Dalhousie University
Faculty of Computer Science

CSCI 3132 – Object Orientation and Generic Programming

Week 10 – Design Patterns

Design Patterns

What are Design Patterns?

- A general repeatable solution to a commonly occurring problem (Software engineering)
  - Template for how to solve a problem that can be repeatedly used in several different situations
  - Not a finished design that can be transformed directly into code
- Design patterns
  - Speed up the development process
  - Prevent subtle issues that can cause major problems
  - Improve code understanding for others who are familiar with the patterns
  - Allow developers to communicate using well-known and well-understood names for software interactions

Ref: Design Patterns Explained Simply

Elements of a Design Pattern

- Each pattern has four essential elements:
  - The pattern name
    - A handle to describe a design problem.
    - Makes it easier to communicate and design at a higher level
  - The problem
    - Explains the problem and its context
    - Describes when to apply the pattern
  - The solution
    - Describes the elements that make up the design, their relationships, responsibilities, and collaborations
    - Doesn't describe a concrete design or implementation, as pattern is more like a template to fit our problem
  - The consequences
    - Results and trade-offs of applying the pattern
    - Critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern

Ref: [GOF]

Classification of Design Patterns

- Creational design patterns
  - Related to class instantiation
  - Can be further sub-divided into:
    - Class-creation patterns that effectively use inheritance in the instantiation process
    - Object-creation patterns that use delegation effectively to get the job done
- Structural design patterns
  - Deal with class and object composition
    - Structural class-creation patterns that use inheritance to compose interfaces
    - Structural object-creation patterns that define ways to compose objects to obtain new functionality
- Behavioral design patterns
  - Patterns that are concerned with communication between objects

Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Ref: [GOF]

Creational Patterns

- Abstract factory
  - Creates an instance of several families of classes
- Builder
  - Separates object construction from its representation
- Factory method
  - Creates an instance of several derived classes
- Prototype
  - A fully initialized instance to be copied or cloned
- Singleton
  - A class of which only a single instance can exist

Structural Patterns

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object's interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Façade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Proxy
  - An object representing another object

Behavioral Patterns

- Chain of responsibility: A way of passing a request between a chain of objects
- Command: Encapsulate a command request as an object
- Interpreter: A way to include language elements in a program
- Iterator: Sequentially access the elements of a collection
- Mediator: Defines simplified communication between classes
- Memento: Capture and restore an object's internal state
- Observer: A way of notifying change to a number of classes
- State: Alter an object's behavior when its state changes
- Strategy: Encapsulates an algorithm inside a class
- Template method: Defer the exact steps of an algorithm to a subclass
- Visitor: Defines a new operation to a class without change

Patterns covered

Included in exam:

- Singleton - [GoF, 1995], [Lethbridge, 2002]
- Strategy - [GoF, 1995]
- Player-Role - [Lethbridge, 2002]
- Abstraction-Occurrence - [Lethbridge; 2002]
- General-Hierarchy - [Lethbridge; 2002]
- Observer - [Priestley, 2004], [GoF, 1995]

How we will Study Patterns

- Context:
  - In which situations should the pattern be employed?
- Problem:
  - What is the problem we are trying to solve?
- Forces:
  - What are the issues and concerns that need to be considered?
  - What is the criteria for evaluating a good solution?
- Solution:
- Antipatterns :
  - What are the common mistakes made in the context of the problem?

Ref: [LET]

Singleton

Object can have One and only One Instance

The Singleton Pattern

- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
  - Examples:
    - The Main Window in a GUI where only one main window can exist.
    - Objects handling registry settings or application preferences.
    - Objects handling thread pools in a multi-threaded application.
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?

The Singleton Pattern

– ***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

The Singleton Pattern – Discussion

Q: Say we have a `MyReg` class that handles Registry or configuration of our application.
How do you create a single object?

A: `MyReg* reg = new MyReg();`

Q: Can other objects call new on `MyReg` again?

A: Yes, as many times as they want.

Q: Can we prevent other objects from creating `MyReg` objects? How?

A: Yes, by making constructor of `MyReg` private.

The Singleton Pattern

Q: If constructor is private, how can the class be instantiated?

A: The code in `MyReg` could instantiate its object

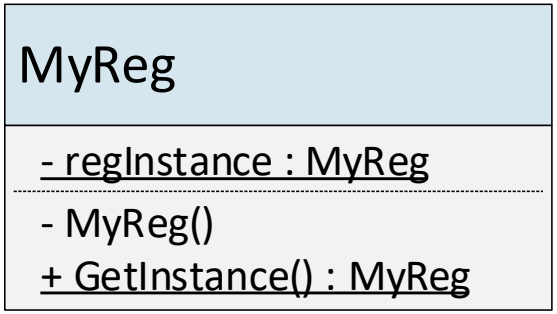
Solution:

Use a static method

The Singleton Pattern – Solution

```
#include <iostream>
class MyReg{
    static MyReg* regInstance;
    MyReg(){
        std::cout<<" Yay, I've been created";
    }
public:
    static MyReg* GetInstance();
};

MyReg* MyReg::regInstance = 0;
MyReg* MyReg::GetInstance(){
    if (regInstance == nullptr){
        MyReg* regInstance = new MyReg();
    }
    return regInstance;
}
int main(){
    MyReg::GetInstance();
}
```



The Singleton Pattern – Solution

```
#include <iostream>
class MyReg{
    MyReg(){
        std::cout<<"Yay, I've been created";
    }
public:
    static MyReg& GetInstance(){
        static MyReg regInstance;
        return regInstance;
    }
};
int main(){
    MyReg::GetInstance();
}
```

Do I Really Need a Singleton?

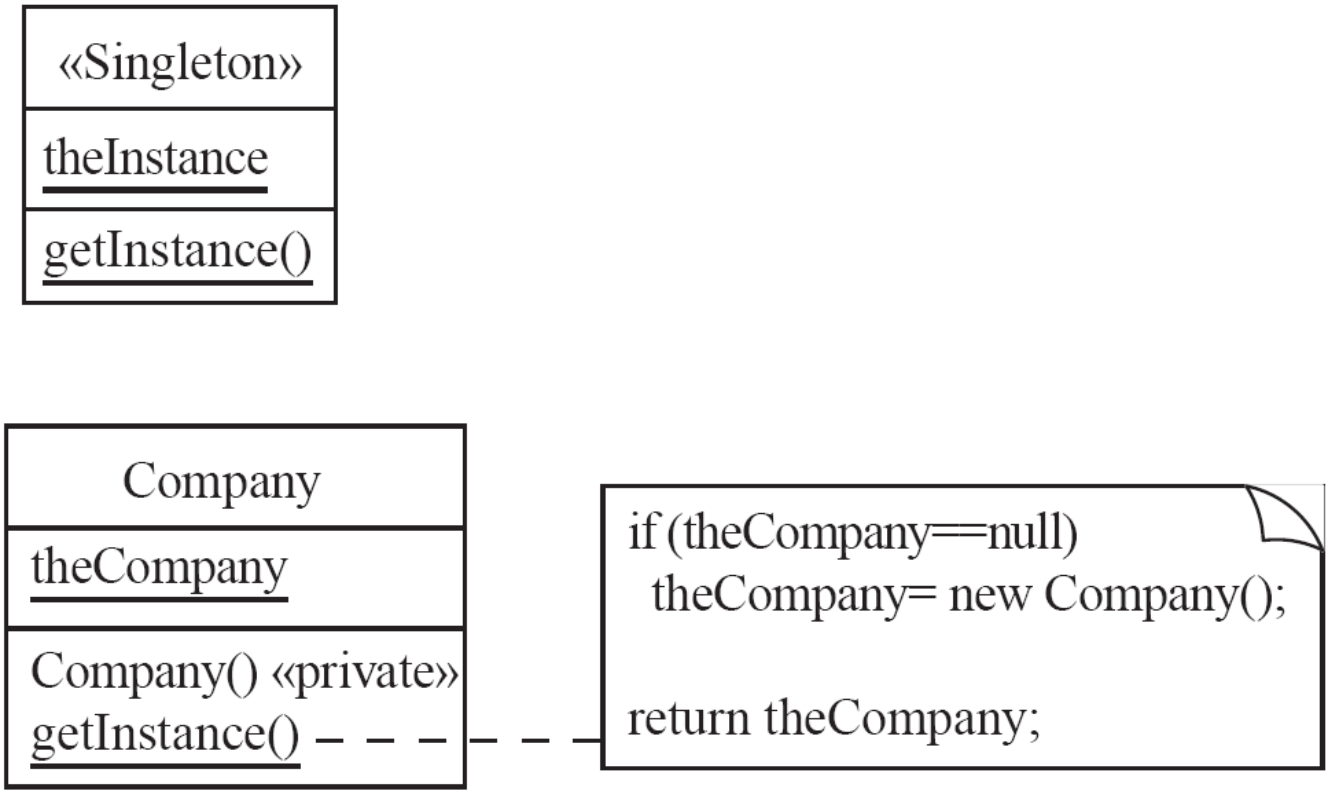
Q: Why use a Singleton pattern instead of assigning an object to a global variable?

A1: You could.

A2: Object assigned to a global variable must be created when application begins. This could be costly for resource intensive apps that may end up not using it.

A3: Singleton pattern is a time-tested method of ensuring that you provide **global access** to an object that is only ever going to have **exactly one instance**

Singleton – Summary



Ref: [LET]

Singleton – Exercise

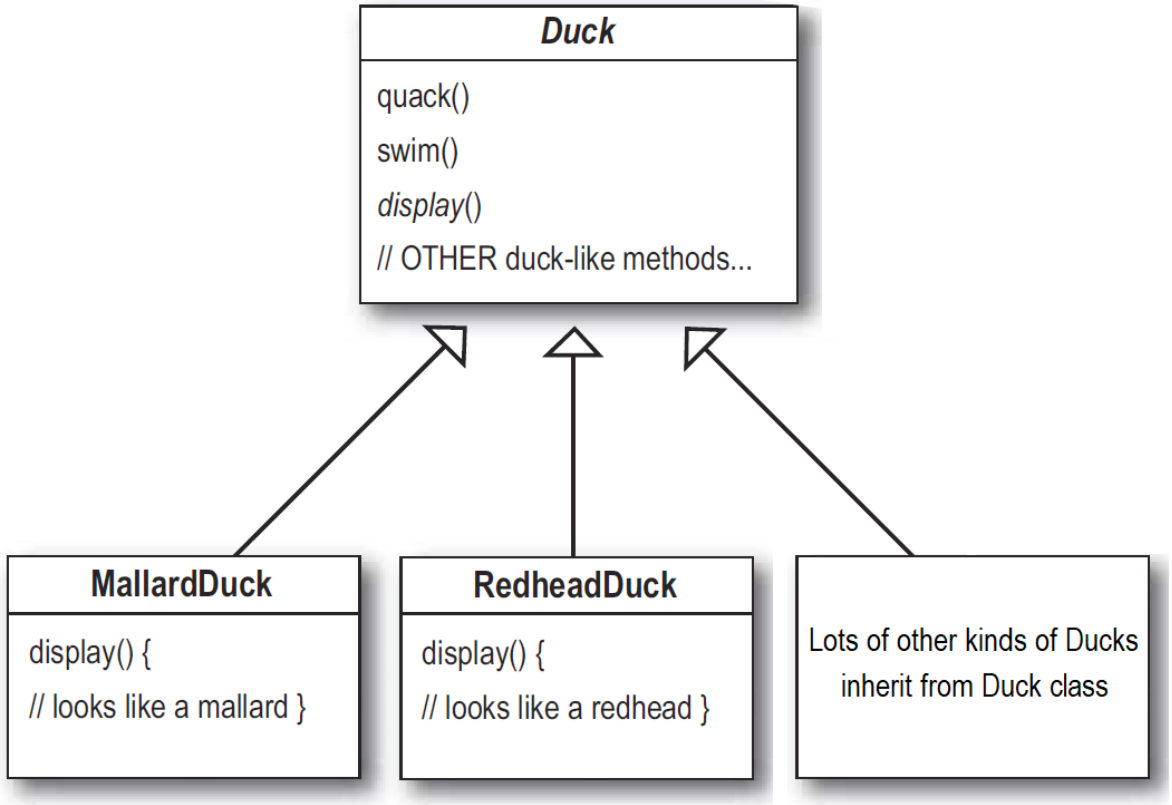
- Generalize the Singleton pattern so that a class could be limited to have a maximum of N instances.

SimUDuck App

(The Strategy Pattern)

Simple SimUDuck App

SimUDuck is a highly successful duck pond simulation game that shows many duck species swimming and making quacking sounds. Its OO design is as shown below. [FRE]

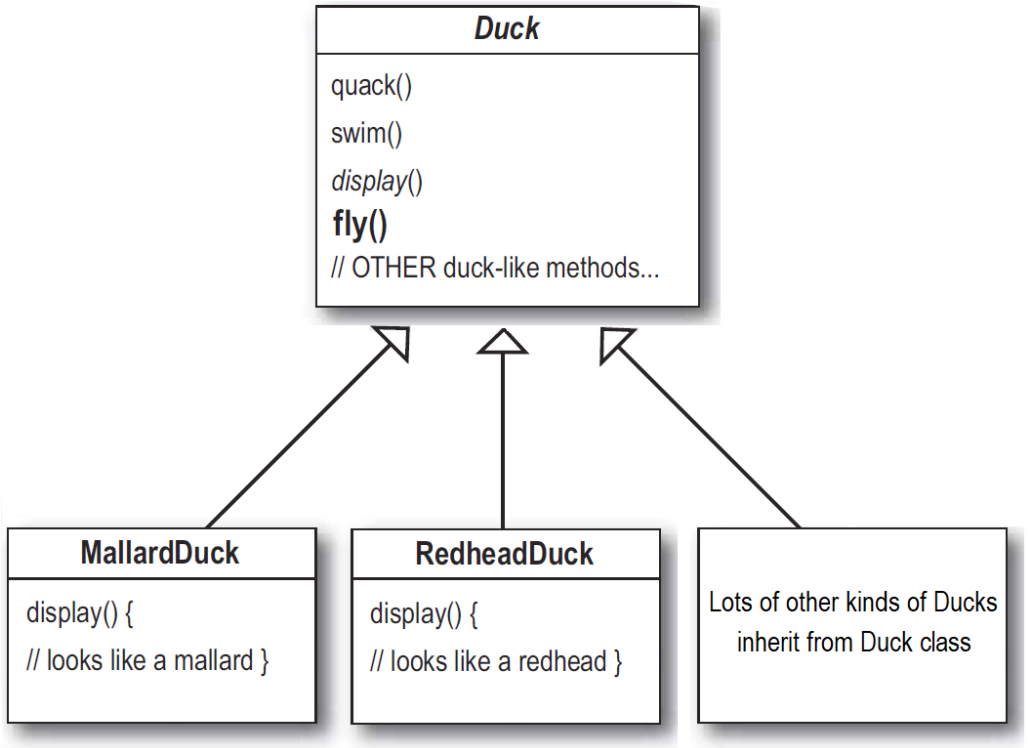


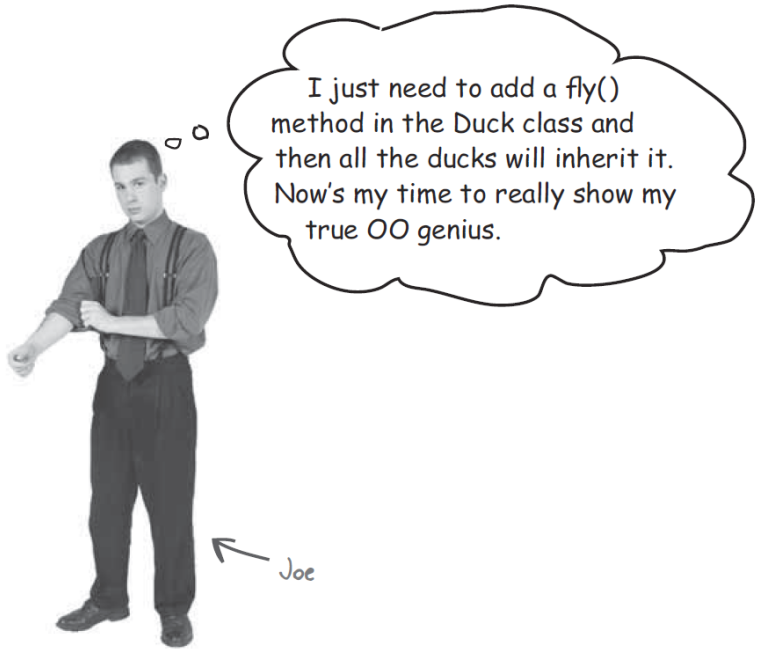
Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Improving SimUDuck App

"After a week-long off-site brainstorming session over golf, company executives thinks its time for a big innovation" [FRE].

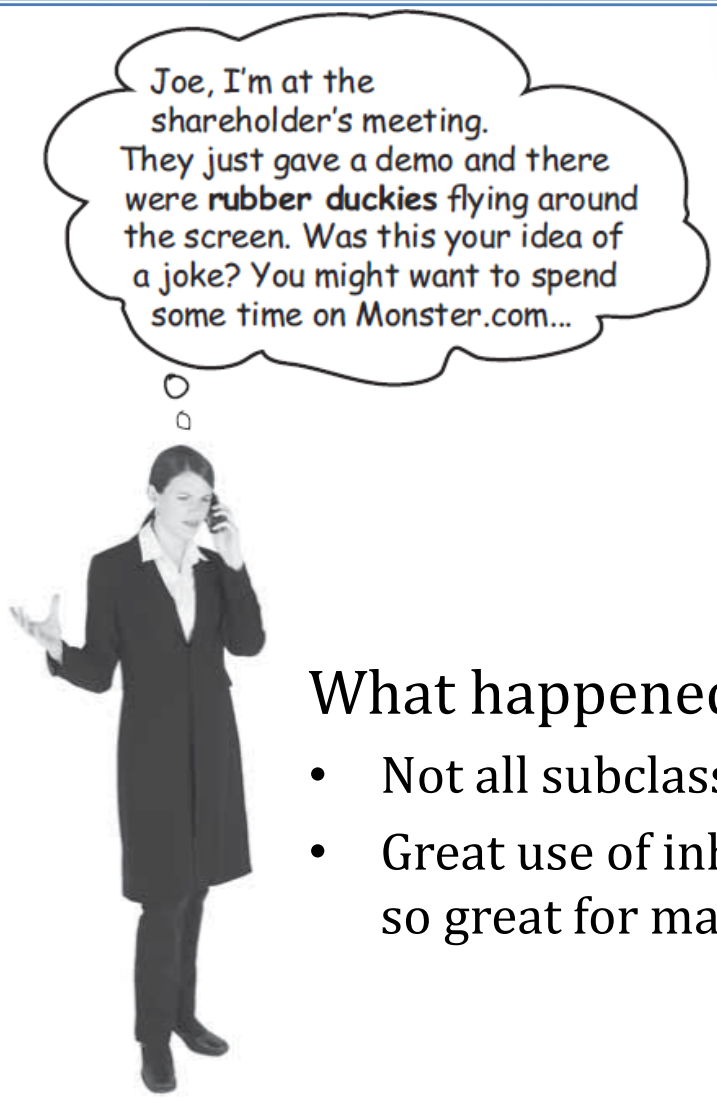
THE DUCKS NEED TO FLY!





Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Improving SimUDuck App



What happened?

- Not all subclasses of ducks should fly
- Great use of inheritance for reuse, not so great for maintenance!

Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Improving SimUDuck App

Solution?

- Override the **`fly()`** method in **`RubberDuck`**

RubberDuck

```
quack()  { // squeak}
display() { .// rubber duck }
fly() {
  // override to do nothing
}
```

How about a Wooden Decoy Duck?

DecoyDuck

```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Challenge Yourself!

Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

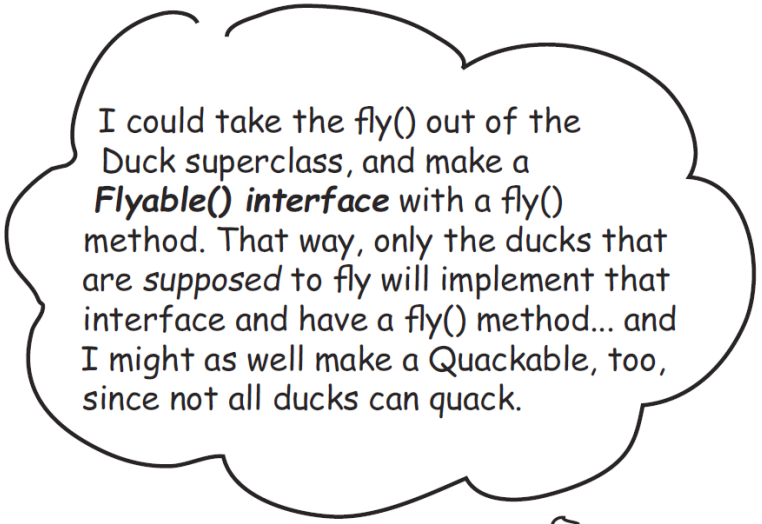
- [x] A. Code is duplicated across subclasses.
- [x] B. Runtime behavior changes are difficult.
- [ ] C. We can't make ducks dance.
- [x] D. Hard to gain knowledge of all duck behaviors.
- [ ] E. Ducks can't fly and quack at the same time.
- [x] F. Changes can unintentionally affect other ducks.

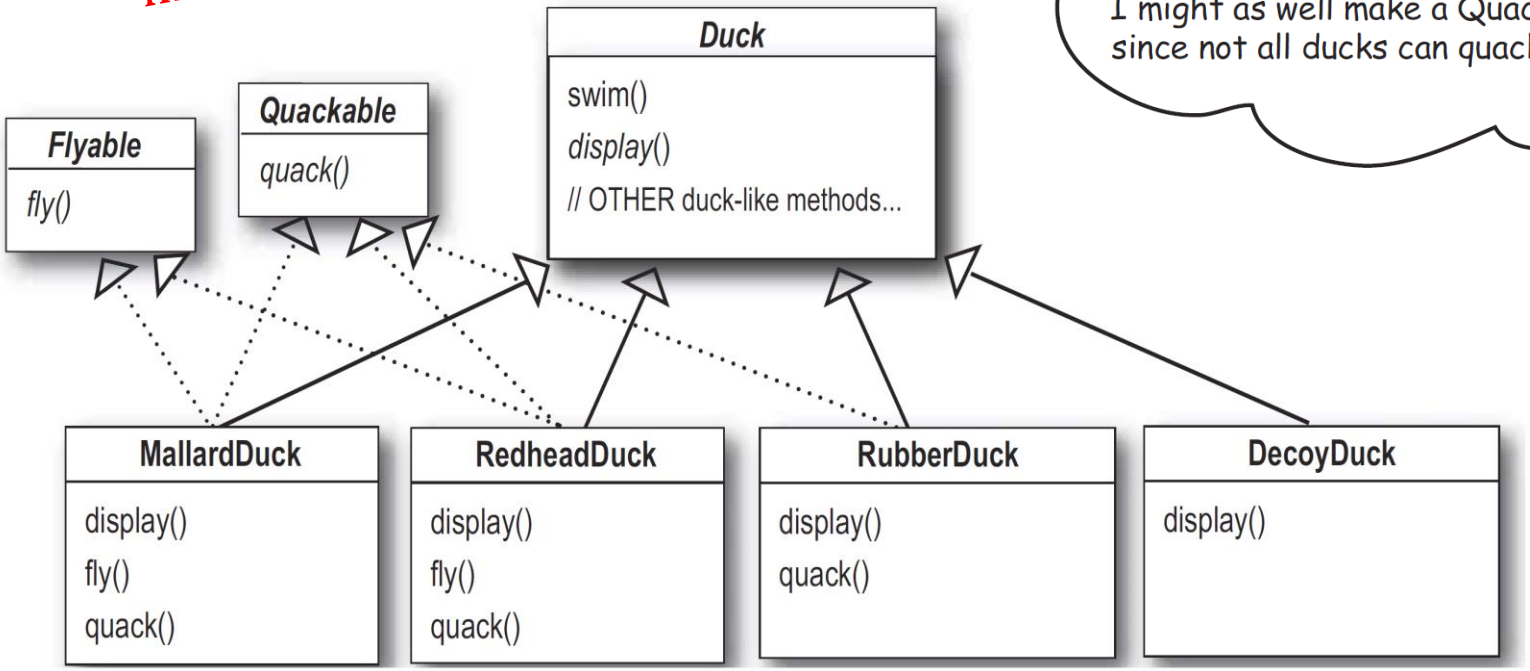
Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Interfaces

Problems?

- Duplicate code
- Changes in behaviour may require modification in all subclasses





Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Identifying the Problem

- Inheritance didn't work well
  - Duck behaviour keeps changing across subclasses
  - Not appropriate to have those behaviours for all subclasses
- Interfaces didn't work well
  - Code duplication in subclasses
  - Behaviour modification is difficult.

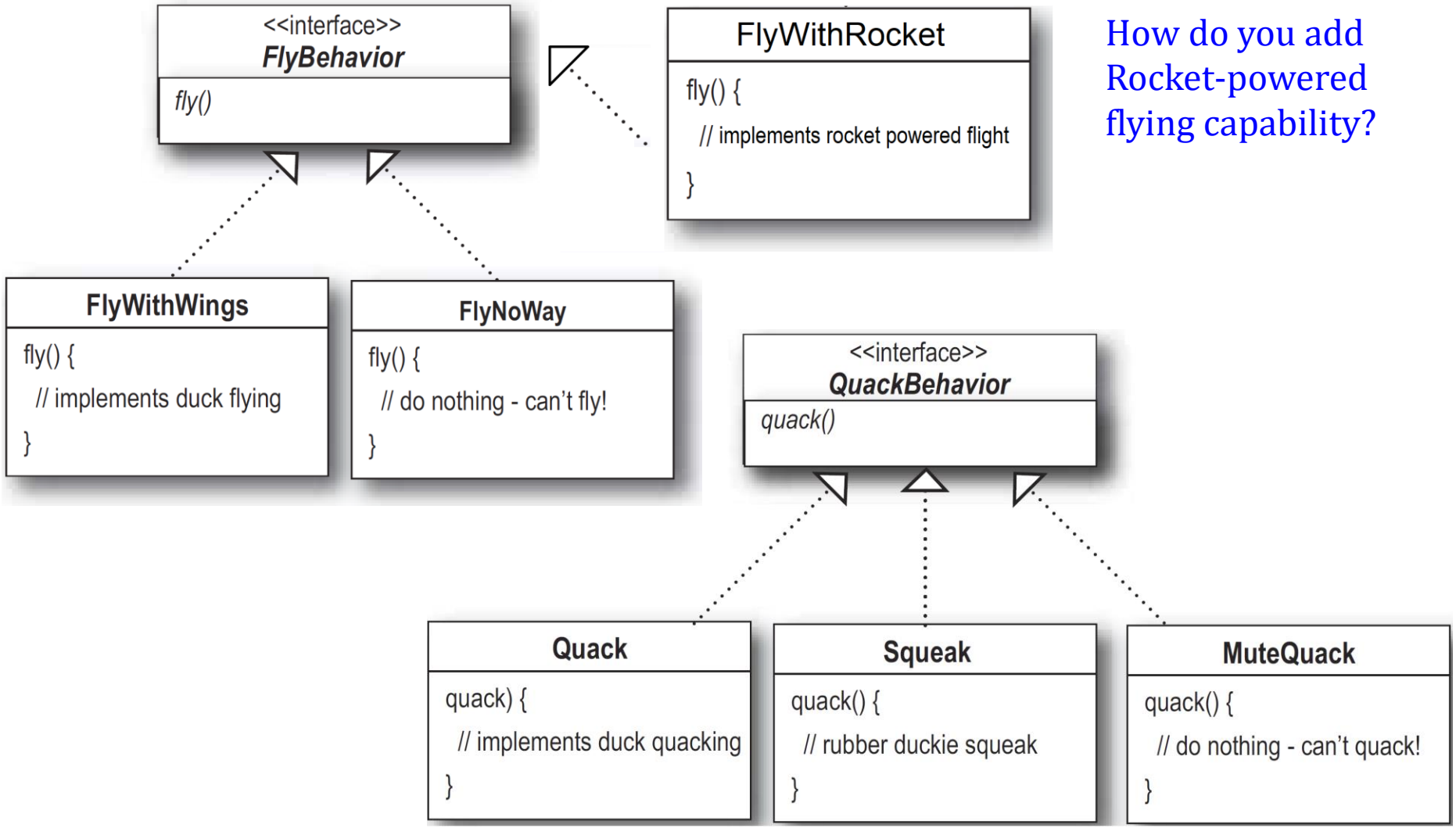
Design Principles:

1. "Take the parts that vary and '**encapsulate**' them"
2. "Program to an **interface**, not an implementation"

Declared variable of supertype

Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Improving SimUDuck



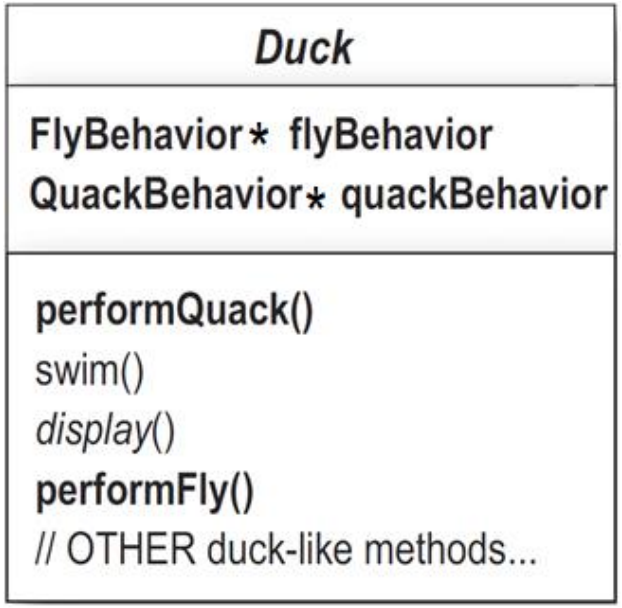
How do you add Rocket-powered flying capability?

Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Integrating the Duck Behaviours

```
class Duck{
protected:
    QuackBehaviour* quackBehaviour = nullptr;
    FlyBehaviour* flyBehaviour = nullptr;
    // other code

public:
    void performQuack(){
        quackBehaviour->quack();
    }
    void performFly(){
        flyBehaviour->fly();
    }
```



Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Setting Ducks' Behaviours

```
class MallardDuck : public Duck{
public:
    MallardDuck(){
        quackBehaviour = new Quack();
        flyBehaviour = new FlyWithWings();
    }
};

class RubberDuck : public Duck{
public:
    RubberDuck(){
        quackBehaviour = new MuteQuack();
        flyBehaviour = new FlyNoWay();
    }
};
```

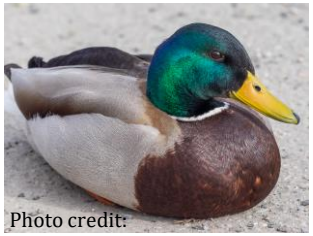


Photo credit:
Bengt Nyman, Wikipedia

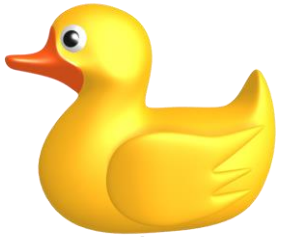


Image: iconbug.com, CC

Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

Testing SimUDuck

```
int main(){

    Duck* duck = new MallardDuck();
    duck->display();
    duck->performQuack();
    duck->performFly();

    duck = new RubberDuck();
    duck->display();
    duck->performQuack();
    duck->performFly();

    delete duck;
}
```

```
Mallard Duck
   I really quack!
   I really fly!

Rubber Duck
   ..............!
   What is fly?
```

Improving Further

- Shouldn't member variables be private?
- Can we change Ducks' behaviours at runtime?

```
class Duck{
    QuackBehaviour* quackBehaviour = nullptr;
    FlyBehaviour* flyBehaviour = nullptr;
    // other code

public:
    Duck(){}
    void setFlyBehaviour(FlyBehaviour* f) { flyBehaviour = f; }
    void setQuackBehaviour(QuackBehaviour* q) { quackBehaviour = q; }
    //other code
};
```

Changing Behaviour at Runtime

```
int main(){

    Duck* duck = new RubberDuck();
    duck->display();
    duck->performQuack();
    duck->performFly();
    duck->setQuackBehaviour(new Squeak());
    duck->setFlyBehaviour(new FlyWithRocket());
    duck->performQuack();
    duck->performFly();

    delete duck;
}
```

```
Rubber Duck
   ............!
   What is fly?
   Squeak...Squeak!
   Rocket Power ... Yay!
```

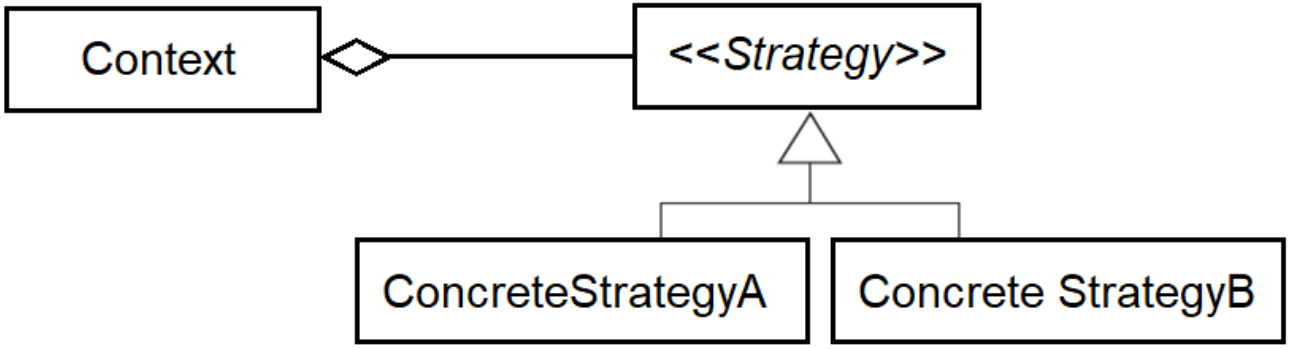
Strategy Pattern

- ***Context***:
  - Often in a domain model, you find a set of classes that differ only in their behaviour or in the algorithms they use.
  - The members of such a set share common information but could implement different behaviours or algorithms.
- ***Problem***:
  - What is the best way to define a family of algorithms (or behaviours), encapsulate each one and make them interchangeable?
- ***Forces***:
  - You want to avoid the client from being exposed to complex, algorithm-specific data structures by separating the algorithms (or behaviours) from the client.

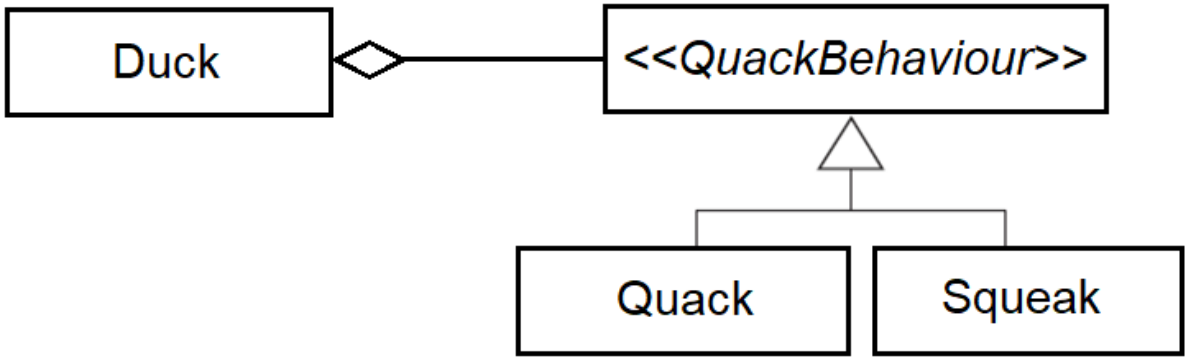
Ref: [GOF]

Strategy Pattern

- Solution



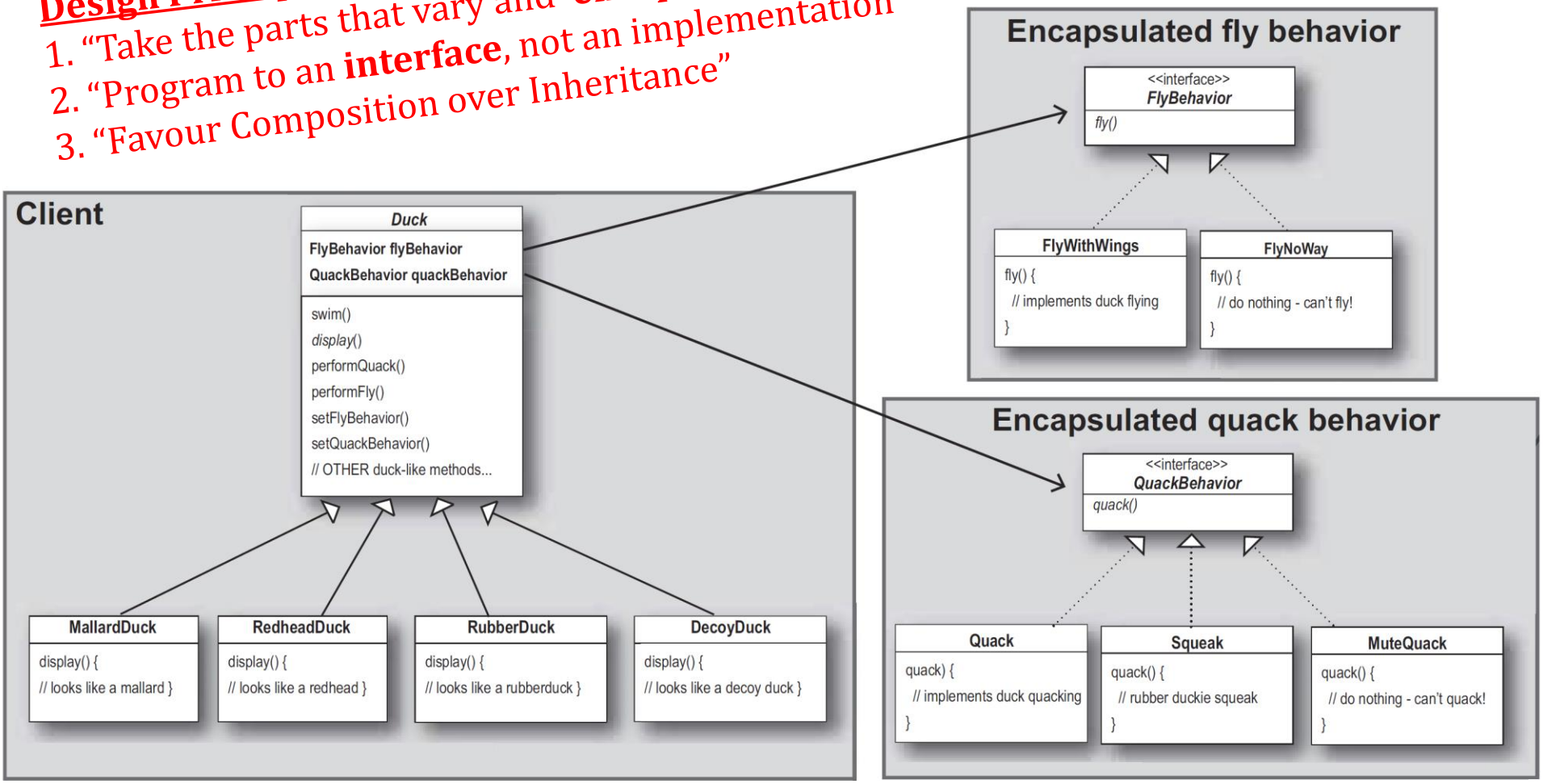
- Example



Strategy Pattern in SimUDuck

Design Principles

1. "Take the parts that vary and '**encapsulate**' them"
2. "Program to an **interface**, not an implementation"
3. "Favour Composition over Inheritance"



Reference: *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

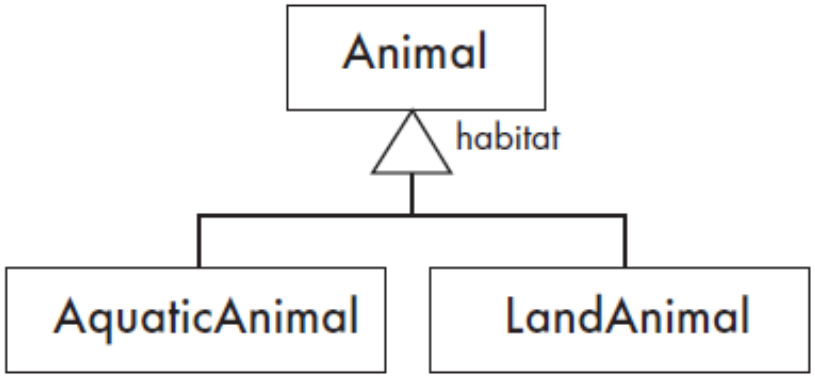
The Player-Role Pattern

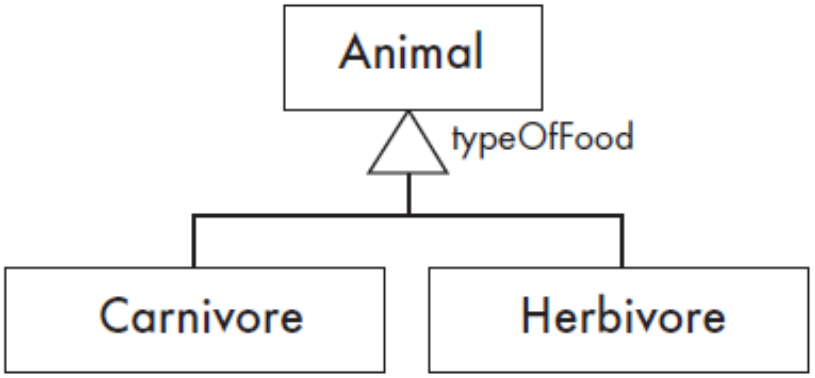
The Player-Role Pattern

- ***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

- 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.
- Consider the following scenario
  - more than one possible generalization set sharing the same superclass

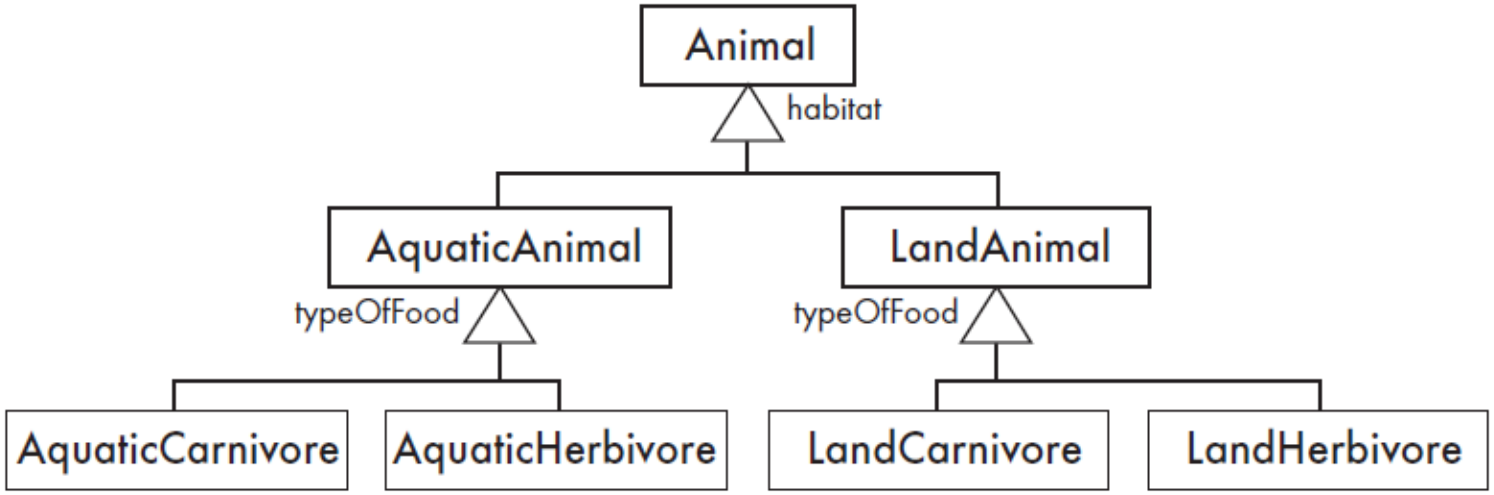




Ref: [LET]

Player-Role – Antipatterns

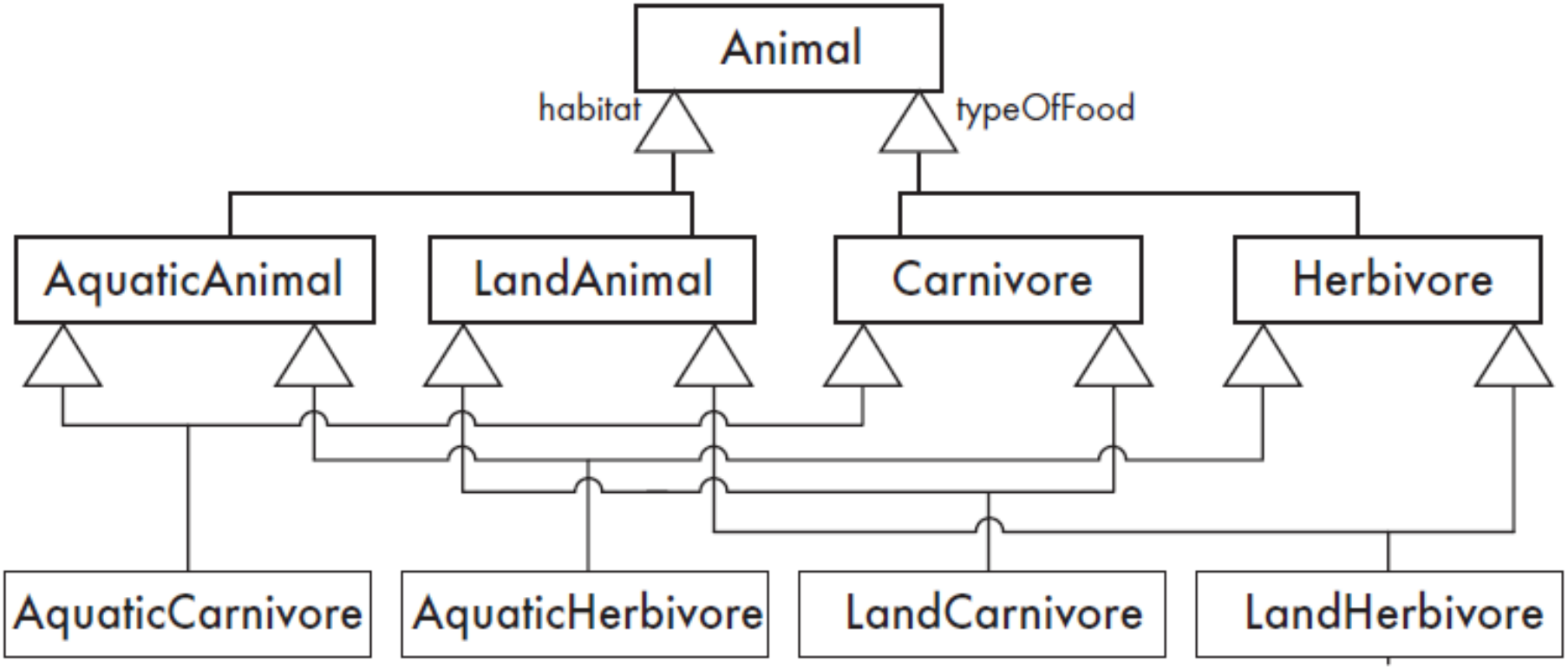
- Create a higher-level generalization set (here habitat), and then to have generalization sets with duplicate labels at a lower level in the hierarchy
  - all the features associated with the second generalization set would also have to be duplicated
  - the number of classes can grow very large.
- What if you wanted to add Omnivores?



Ref: [LET]

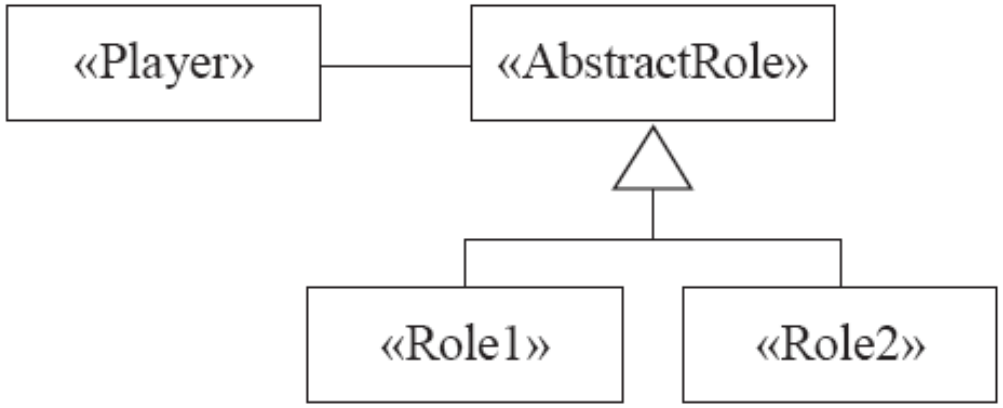
Player-Role – Antipatterns

- Another solution is multiple hierarchy
  - Complex with a large number of classes
  - Cannot be implemented in certain programming languages



Player-Role

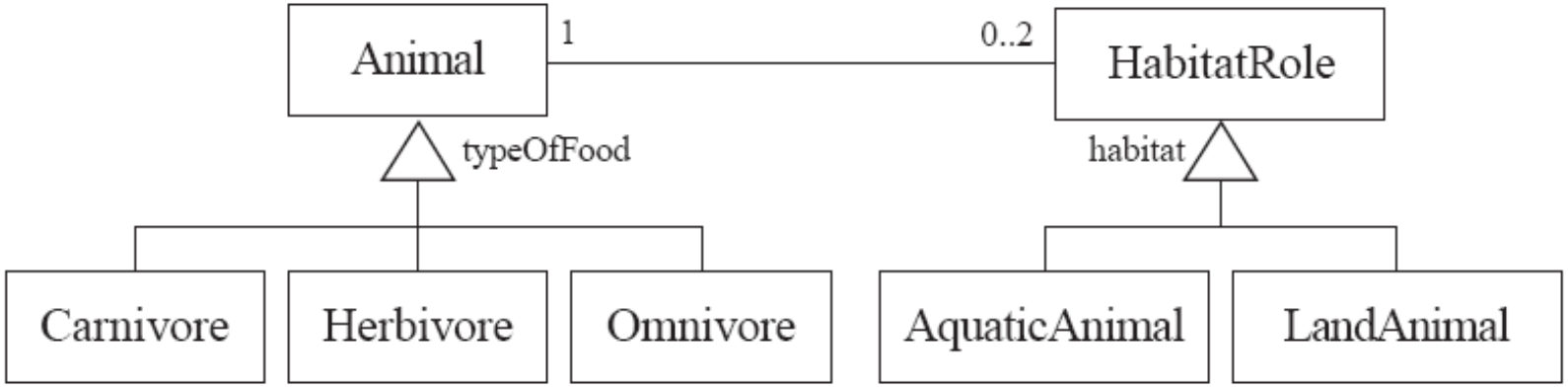
– ***Solution**:*



- Create a ≪*Player*≫ class to represent the object that plays different roles.
- Create an association from this class to an abstract ≪*Role*≫ class, which is the superclass of a set of possible roles.
- The subclasses of this ≪Role≫ class encapsulate all the features associated with the different roles.

Player-Role

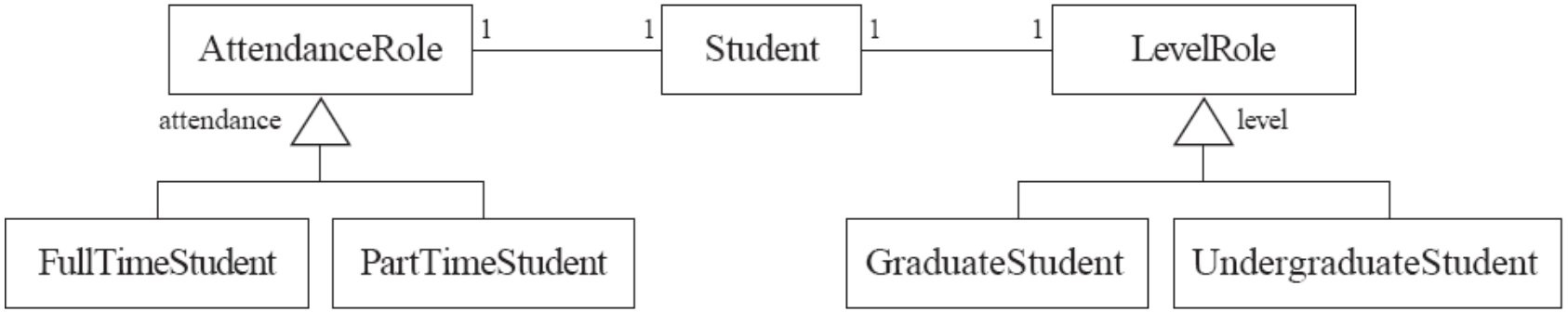
- Example 1:



Ref: [LET]

Player-Role

- Example 2:



Ref: [LET]

Player-Role Exercise

- An organization has three categories of employee: professional staff, technical staff and support staff. The organization also has departments and divisions. Each employee belongs to either a department or a division
  - Assume that people will never need to change from one category to another

More Player-Role Exercises

Consider how you would apply the Player–Role pattern in the following circumstances.

a) Users of a system have different privileges.

b) Managers can be given one or more responsibilities.

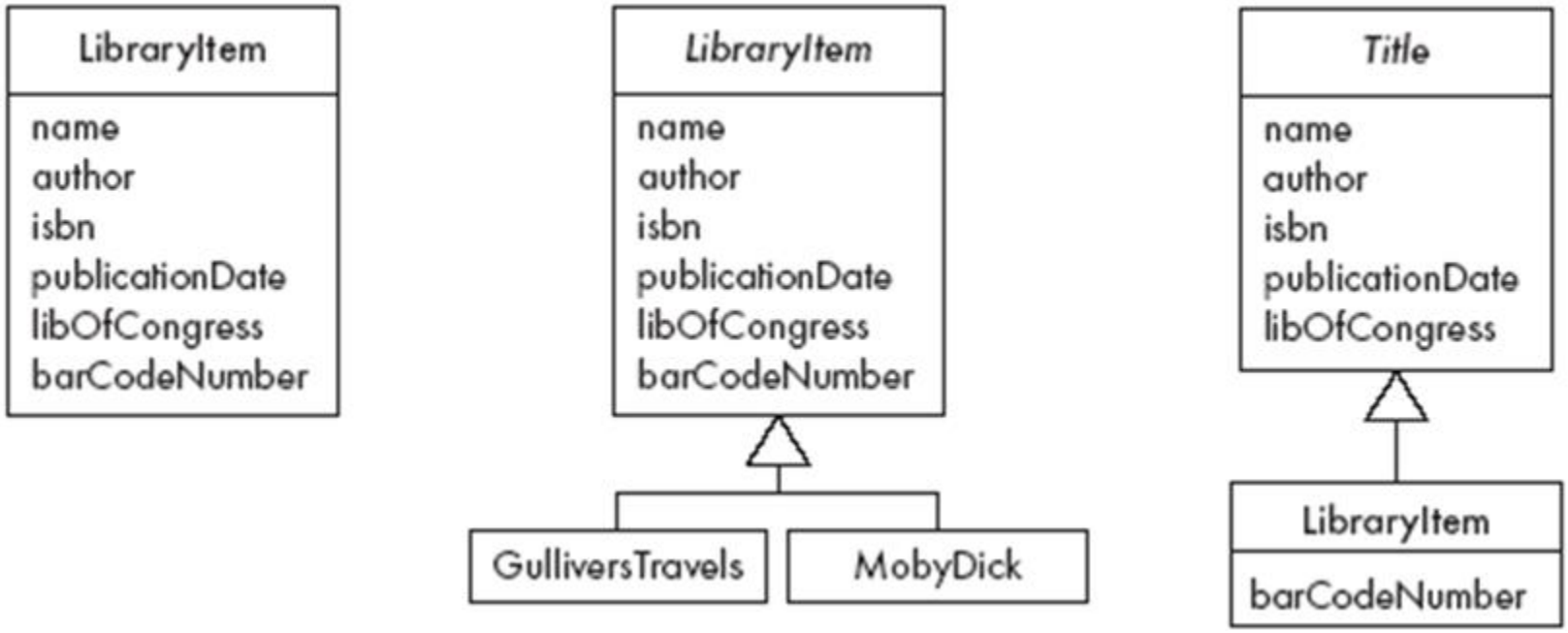
c) Your favorite sport (football, baseball, etc.) in which players can play at different positions at different times in a game or in different games.

The Abstraction-Occurrence Pattern

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

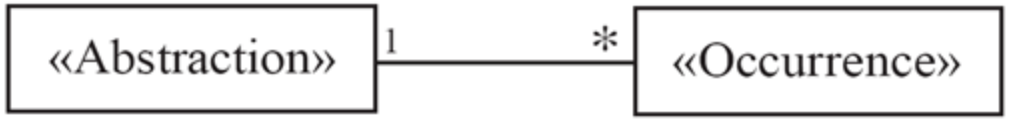


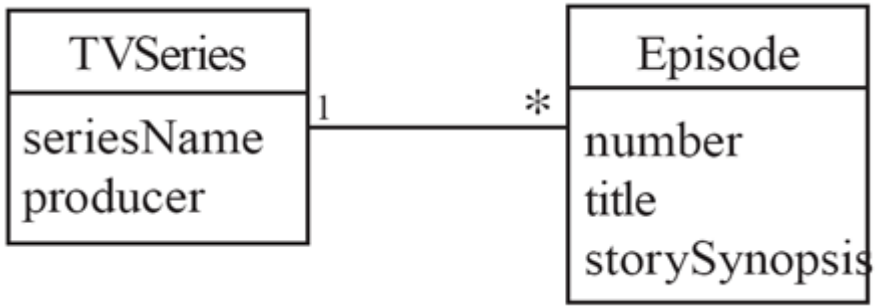
- Antipatterns:
  - Above examples are antipatterns, i.e. how not to do something.

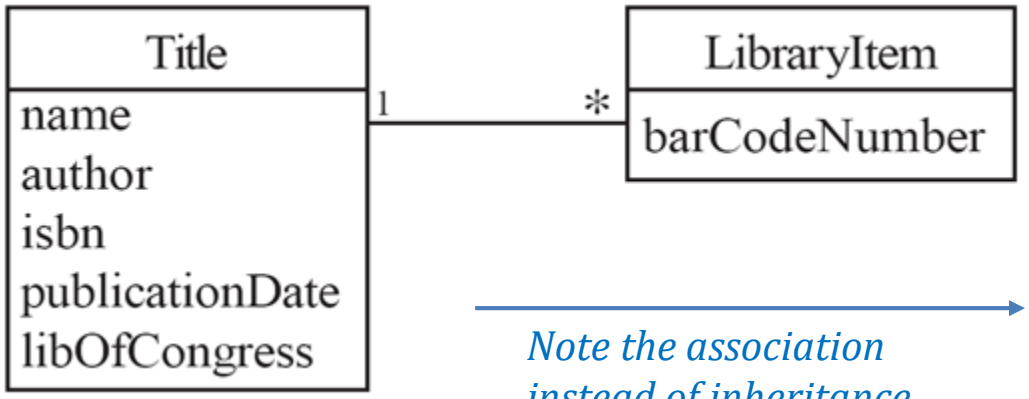
Ref: [LET]

Abstraction-Occurrence

– ***Solution:***





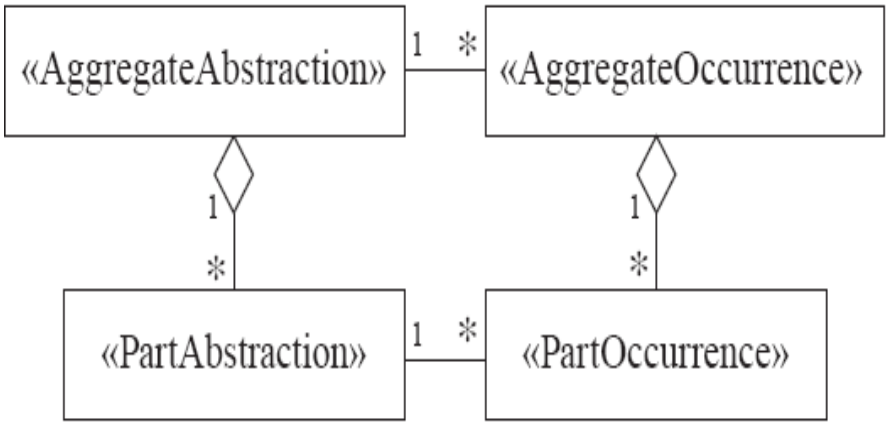


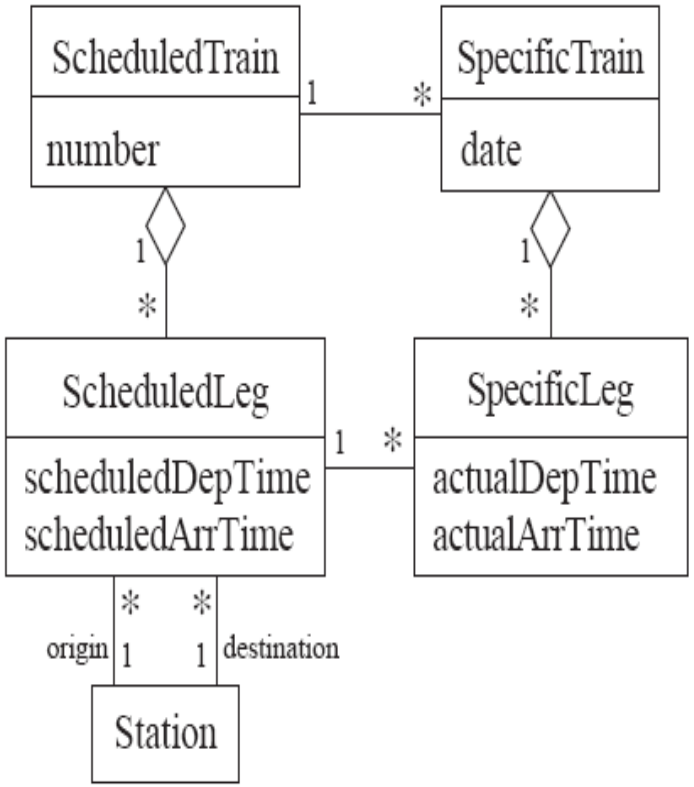
Note the association instead of inheritance

Ref: [LET]

Abstraction-Occurrence

- Square variant

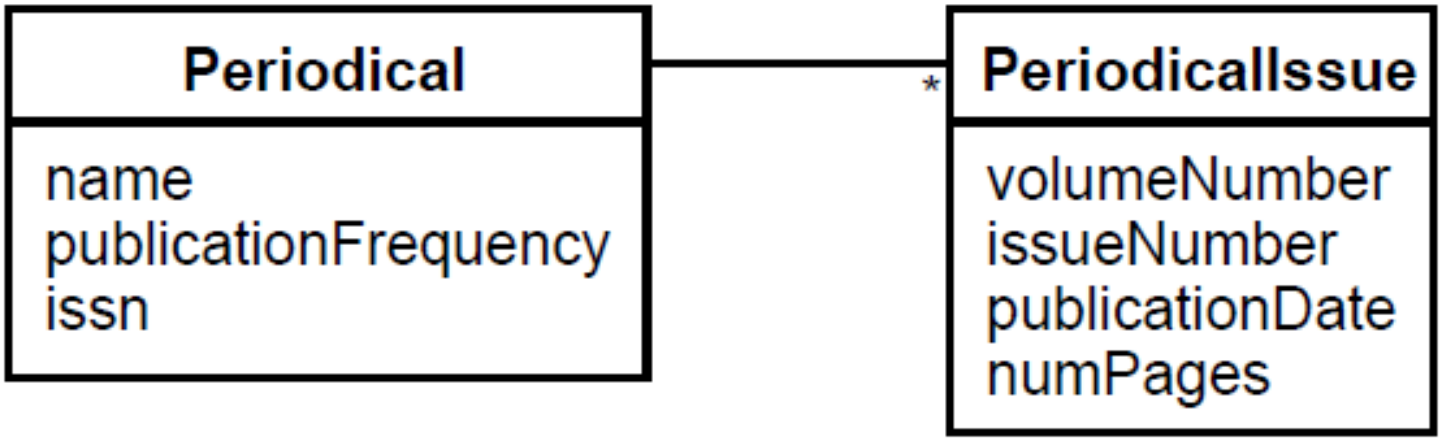




Ref: [LET]

Abstraction – Occurrence Exercise

- Apply the Abstraction–Occurrence pattern for the issues of a periodical. Show the two linked classes, the association between the classes, and the attributes in each class.



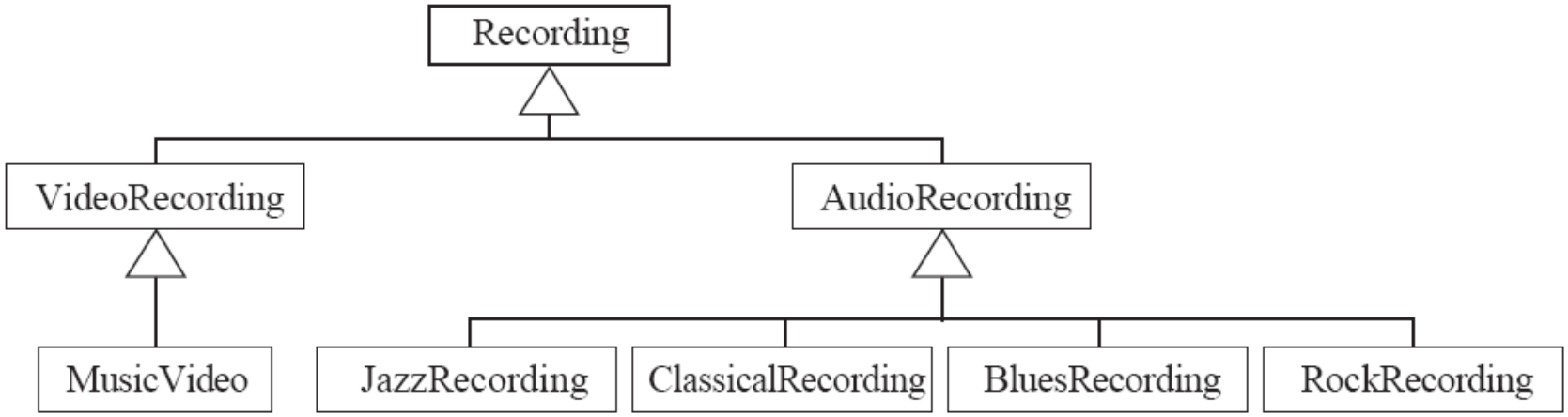
The General Hierarchy Pattern

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

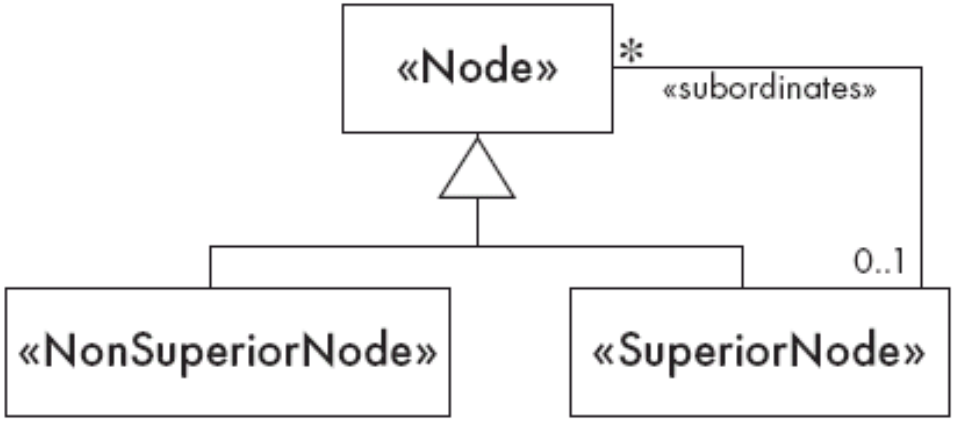
General Hierarchy

- Antipattern:



General Hierarchy

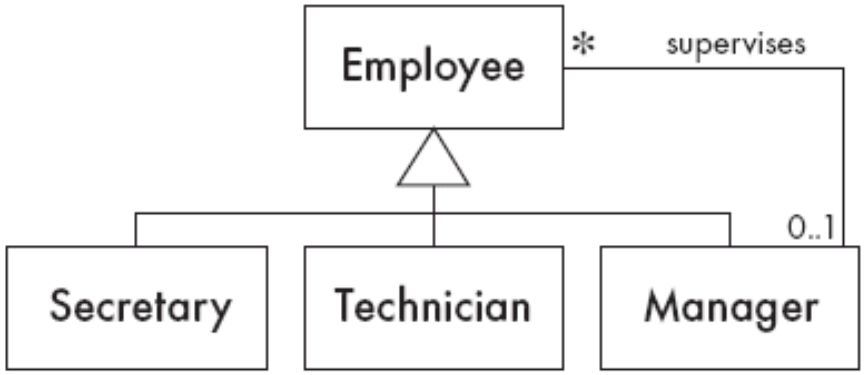
– ***Solution:***

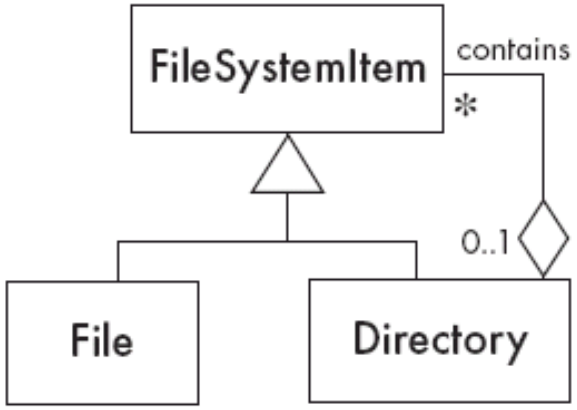


Ref: [LET]

General Hierarchy

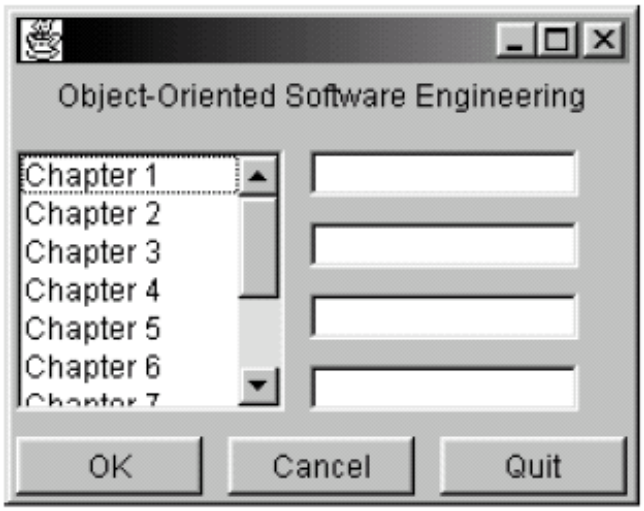
– ***Solution:***

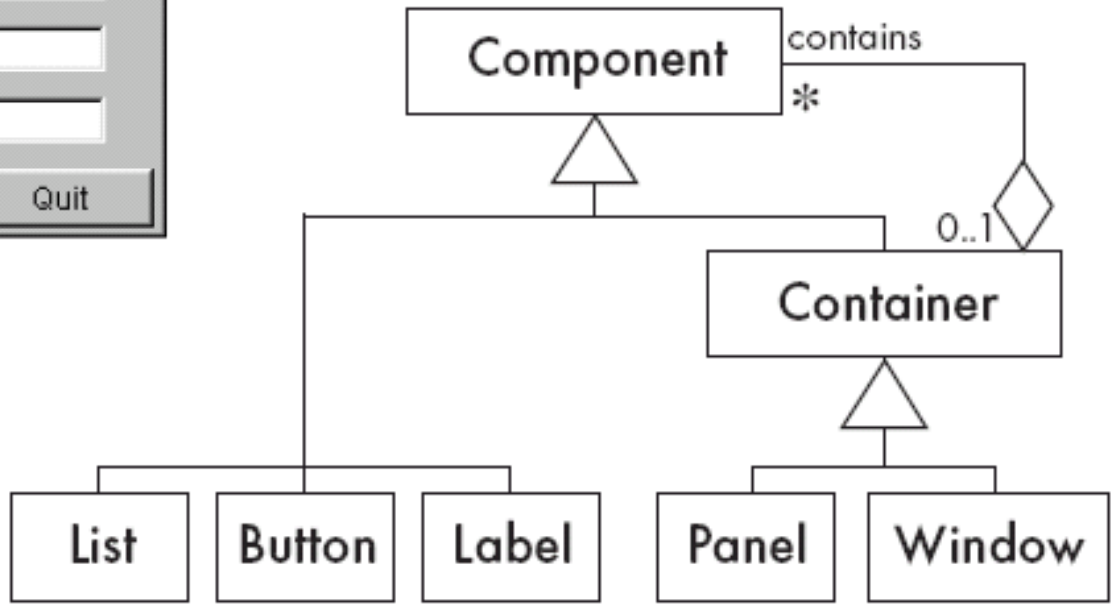




Ref: [LET]

General Hierarchy – Example

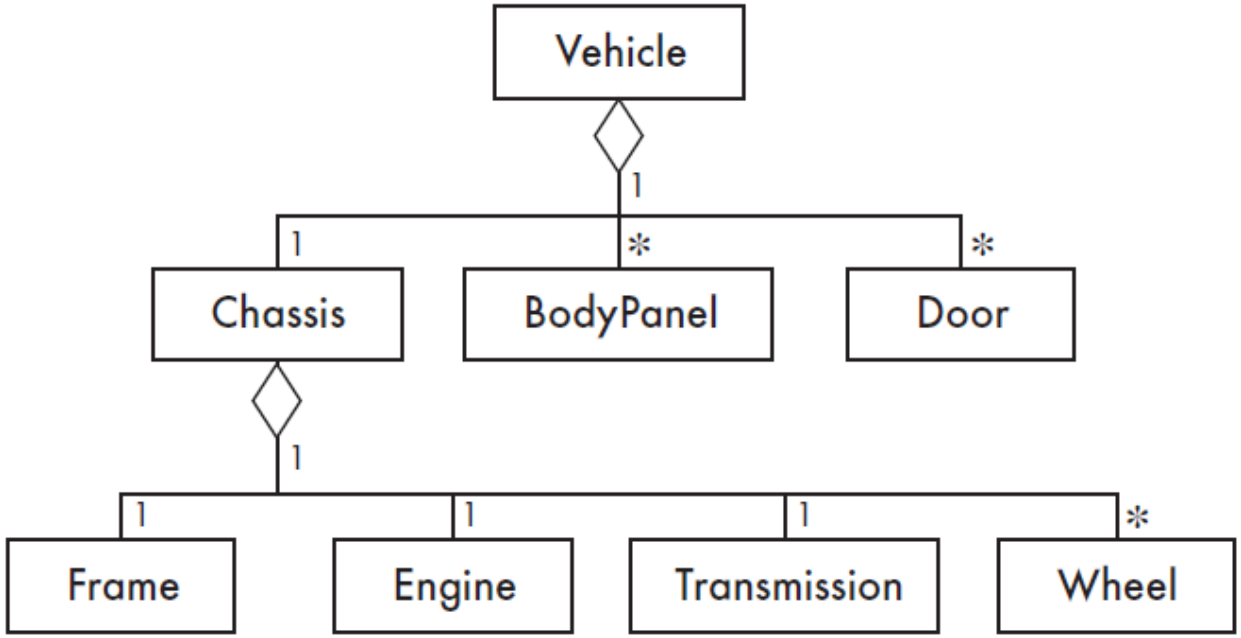




Ref: [LET]

General Hierarchy – Example

- Figure below shows a hierarchy of vehicle parts. Show how this hierarchy might be better represented using the General Hierarchy pattern



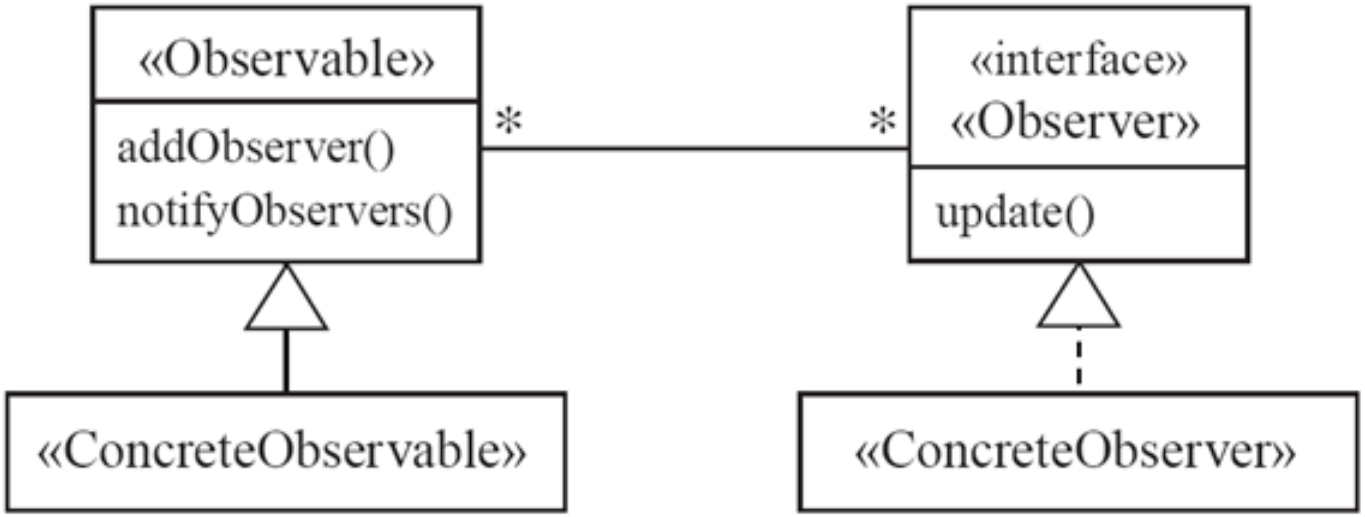
The Observer Pattern

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

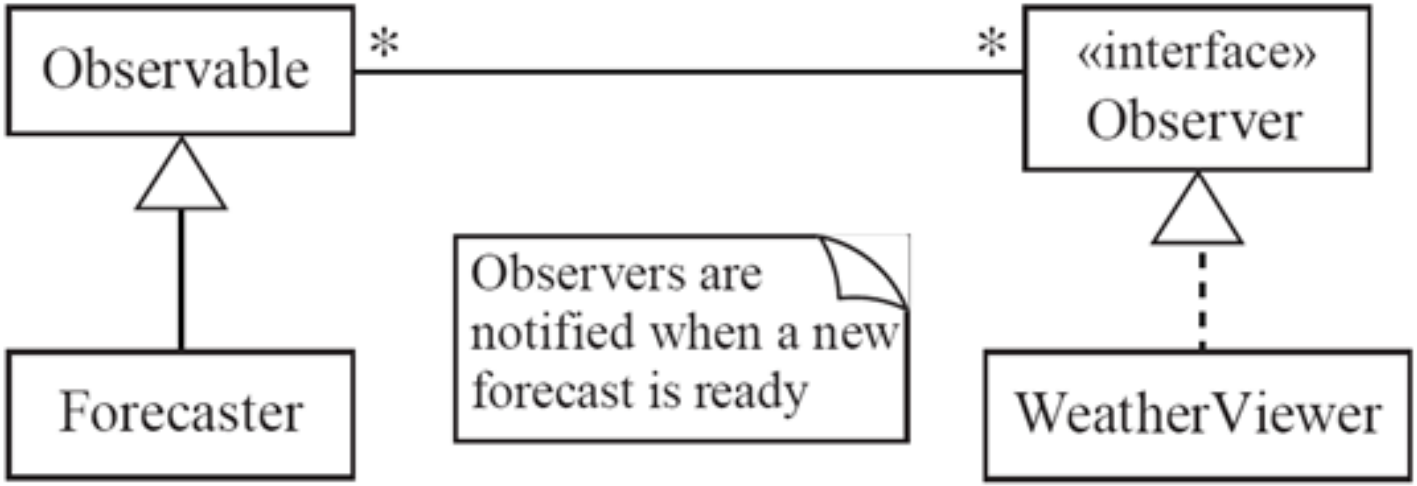
Observer

– ***Solution:***



Ref: [LET]

Observer – Example



Ref: [LET]

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.

Other Patterns

(Not Included in Exam but Included in Slides for Your Reference)

Delegation

Adapter

Façade

Immutable

Read-only

Proxy

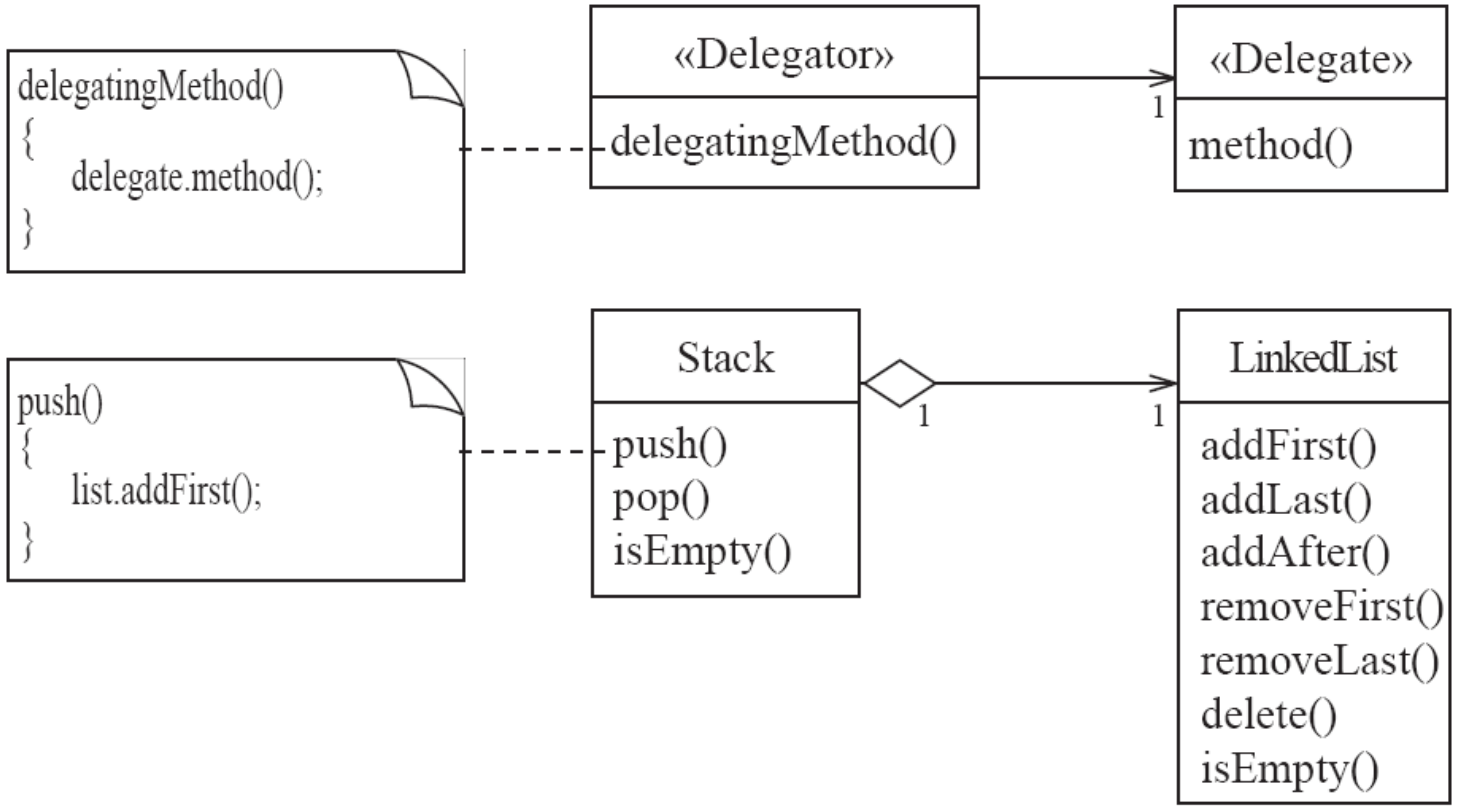
Factory

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 is-a 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

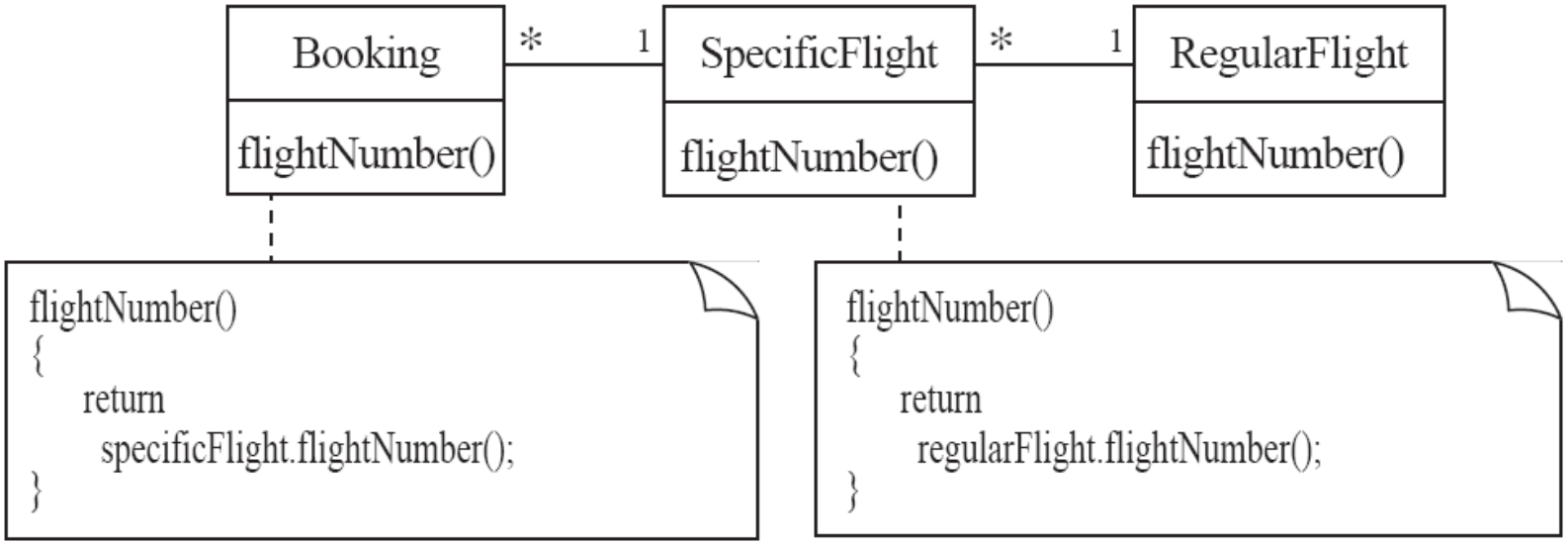
– ***Solution:***



Ref: [LET]

Delegation

Example:



Ref: [LET]

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
    - consider having many different methods in the «Delegator» call the delegate's method
  - Access non-neighboring classes

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

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

The Law of Demeter

A fundamental principle of the Delegation pattern is that a method should only communicate with objects that are neighbors in the class diagram. This is a special case of the 'Law of Demeter', which was formulated by a team from Northeastern University in Boston.

The Law of Demeter says, in short, '**only talk to your immediate friends**'. In software design, this means that a method should only access data passed as arguments, linked via associations, or obtained via calls to operations on other neighboring data. The rationale is that this limits the impact of changes, and makes it easier for software designers to understand that impact. If each method only communicates with its neighbors, then it should only be impacted when those neighbors change, not when changes occur in more distant parts of the system.

Ref: [LET]

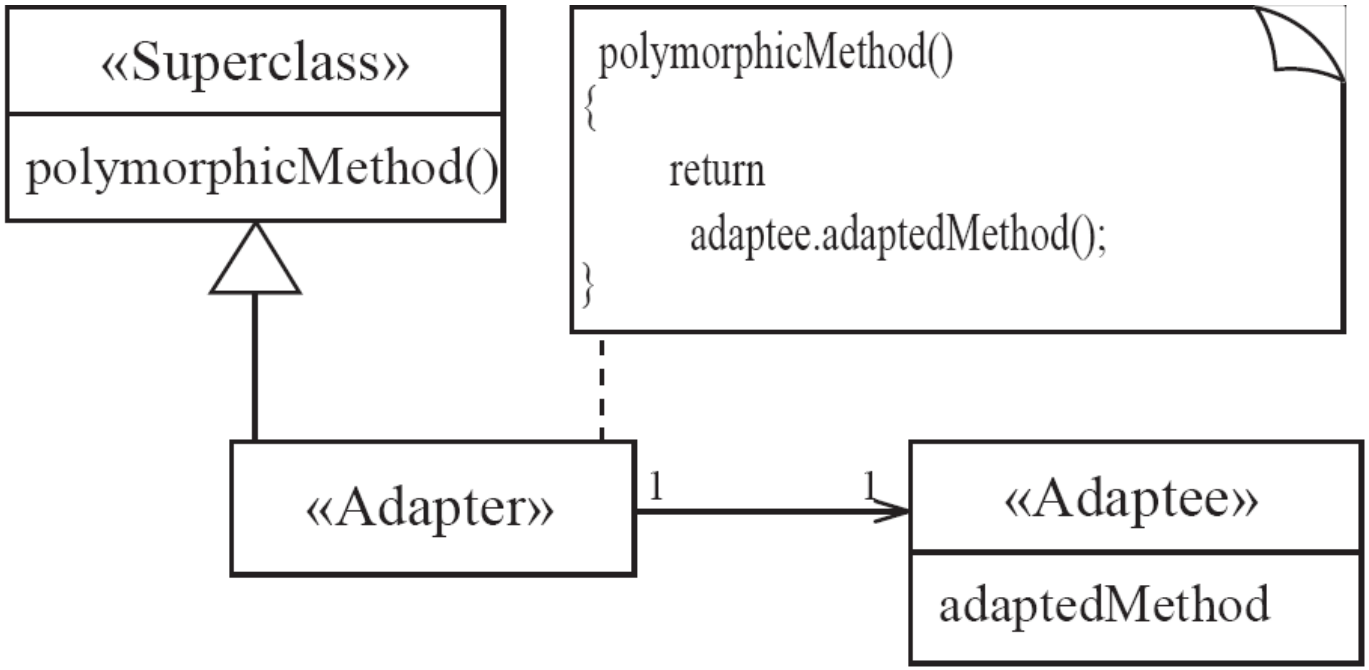
The Adapter Pattern

- ***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

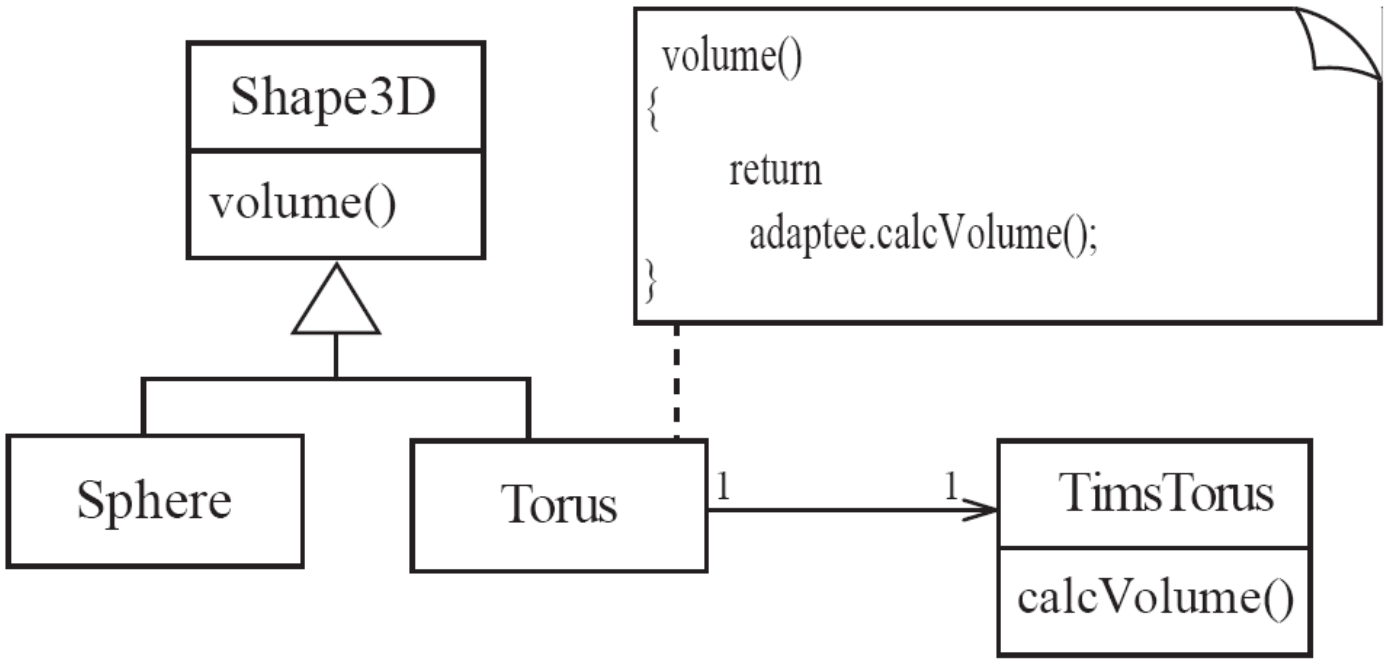
– ***Solution:***



Ref: [LET]

Adapter

- Example:



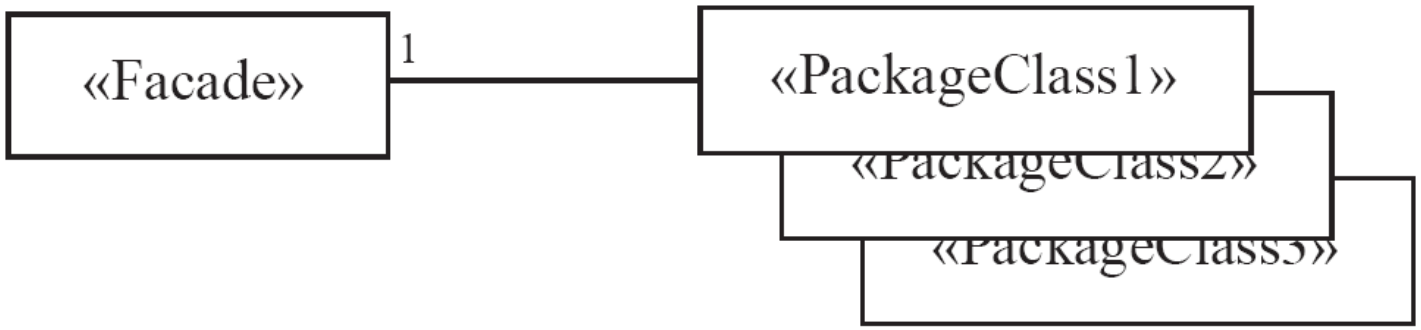
Ref: [LET]

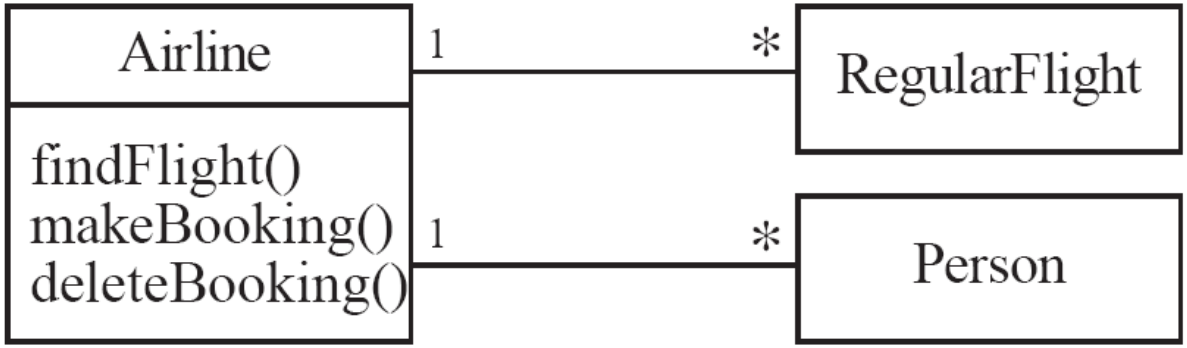
The Façade Pattern

- ***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:***





Ref: [LET]

The Immutable Pattern

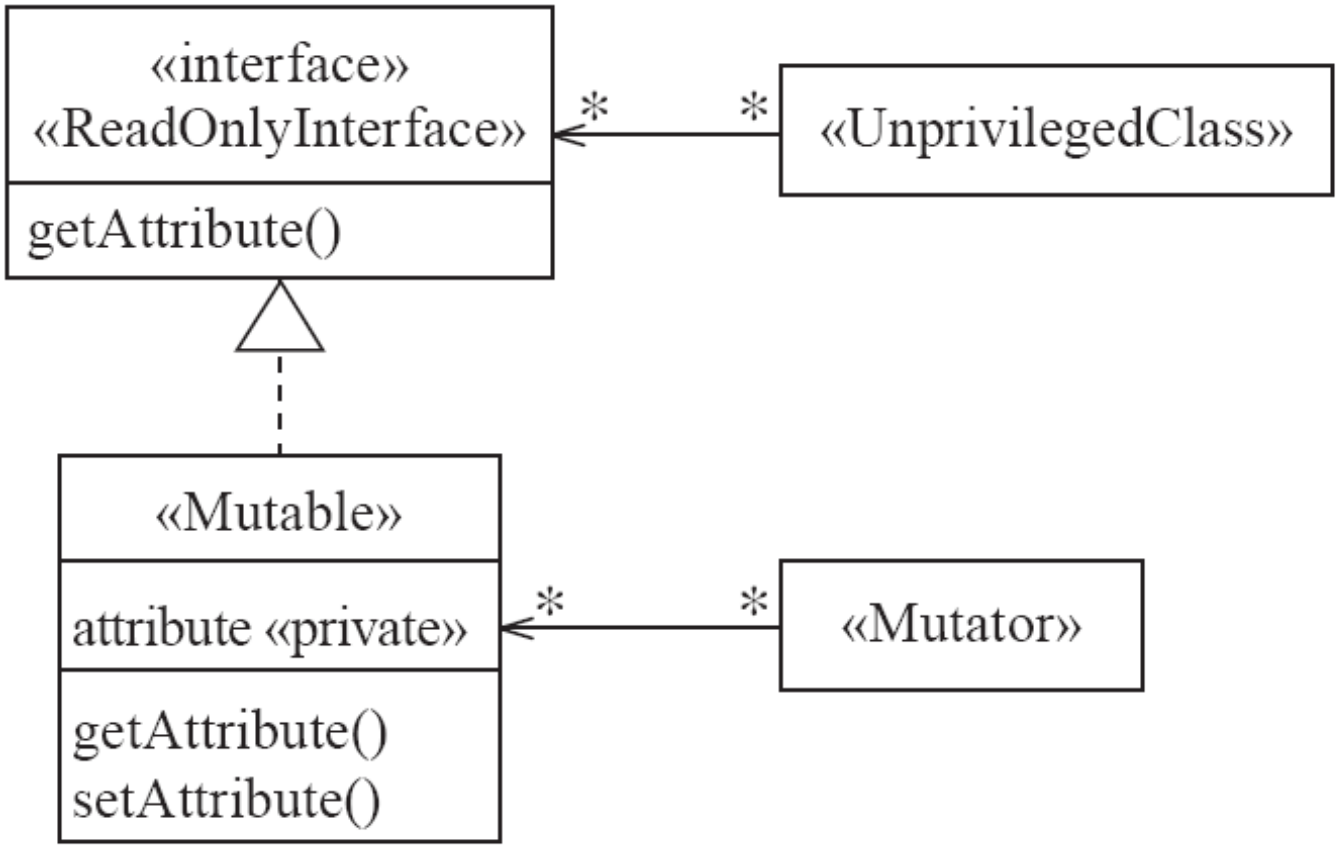
- ***Context***:
  - An immutable object is an object that has a state that never changes after creation
- ***Problem***:
  - How do you create a class whose instances are immutable?
- ***Forces***:
  - There must be no loopholes that would allow 'illegal' modification of an immutable object
- ***Solution***: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

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

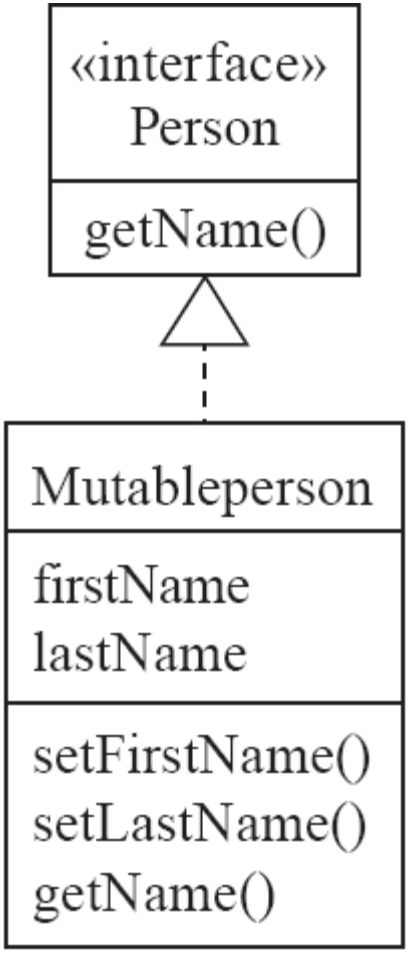
– ***Solution:***



Ref: [LET]

Read-only Interface

- Example:



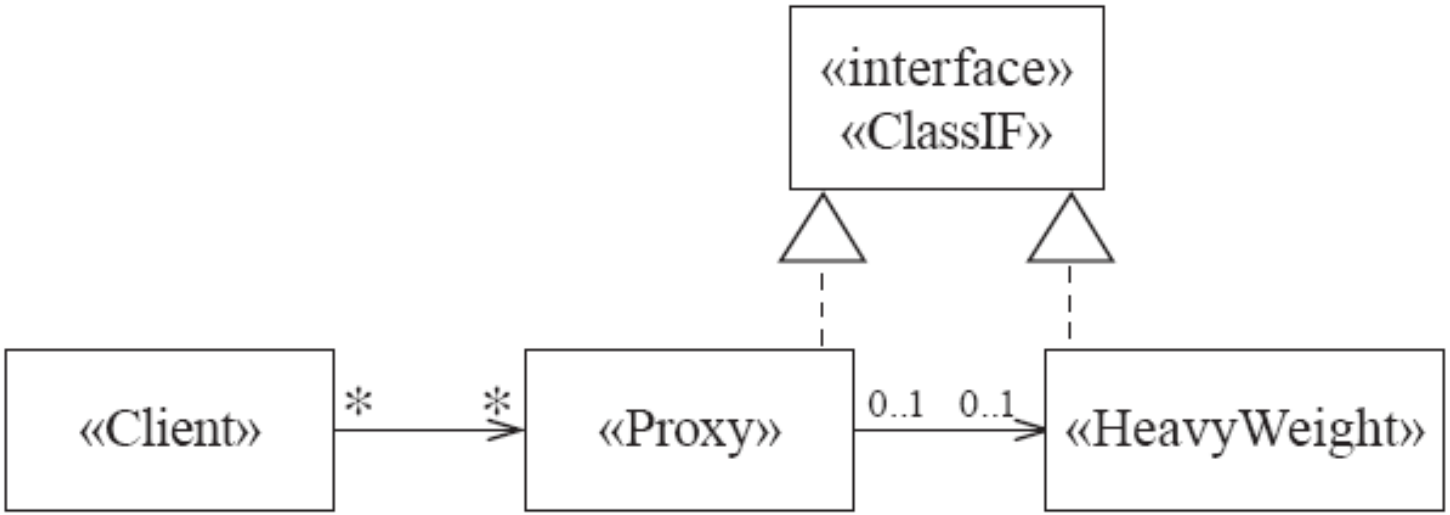
Ref: [LET]

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?
- ***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

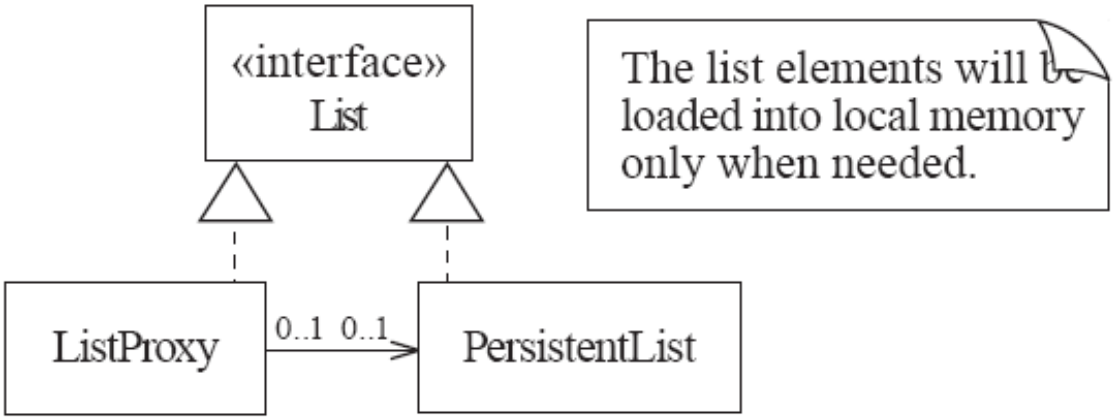
- ***Solution:***

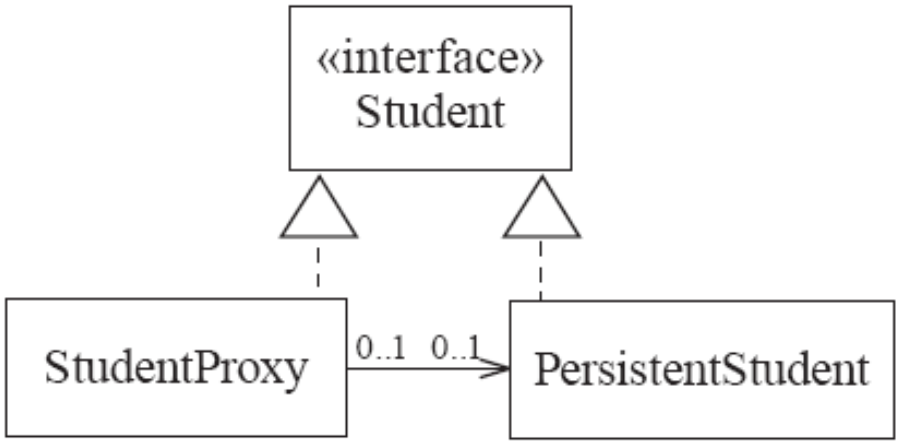


Ref: [LET]

Proxy

- Examples:





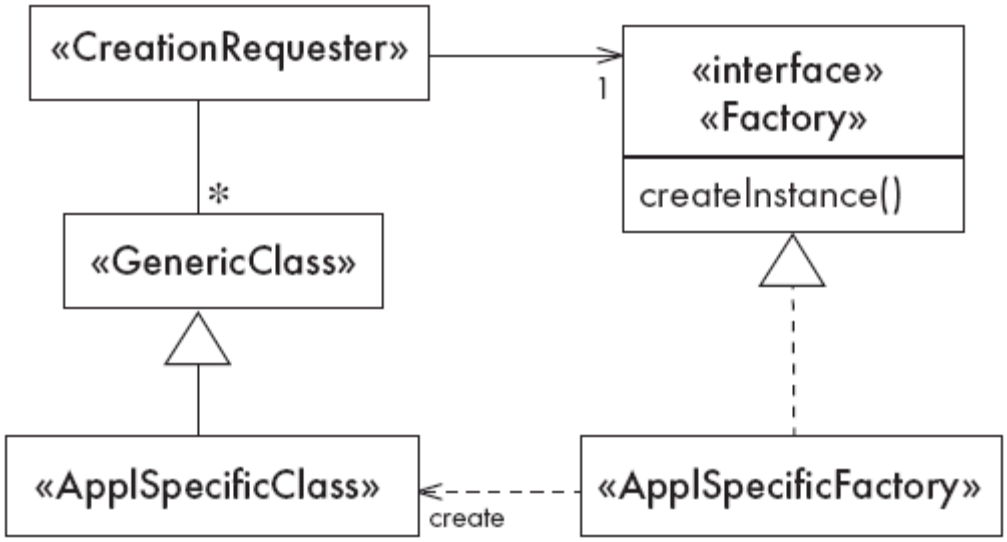
Ref: [LET]

The Factory Pattern

- ***Context***:
  - A reusable framework needs to create objects; however the class of the created objects depends 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***:
  - We 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 is 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

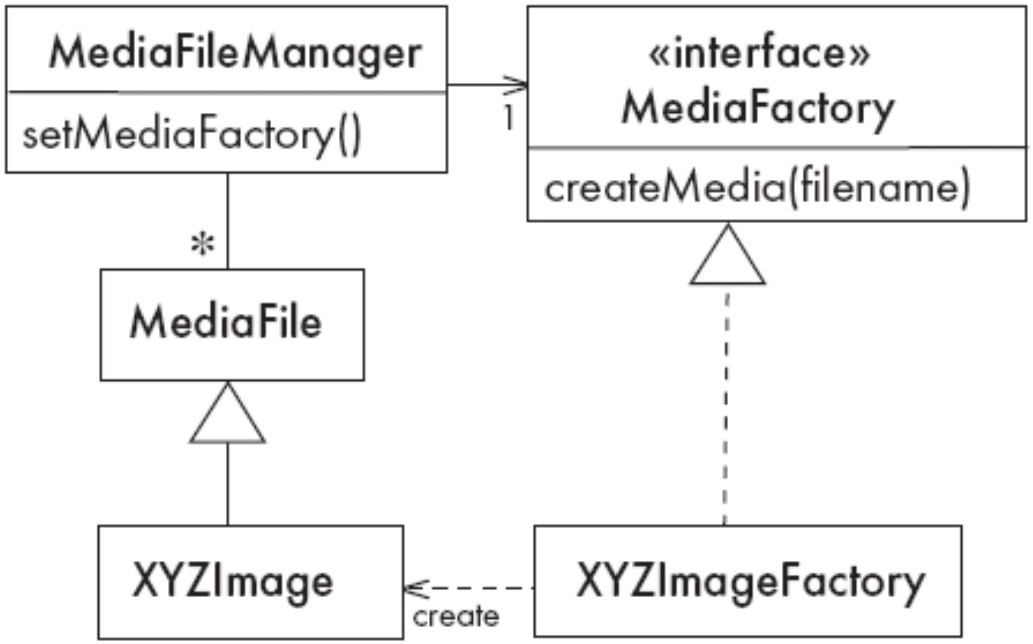
- Solution



Ref: [LET]

The Factory Pattern

- Example



Ref: [LET]

What's Wrong with Design Patterns?

- Lacks formal foundations
  - Study of design patterns has been excessively ad hoc
    - The GoF were convicted by 2/3rd of the jurors in OOPSLA 99 in a show trial for their numerous crimes against CS
  - Leads to inefficient solutions
    - Using a well-factored implementation rather than a just good enough design pattern is always more efficient
  - Targets the wrong problem
    - Need for patterns results from using computer languages or techniques with insufficient abstraction ability
      - Peter Norvig demonstrates that 16 out of the 23 patterns in the Design Patterns book are simplified or eliminated in Lisp[1]
  - Does not differ significantly from other forms of abstractions

[1] http://norvig.com/design-patterns/design-patterns.pdf

More Design Principles

- The Single Responsibility Principle (SRP)
  - "There should never be more than one reason for a class to change"
- The Open Closed Principle (OCP)
  - "A module should be open for extension, but closed for modification"
- The Liskov Substitution Principle (LSP)
  - "Subclasses should be substitutable for their base classes."
- The Dependency Inversion Principle (DIP)
  - "Depend upon Abstractions. Do not depend upon concretions."
- The Interface Segregation Principle (ISP)
  - "Many client specific interfaces are better than one general purpose interface"

Design Principles

1. "Take the parts that vary and '**encapsulate**' them"
2. "Program to an **interface**, not an implementation"
3. "Favour Composition over Inheritance"

[REF] "Design Principles and Design Patterns", Robert C. Martin, www.objectmentor.com

References

- Figures and adapted text in these slides are taken from:

  [LET] *Timothy C. Lethbridge, Robert Laganiere,* "Object Oriented Software Engineering", 2nd Edition, Chapter 6, pp. 221 – 252.

- Other sources used:

  [FRE] *Freeman et. al.,* "Head First Design Patterns", 1st Edition, 2004. (O'Reilly)

  [GOF] *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,* Design Patterns – Elements of Reusable Object-Oriented Software, 1994. (Addison-Wesley)