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

In this lecture:

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

Week 12:

- Observer - [Priestley, 2004], [GoF, 1995]
- Adapter* - [GoF, 1995]
- Façade * - [GoF, 1995], [Lethbridge, 2002]
- State * - [Priestley, 2004], [GoF, 1995]

* If time permits

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

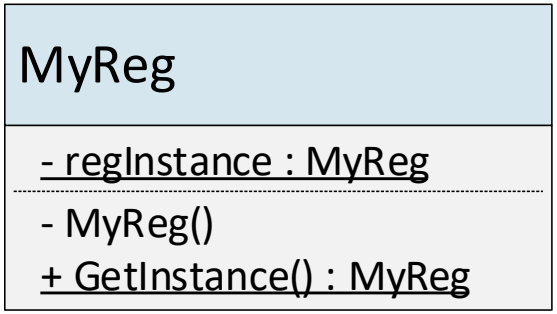
Solution:

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?

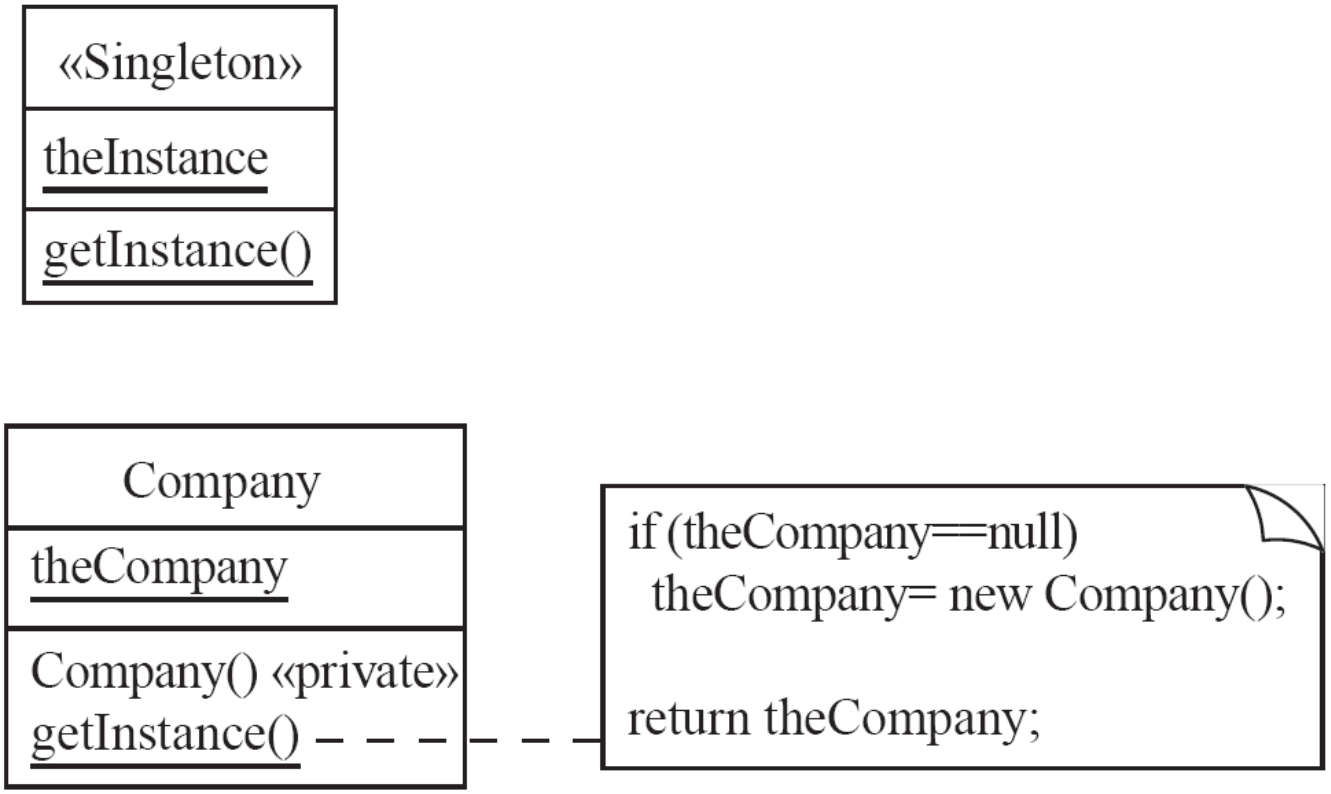
Q: Why use a Singleton pattern instead of assigning an object to a global variable?

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]

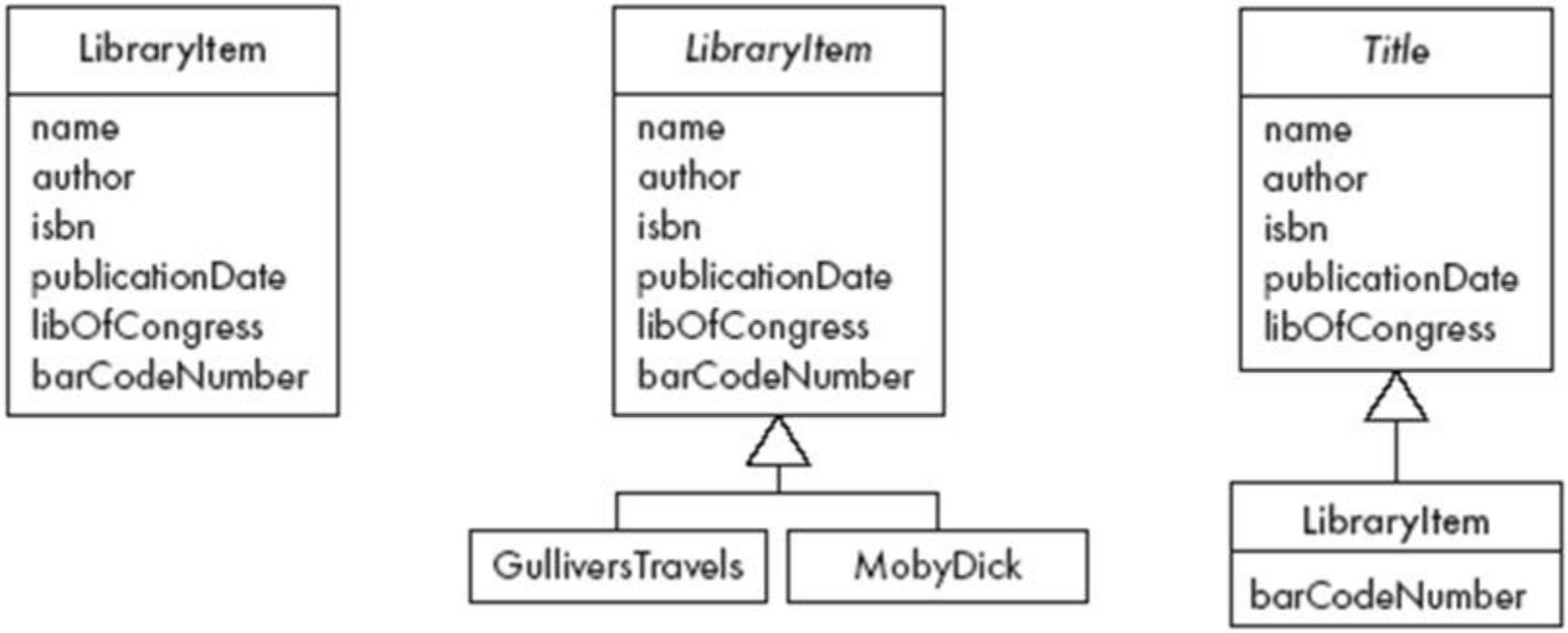
Singleton – Exercise

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

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

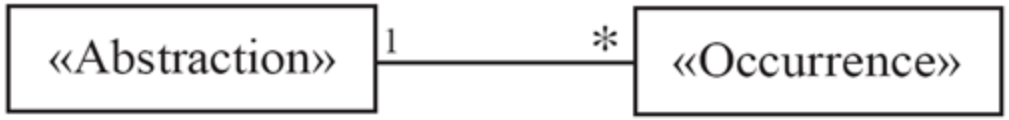


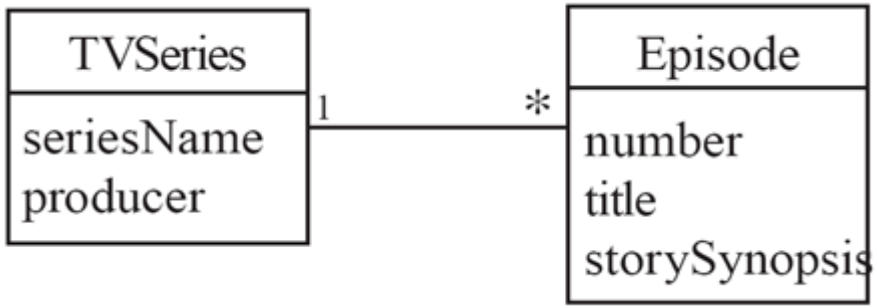
- Antipatterns:
  - Above examples are antipatterns, i.e. how not to do something.

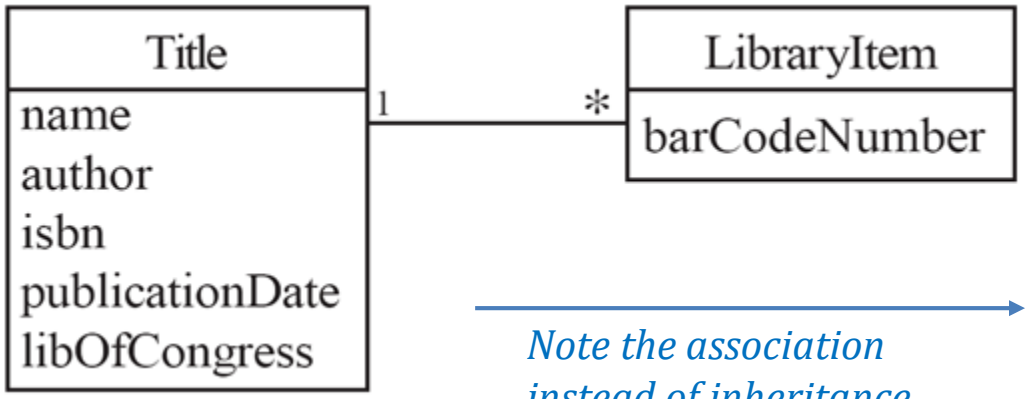
Ref: [LET]

Abstraction-Occurrence

– ***Solution:***





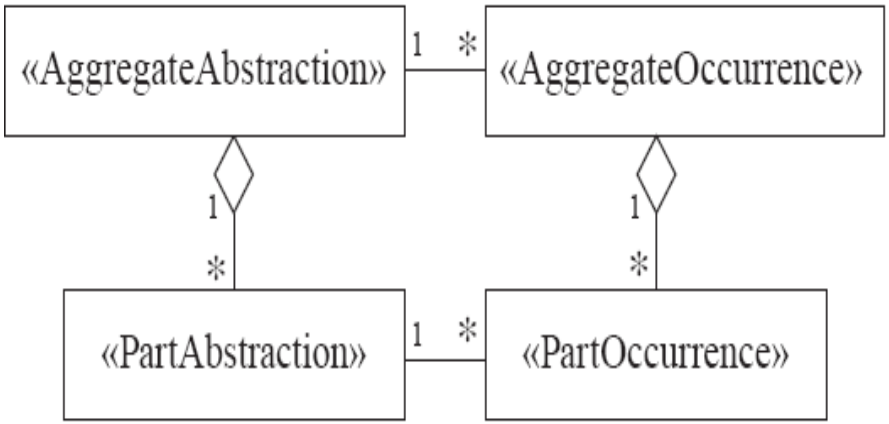


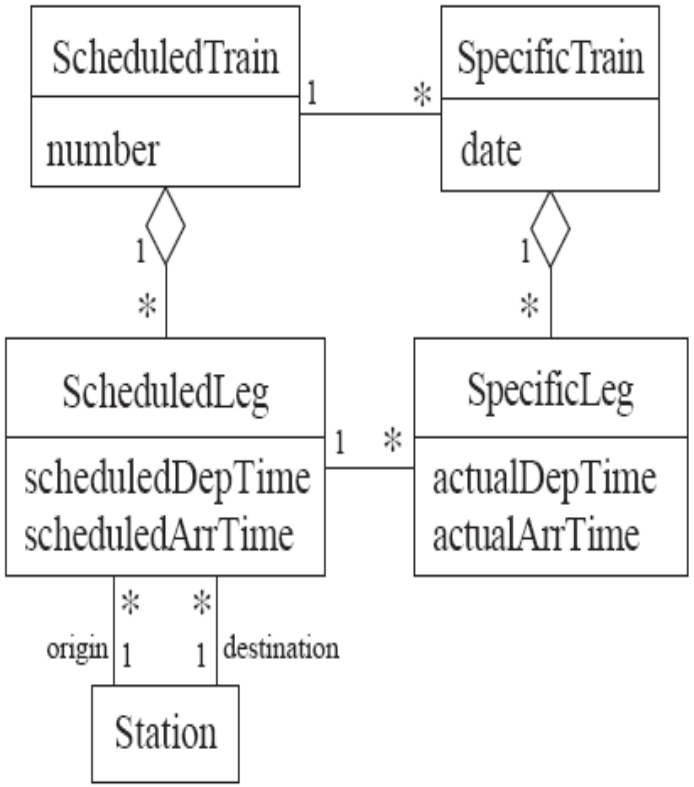
Note the association instead of inheritance

Ref: [LET]

Abstraction-Occurrence

- Square variant

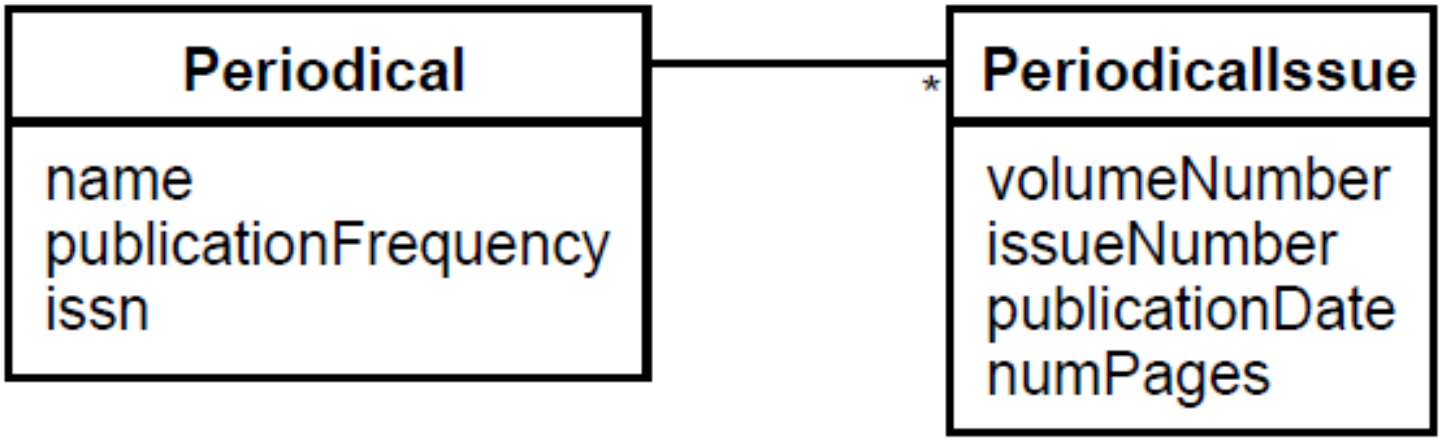




Ref: [LET]

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.

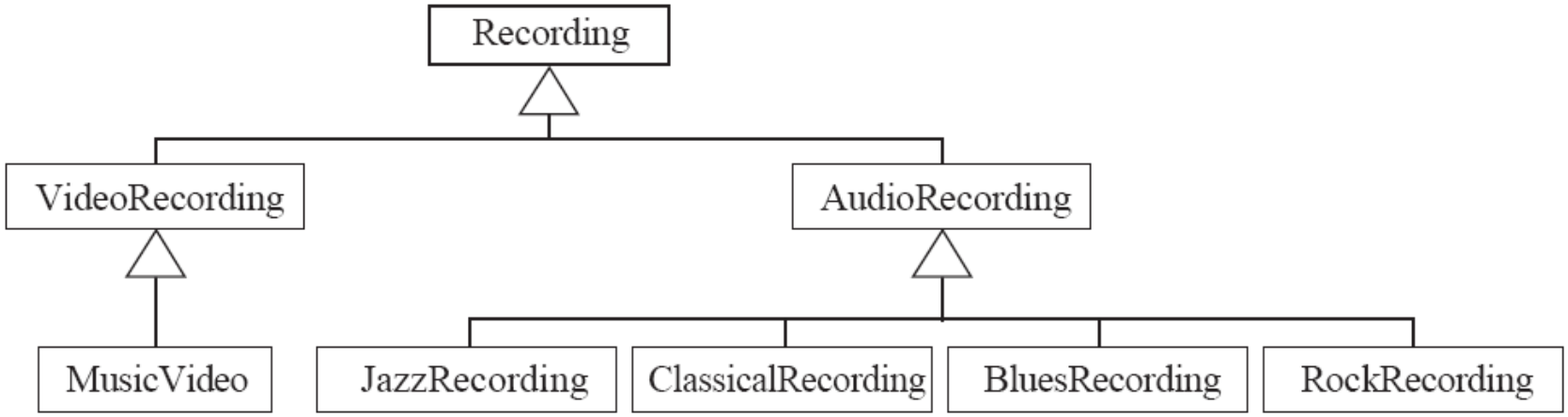


The General Hierarchy Pattern

- ***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

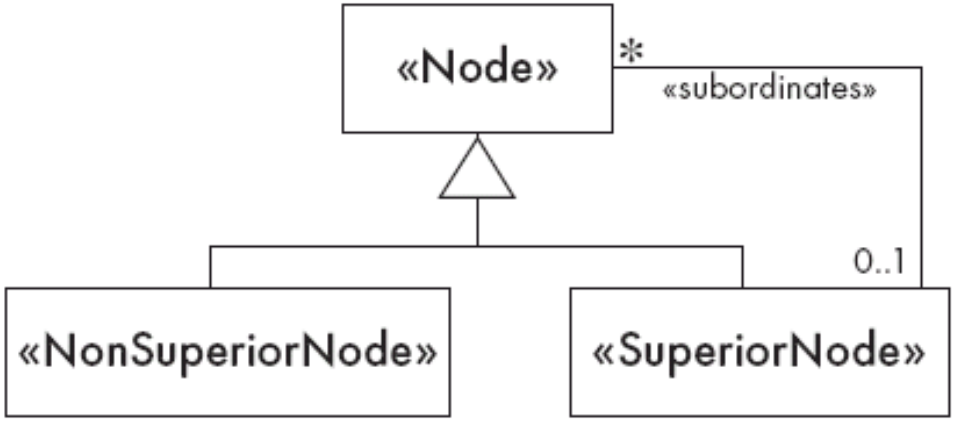
General Hierarchy

- Antipattern:



General Hierarchy

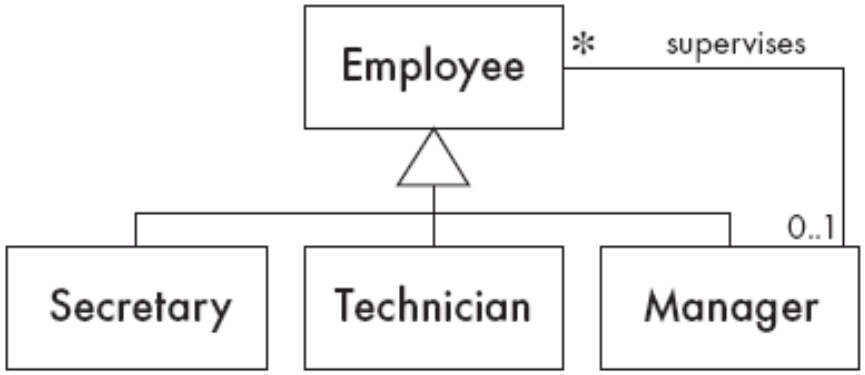
– ***Solution:***

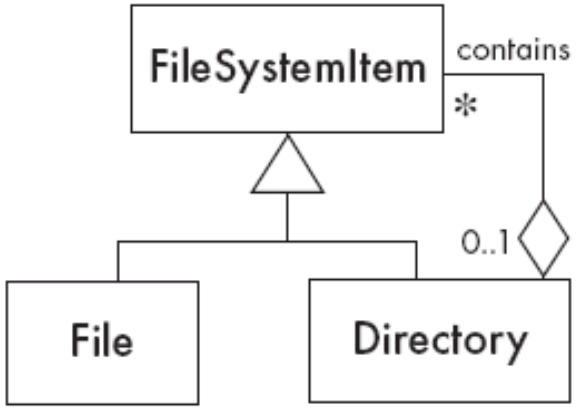


Ref: [LET]

General Hierarchy

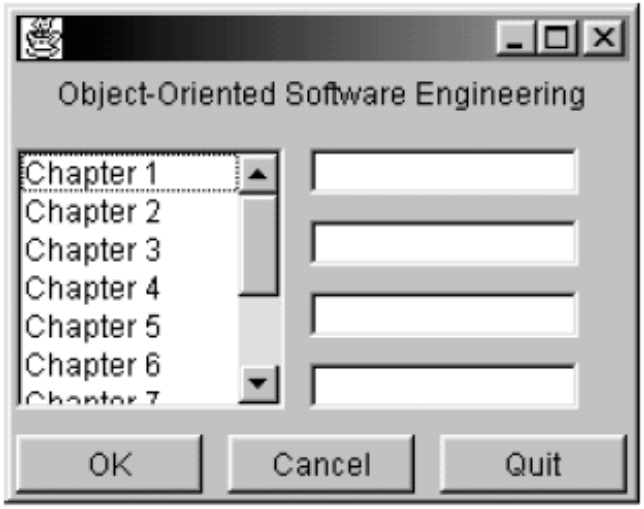
– ***Solution:***

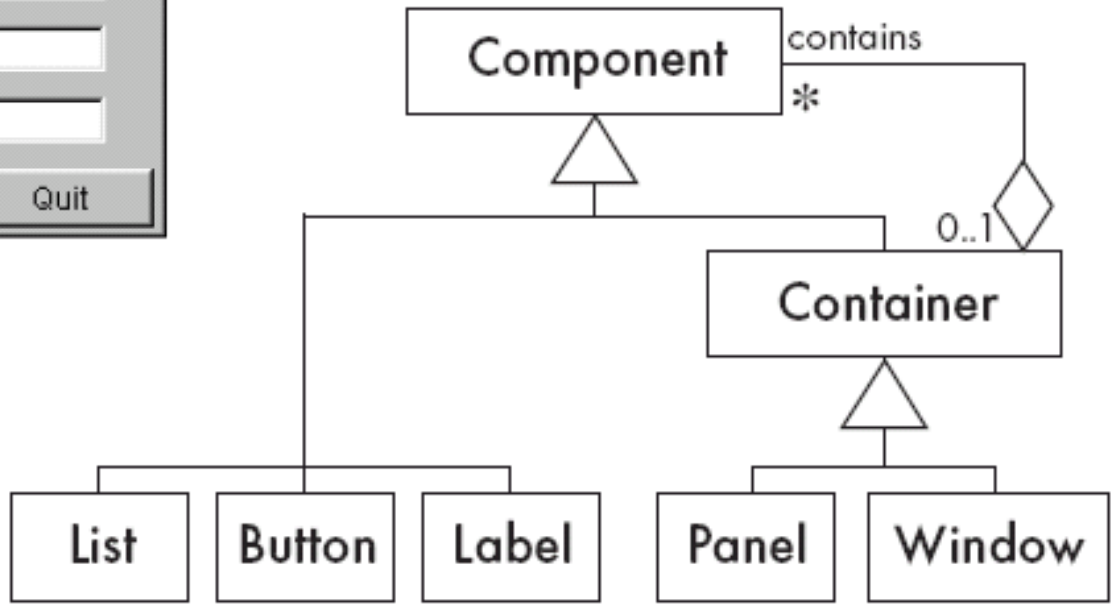




Ref: [LET]

General Hierarchy – Example

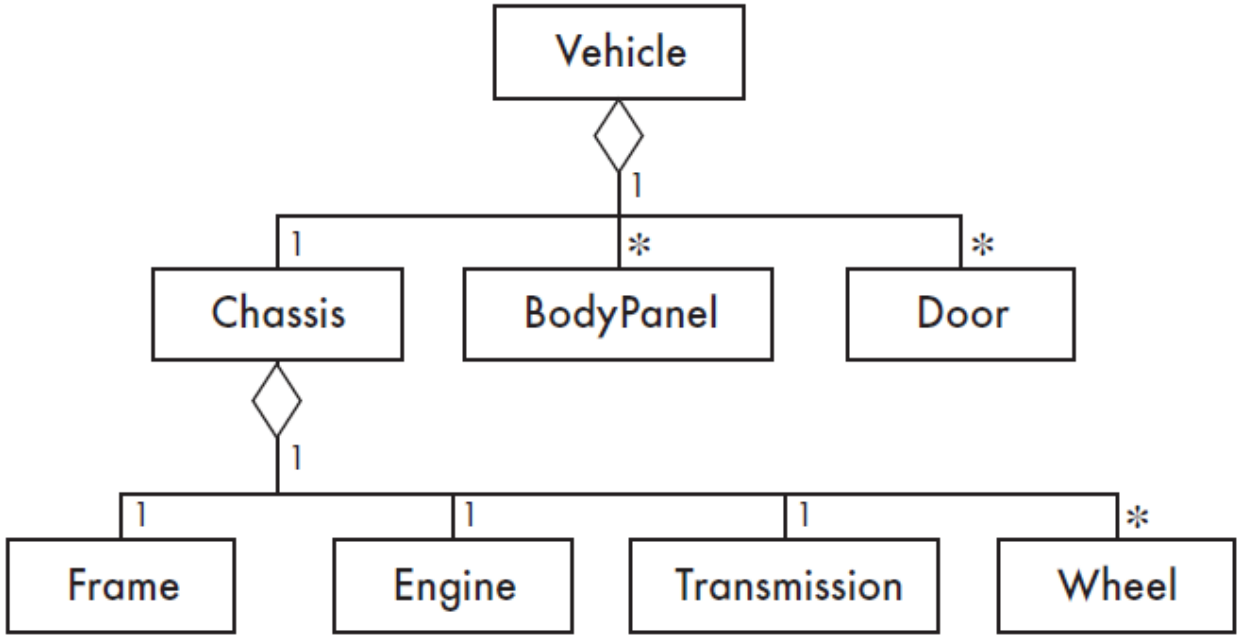




Ref: [LET]

General Hierarchy – Example

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

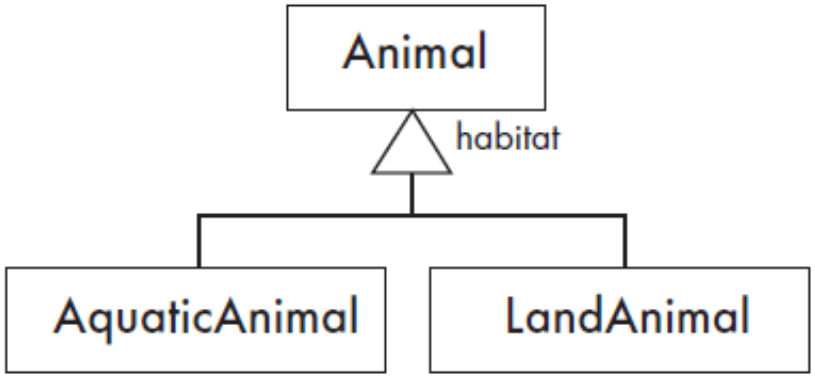


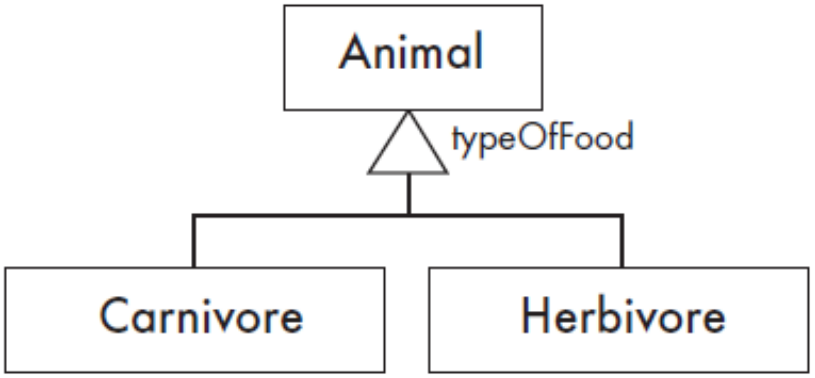
The Player-Role Pattern

- ***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

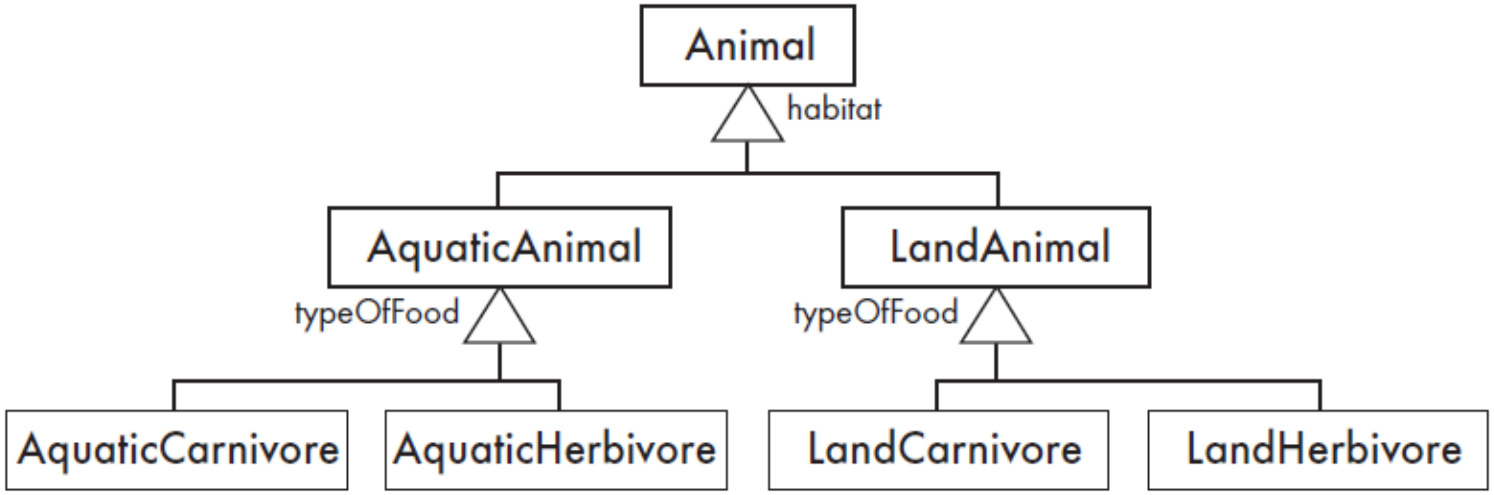




Ref: [LET]

Player-Role – Antipatterns

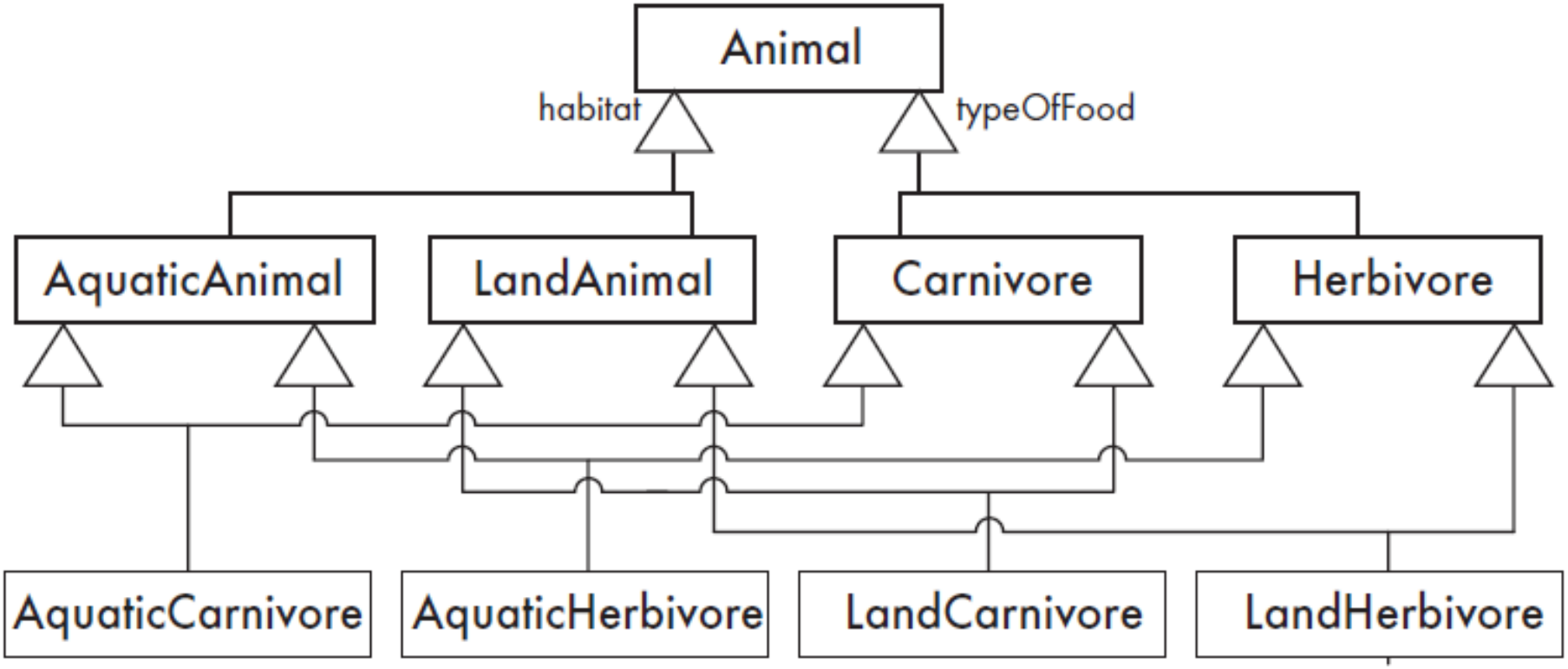
- Create a higher-level generalization set (here habitat), and then to have generalization sets with duplicate labels at a lower level in the hierarchy
  - all the features associated with the second generalization set would also have to be duplicated
  - the number of classes can grow very large.
-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



Ref: [LET]

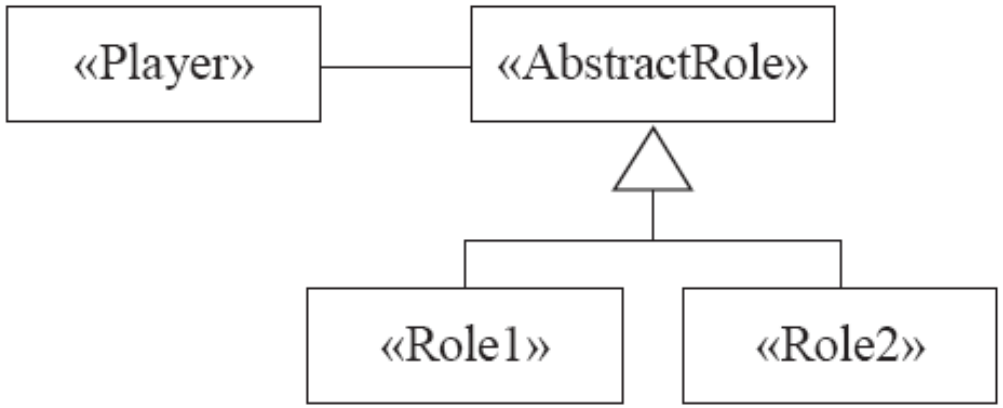
Player-Role – Antipatterns

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



Player-Role

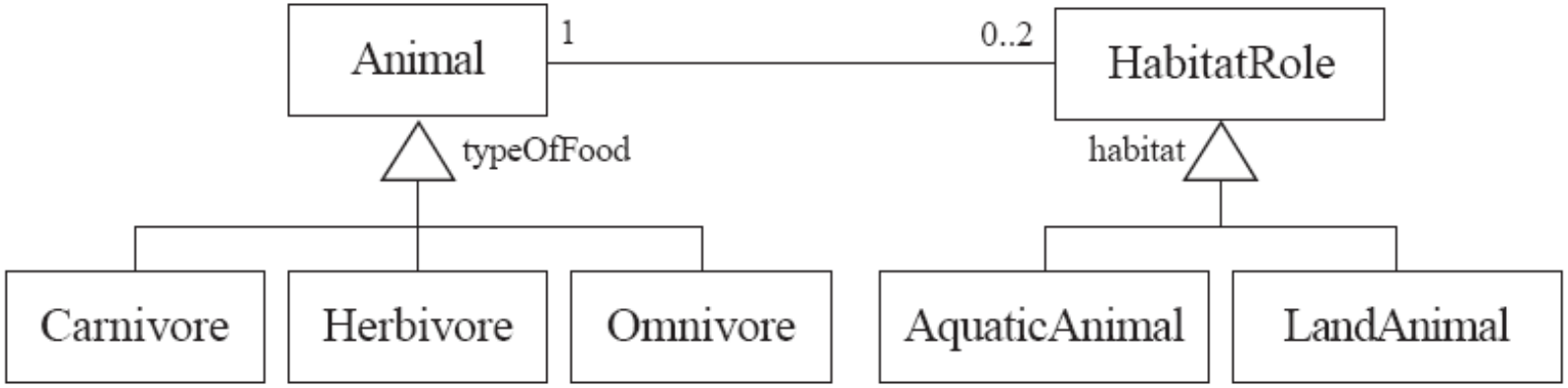
– ***Solution**:*



- 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

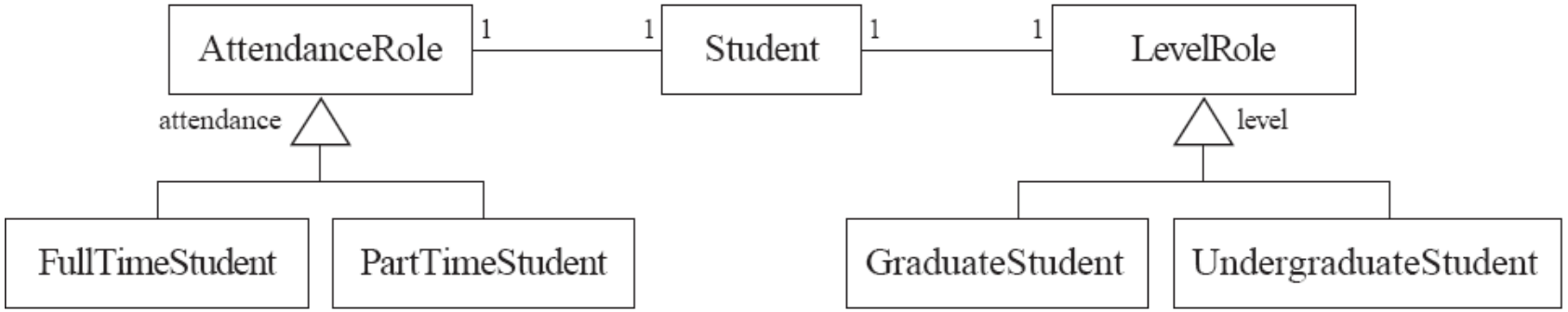
- Example 1:



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

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.

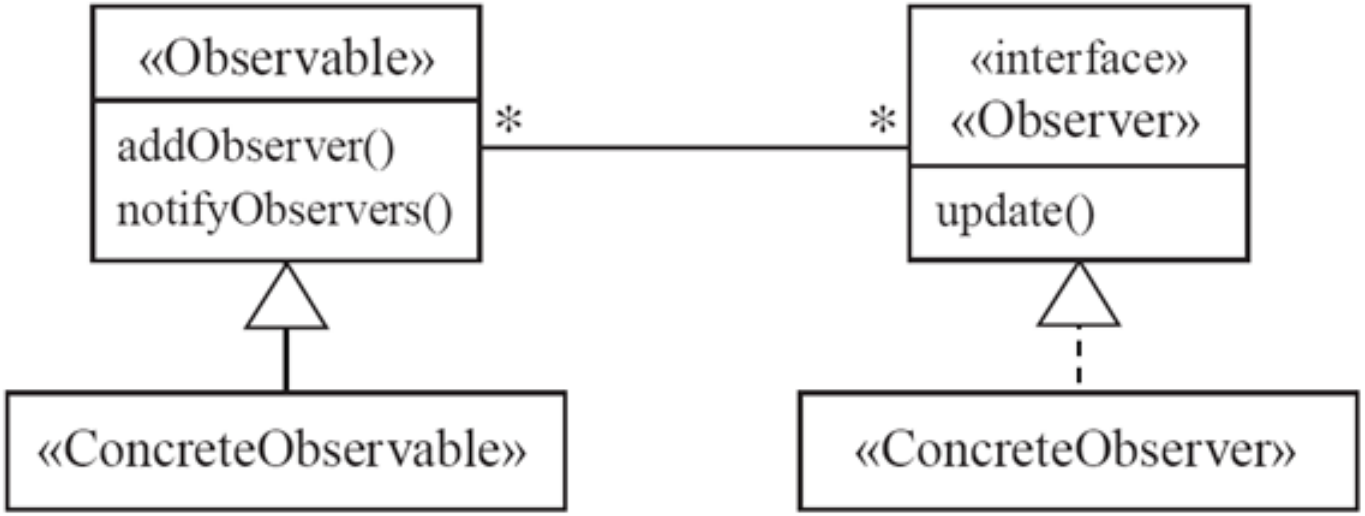
c) Your favorite sport (football, baseball, etc.) in which players can play at different positions at different times in a game or in different games.

The 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

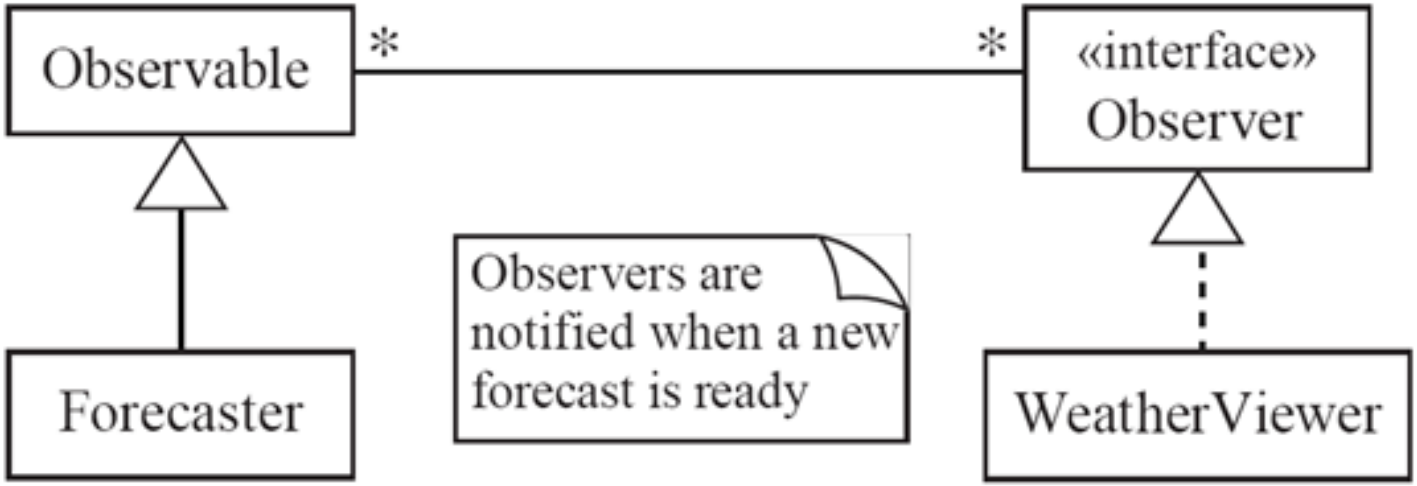
Observer

– ***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.