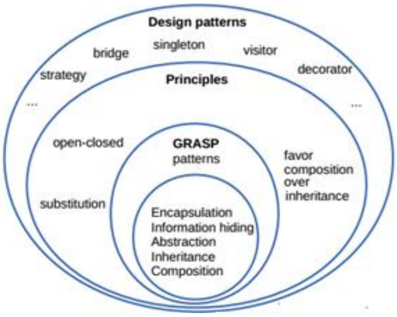
260CT
Software Engineering

Dr. Yih-Ling Hedley
Email: aa0817@coventry.ac.uk

Object-Oriented (OO) Design



Design patterns

bridge singleton visitor

strategy decorator

... ...

Principles

open-closed

substitution

favor composition over inheritance

GRASP patterns

Encapsulation
Information hiding
Abstraction
Inheritance
Composition

OO Software Design Concepts

OO Design Concepts

- Abstraction
  - Data Abstraction
  - Procedure Abstraction
- Encapsulation
- Information Hiding
  - Procedure Hiding
  - Data Hiding
- Modularity
  - Coupling
  - Cohesion
- Inheritance

Abstraction

- **Data abstraction:** The developer and other objects in the system have **a high level summary view** (an abstract view) of **what data items are**. It contains the required info about an object.
- **Procedure abstraction: A high level summary view of the operations** provided. The details and how the methods are coded are ignored.

Room
roomID: String **capacity : Integer** **type : String**

renovate()
splitRoom()

Encapsulation: Classes/Objects

- The **attributes (data)** relating to an object and the **methods/operations** that act upon them are all **encapsulated** inside a single software "capsule".

A class

Room
roomID: String **capacity : Integer** **type : String**

renovate()
splitRoom()

⇨

An instance = an object

roomAB107: Room
roomID = "AB107" **capacity = 200** **type - "Lecture Theatre"**

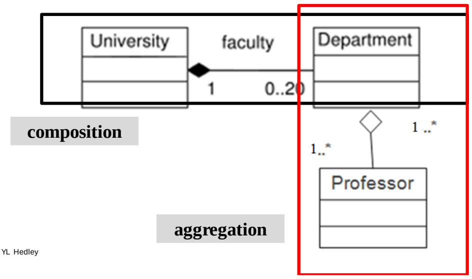
roomAS116: Room
roomID = "AS116" **capacity = 45** **type - "Lab"**

Encapsulation: Aggregation/Composition

- **Encapsulation**
  - via **aggregation and composition** to encapsulate components
    - encapsulates a group of classes collectively, as a complex **whole** is made of similar components (**parts**)

Composition vs Aggregation: Exercise 1

- **Encapsulation**
  - **Aggregation? Composition?**



Aggregation/Composition: Exercise 2

- Consider the code below, is it implemented as composition or aggregation?

```
public class University{

private Department[ ] departs;

public University( ) {
departs = new Department[20] ;
…
}
```

Composition vs Aggregation: Feedback 1

- **Composition**

```
public class University{

private Department[ ] departs;

public University( ) {

/* only the University instance has access to the list of Department.
When this University object is destroyed, the list of Department
objects will not be available */
departs = new Department[20] ;
…
}
```

Composition vs Aggregation: Feedback 2

- **Aggregation**

```
public class Department{
private Professors [ ] professors;
public Department( Professor[ ] pros) {

/*the list of Professor objects is created outside and
is passed as argument to Department constructor
When this Department object is destroyed,
the list of Professor objects is still available to
other Department objects or other objects*/
this. professors = pros;
…
}
```

Information Hiding

- **Data Hiding:** the developer and other objects in the system have **no direct access to** the **attributes (which are private)** or **the detail of how the attributes** are stored.
- **Procedure Hiding:** The developer and other objects in the system do not know the detail of how the methods work. The **name of a method is public**, but the **code body of the method is private**.

Room

-roomID: String
-capacity : Integer
-type : String

+renovate()
+splitRoom()
+getRoomID(): String
+setRoomID(id: String)

The access to the data via accessor methods (e.g getRoomId) ; modification of the data via mutator methods (e.g. setRoomID)

Cohesion: Operation cohesion

- **Cohesion:** a group of clearly defined processes that are functional related
  - **Operation cohesion**: the degree of an operation achieves a single functional requirement
    - High operation cohesion = good design

Student

-studID: String
-studName: String
-DOB: Date
-email: String

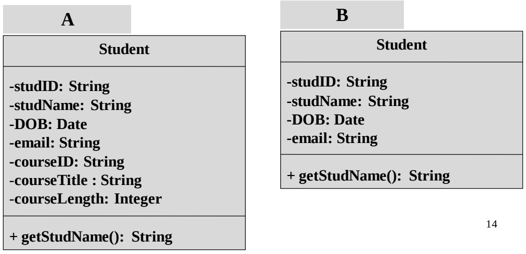
+ getStudID(): String
+ getStudName() : String
+ getStudDOB():Date
+ setStudName(name: String)
+ changetStudEmail(email: String)

Cohesion: Class Cohesion

- **Cohesion:**
  - **Class cohesion**: the degree of a class achieves a single requirement
    - A class should only have attributes and operations related to its objects
    - Data classes should just handle data.
    - ***High class cohesion = good design***

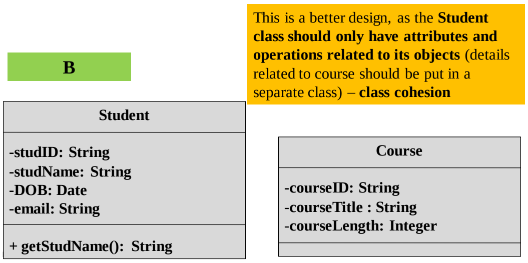
OO Design Concepts: Exercise 1

- Consider the code below, which of the following is a better design? Why?



OO Design Concepts: Exercise 1 Feedback

- Consider the code below, which of the following is a better design? Why?

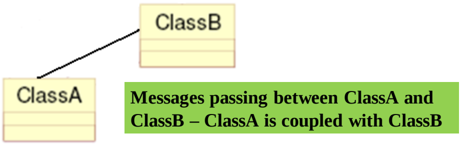

This is a better design, as the **Student class should only have attributes and operations related to its objects** (details related to course should be put in a separate class) – **class cohesion**

Cohesion: Specialisation Cohesion

- **Cohesion:**
  - **Specialisation cohesion**: semantic cohesion of inheritance hierarchies. Super-class and its subclasses should be closely related.
    - ***High specialisation cohesion = good design***

Coupling: Interaction Coupling

- **Coupling:** degree of interaction between objects
  - **Interaction coupling** – the number of **message types** an object sends to other objects and number of **parameters passed** with the messages
    - ***Low Interaction coupling = good design***

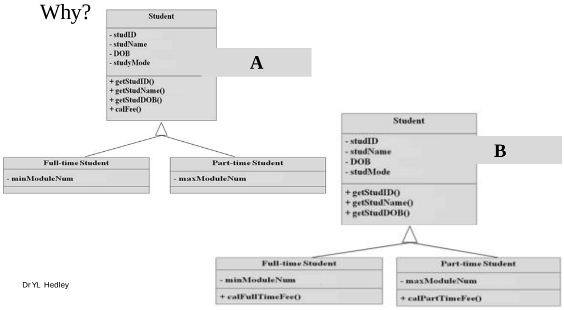

Messages passing between ClassA and ClassB – ClassA is coupled with ClassB

Coupling: Inheritance Coupling

- **Coupling:**
  - **Inheritance coupling** – the features from a subclass inherited from its superclass
    - ***High inheritance coupling*** (i.e. Subclass inherits more its superclass features) ***= good design***

OO Design Concepts: Exercise 2

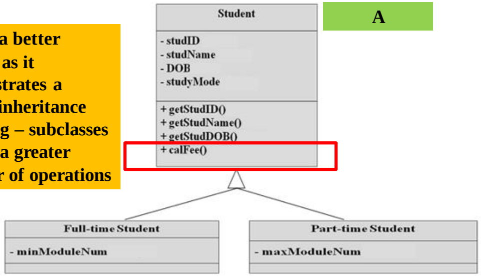
- Which of the following demonstrates a better design? Why?



OO Design Concepts: Exercise 2 Feedback

- Which of the following demonstrates a better design? Why?

This is a better design, as it demonstrates a higher inheritance coupling – subclasses inherit a greater number of operations



OO Design Concepts: Exercise 3

- Method overloading? Or overriding?

Overriding is to give a specific implementation to the inherited method of parent class.

Overriding requires base and child classes

```
class Student{
  public int calFee( ) {
          return 9000;
  }
}
class PartTimeStudent extends Student{
  public int calFee( ) {
          return 4500;
  }
  public static void main(String args[]) {
    Student obj = new PartTimeStudent ();
    int fee= obj. calFee();
    System.out.println(“part time fee is: "+ fee);
  }
}
```

OO Design Concepts: Exercise 3.1

- Differences between overriding and overloading?

Overloading:
a class can have more than one static method of same name.
overloading is being done in the same class whereas overriding requires parent and child classes.

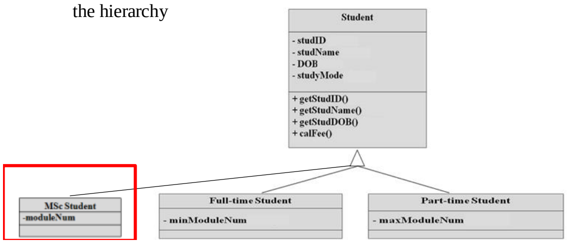
```
class Student{
private String email;
private int yearOfStudy;
  public void updateDetails( String em) {
          em = email;
  }
  public void updateDetails(int year) {
          yearOfStudy  = year;
  }
  public static void main(String args[]) {
    Student obj = new Student();
    obj .updateDetails(“john@gmail.com”);
    obj .updateDetails(4);
  }
}
```

OO Design Concepts: Exercise 3.1 Feedback

- Differences between overriding and overloading:
  - for overloading, methods are in the same class, whereas overriding requires parent and child classes.
  - Overloading happens at compile-time while Overriding happens at runtime
  - **Static binding** is being used for overloaded methods and **dynamic binding** is for overriding methods.
  - Overloading gives better performance as the binding of overridden methods is performed at runtime.
  - Argument list should be different for method overloading whereas argument list should be same in method Overriding.

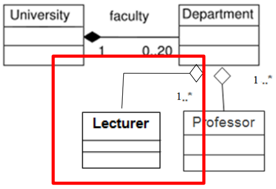
Design Concept: Inheritance for Reusability

- **Generalisation/Specialisation (Inheritance)**
  - allows the creation of new specialised classes when needed, as new specialised subclasses will inherit the characteristics of existing superclasses; e.g. class, MSc Student, can be added to the hierarchy



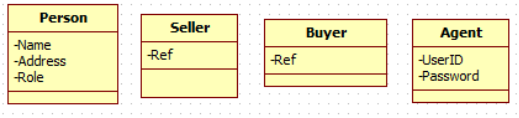
Design Concept:
Encapsulation for Reusability

- **Aggregation and composition to encapsulate components**
  - encapsulates a group of classes collectively for a **reuse subassembly**, as a complex **whole** is made of similar components (**parts**); more parts can be added to the whole component



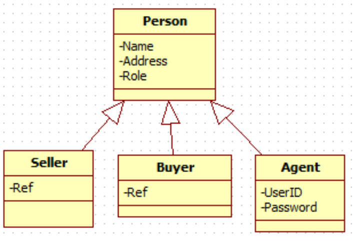
Design Principle:
Favour Composition/aggregation over Inheritance 1

- Example: A buyer, seller or an agent in an estate agent appointment booking system.



Design Principle:
Favour Composition/aggregation over Inheritance 2

- Question: Is the following via inheritance a good design?



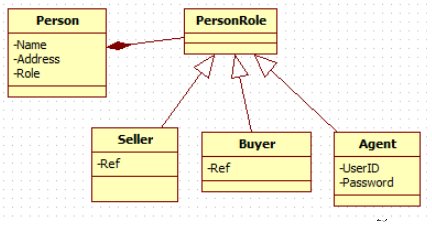
Design Principle:
Favour Composition/aggregation over Inheritance 3

- Criterion for inheritance: **when a subclass expresses "is a special kind of" and not "is a role played by a":** A buyer, seller or an agent is a role a person plays. Buyer, Seller and Agent are special kinds of person roles.

- Criterion for inheritance: **An instance of a subclass never needs to become an object of another class:** A instance of a subclass of Person could change from Buyer to Seller to Agent over time.

Design Principle:
Favour Composition/aggregation over Inheritance 4

- **Design with composition/aggregation** over inheritance: designs can be made more reusable and simpler by favouring composition



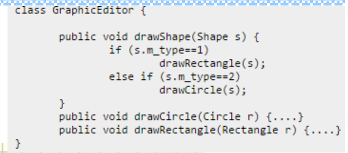
Design Principle: Open-Closed 1

- **Open-Closed Principle**: Software entities should be open for extension, whilst keeping closed for modification
  - **Open For Extension** - The behaviour of the module can be extended to meet new requirements
  - **Closed For Modification** – change to the source code of the module should be kept to a minimum
  - Via: Abstraction, Polymorphism, Inheritance, Interfaces

Source: Effective Java, by Joshua Bloch

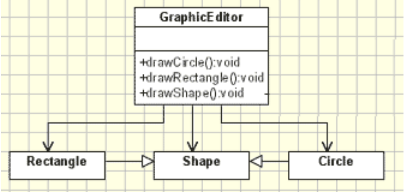
Design Principle: Open-Closed 2

- Example: Graphic editor



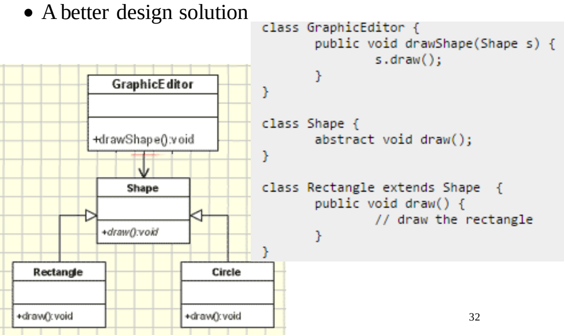
```
class GraphicEditor {

        public void drawShape(Shape s) {
                if (s.m_type==1)
                        drawRectangle(s);
                else if (s.m_type==2)
                        drawCircle(s);
        }
        public void drawCircle(Circle r) {....}
        public void drawRectangle(Rectangle r) {....}
}
```



Design Principle: Open-Closed 2.1

- A better design solution



```
class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
}
```

Design Principle: Liskov Substitution 1

- **Liskov Substitution principle:** functions that use references to base (super) classes must be able to use objects of derived (sub) classes without knowing it
  - 'In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.) – *Barbara Liskov'*

- Question: Can a Square class inherit from a Rectangle class, as a square is a type of rectangular shape?

Design Principle: Liskov Substitution 1.1

- No, a mathematical square might be a rectangle, as the area can be calculated by setting width and height. However, a Square object is not a Rectangle object, because the behaviour of a Square object is not consistent with that of a Rectangle object, as the width and height may differ for a Rectangle object whereas width and height are identical for a Square object. They behave differently, for example, when calculating their areas.

Example: LSP
http://prasadhonrao.com/solid-principles-liskov-substitution-principle-lsp/