Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 6:

Using Design Patterns

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:

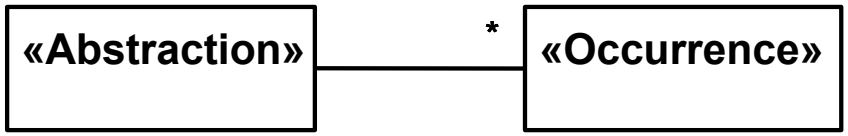
- Who developed or inspired the pattern.

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

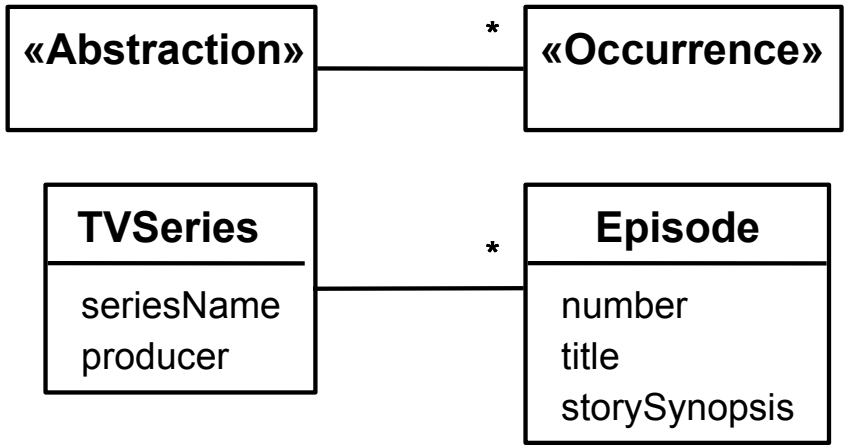
Abstraction-Occurrence

- ***Solution:***



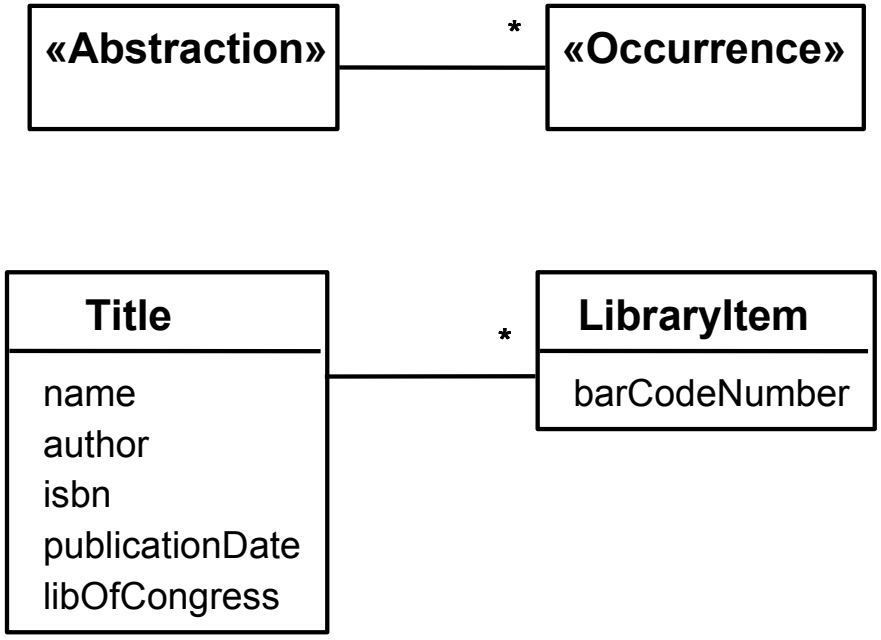
Abstraction-Occurrence

- ***Solution:***



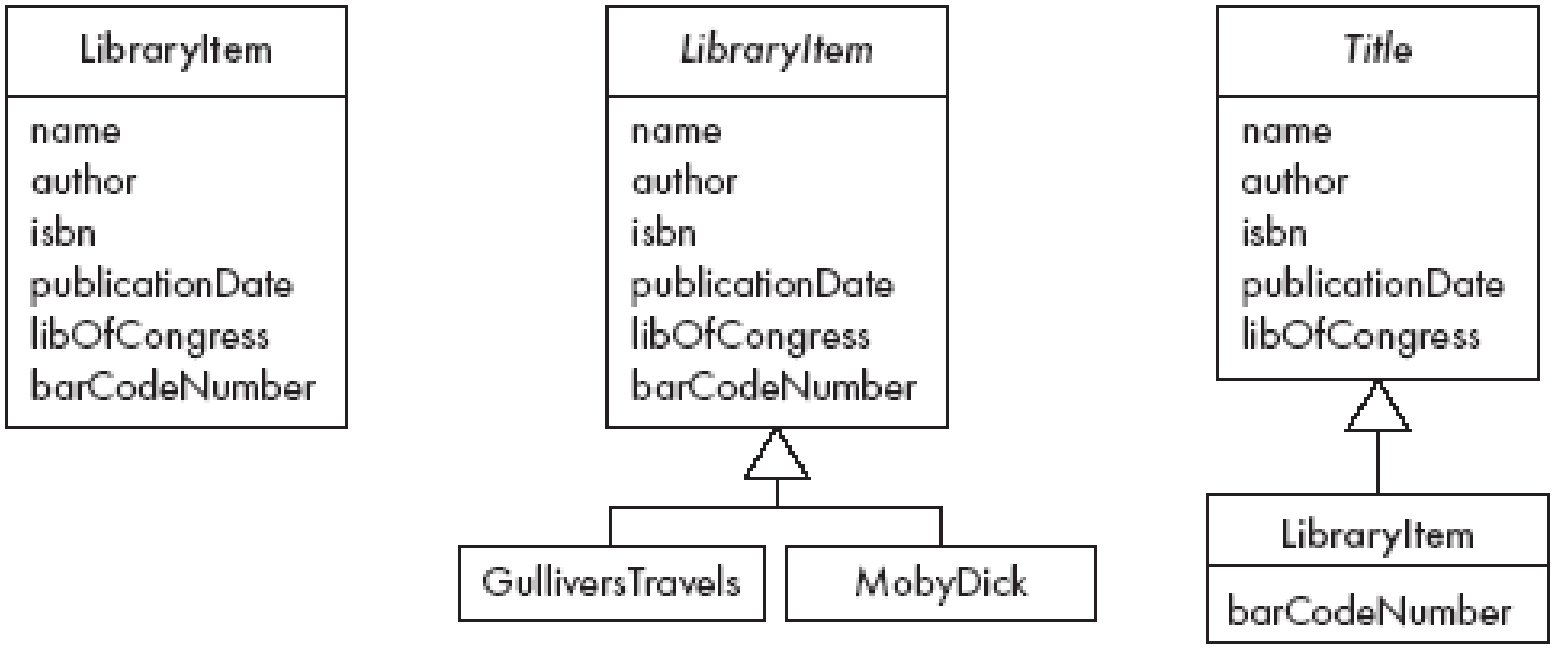
Abstraction-Occurrence

- ***Solution:***



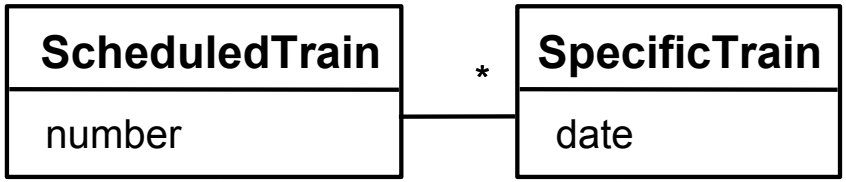
Abstraction-Occurrence

Antipatterns:



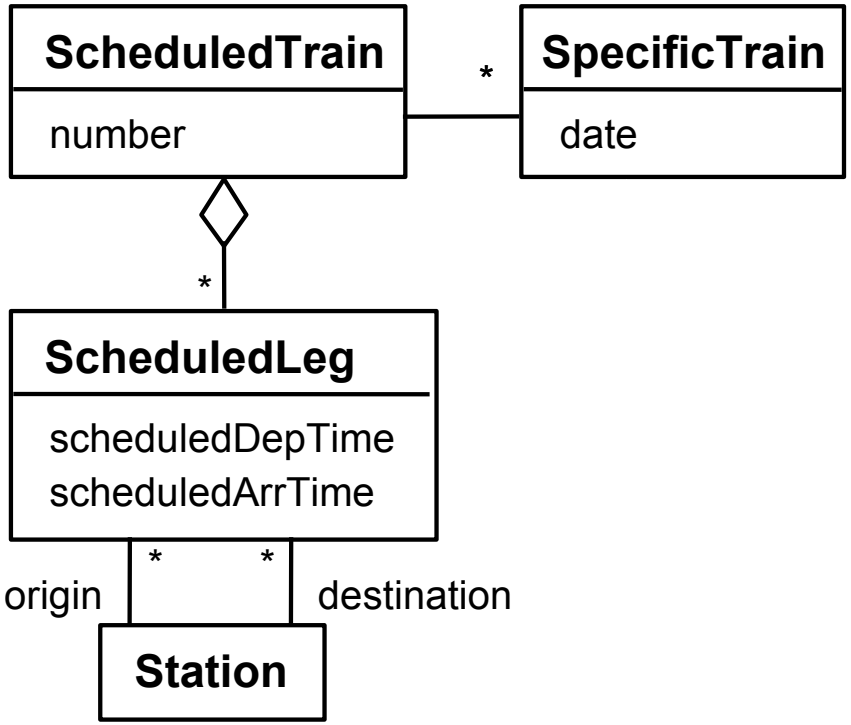
Abstraction-Occurrence

Square variant



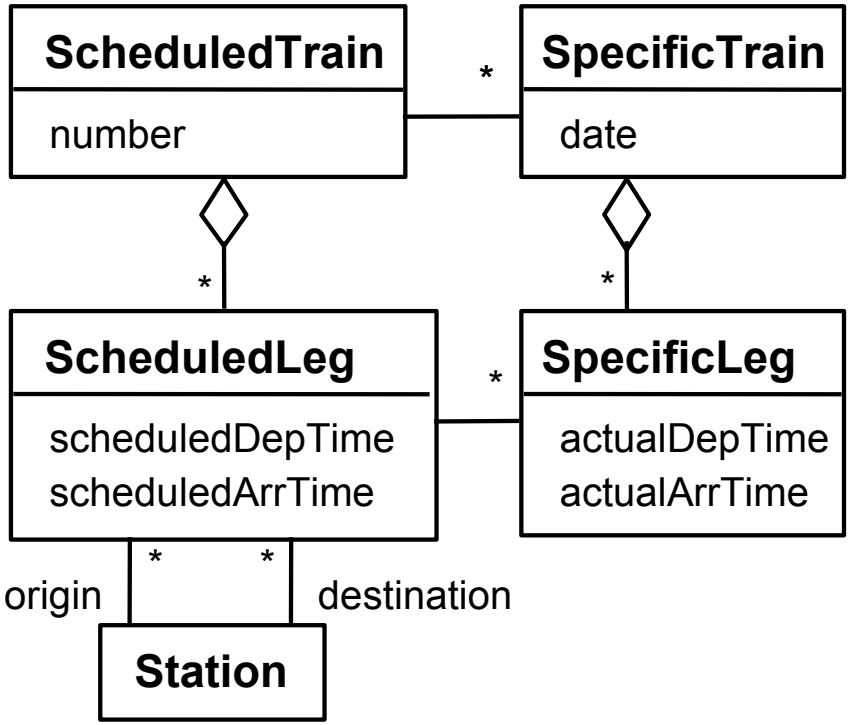
Abstraction-Occurrence

Square variant



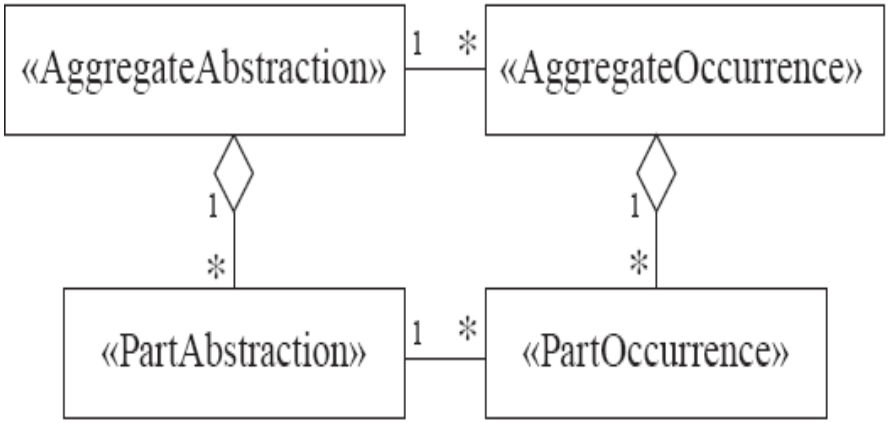
Abstraction-Occurrence

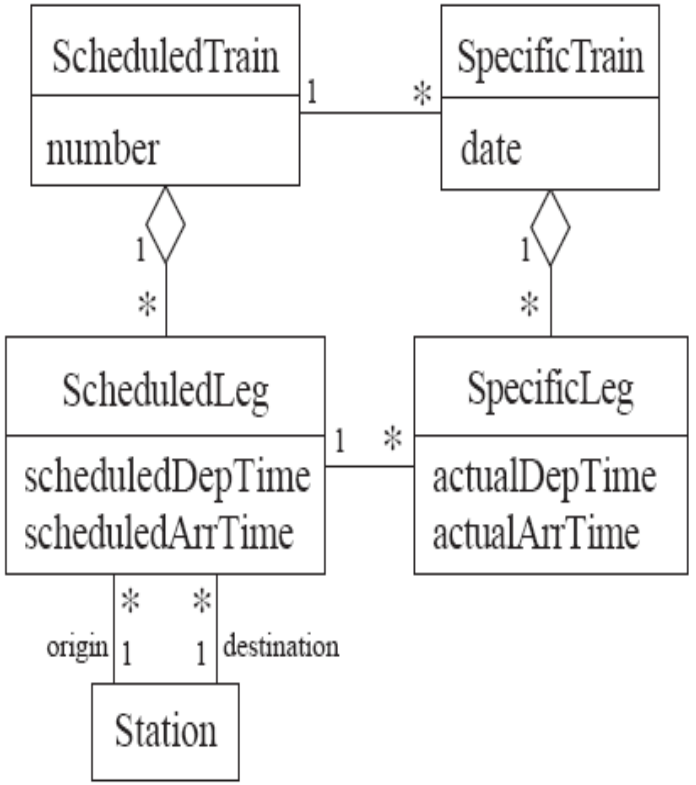
Square variant



Abstraction-Occurrence

Square variant





Exercise E6.1

Apply the Abstraction-Occurrence pattern in the following situations. For each situation, show the two linked classes, and the attributes in each class.

a) The issues of a periodical

Exercise E6.1

Apply the Abstraction-Occurrence pattern in the following situations. For each situation, show the two linked classes, and the attributes in each class.

a) The issues of a periodical

b) The copies of the issues of a periodical

Exercise E6.1

Apply the Abstraction-Occurrence pattern in the following situations. For each situation, show the two linked classes, and the attributes in each class.

a) The issues of a periodical

b) The copies of the issues of a periodical

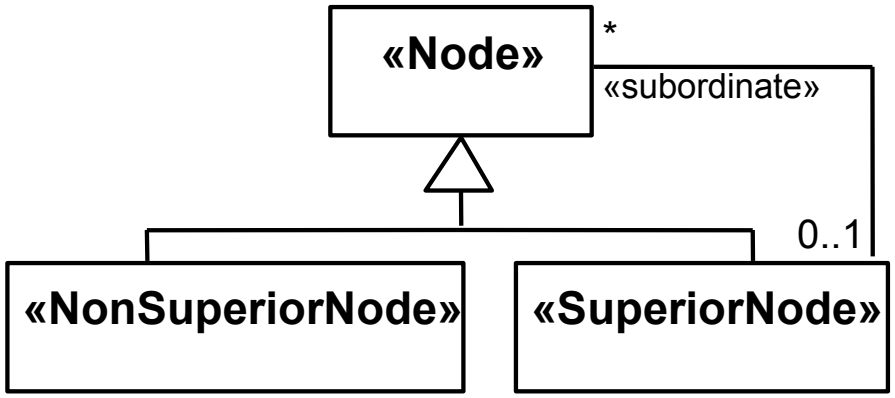
c) The repeats and re-runs of the same television program

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

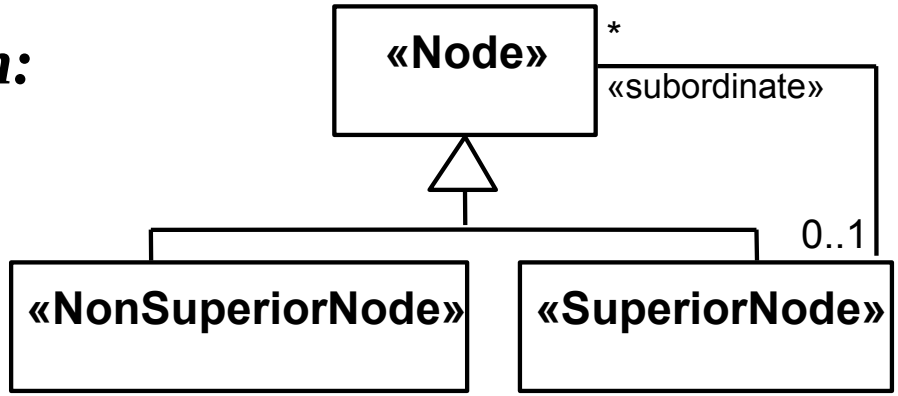
General Hierarchy

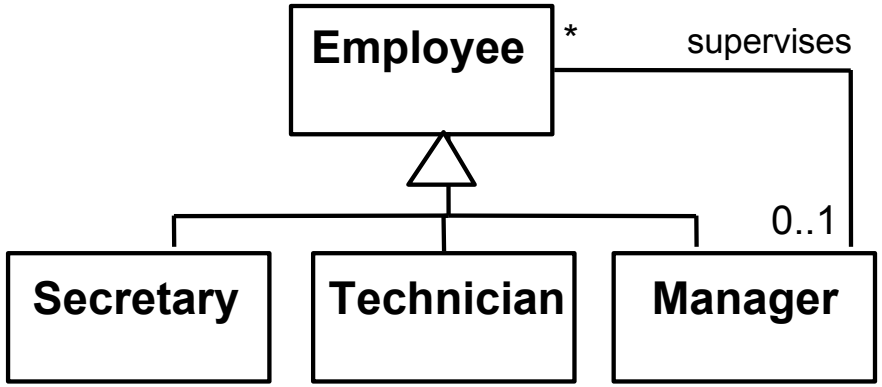
- ***Solution:***



General Hierarchy

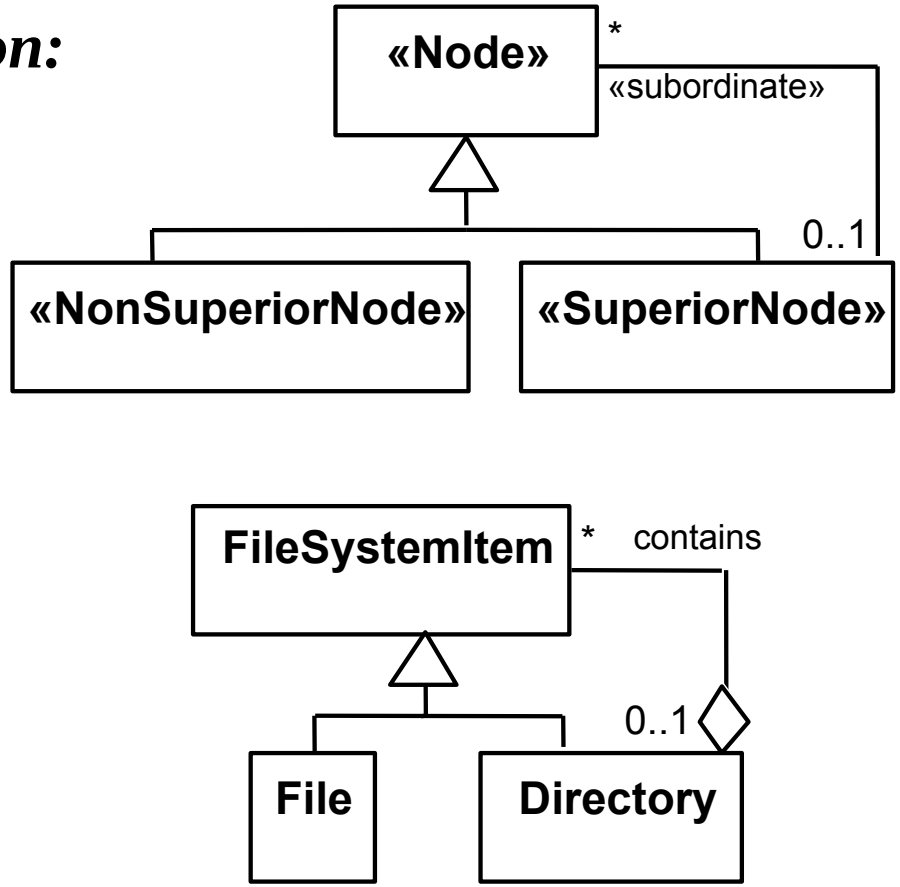
- ***Solution:***





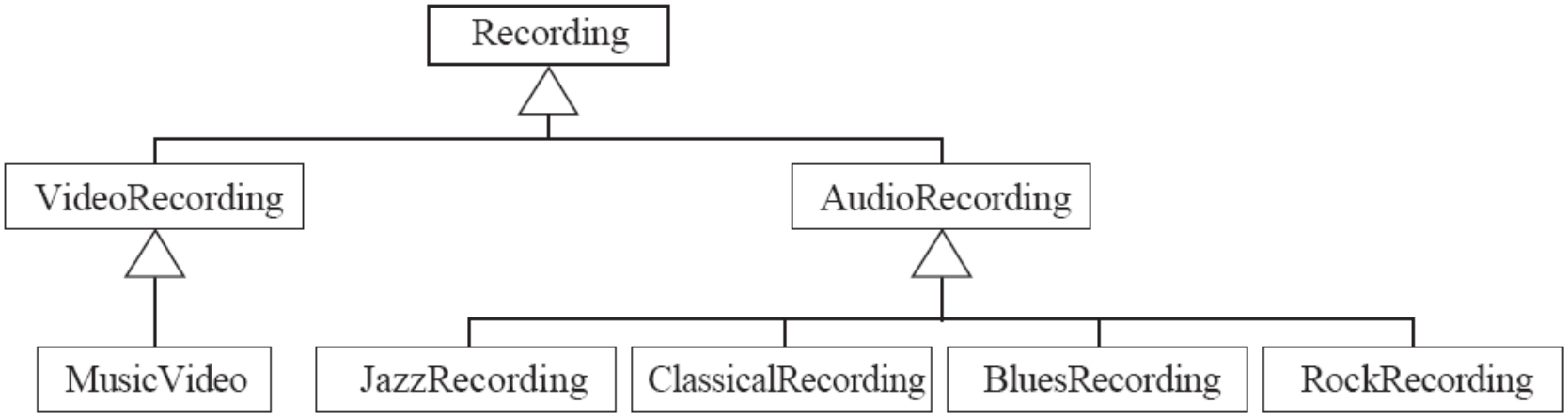
General Hierarchy

- ***Solution:***



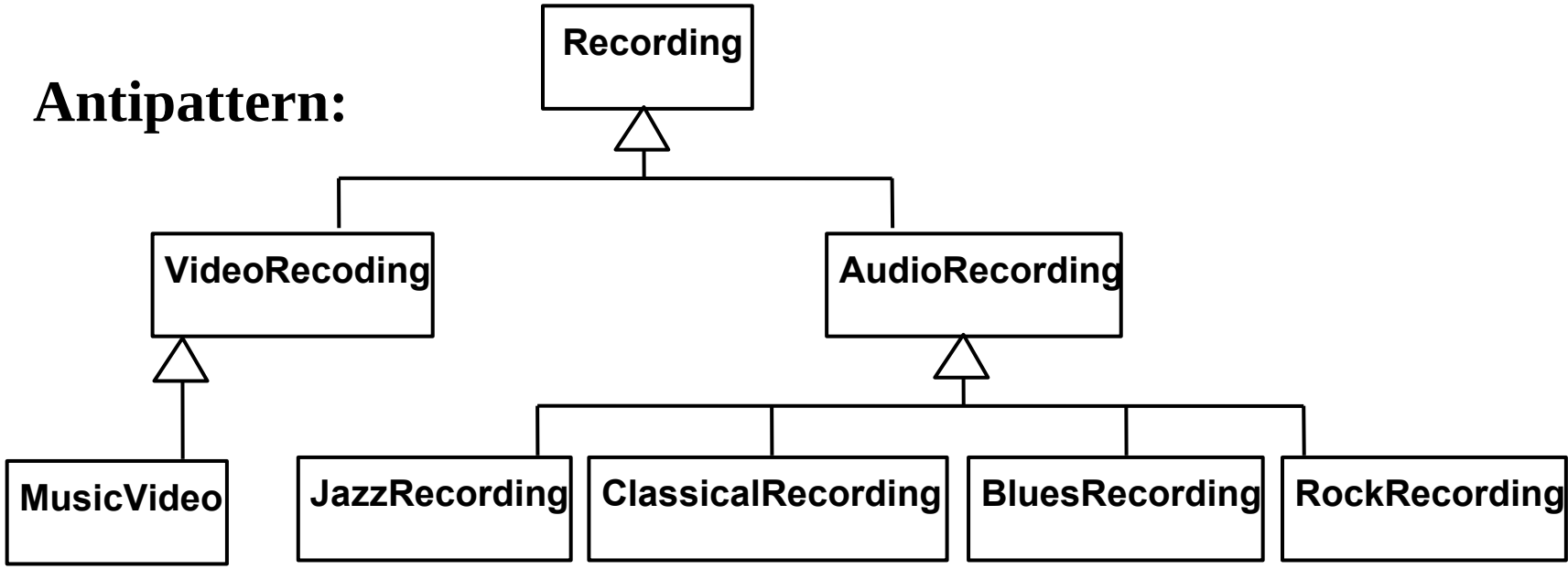
General Hierarchy

Antipattern:

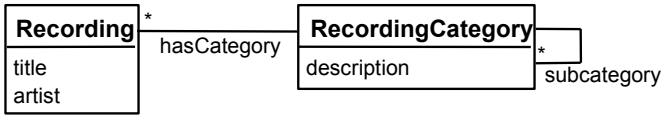


General Hierarchy

Antipattern:



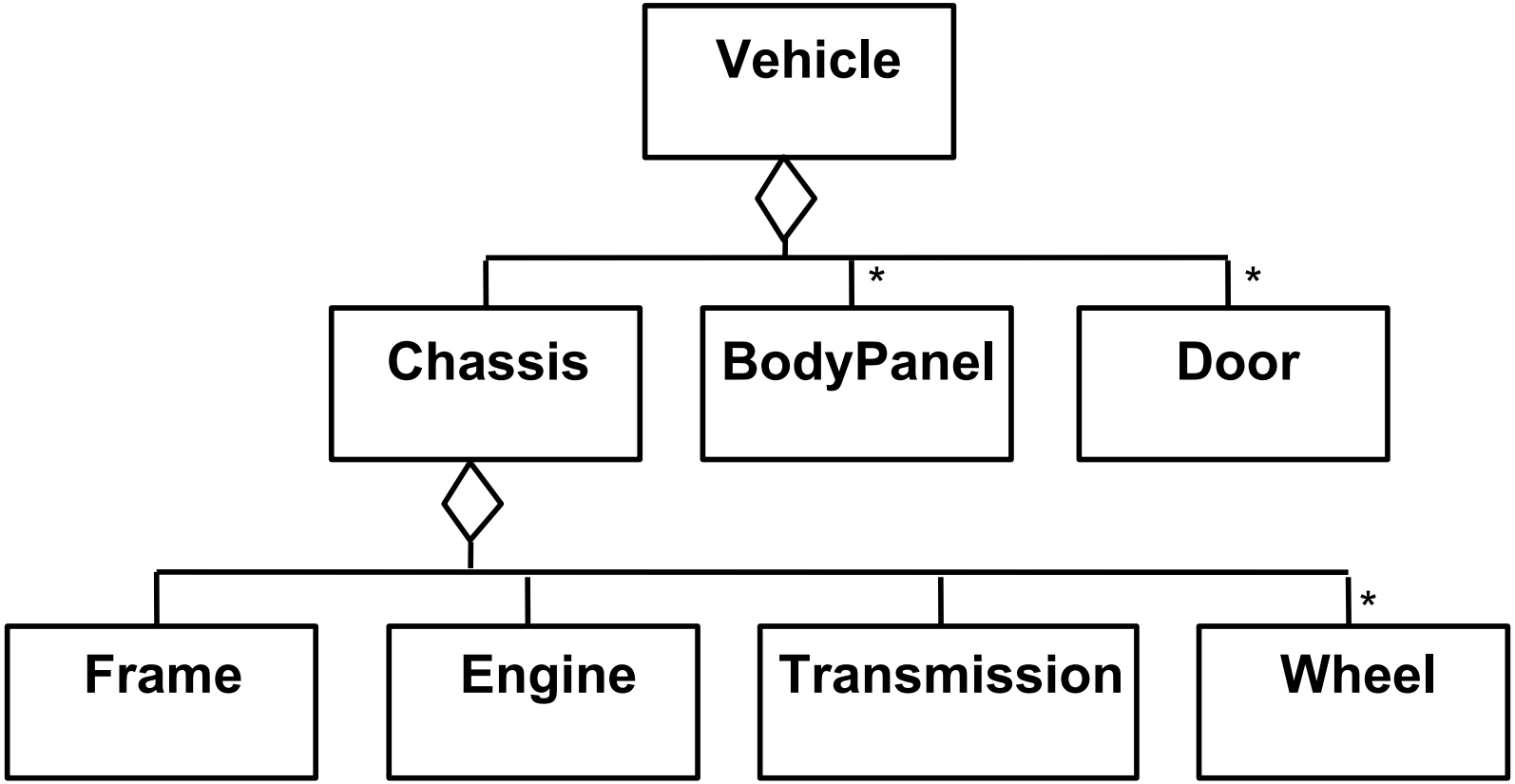
Better:



Exercise E6.2

Figure 5.20 (see next slide) shows a hierarchy of vehicle parts. Show how this hierarchy might be better represented using the General Hierarchy pattern (or more precisely, by the Composite pattern).

Fig. 5.20 Aggregation hierarchy

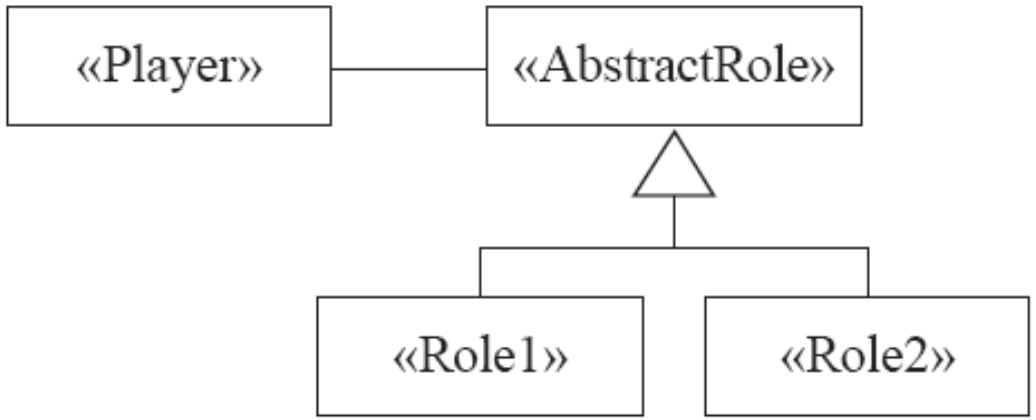


6.4 The Player-Role Pattern

- ***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?
- ***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

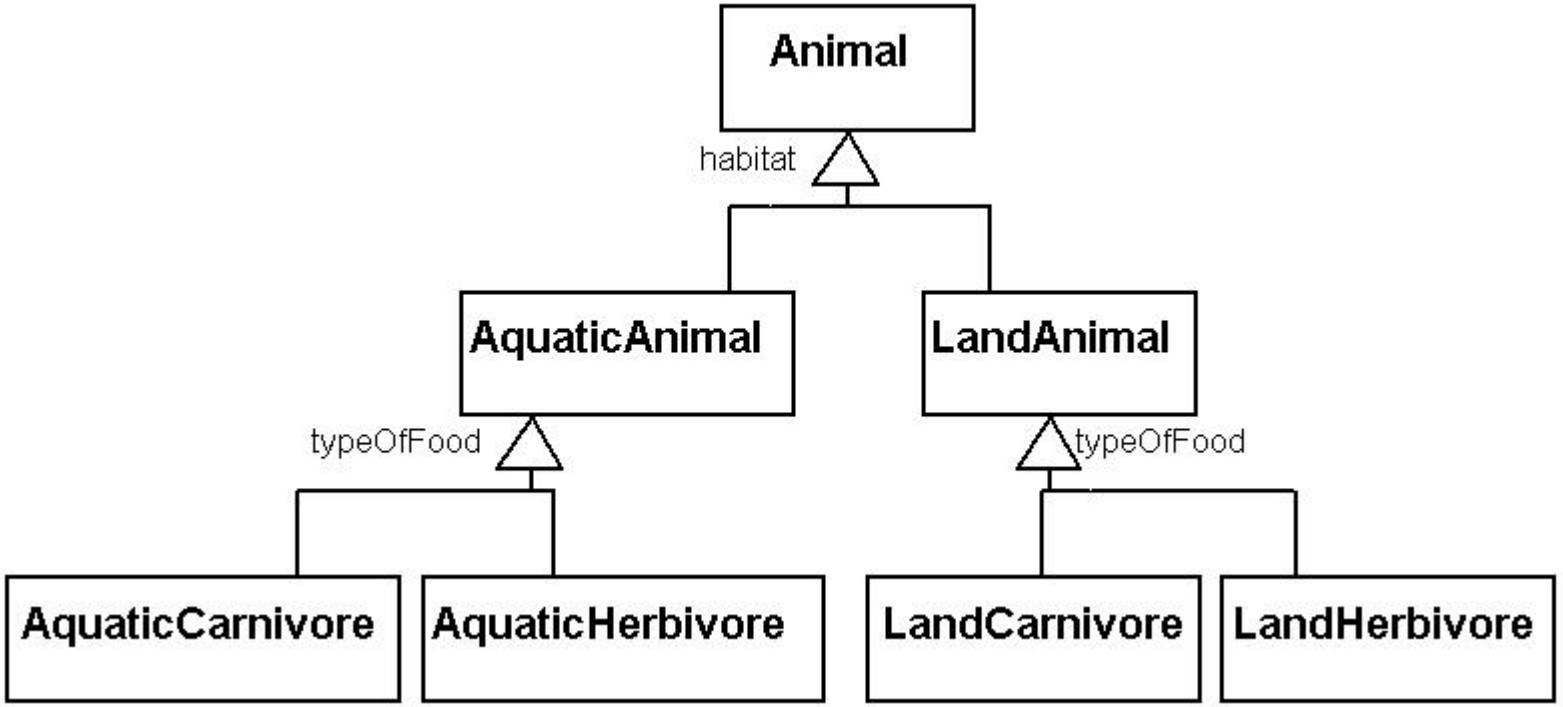
Player-Role

- ***Solution**:*



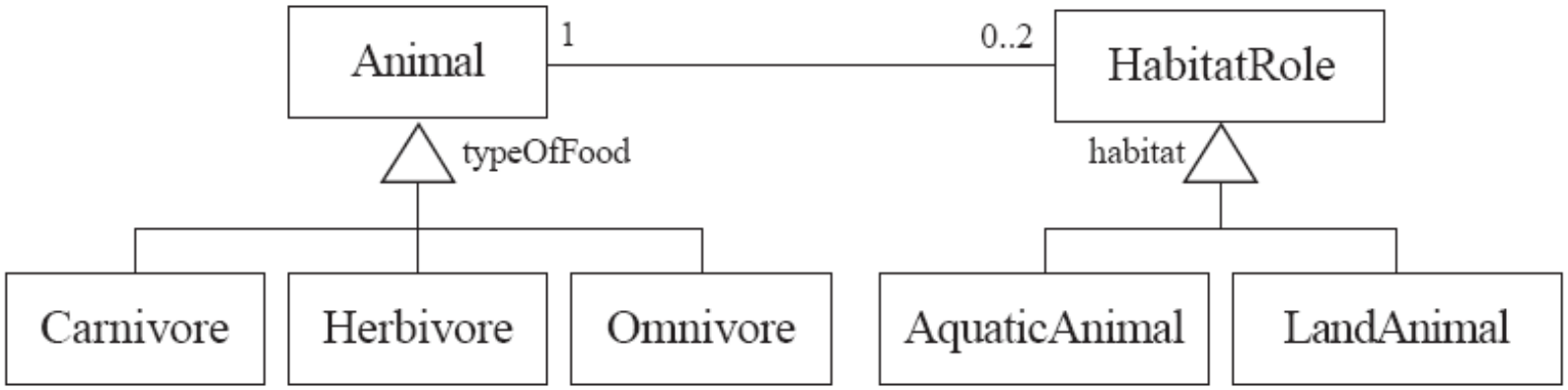
Player-Role

- Example of redundant hierarchy



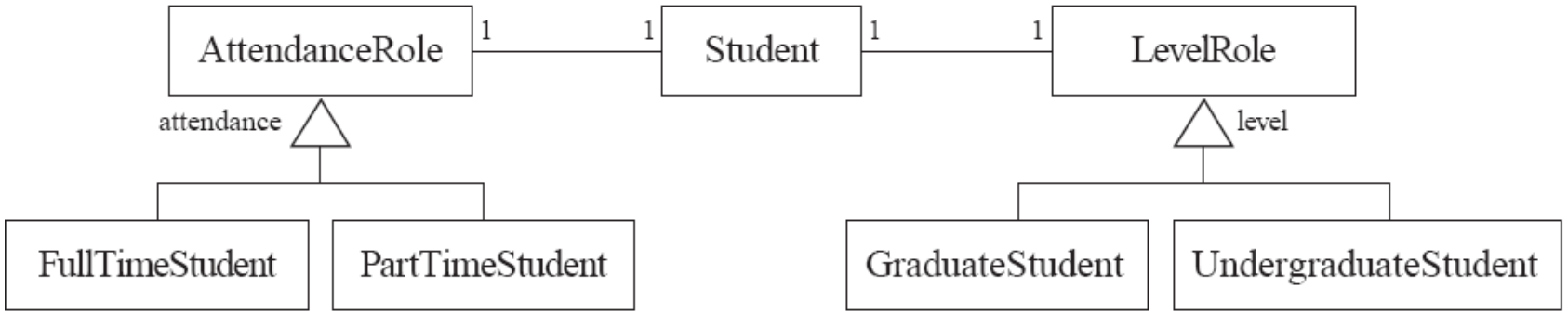
Player-Role

Example 1:



Player-Role

Example 2:



Player-Role

Antipatterns:

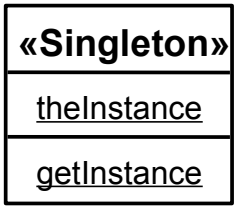
- Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.
- Create roles as subclasses of the «Player» class.

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

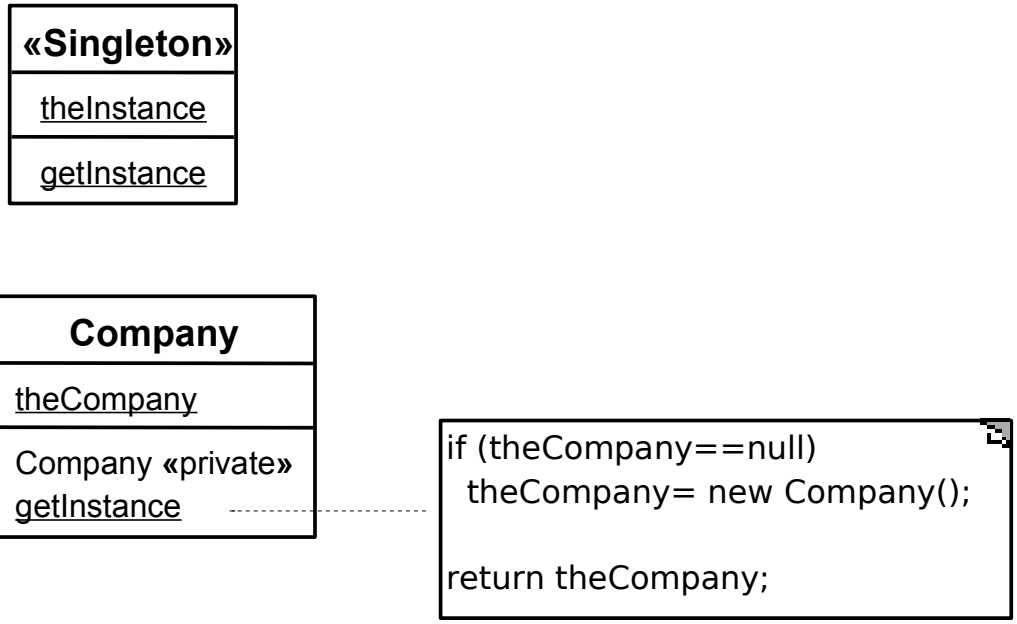
Singleton

- ***Solution:***



Singleton

- ***Solution:***

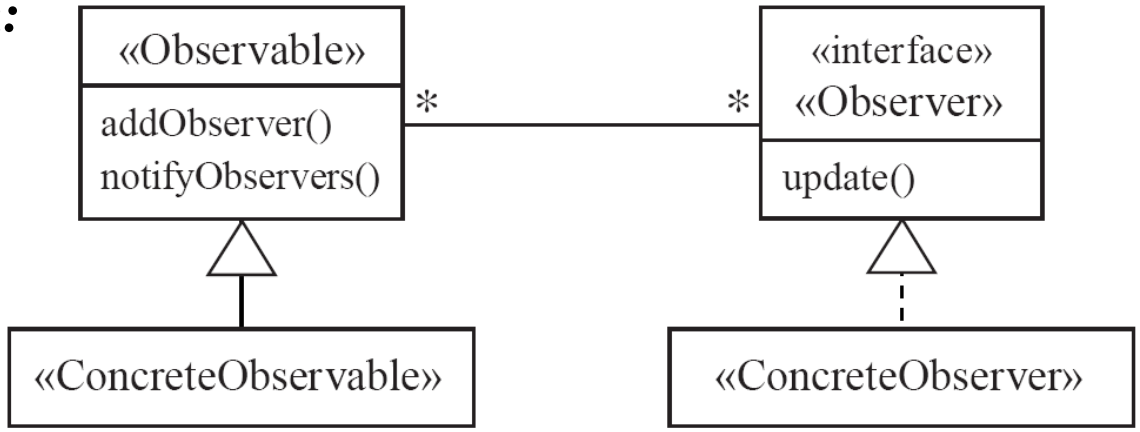


6.6 The Observer Pattern

- ***Context***:
  - —When an 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?
- ***Forces***:
  - —You want to maximize the flexibility of the system to the greatest extent possible

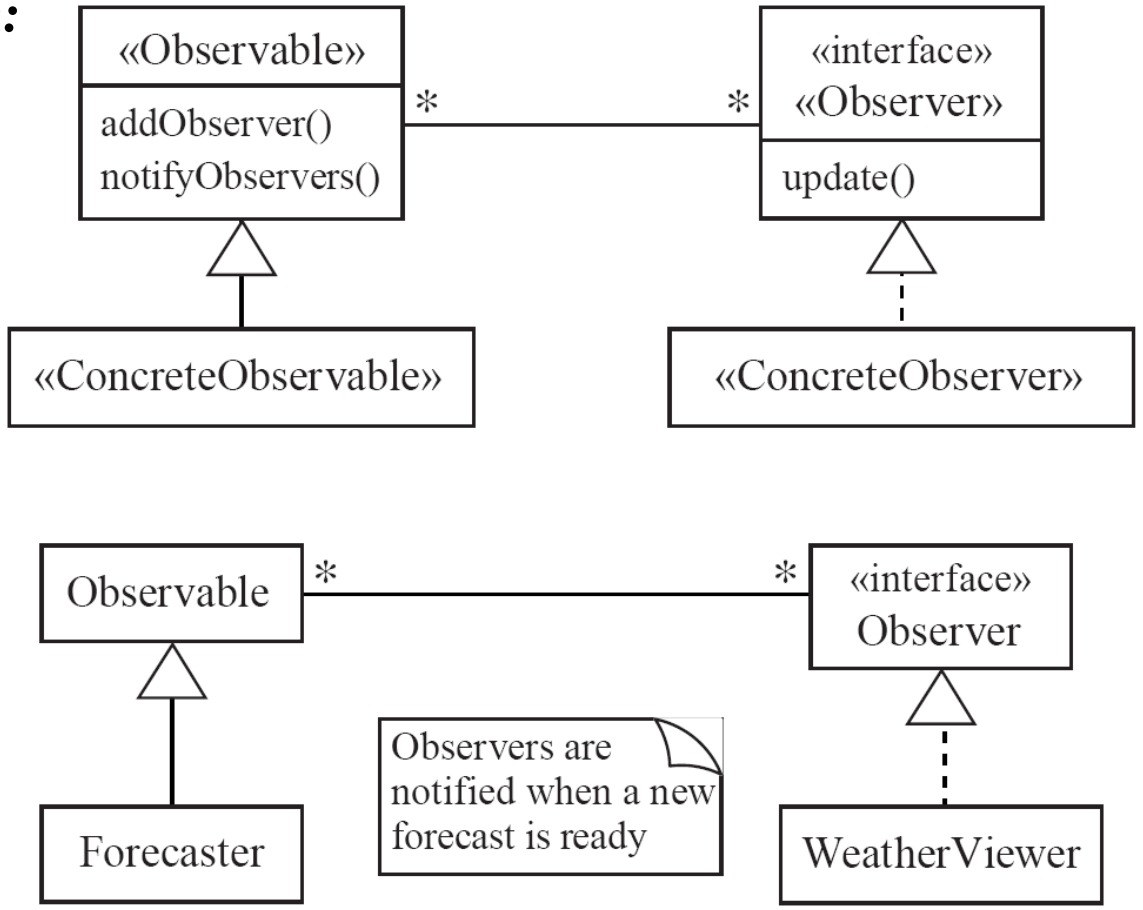
Observer

- ***Solution:***



Observer

- ***Solution:***



Observer

Antipatterns:

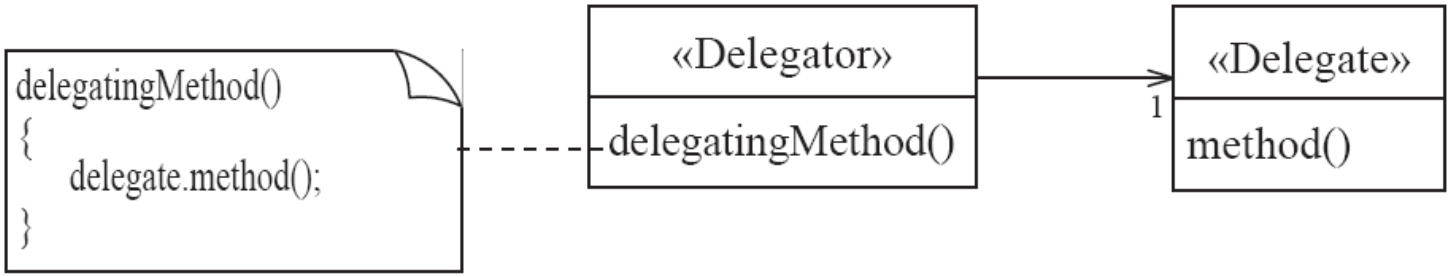
- Connect an observer directly to an observable so that they both have references to each other.
- Make the observers *subclasses* of the observable.

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

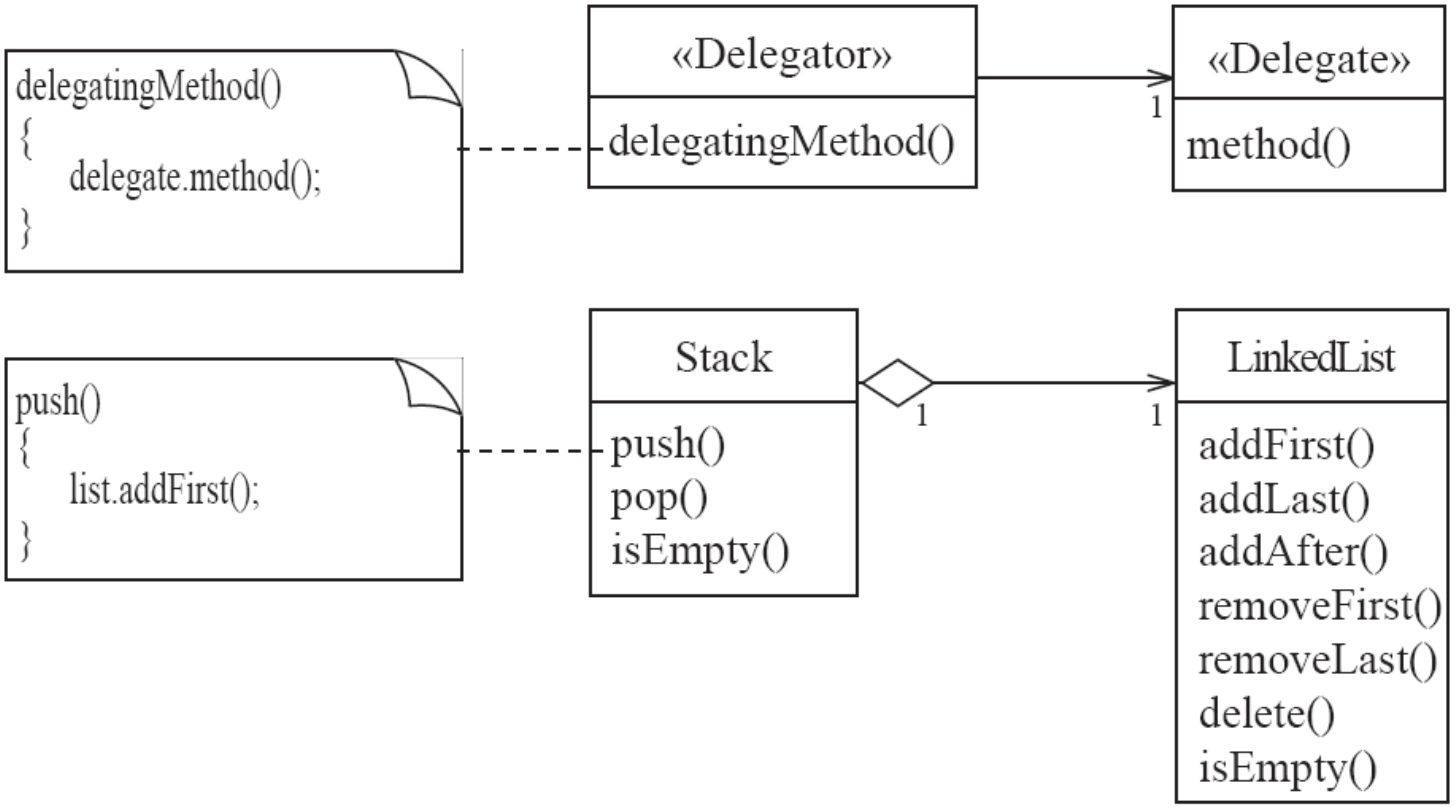
Delegation

- ***Solution:***



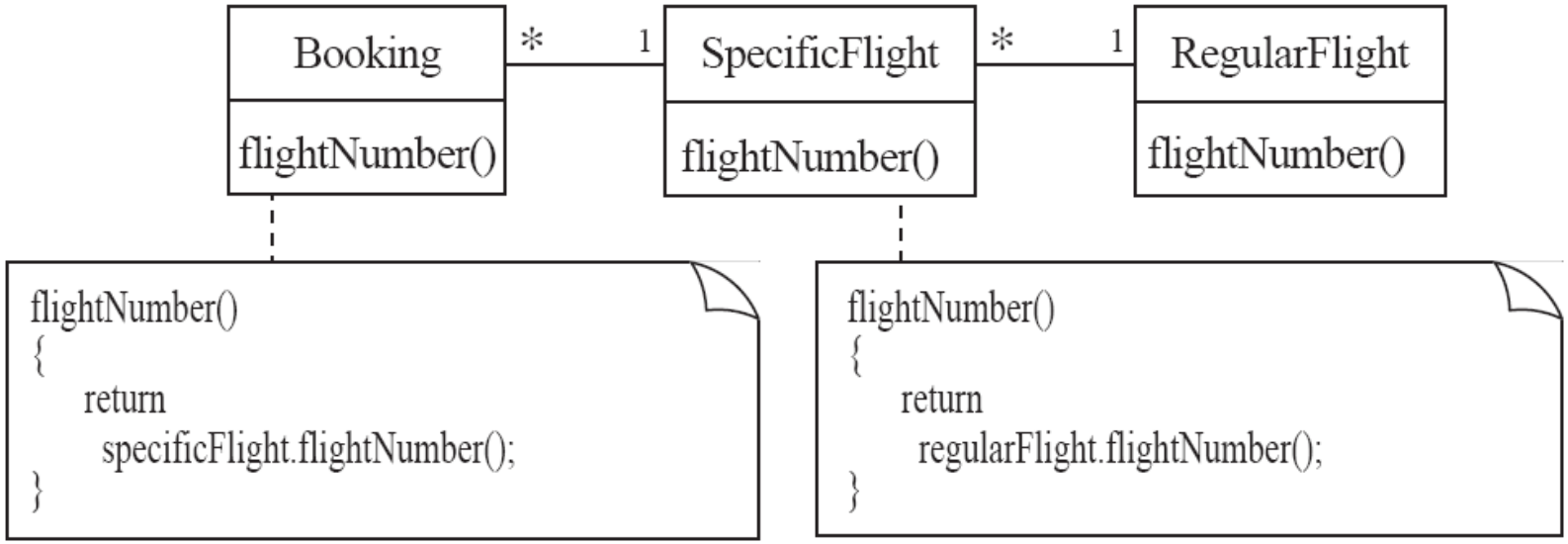
Delegation

- ***Solution:***



Delegation

Example:



Delegation

Antipatterns

- Overuse generalization and *inherit* the method that is to be reused
- Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate»
  - —having many different methods in the «Delegator» call the delegate's method
- Access non-neighboring classes

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

return getRegularFlight().flightNumber();
```

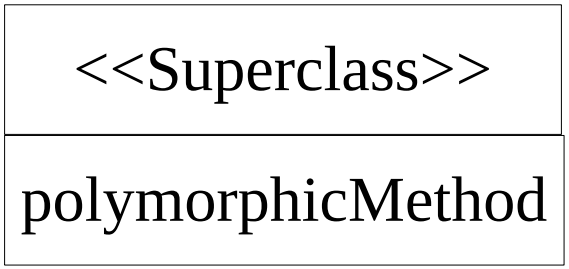
6.8 The Adapter Pattern

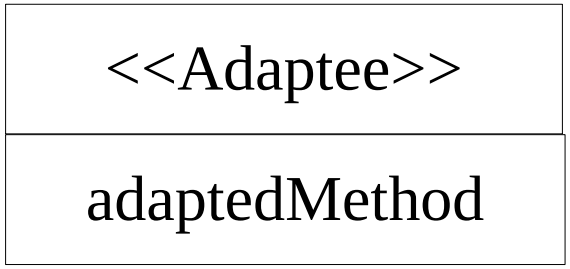
- ***Context***:
  - —You are building an inheritance hierarchy and want to incorporate it into an existing class.
  - —The reused class is also often already part of its own inheritance hierarchy.
- ***Problem***:
  - —How to obtain the power of polymorphism when reusing a class whose methods
    - 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

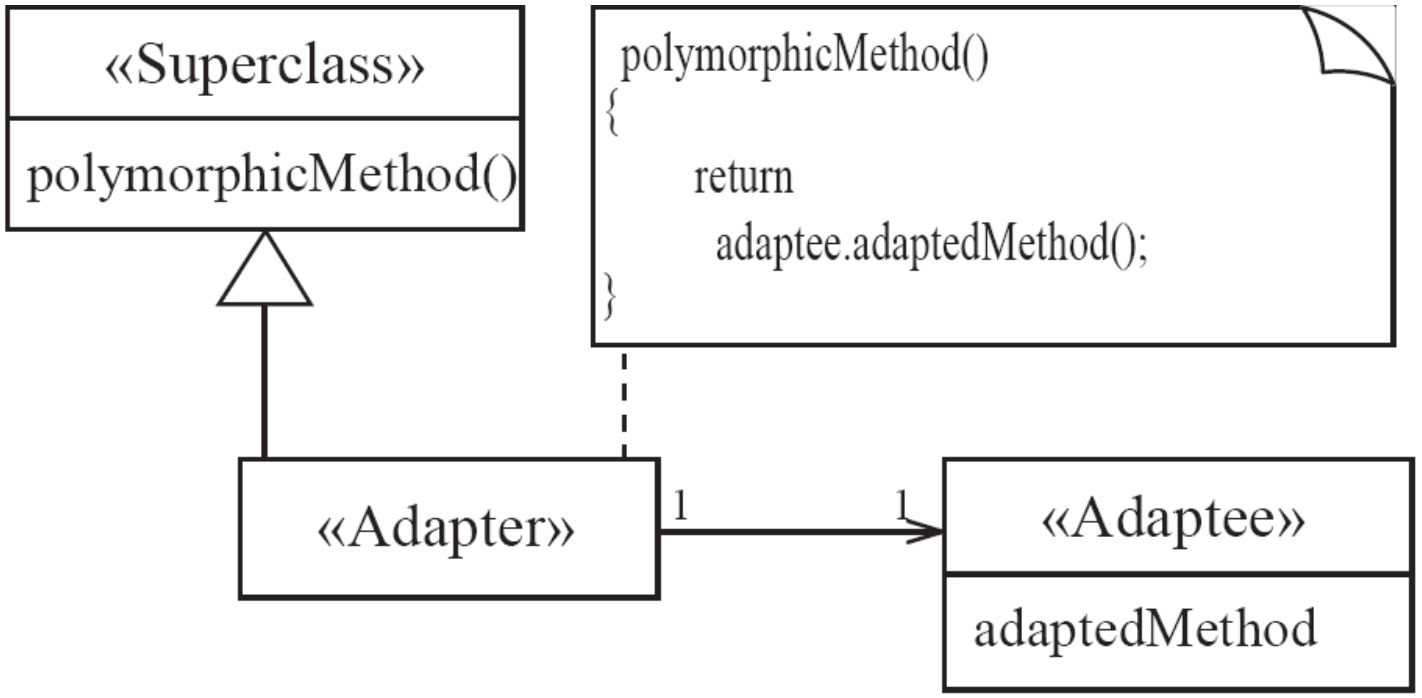
- ***Solution:***





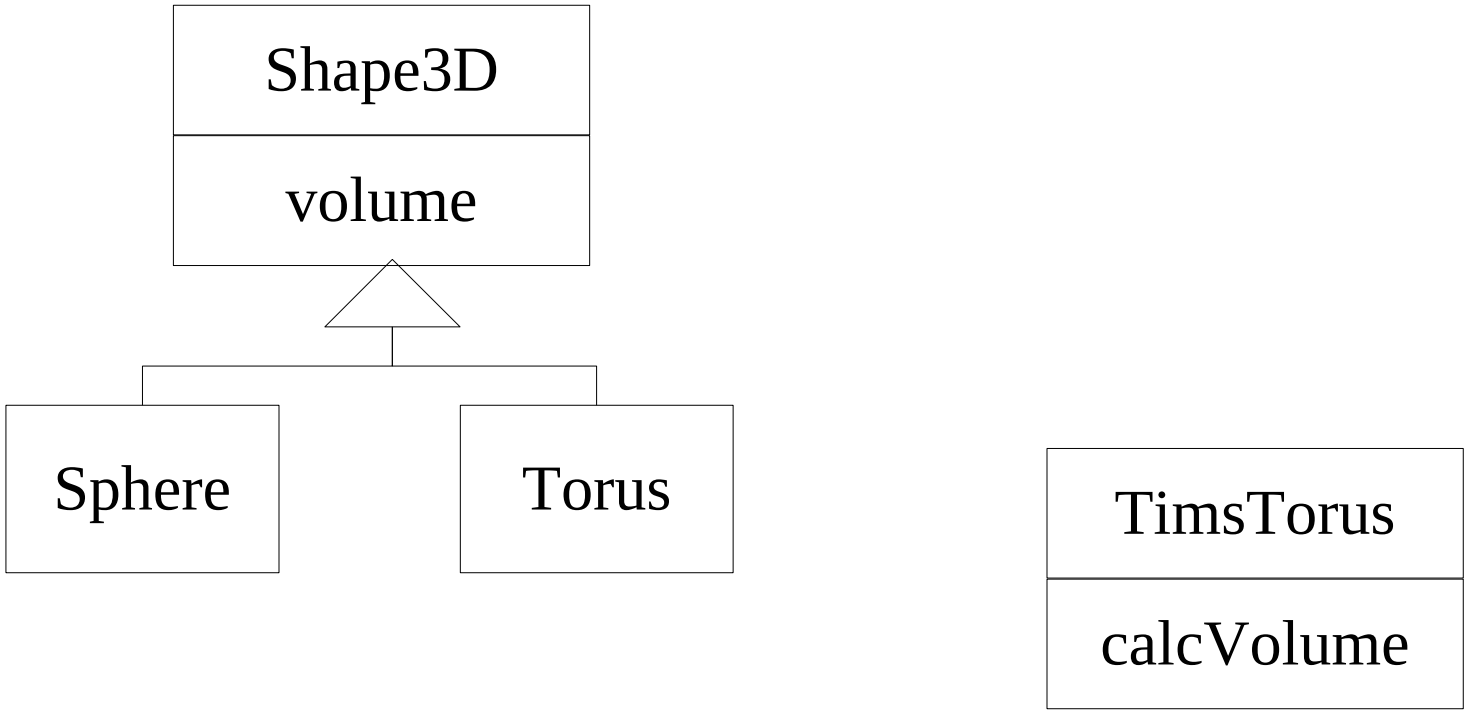
Adapter

- ***Solution:***



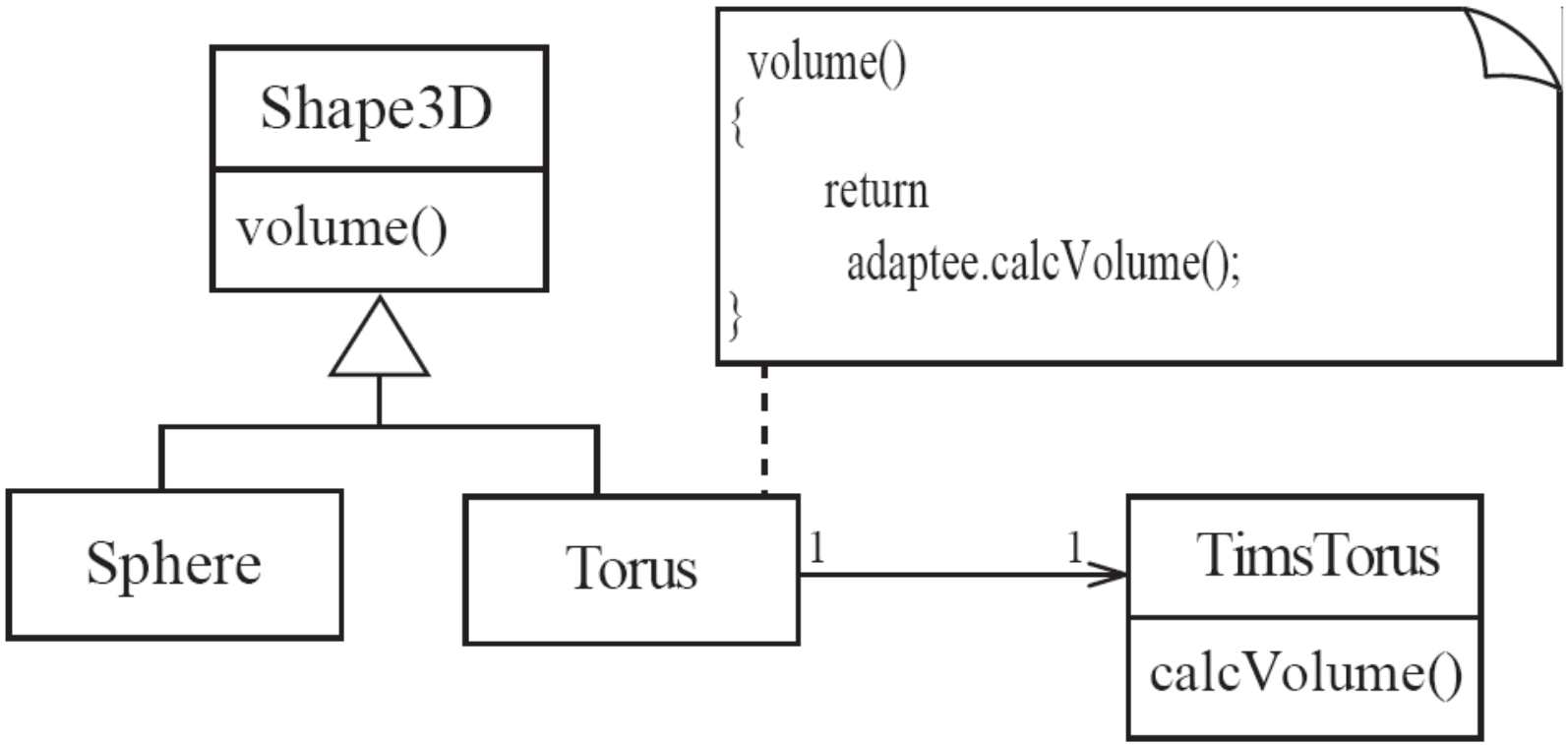
Adapter

Example:



Adapter

Example:

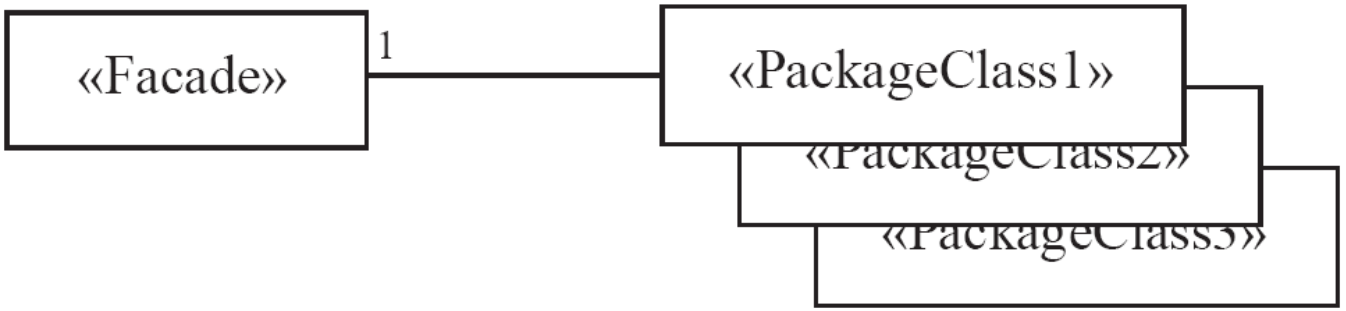


6.9 The Façade Pattern

- ***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.

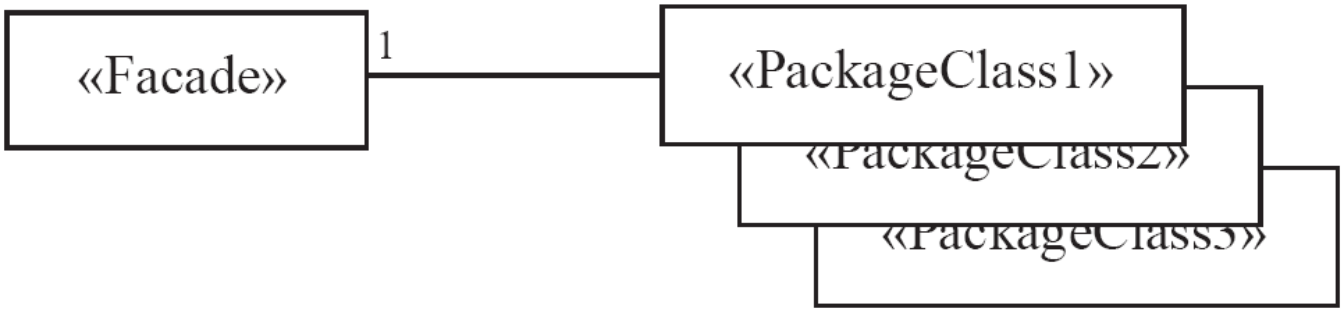
Façade

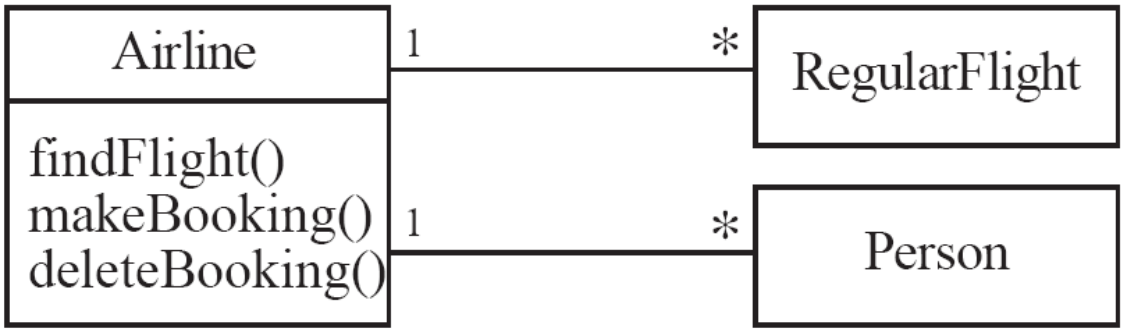
- ***Solution:***



Façade

- ***Solution:***





6.10 The Immutable Pattern

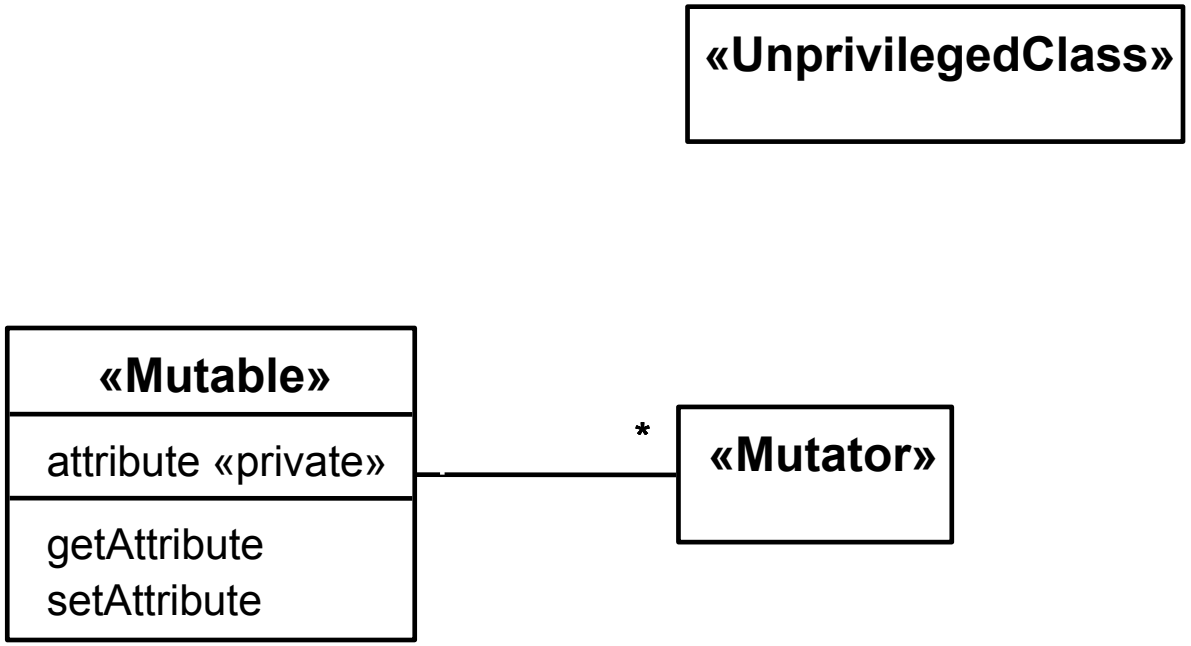
- ***Context***:
  - —An immutable object is an object that has a state that never changes after creation
- ***Problem***:
  - —How do you create a class whose instances are immutable?
- ***Forces***:
  - —There must be no loopholes that would allow 'illegal' modification of an immutable object
- ***Solution***:
  - —Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
  - —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.

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

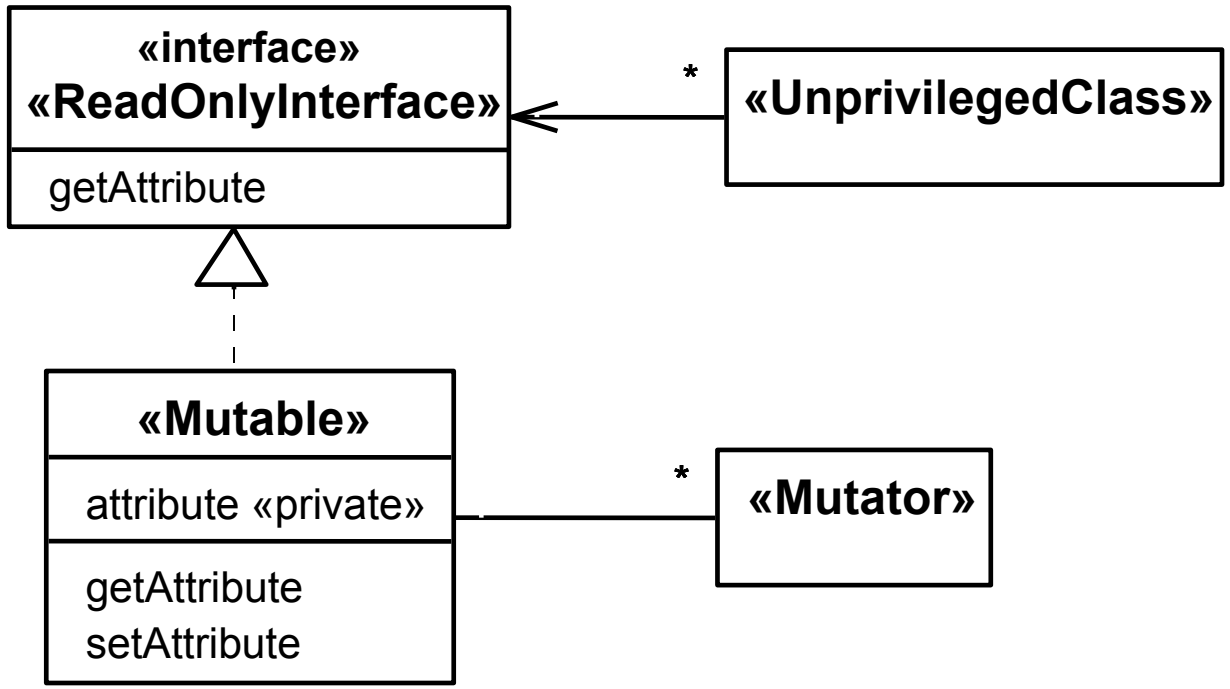
Read-only Interface

- ***Solution:***



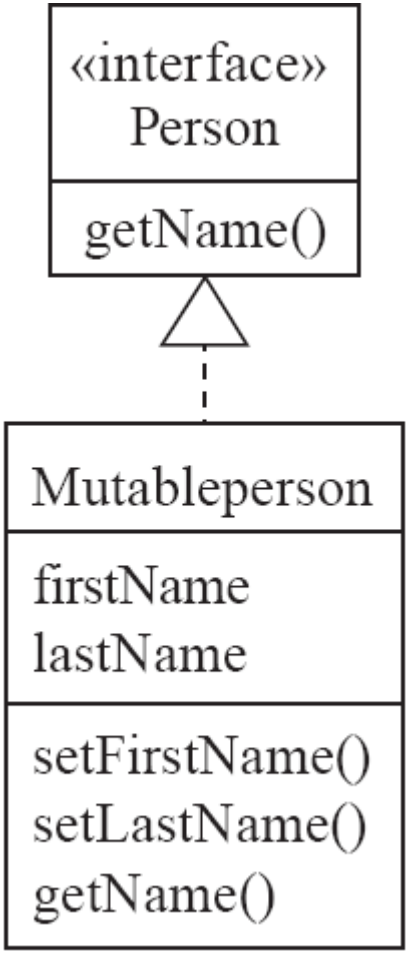
Read-only Interface

- ***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

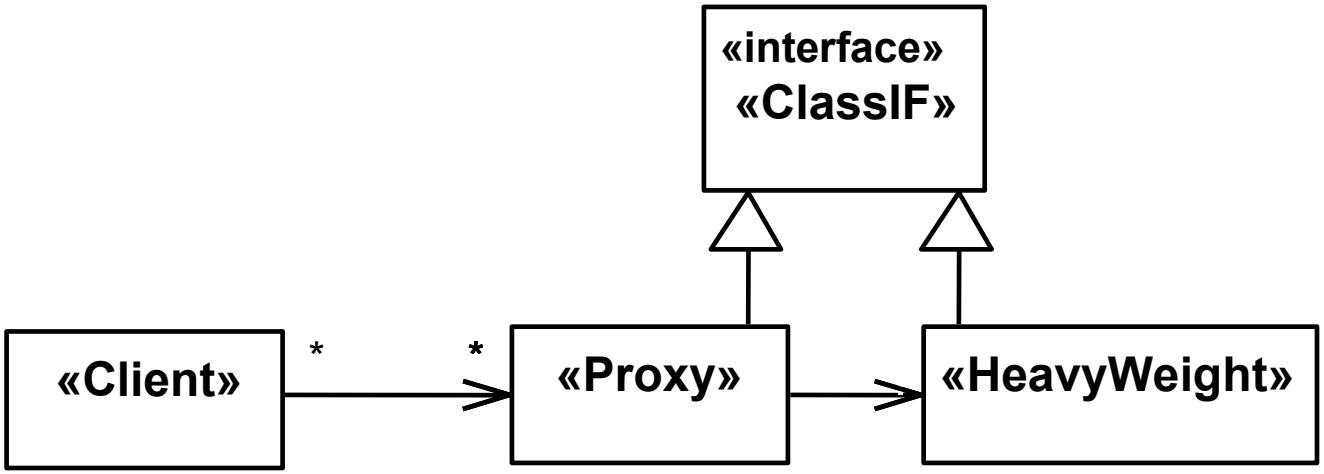
- ***Context***:
  - —Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).
  - —There is a time delay and a complex mechanism involved in creating the object in memory
- ***Problem***:
  - —How to reduce creating instances of a heavyweight class?
- ***Forces***:
  - —We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.
  - —It is also important for many objects to persist from run to run of the same program

Proxy

- ***Solution:***

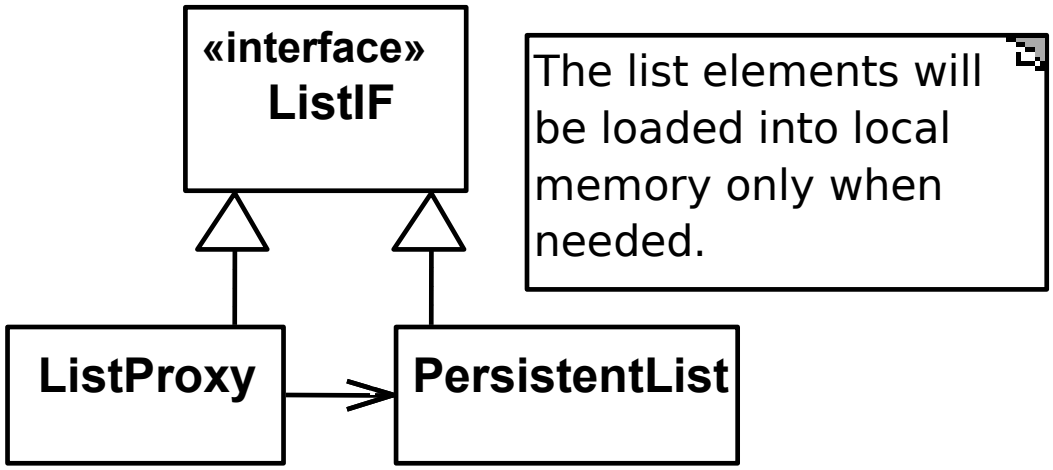
Proxy

- ***Solution:***



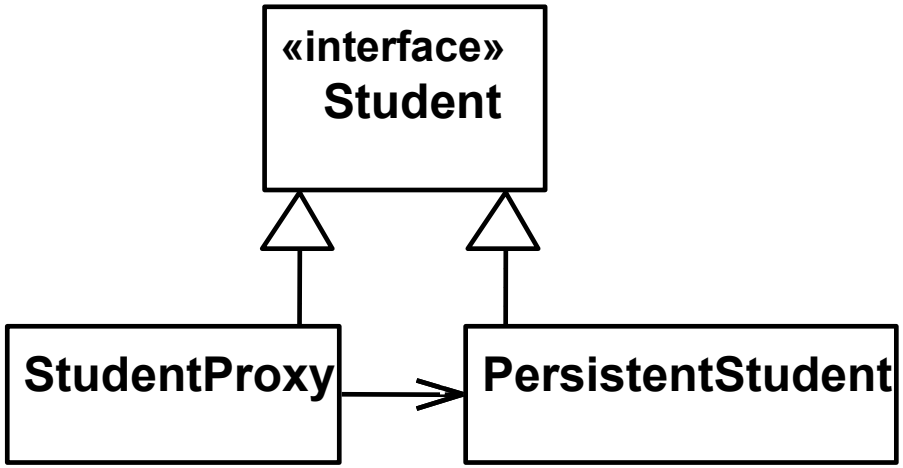
Proxy

Example:



Proxy

Example:



Exercise

Find the most appropriate design pattern for the following problems:

You are building an inheritance hierarchy of products that your company sells; however, you want to reuse several classes from one of your suppliers. You cannot modify your suppliers' classes. How do you ensure that the facilities of the suppliers' classes can still be used polymorphically?

Exercise

Find the most appropriate design pattern for the following problems:

You are building an inheritance hierarchy of products that your company sells; however, you want to reuse several classes from one of your suppliers. You cannot modify your suppliers' classes. How do you ensure that the facilities of the suppliers' classes can still be used polymorphically?

Adapter

Exercise (cont'd)

You want to reuse a method in one of the classes from one of your suppliers. You cannot subclass your supplier's class. How do you most effectively make use of the already existing method?

Exercise (cont'd)

You want to reuse a method in one of the classes from one of your suppliers. You cannot subclass your supplier's class. How do you most effectively make use of the already existing method?

Delegation

Exercise (cont'd)

You want to allow operations on instances of **RegularPolygon** that will distort them such that they are no longer regular polygons. How do you allow the operations without raising exceptions?

Exercise (cont'd)

You want to allow operations on instances of **RegularPolygon** that will distort them such that they are no longer regular polygons. How do you allow the operations without raising exceptions?

Immutable

Exercise (cont'd)

Your program manipulates images that take a lot of space in memory. How can you design your program so that images are only in memory when needed, and otherwise can only be found in files?

Exercise (cont'd)

Your program manipulates images that take a lot of space in memory. How can you design your program so that images are only in memory when needed, and otherwise can only be found in files?

Proxy

Exercise (cont'd)

You have created a subsystem with 25 classes. You know that most other subsystems will only access about 5 methods in this subsystem; how can you simplify the view that the other subsystems have of your subsystem?

Exercise (cont'd)

You have created a subsystem with 25 classes. You know that most other subsystems will only access about 5 methods in this subsystem; how can you simplify the view that the other subsystems have of your subsystem?

Facade

Exercise (cont'd)

You need to represent insects and insect specimens that are collected in a state forest. For every kind of insect in the forest, many specimens of that type will be collected for analysis. How do you represent specimens without duplicating common information like scientific name, or the name of the forest in which they are collected?

Exercise (cont'd)

You need to represent insects and insect specimens that are collected in a state forest. For every kind of insect in the forest, many specimens of that type will be collected for analysis. How do you represent specimens without duplicating common information like scientific name, or the name of the forest in which they are collected?

Abstraction-Occurrence

Exercise (cont'd)

You are developing a stock quote framework. Some applications using this framework will want stock quotes to be displayed on a screen when they become available; other applications will want new quotes to trigger certain financial applications; yet other applications might want both of the above, plus having quotes transmitted wirelessly to a network of pagers. How can you design the framework so that various different pieces of application code can react in their own way to the arrival of new quotes?

Exercise (cont'd)

You are developing a stock quote framework. Some applications using this framework will want stock quotes to be displayed on a screen when they become available; other applications will want new quotes to trigger certain financial applications; yet other applications might want both of the above, plus having quotes transmitted wirelessly to a network of pagers. How can you design the framework so that various different pieces of application code can react in their own way to the arrival of new quotes?

Exercise (cont'd)

You would like users in an airline reservation system to be treated as employees in one setting but as customers in another. You want to represent users as objects but you don't want any objects to change class.

Exercise (cont'd)

You would like users in an airline reservation system to be treated as employees in one setting but as customers in another. You want to represent users as objects but you don't want any objects to change class.

Player-Role

Exercise (cont'd)

You want to represent genealogical information in a family tree. You need to represent that some bloodlines end without any descendants. You don't want your class diagram to have to change every time a new family is added as a node to the family tree.

Exercise (cont'd)

You want to represent genealogical information in a family tree. You need to represent that some bloodlines end without any descendants. You don't want your class diagram to have to change every time a new family is added as a node to the family tree.

General Hierarchy

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. However 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.*