem is if you load one object from a database or server, how can you avoid loading all other objects that are linked to it?
- ***Forces***:
  - —We want to be able to program the application **as if** all the heavyweight objects were located in the memory
  - —The details of how heavyweight objects are actually stored and loaded should be transparent

Proxy: Solution



Proxy: Example





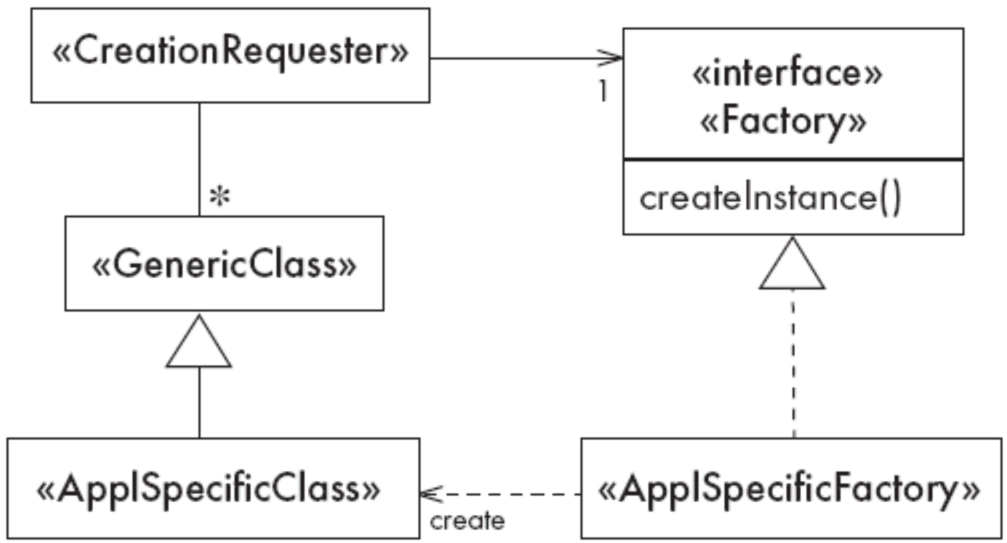
Proxy Example

```
public interface IPicture{
   void draw();
}

public class  Picture implements IPicture {
   public Picture (String fileName){  loadPicture(fileName); }
   public draw(){ //actual code goes here }
}

public class PictureProxy  implements IPicture{
     private String fileName;
     private Picture realPicture = null;

     public PictureProxy(String fileName){
        this.fileName = fileName;
     }

    public draw(){
       if (realPicture == null) realPicture = new Picture(fileName);
       realPicture.draw();
    } }
```

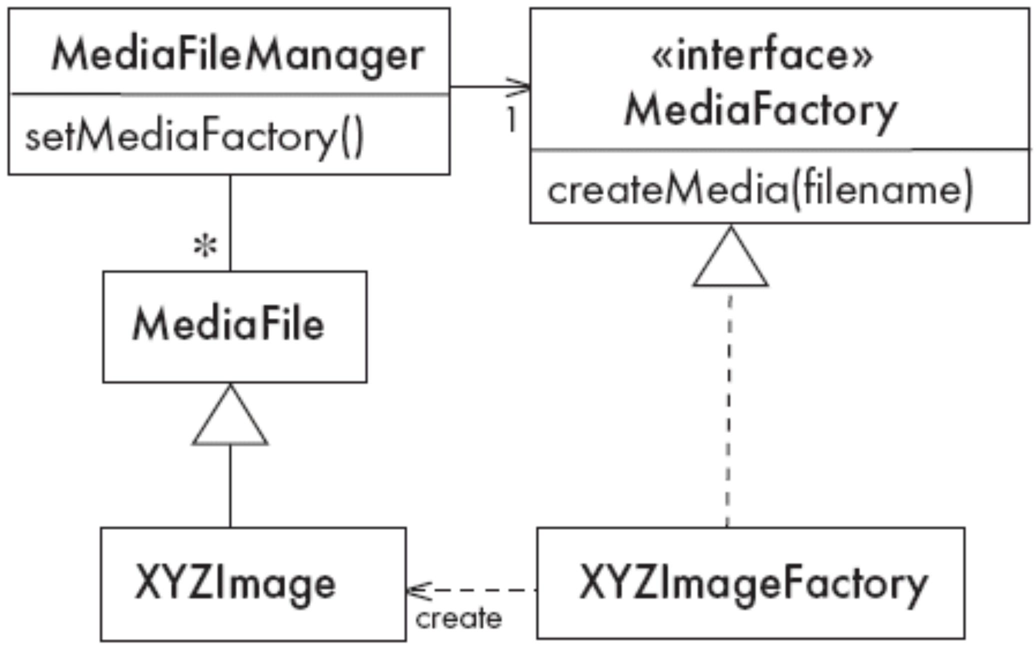
6.13 The Factory Pattern

- ***Context***:
  - —You have a reusable framework that needs to create objects as part of its work. However, the class of the created objects will depend on the application.
- ***Problem***:
  - —How do you enable a programmer to add new application-specific class into a system built on such a framework?
- ***Forces***:
  - —You want to have the framework create and work with application-specific classes that the framework does not yet know about.
- ***Solution***:
  - —The framework delegates the creation of application-specific classes to a specialized class, the Factory.
  - —The Factory implements a generic interface defined in the framework.
  - —The factory interface declares a method whose purpose is to create some subclass of a generic class.

The Factory Pattern: Solution



The Factory Pattern: Example



Factory pattern: Example (framework)

```
 public abstract class MediaFile{
   protected String fileName;
   public abstract void play();
   public abstract void display();
   public abstract void load();
}

public interface MediaFactory{
   MediaFile createMediaFile(String fileName);
}

public class MediaFileManager{
   private MediaFactory factory;
   private Vector<MediaFile> files;

   public void setMediaFactory(MediaFactory f){ factory = f; }

   public void addMediaFile(String fileName){
      files.add(factory.createMediaFile(fileName));
   }
}
```

Factory pattern: Example (customization of the framework)

```
public class XYZImage extends MediaFile{
   public XYZImage(String fn){
        fileName = fn;
   }

   public void play(){//actual code goes in here}
   public void display(){//actual code goes in here}
   public void load(){//actual code goes in here}
}

public class XYZImageFactory implements MediaFactory {

   public XYZImageFactory(MediaFileManager m){
      m.setMediaFactory(this); }

   public MediaFile createMediaFile(String fileName){
      return (MediaFile) new XYZImage(fileName);}
}
```

A Variation of Factory Pattern: Example

ServicesFactory
accountingAdapter : IAccountingAdapter inventoryAdapter : IInventoryAdapter taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter getInventoryAdapter() : IInventoryAdapter getTaxCalculatorAdapter() : ITaxCalculatorAdapter ...

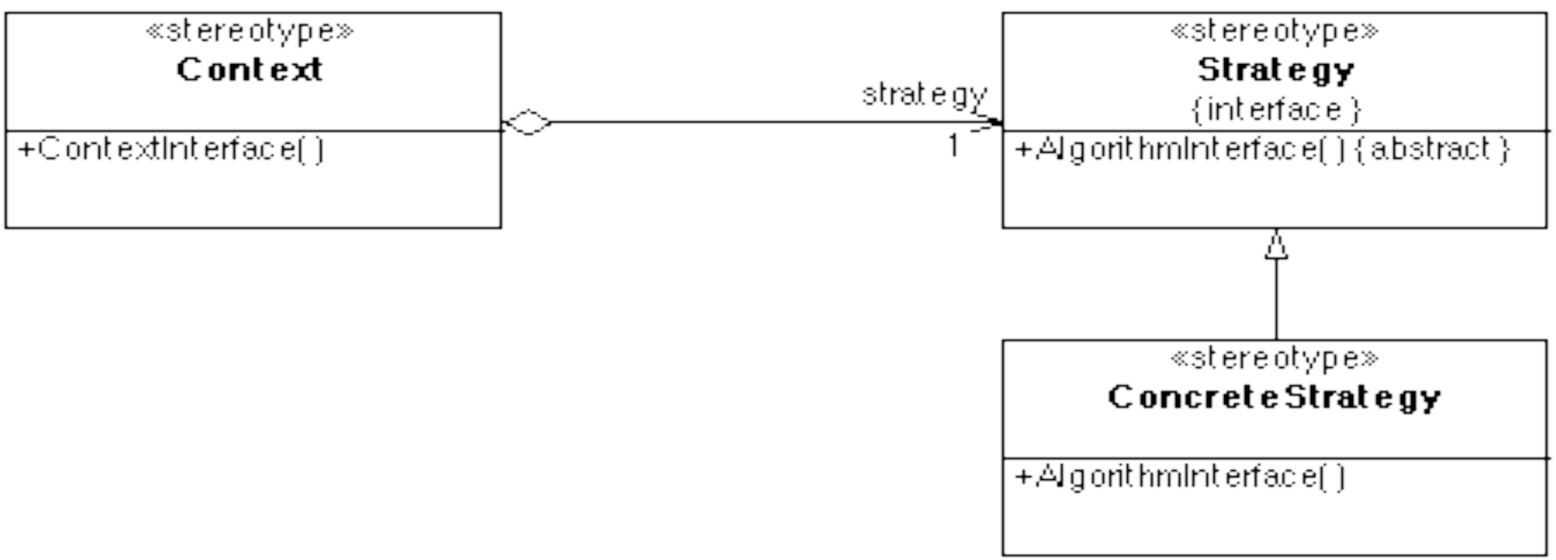
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class: read it from an
  // external property

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

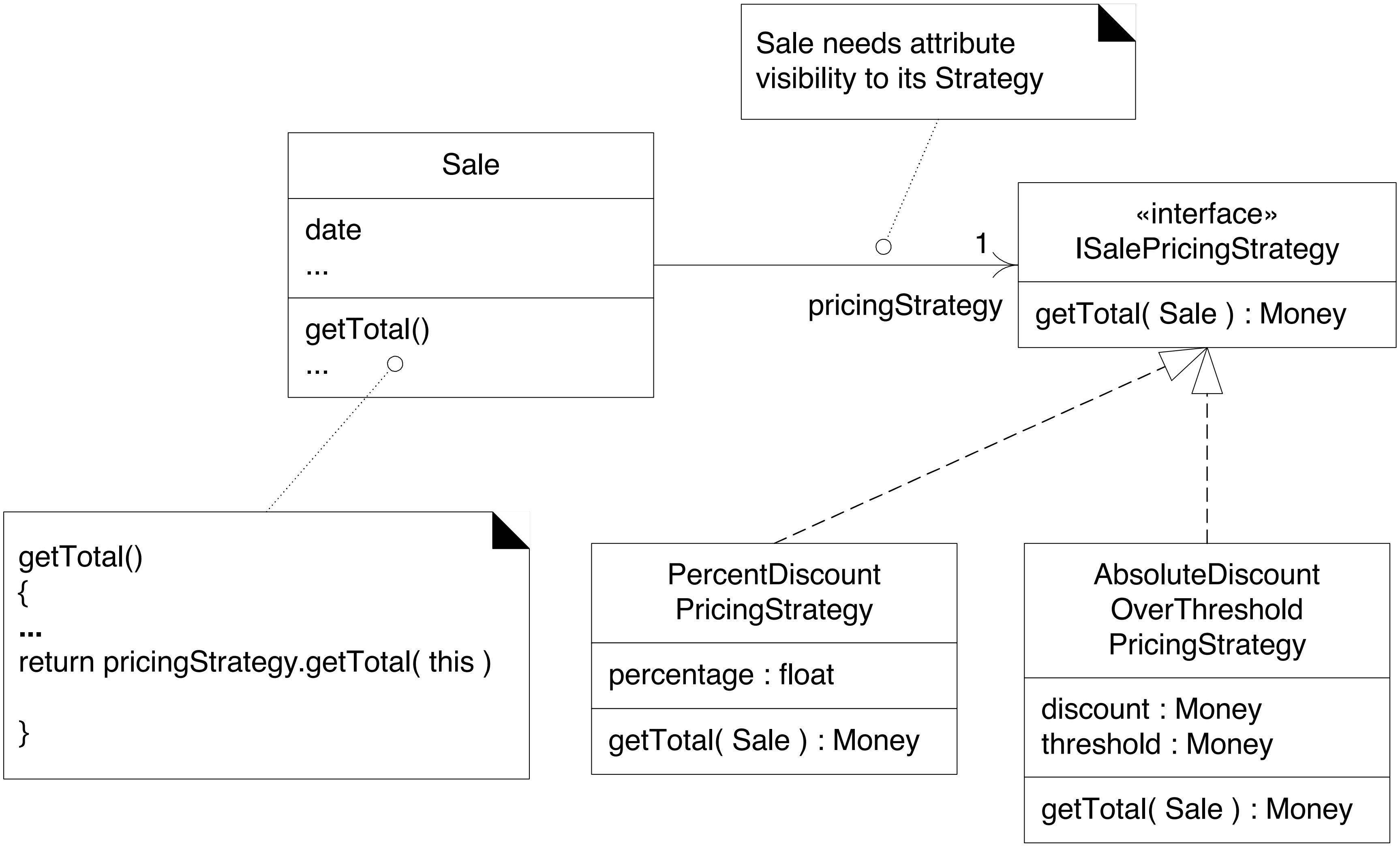
The Strategy Pattern

- ***Problem***:
  - —How to design for varying, but related, algorithms or policies?
  - —How to design for the ability to change these algorithms and policies?
- ***Solution***:

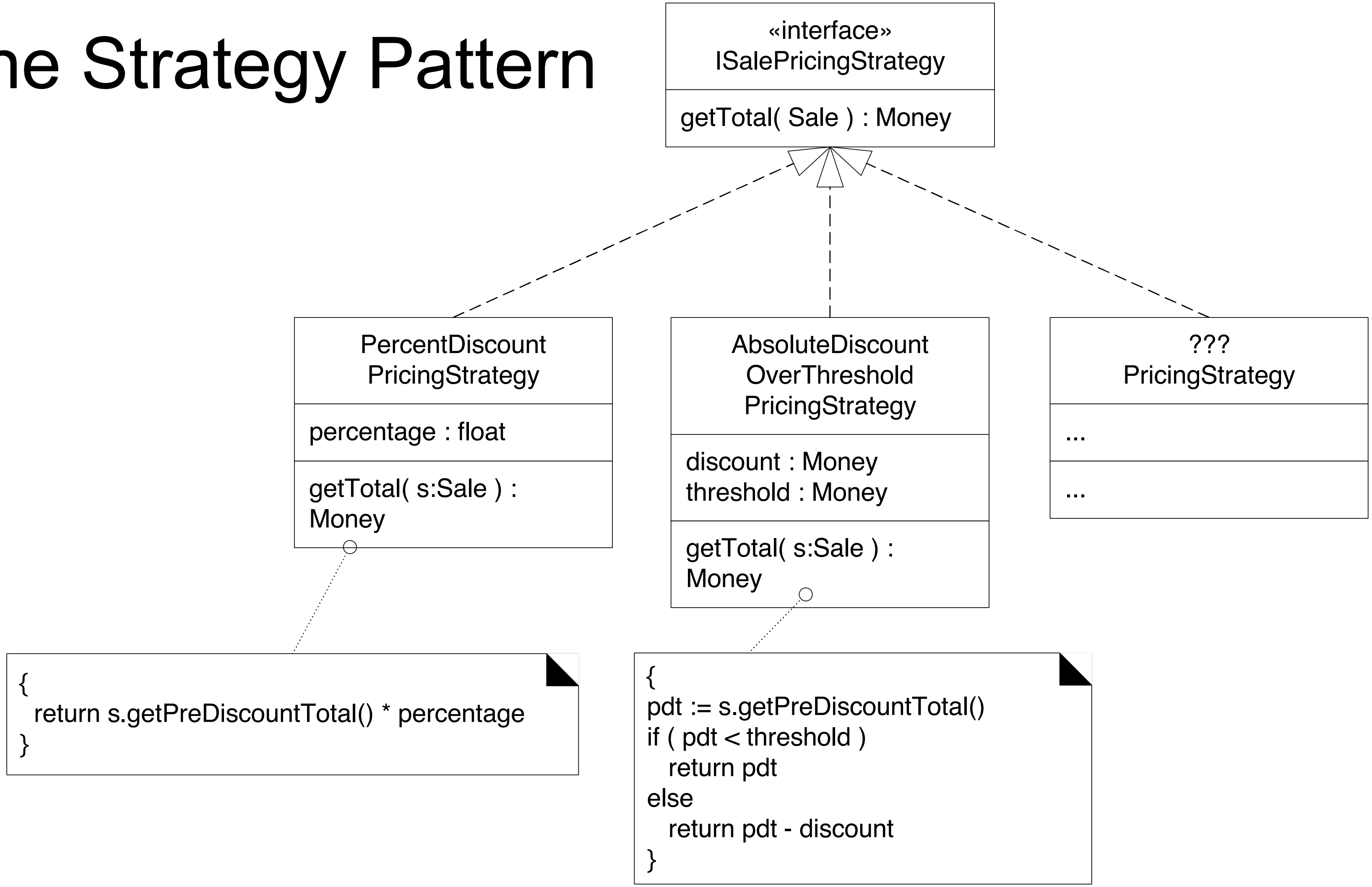
  Define each algorithm/policy/strategy in a separate class, with a common interface.



The Strategy Pattern

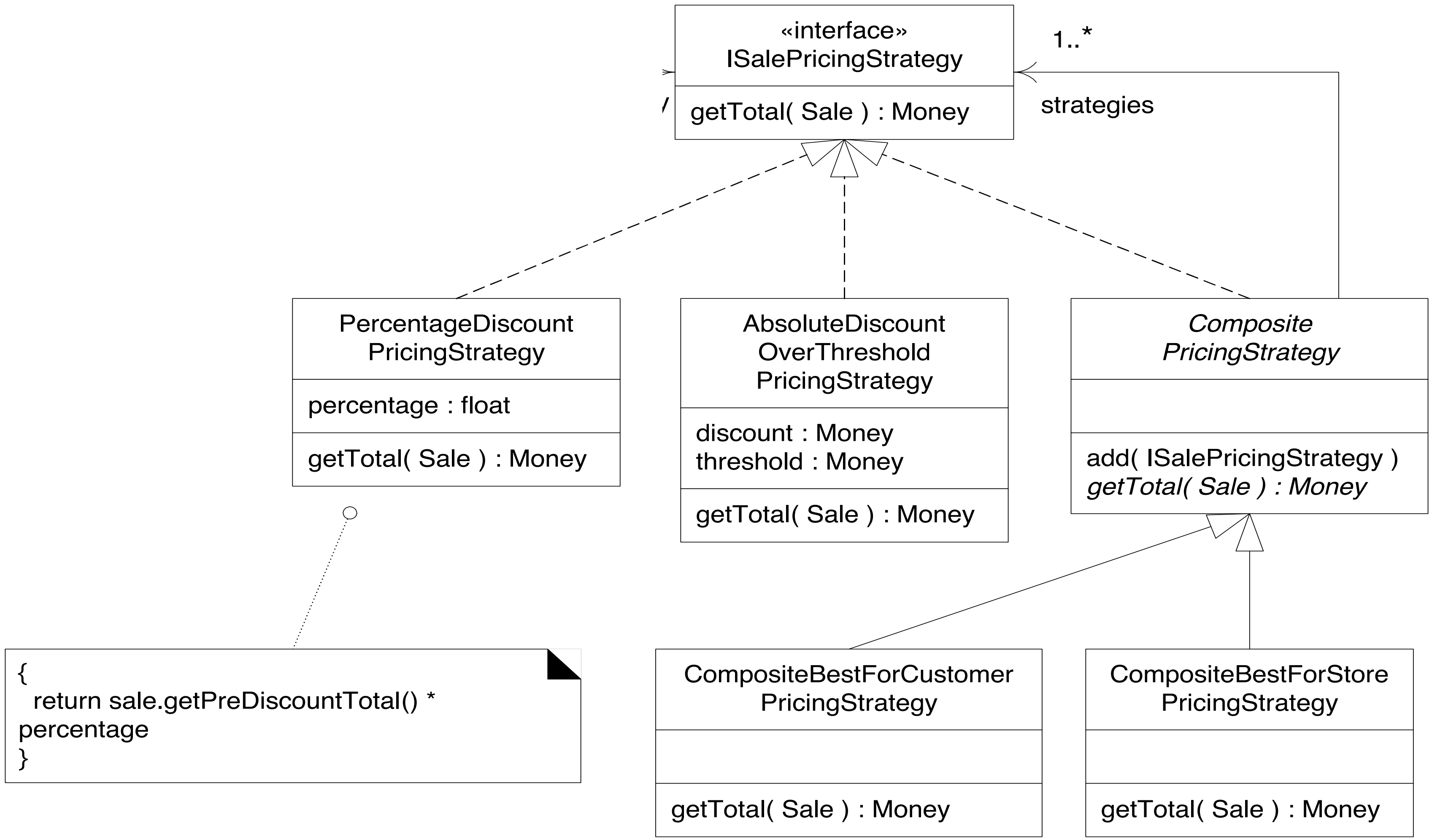


The Strategy Pattern



- Issue: Provide more complex pricing logic.
  - Pricing strategy varying over time
  - Example: Different kinds of sales.

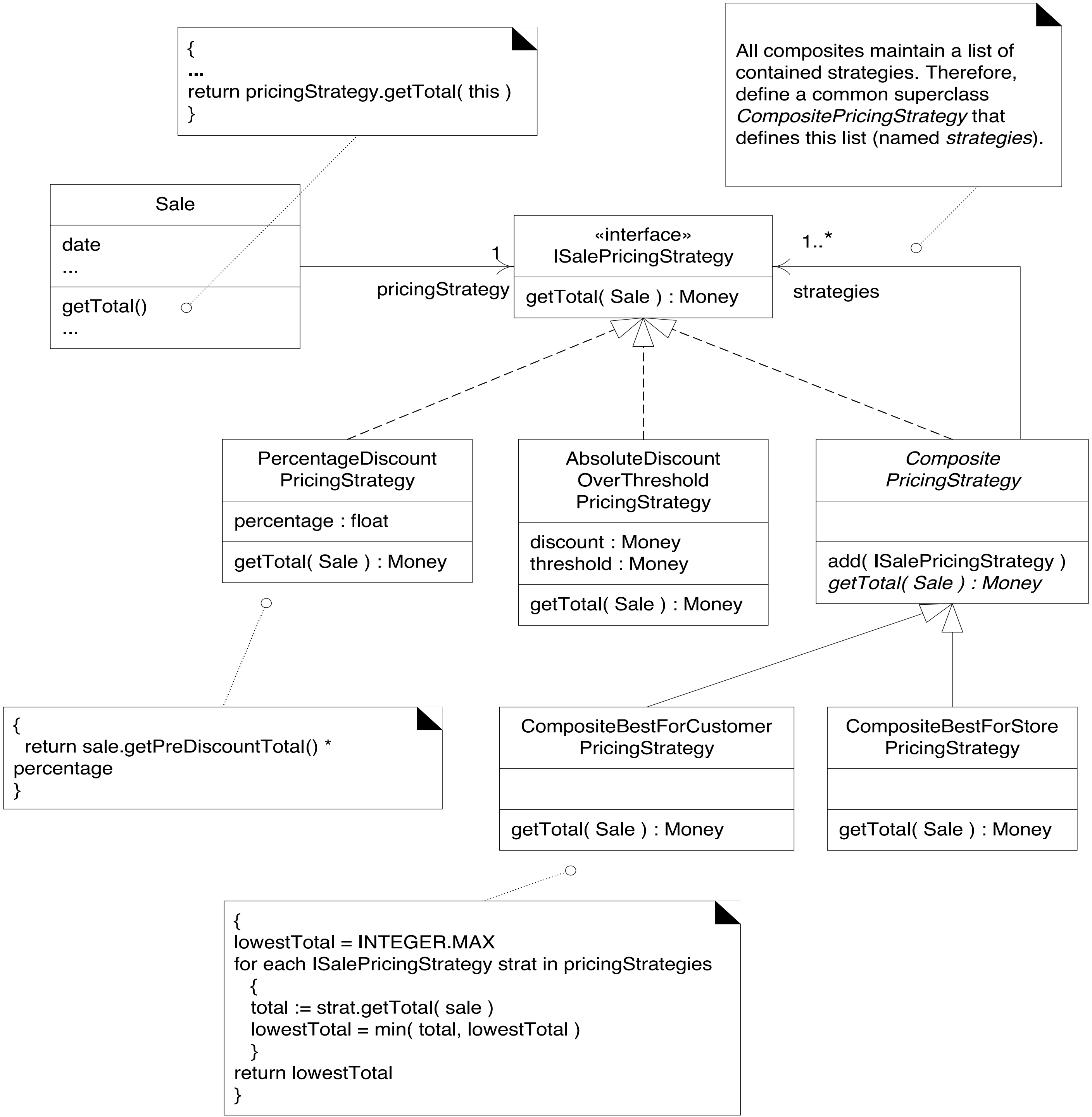
Composite Pricing Strategies



Composite Pricing Strategies

CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money

```
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
   {
   total := strat.getTotal( sale )
   lowestTotal = min( total, lowestTotal )
   }
return lowestTotal
}
```


{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).
Sale
date
...
getTotal()
...
pricingStrategy
1
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
strategies
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() * percentage
}
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
}

```java
public abstract class CompositePricingStrategy
  implements ISalePricingStrategy {

   protected List strategies = new ArrayList();

    public add (ISalePricingStrategy s) {
      strategies.add(s);
    }

   public abstract Money getTotal( Sale sale );
}

public class ComputeBestForCustomerPricingStrategy
  extends CompositePricingStrategy {

  Money lowestTotal = new Money( Integer.MAX_VALUE );

  for (Iterator i = strategies.iterator(); i.hasNext(); ) {
     ISalePricingStrategy strategy = (ISalePricingStrategy) i.next();
     Money total = strategy.getTotal( sale );
     lowestTotal = total.min( lowestTotal);
  }
  return lowestTotal;
}
```

6.15 Difficulties and Risks When Creating Class Diagrams

- **Patterns are not a panacea:**
  - Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern.
  - This can lead to unwise design decisions .
- *Resolution:*
  - *Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.*
  - *Make sure you justify each design decision carefully.*

Difficulties and Risks When Creating Class Diagrams

- **Developing patterns is hard**
  - Writing a good pattern takes considerable work.
  - A poor pattern can be hard to apply correctly
- *Resolution:*
  - *Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.*
  - *Take an in-depth course on patterns.*
  - *Iteratively refine your patterns, and have them peer reviewed at each iteration.*