Lecture 3: Design Patterns I

Today's Plan

- 1st hour:
  - Quiz 3 (assessed, correctness)
  - Abstraction-Occurrence, Singleton, Façade, Factory
- Break
- 2nd hour:
  - PI 3 (assessed, participation)
  - Player-Role, General Hierarchy, Delegation, Adaptation,
- Long break
- 13:00-14:00 OR 14:00-15:00 in BO1028: Lab 3 Design Patterns Part 1

Quiz 3: Design Patterns

QUIZ

- Go to YACRS
  - https://classresponse.gla.ac.uk
- Join session : **1171**
  - prepare for quiz.
  - 120 sec per question (I'll call out 60 sec, 30 sec, 5 sec)

Q3.1Design Patterns

Design patterns have a standard description format composed of 7 key elements. Which of the following options IS NOT one of these elements?

- A: Solution
- B: Anti-patterns
- C: Context
- D: Discussion

Q3.1 Design Patterns I (solution)

TODO

Design patterns have a standard description format composed of 7 key elements. Which of the following options IS NOT one of these elements?

- A: Solution <– The recommended way to apply the pattern
- B: Anti-patterns <– Incorrect applications of the pattern
- C: Context <– The general situation where a pattern should be used.
- D: Discussion

Q3.2: Design Patterns II

Patterns should normally be illustrated using a simple diagram. These diagrams:

- A: Follow a formal syntax that must be used in all pattern diagrams

- B: Have an informal syntax with elements from UML
- C: Do not have any standard syntax
- D: Patterns are not represented with diagrams

Q2.2: Design Patterns II (solution)

Patterns should normally be illustrated using a simple diagram. These diagrams:

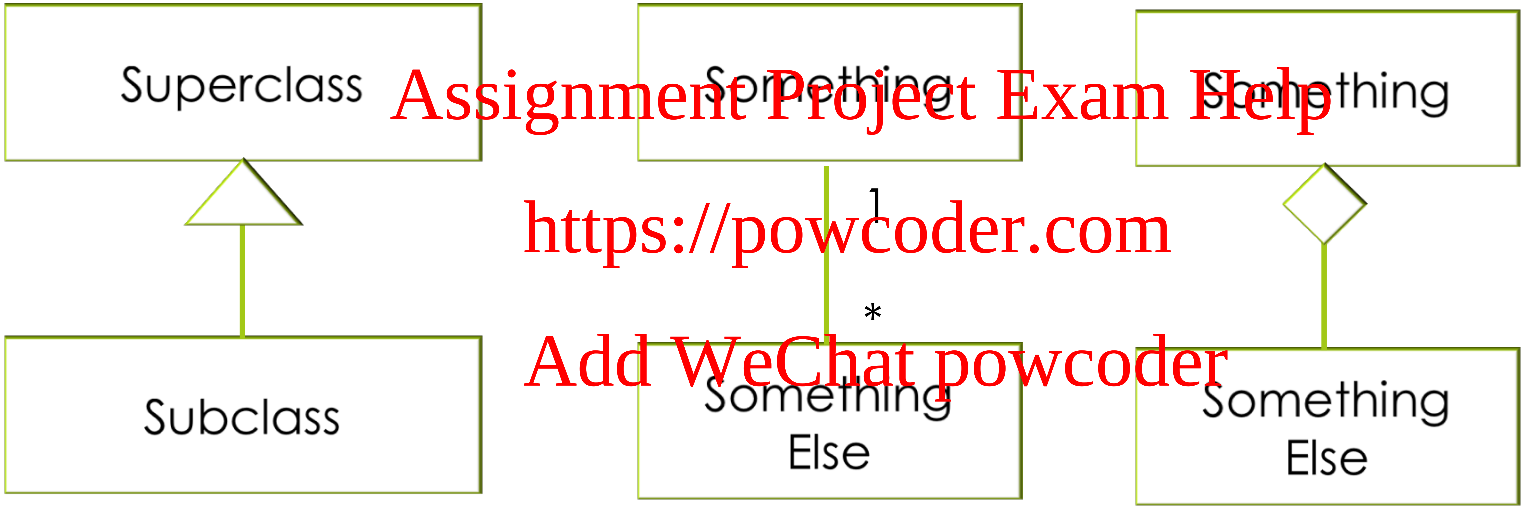
- B: Have an informal syntax with elements from UML
  - For example, see Figure 6.1 and 6.2 in your book.

Q3.3: Abstraction Occurrence

The abstraction occurrence pattern is an example of a:

- A: "*is a*" relationship
- B: "*Many-to-many*" relationship
- C: "*One-to-one*" relationship
- D: "*has a*" relationship

Q3.3: Abstraction-Occurance



Is A

Has A

Is an aggregation of…

"Is a" and "has a" relationships

Design patterns (in this course)

This week:

- Abstraction-Occurrence
- Façade
- Singleton
- The Factory

- Player-Role
- Delegation
- Adapter
- General Hierarchy

Next week:

- Observer
- Immutable
- Read-Only Interface
- The Proxy

For Each Pattern

- You must understand its context, problem, forces, solution, anti-patterns, and graphical representation.

- You should have a good understanding such that you could write a basic implementation.

Design Patterns Gameplan

- Design Patterns are important, we'll spend two weeks on this chapter.

- Now:
  - **Abstraction-Occurrence, Singleton, Factory, Façade**

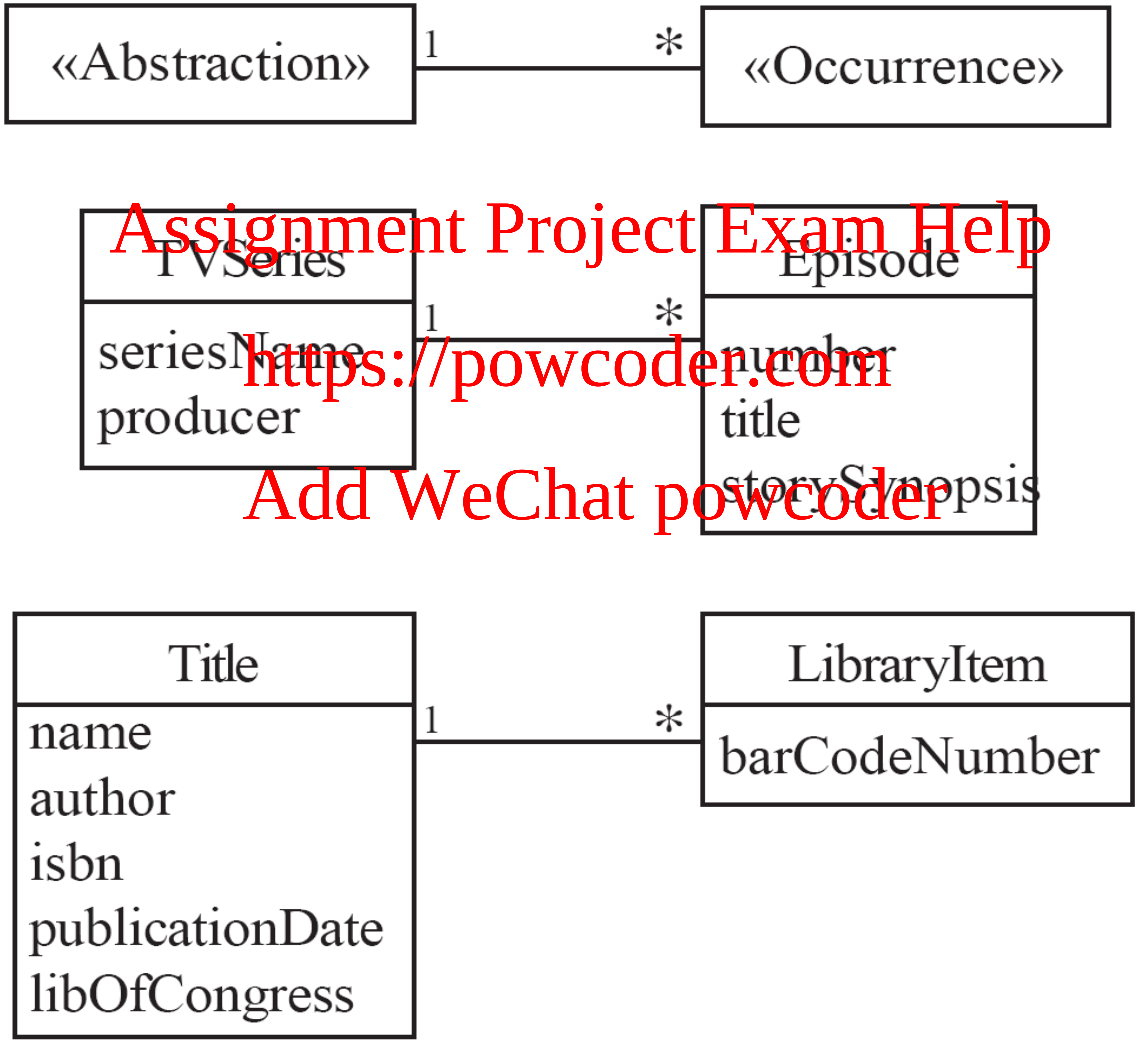
Abstraction Occurrence

- Context: A set of related object that we will call occurrences, the members of such a set share common information but differ in other important ways.
  - Episodes in a television series
  - Flights offered daily on a flight schedule
  - Series of lectures in a course

- Solution: Create an abstraction class that contains the data that is common to all the members of a set of occurrences. Then create an Occurrence class representing the occurrence of this abstraction.

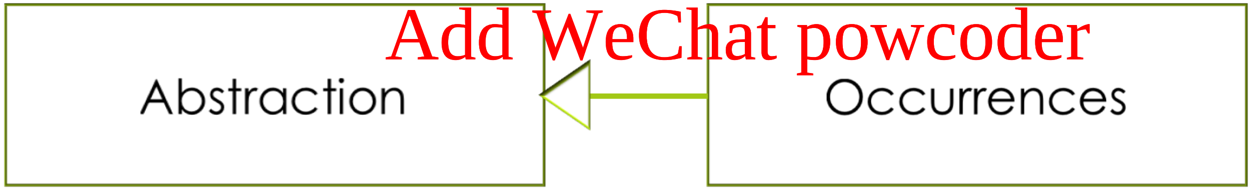
- Anti-patterns: Using a single class with inheritance.

Abstraction Occurrence



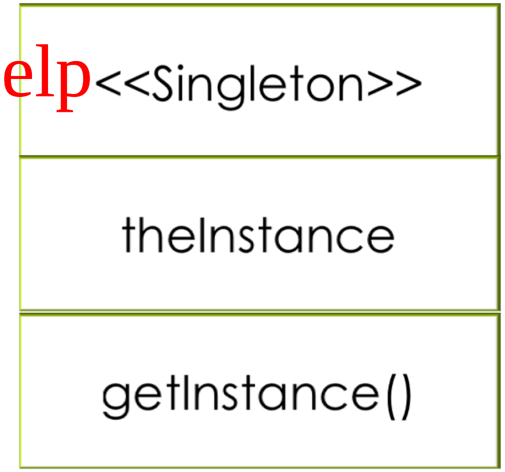
Abstraction Occurrence Antipattern

- Why is the following use of abstraction occurrence incorrect?



Singleton

- Context: Use in scenarios where you only want one instance to exist.
  - Company class
  - MainWindow class in UI
- Solution: A private class variable stores the single instance. A public static method returns the instance. The constructor is private.
- Note: Pattern should not be overused.

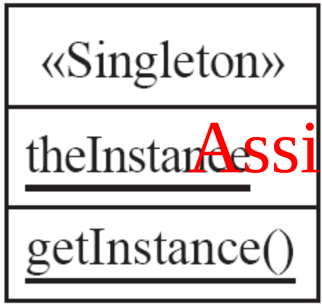


<<Singleton>>

theInstance

getInstance()

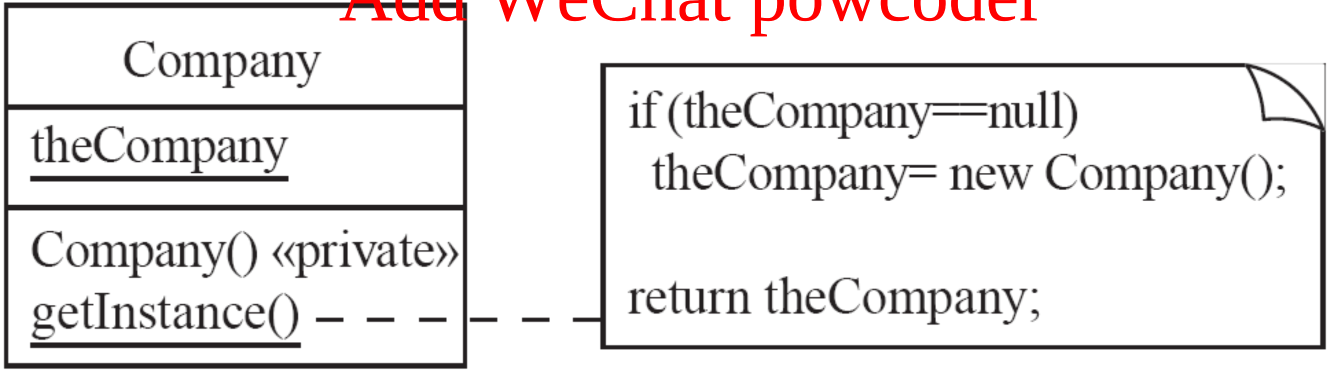
Singleton



«Singleton»

theInstance

getInstance()



Company

theCompany

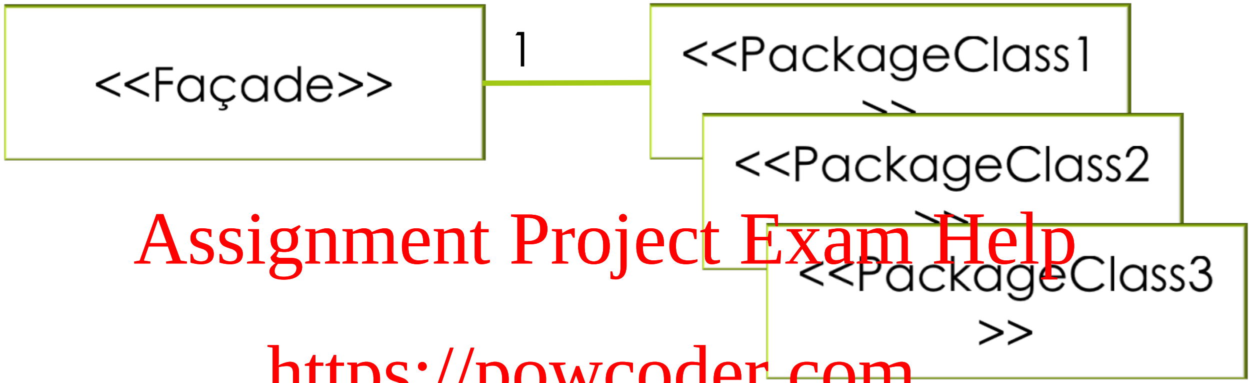
Company() «private»

getInstance()

```
if (theCompany==null)
  theCompany= new Company();

return theCompany;
```

Façade



<<Façade>>

1

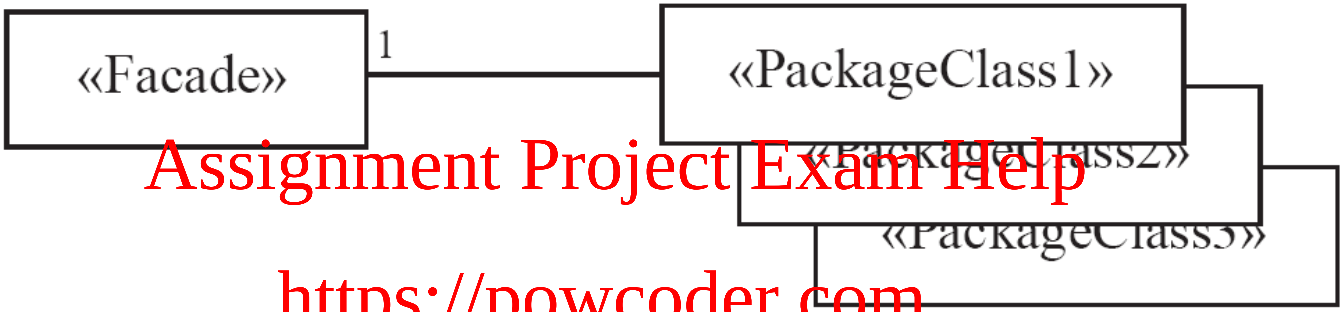
<<PackageClass1 >>

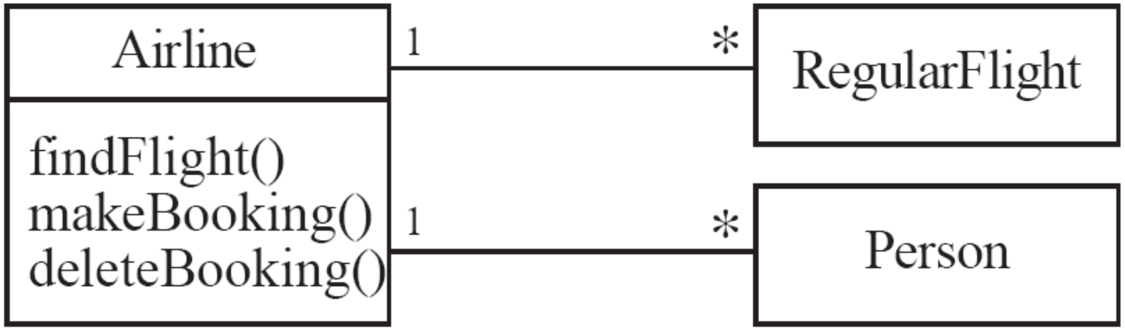
<<PackageClass2 >>

<<PackageClass3 >>

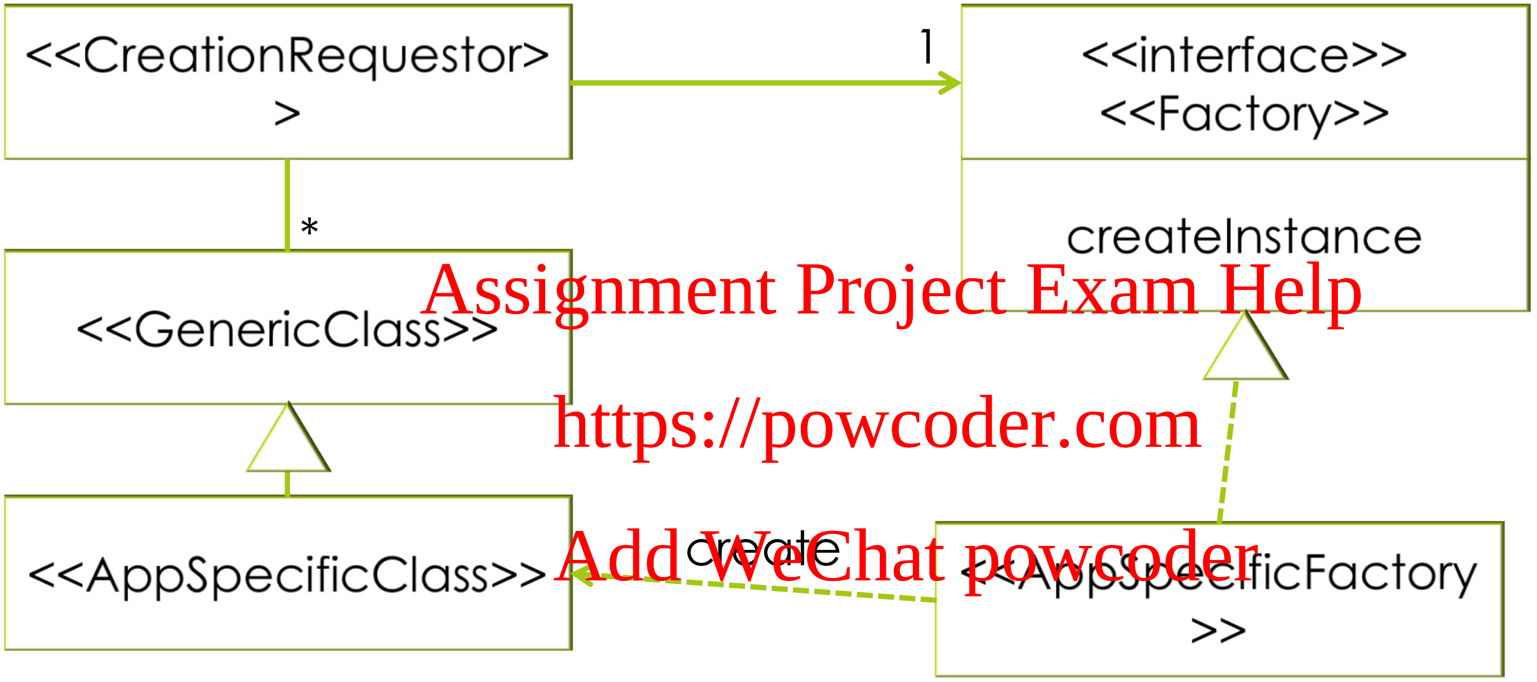
- Context: One application contains several complex packages. A programmer working with such packages has to manipulate many different classes.
  - Airline uses a large number of classes which could be simplified
- Solution: Create a façade class, which provided a set of public methods that simplifies the usage of the package.

Façade


«Facade»
1
«PackageClass1»
«PackageClass2»
«PackageClass3»

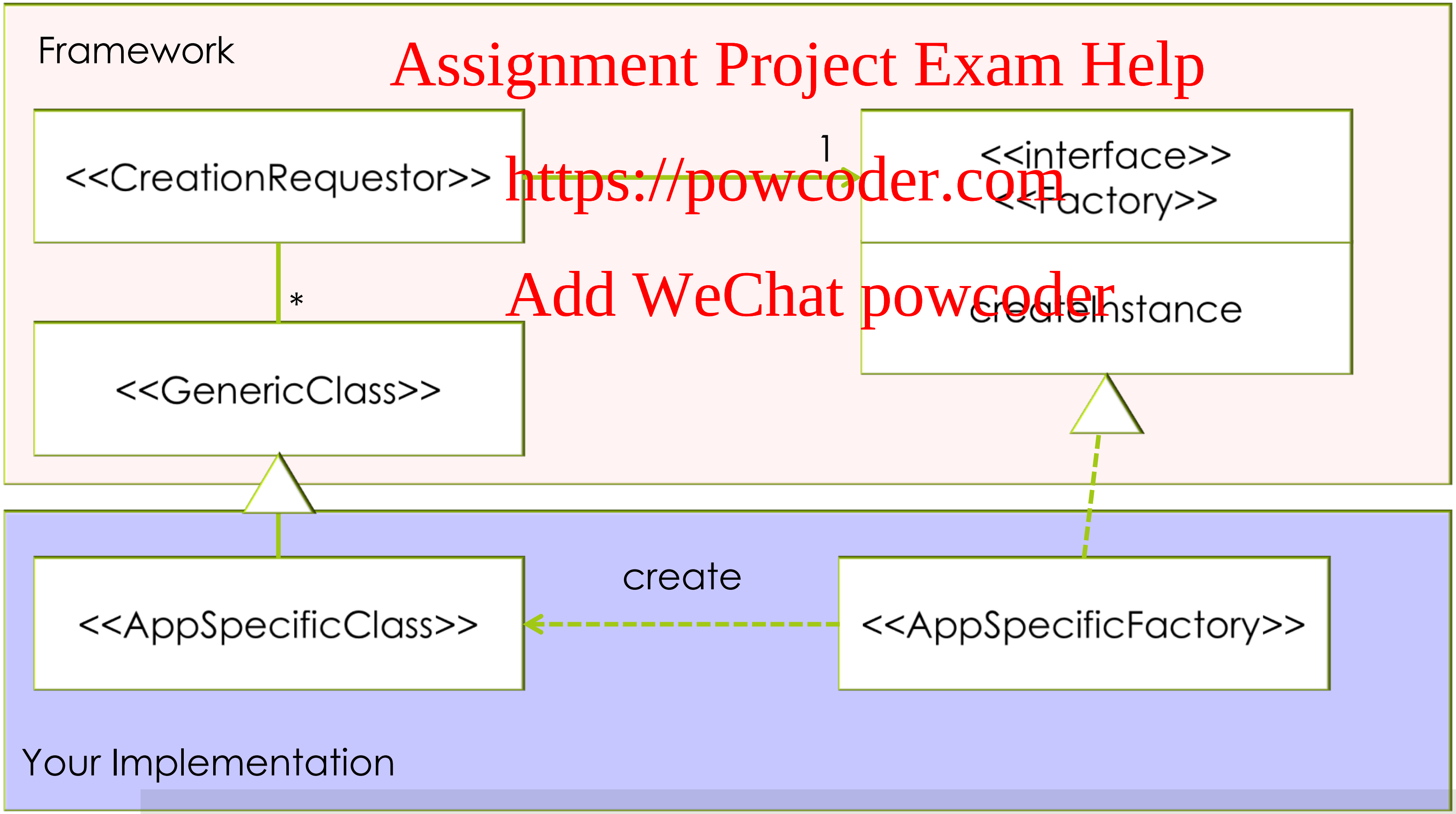

Airline
findFlight()
makeBooking()
deleteBooking()
1
*
RegularFlight
1
*
Person

Factory



- Context: You have a **framework** that needs to create objects as part of its work. However, the class of the created objects will depend on the application.
  - MediaFileManager framework which needs to make different kinds of media files

Factory Pattern



Framework

<<CreationRequestor>>

1

<<interface>>
<<Factory>>

createInstance

*

<<GenericClass>>

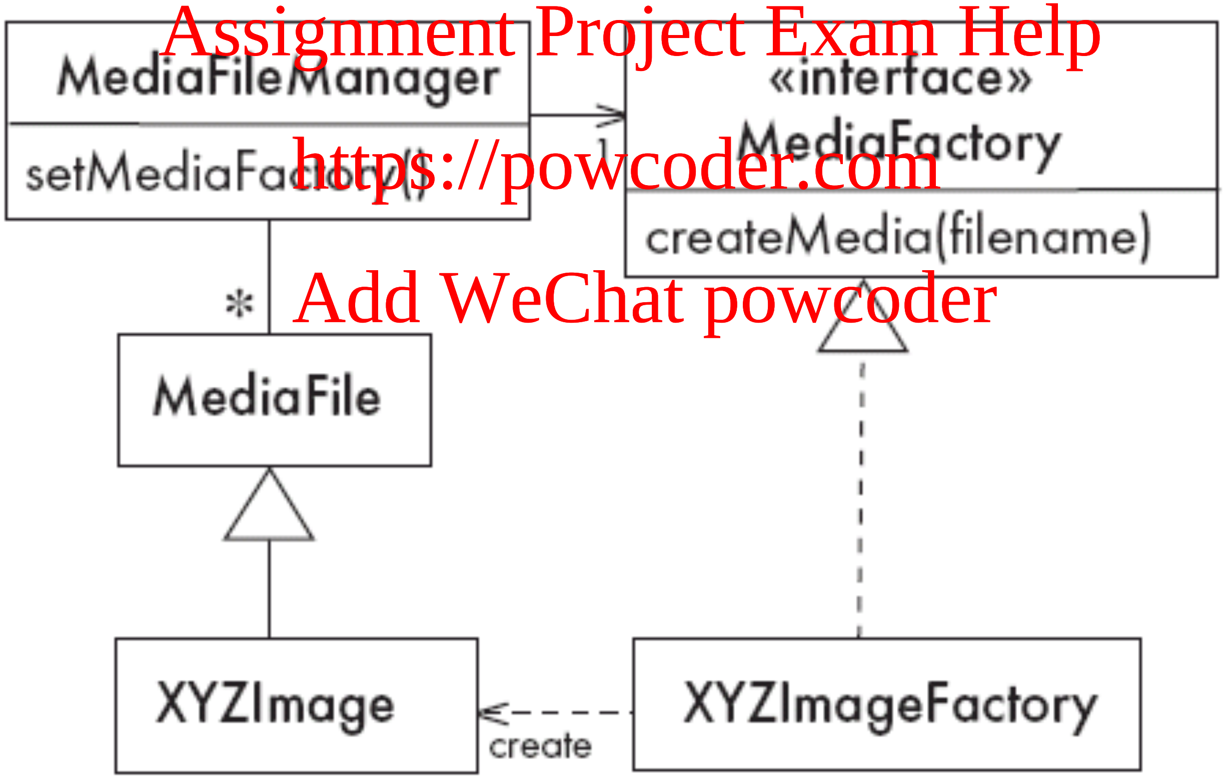
create

<<AppSpecificClass>>

<<AppSpecificFactory>>

Your Implementation

Factory Pattern



Factory

- Solution: The framework delegates creation of **specific classes** to an app specific *factory*. The *factory* implements an interface defined in the framework. The **specific class** extends a generic class defined in the framework.

PI 3: Questions

- Join YACRS Now!
- Join Session: **1172**

PI 3.1: Singleton Pattern

The Singleton pattern ensures other classes can’t create multiple instantiations of the singleton object by:

- A: Not using a constructor
- B: Using a private constructor
- C: Using a protected constructor
- D: Using a public constructor

PI 3.1: Singleton Pattern (solution)

- The Singleton pattern ensures other classes can create multiple instantiations of the singleton object by:

- A: Not using a constructor
- B: Using a private constructor <– Only a private constructor ensure other classes can't create multiple instantiations of the Singleton object
- C: Using a protected constructor
- D: Using a public constructor

PI3.2: Singleton Pattern

A system for managing a list of company contacts keeps each company in the database using the Singleton pattern. This way, a company's records are not duplicated throughout the system.

A: This is a good use of the Singleton Pattern.

B: This is a good use of the Singleton Pattern, but it introduces unnecessary complexity.

C: This is a bad use of the Singleton Pattern, and it introduces unnecessary complexity.

D: This is a bad use of the Singleton Pattern, but it will functionally work as the database of companies expands.

PI3.2: Singleton Pattern (solution)

A system for managing a list of company contacts keeps each company in the database using the Singleton pattern. This way, a company's records are not duplicated throughout the system.

A: This is a good use of the Singleton Pattern

Are Singletons conceptually what we need?

B: This is a good use of the Singleton Pattern, but it introduces unnecessary complexity.

There is unnecessary coupling because this is a bad use of this pattern.

C: This is a bad use of the Singleton Pattern, and it introduces unnecessary complexity.

Correct.

D: This is a bad use of the Singleton Pattern, but it will functionally work as the database of companies expands.

Imagine how maintaining this class will work. How will you add new companies? You would need to implement a new Class every time you wanted to add a company. If you feel like you have to fight the pattern to make it work, it might not be the right pattern.

PI3.3: Façade Pattern

How many of the following statements about the Façade pattern are true:

-The Façade pattern simplifies a complex group of packages.

-Once the Façade is defined, it can be easier to make small changes to how to packages work together.

-A Façade class is useful because it does not introduce coupling.

A: 0

B: 1

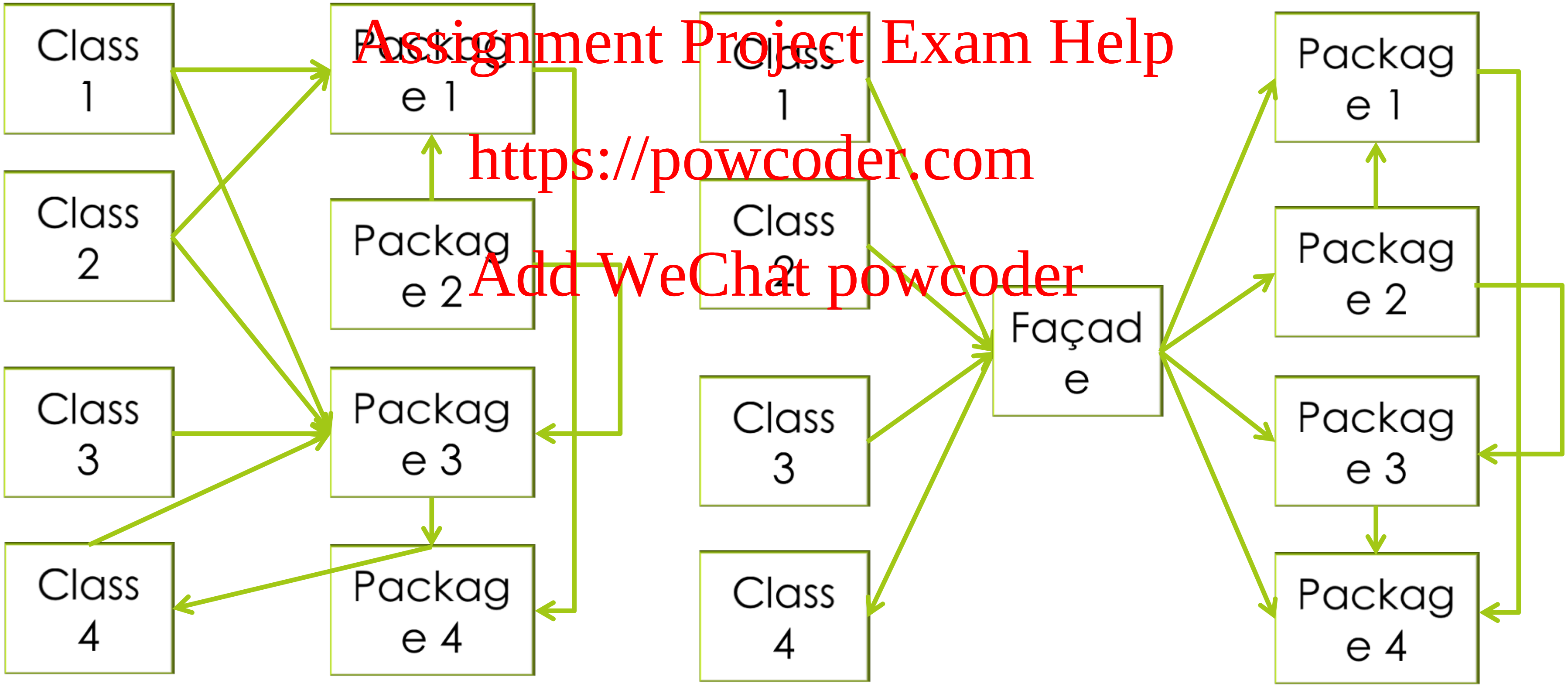
C: 2

D: 3

PI3.3: Façade Pattern (solution)

The Façade pattern simplifies a complex group of packages.

Once the Façade is defined, it can be easier to make small changes to how to packages work together.



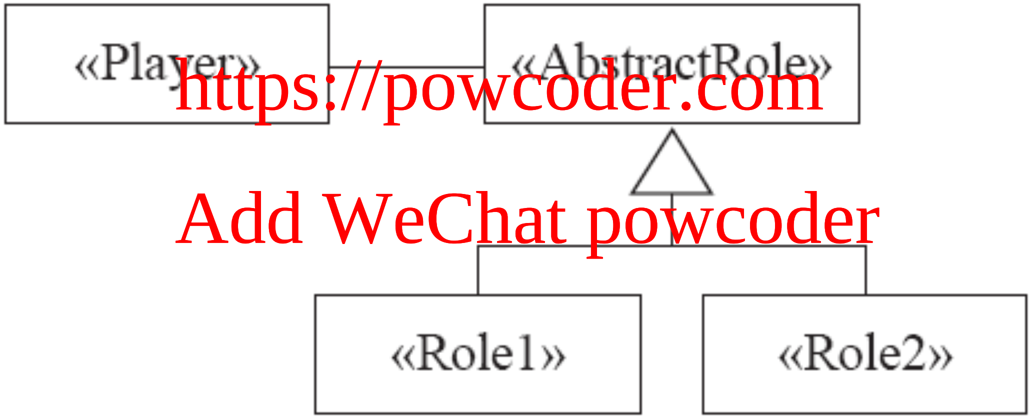
Design Patterns (cont.)

- Now:
  - Player Role, Delegation, Adaptor, General Hierarchy

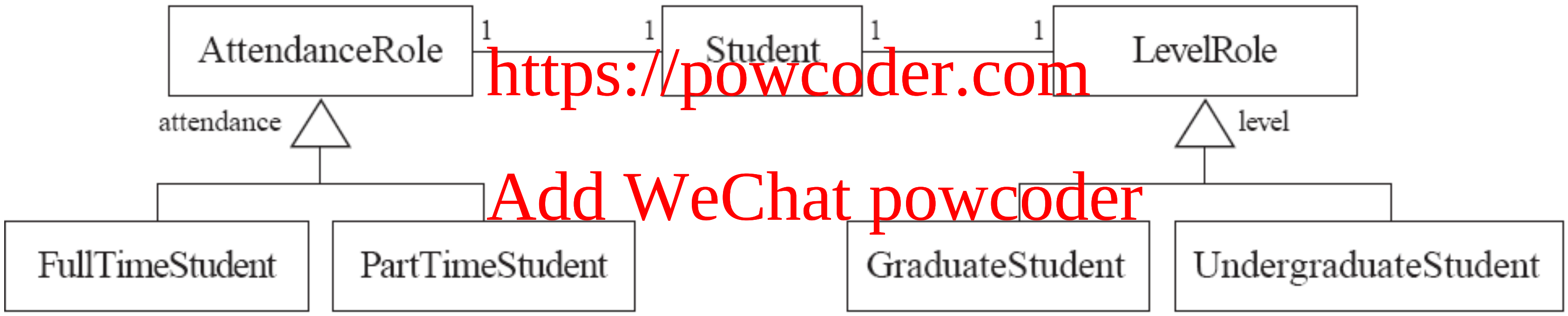
Player-Role

- ***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



Player-Role

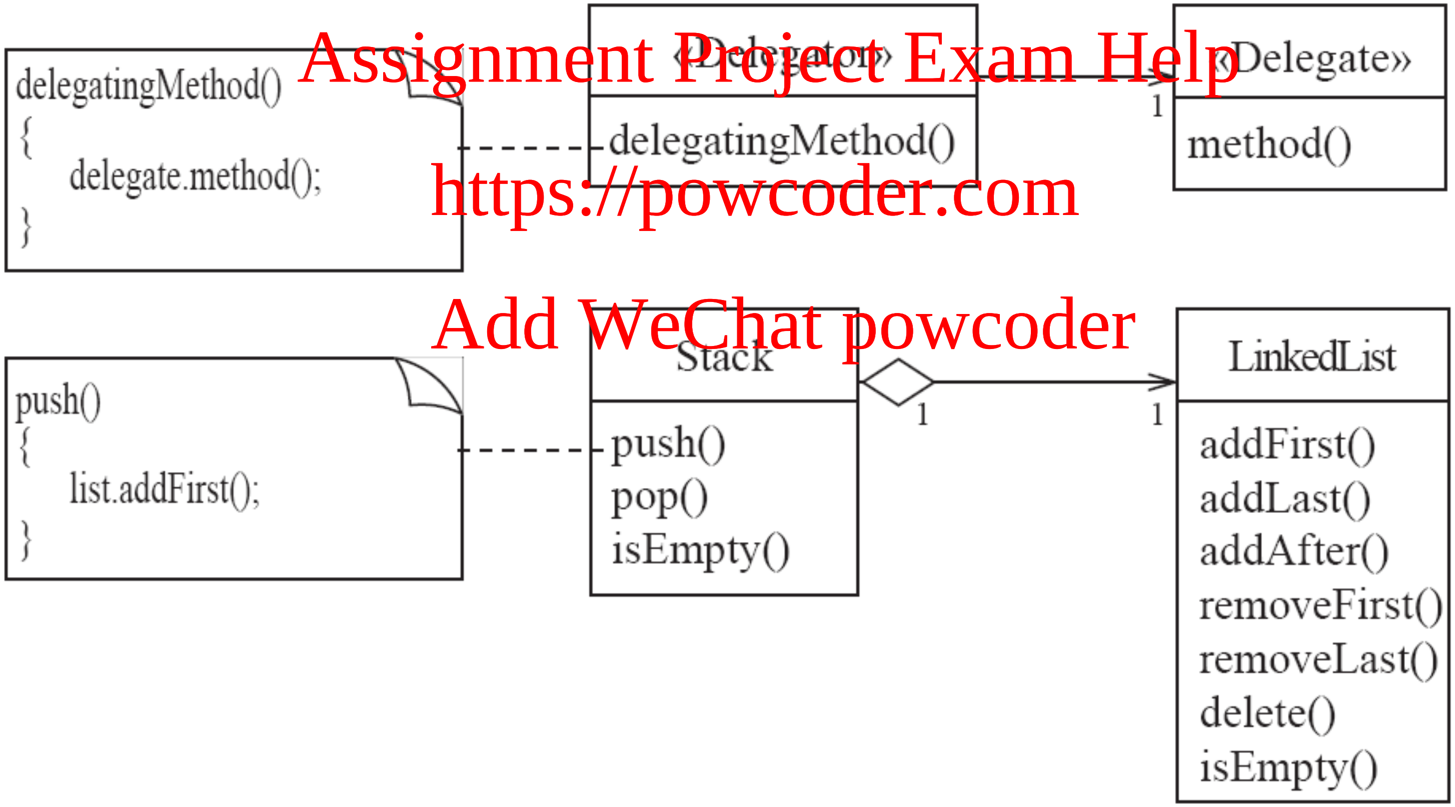

AttendanceRole
1
1
Student
1
1
LevelRole
attendance
level
FullTimeStudent
PartTimeStudent
GraduateStudent
UndergraduateStudent

The Delegation Pattern

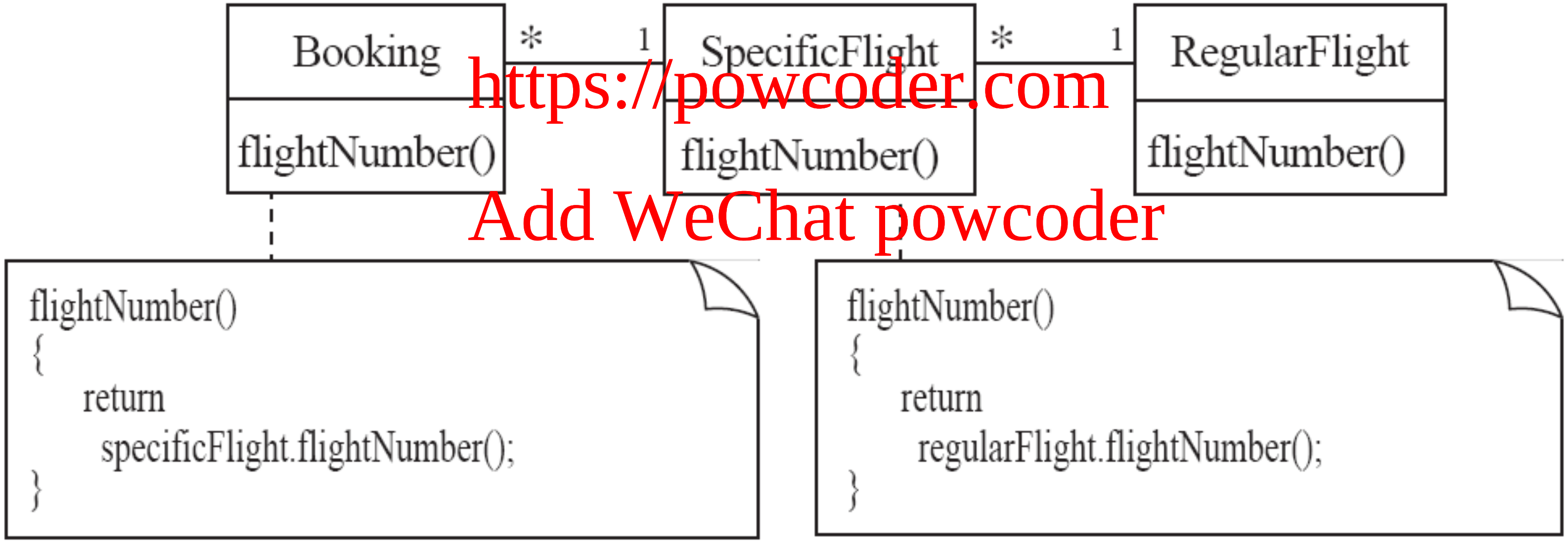
- ***Context:***
  - You are designing a method in a class
  - You realize that another class has a method which provides the required service
  - Inheritance is not appropriate

- Problem:
  - How can you most effectively make use of a method that already exists in the other class?

The Delegation Pattern



The Delegation Pattern



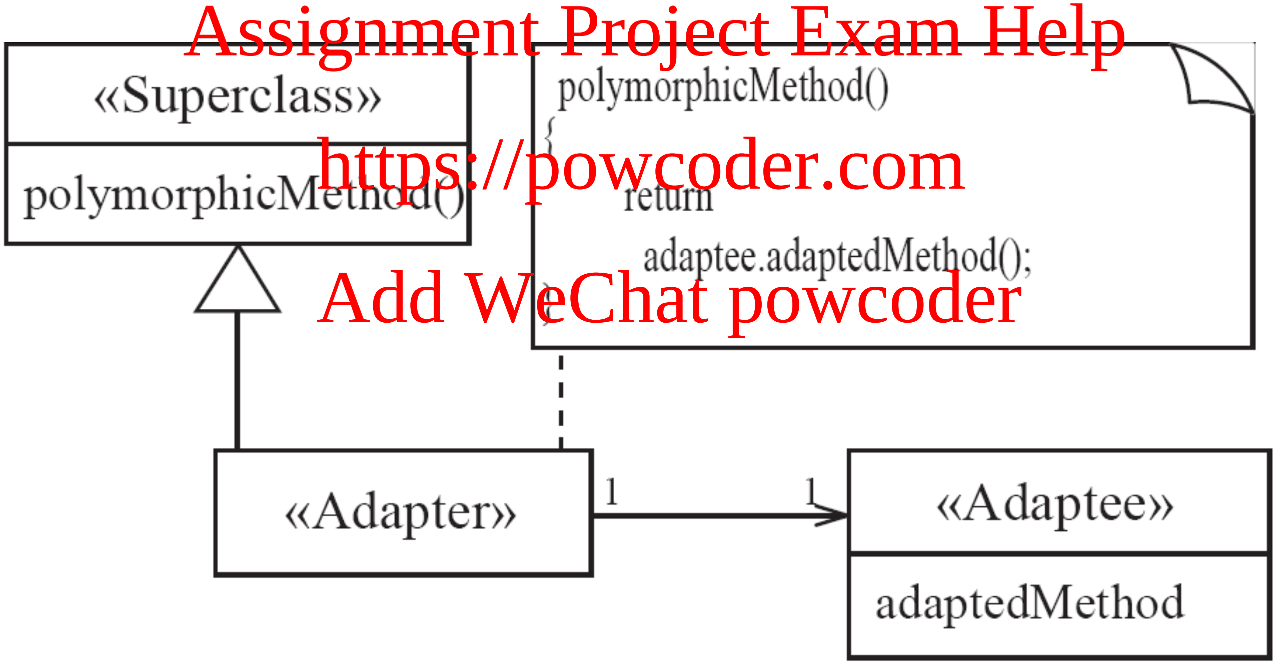
```
flightNumber()
{
    return
      specificFlight.flightNumber();
}
```

```
flightNumber()
{
    return
      regularFlight.flightNumber();
}
```

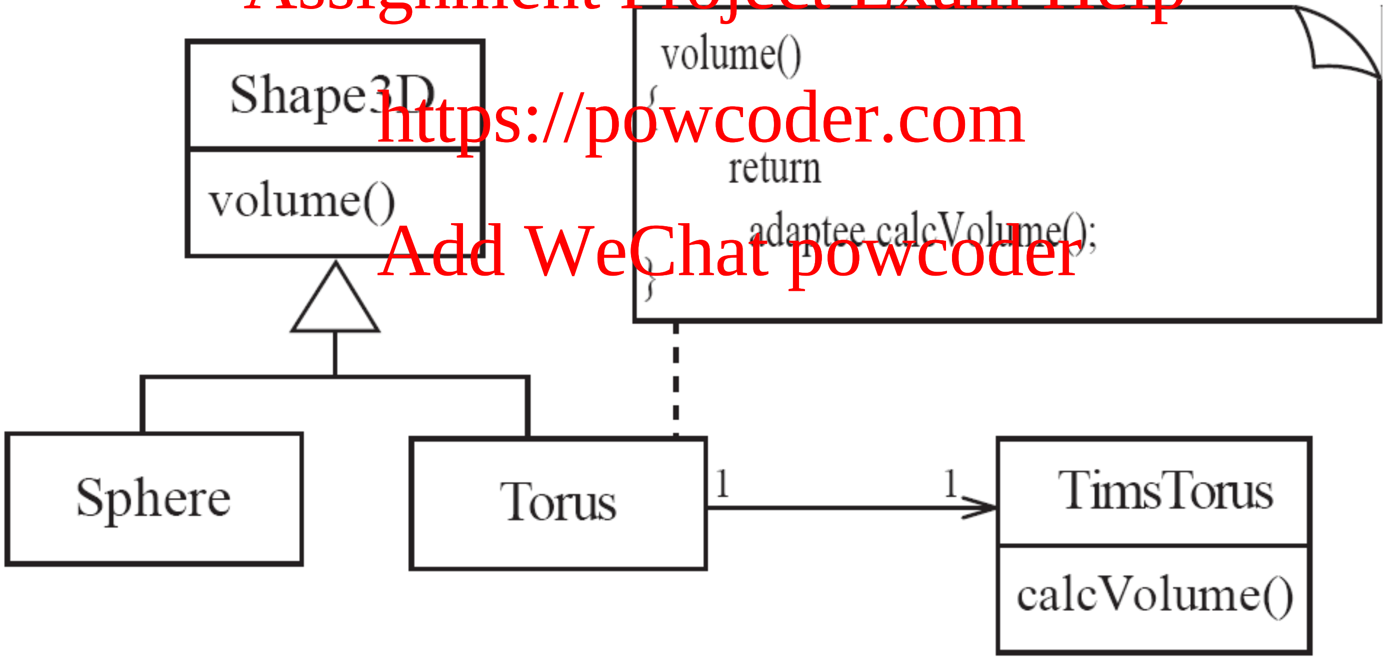
The Adaptor Pattern

- ***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

The Adaptor Pattern



The Adaptor Pattern

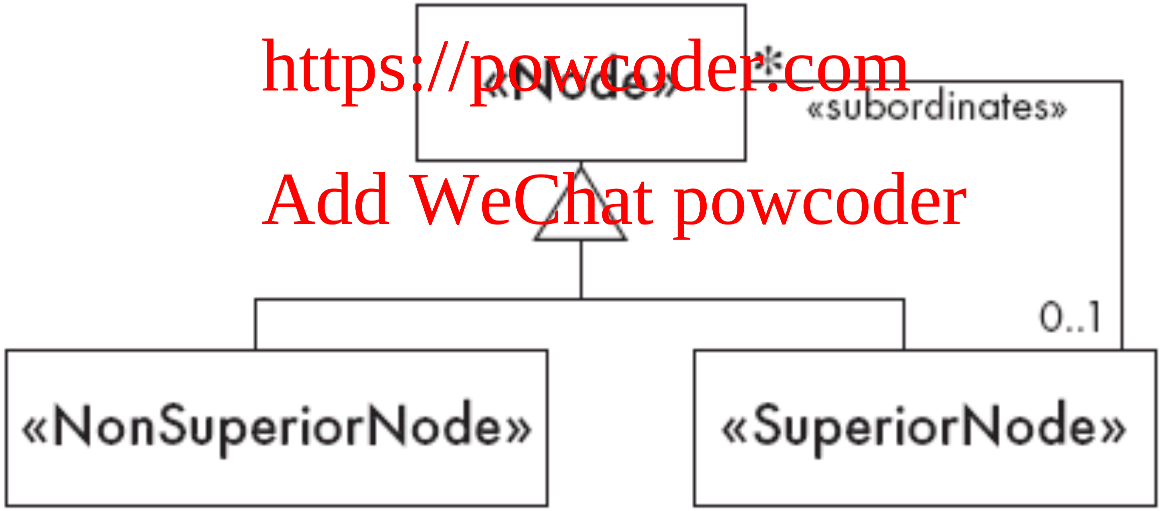


```
volume()
{
    return
      adaptee.calcVolume();
}
```

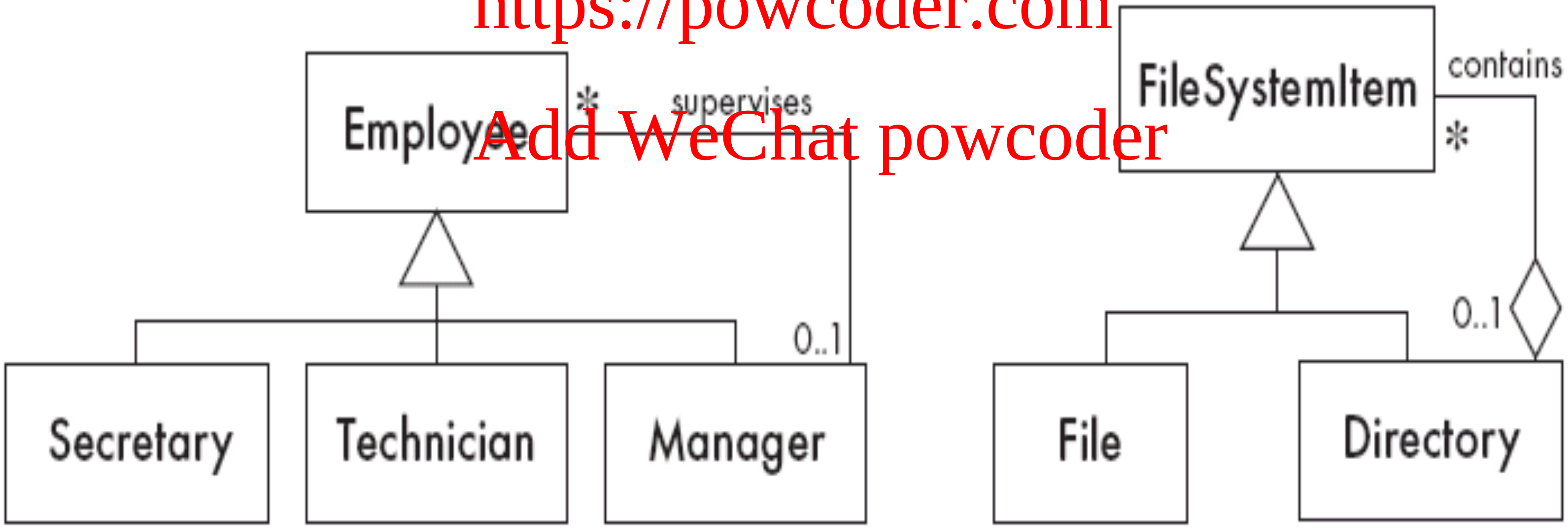
General Hierarchy

- ***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
  - How do you represent a hierarchy of objects, in which some objects cannot have subordinates?
- Problem: How do you represent a hierarchy of objects, in which some objects cannot have subordinates?

General Hierarchy



General Hierarchy



PI Questions

- Join YACRS session: **1172**

PI 3.4: Player Role

The Player-Role pattern is related to the abstraction occurrence. What relates these patterns?

A: They both use inheritance.

B: They both use abstract classes.

C: They both relate abstractions to real world entities.

D: They both use interfaces.

PI 3.4: Player Role (solution)

The Player-Role pattern is related to the abstraction occurrence. What relates these patterns?

A: They both use inheritance.

B: They both use abstract classes.

C: They both relate abstractions to real world entities.

D: They both use interfaces.

PI 3.5: General Hierarchy

The general hierarchy organizes classes into a hierarchy structure. This represents an:

A: Many to Many relationship

B: Inheritance Relationship

C: Superior/Subordinate Nodes

D: Abstract Relationship

PI 3.5: General Hierarchy (solution)

The general hierarchy organises classes into a hierarchy structure. This represents an:

A: Many to Many relationship

B: Inheritance Relationship ← This is an anti-pattern

C: Superior/Subordinate Nodes

D: Abstract Relationship

PI 3.6: Adaptor Pattern

The adaptor pattern allows you to incorporate an existing class into your inheritance hierarchy. This is useful because:

A: It encourages you to write specialised classes.

B: It simplifies code using polymorphism.

C: It prevents incorrect usage of inheritance.

D: It uses interface polymorphism.

PI 3.6: Adaptor Pattern (solution)

The adaptor pattern allows you to incorporate an existing class into your inheritance hierarchy. This is useful because:

A: It encourages you to write specialised classes.

B: It reuses code using polymorphism.

C: It prevents incorrect usage of inheritance. ← You can still use inheritance incorrectly with this pattern.

D: It uses interface polymorphism.

Lab 3

- We'll implement some design patterns in Java.
- See you in the computer lab.

Next Week (i.e. week 4)

- Remaining 4 patterns (OOSE Chapter 6).
- Review all the patterns.