Contents **Topic 09 - OOP Review – Features, Techniques, Practices and Principles**

I OOP Basics, Classes and Objects, Refactoring

Programming - Procedural Approach vs OO Approach

Procedural Approach	Object-Oriented Approach
Specify what tasks to do in each step.	Specify who performs what tasks in each step.

World objects vs OO Programs

The world : Objects **communicate** to complete some tasks

OO Programs : Objects **communicate** to provide some functions to users

Why OOP?

- State-of-the-art, popular approach in the industry
- To handle complexity
- To ease change, e.g., Encapsulation

Relationship between classes (Ref Topic03: A class is the template / blueprint from which objects are made)

- Inheritance ("is-a") [See topics 04]
- Interface ("Implements-a") [See topics 05]
- Dependency ("uses-a") [Next page]

Relationship between objects

- Association ("Knows-a") [Next page]
- Aggregation and Composition ("Has-a") [Next page]

How to maintain good quality of an OOP program - Refactoring

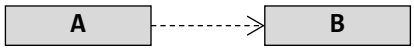
Refactoring

At the beginning, nobody wanted to write poor code;
Gradually, our code starts to look ugly (code smell).
One day we find that it is hard (then towards impossible) to maintain or change.

Refactoring: Improve code structure, without changing its external behavior
- Remove unhealthy dependencies between classes or packages
- Solve for bad class / method responsibilities; Reduce duplicate code and confusion
- Programmers ***refactor continuously*** to keep code as clean, simple, and expressive as possible.

II. The Dependency, Association, Aggregation, Composition relationships

Dependency ("Uses a ")



- We say "Class A [the client] depends on Class B [the supplier]" if:
  Any change to B requires A to be changed and/or ***recompiled***.
- Eg. 1. B is the superclass of A; or B is the interface implemented by A
  2. B is the type of
     a field of class A,
     a parameter of a class A method, or
     a local variable of a class A method
  3. a class B method or field is used / accessed from the code of class A, or
     a B object is created in class A's code

```
class A extends B {
      ..
}
```

```
class A implements B,C,D {
      ..
}
```

```
class A {
      private B b;
      ..
}
```

```
class A {
      private static B b;
      ..
}
```

```
class A {
      public void handleIt(B b) {
            xxx(b);
      }
}
```

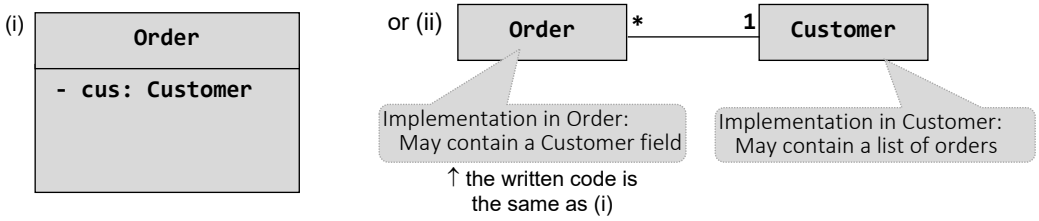
```
class A {
      public void handleIt() {
            B b = Company.getB();
            b.xxx();
      }
}
```

```
class A {
  public void handleIt(C c) {
    c.getB().xxx(); //xxx is B's method
  }
}
```

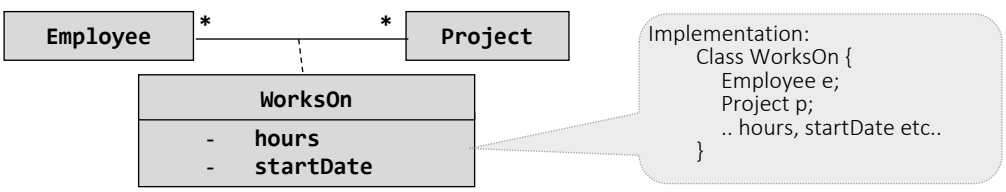
```
class A {
  public void handleIt() {
    (new B()).xxx();
  }
}
```

Association ("Knows a")

- An association is a type of object links.
- Often implemented as object fields, like other attributes
- Often expressed in two notations (depends on your intention):

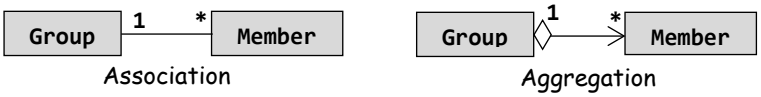


- **Association class**
  If the association has attributes, it should be implemented as a separate class:

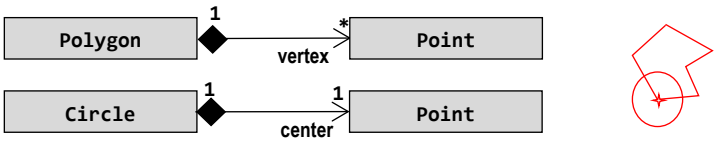


Aggregation and Composition ("Has a")

- **Aggregation** has vague semantics (Aggregations are also associations)



- **Composition** relates to creating and controlling the lifetime of another object
(also think about: Garbage Collection).
Single-owner: When B is composed by A, A owns B.



Suppose a vertex of a polygon p1 is (123, 45) , and
the center of a circle c1 happens to be (123, 45) as well;
Even so, p1 and c1 should link to two different Point objects

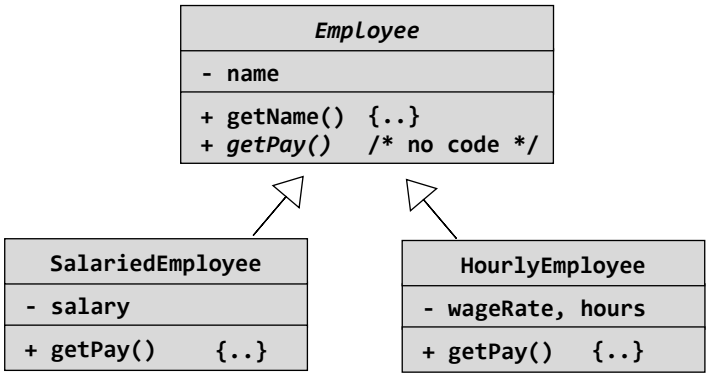
FAQ:
[Q] It is often vague to tell two classes' relationship (*Association? Dependency? Aggregation?..*) Then, use which one?
[A] Fact is: **More than one could be correct!** Therefore, see which can best match your intention (what to tell).

III. OO Techniques: Encapsulation, Abstraction, Generalization, Realization, Delegation

- **Encapsulation** [Topic03] : Combine data and behavior in one package Hide details from the users

- **Abstraction**: specify the framework and hide implementation details.
  - Give the developer a blueprint to follow.
  - Tell the user *what* instead of *how*
  - Eg. in Java: interfaces and abstract methods

- **Generalization**: provide common structure and behavior at an upper level of the hierarchy (available to lower level along inheritance)
  - Eg. We first design SalariedEmployee and HourlyEmployee as 2 separate classes,
    Then we observe common structure and behavior and rewrite as:

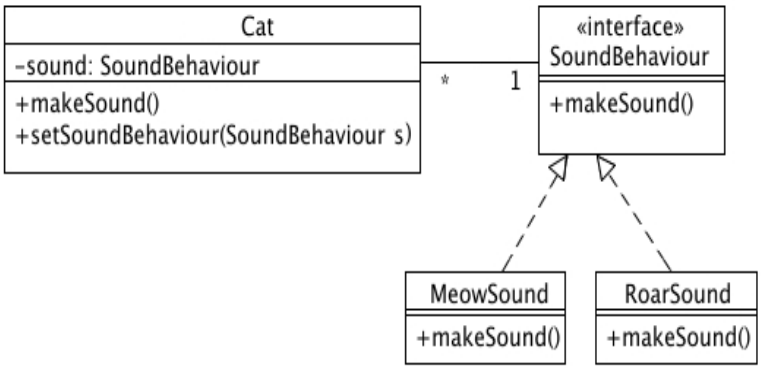


- **Realization**: provide implementation details to realize the abstract blueprint

Examples: .... implements Cloneable
.... implements Comparable <..>
class CmdAddSalary extends RecordedCommand
(CmdAddSalary: Concrete class; RecordedCommand: Abstract class)

- **Delegation:** An object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object.

  Advantage:
  Easily compose behaviors at run-time.
  Often used as an alternative to inheritance.

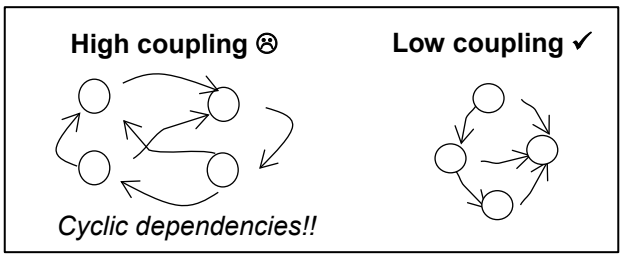



```
interface SoundBehavior {
   void makeSound();
}
-----------------------------------------------------------------------
public class MeowSound implements SoundBehavior {
   public void makeSound() { System.out.println("Meow"); }
}
-----------------------------------------------------------------------
public class RoarSound implements SoundBehavior {
   public void makeSound() { System.out.println("Roar!"); }
}
-----------------------------------------------------------------------
public class Cat {
   private SoundBehavior sound = new MeowSound();
   public void makeSound() { sound.makeSound(); }
   public void setSoundBehavior(SoundBehavior newsound) { sound = newsound; }
}
```

IV. Measurements: Cohesion and coupling

Coupling

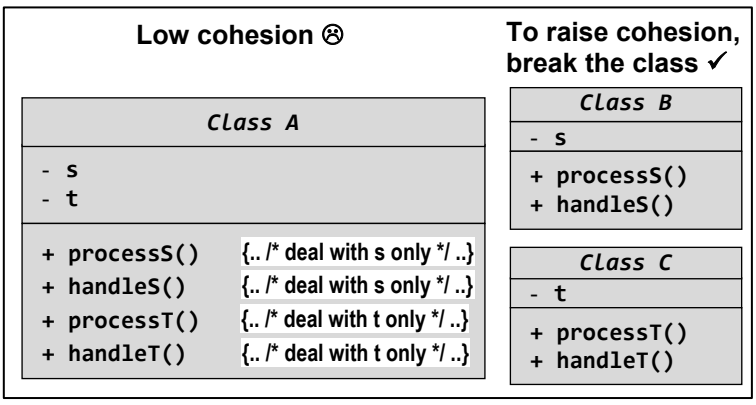
- The degree to which software components depend on each other
- Two classes (methods) are coupled if changing one of them leads to a change of the other.



Cohesion

- The degree to which a class/method has ONE and ONLY ONE purpose.
- A class (method) has a low cohesion if it does many unrelated things or too much work.
- One common mistake of OO design: Too few classes/methods! Measurement: *Cohesion!*

A good design should have **high cohesion**, **low coupling**



V. Principles

Liskov Substituion Principle (LSP)

- LSP: Subclasses must be substitutable for their superclasses.
- -ve case: A subclass's object is not 100% a superclass's object.
  (Partial code reuse only, eg. `BatteryDuck` inheriting a `Duck` which has the `eat()` method)

Dependency Inversion Principle (DIP)

- DIP: High level modules should not depend upon low level modules. Both should depend upon abstractions.
  Abstractions should not depend upon details. Details should depend upon abstractions.
- -ve case: Classes depend too much on each other, changing one will lead to the change of another.
- Example 1: Topic04 P.10

  `Employee` (Abstract), `SalariedEmployee`, `HourlyEmployee` (Details); `main`, `Company` (High level) etc. depends on `Employee` (Abstract)
- Example 2: `Lab12Q2` – `Person`, `Playables` (`Football`, `Piano`, + `Chess`!)

Open-Closed Principle (OCP)

- OCP: Modules should be open for extension, but closed for modification
- Robert Martin: "it should be easy to change the behavior of a module without changing the source code of that module."
- If OCP is applied well, then further changes can be done by adding new code, not by changing old code that already works. (Ref: Lab06 page 1 State-Pattern; Lab06-Q3: add "Disappeared Member")

VI. Design Patterns

- Design patterns are referred to as *best practices* to approach *common object-oriented design problems*
- Examples of design patterns
  - Singleton - Ensure a class has only one instance, and provide a global point of access to it.
  - State - Encapsulate a state as an object
    To allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - Command - Encapsulate a command as an object
    To let you support undoable operations.

[A famous book]

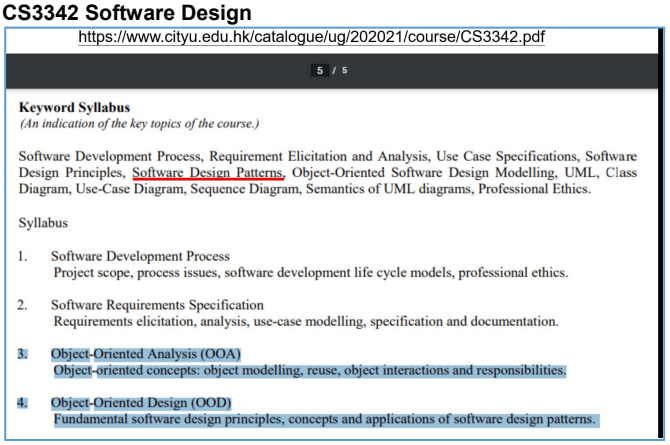
Design Patterns - Elements of Reusable Object-Oriented Software

by *Gang of Four*
Erich Gamma / Richard Helm /
Ralph Johnson / John Vlissides

First published: 1994;
Latest print: 2016 (44th printing)

Looking forward to learn more in your next course:
CS3342 Software Design

https://www.cityu.edu.hk/catalogue/ug/202021/course/CS3342.pdf



5 / 5

Keyword Syllabus
(An indication of the key topics of the course.)

Software Development Process, Requirement Elicitation and Analysis, Use Case Specifications, Software Design Principles, Software Design Patterns, Object-Oriented Software Design Modelling, UML, Class Diagram, Use-Case Diagram, Sequence Diagram, Semantics of UML diagrams, Professional Ethics.

Syllabus

1. Software Development Process
   Project scope, process issues, software development life cycle models, professional ethics.
2. Software Requirements Specification
   Requirements elicitation, analysis, use-case modelling, specification and documentation.
3. Object-Oriented Analysis (OOA)
   Object-oriented concepts: object modelling, reuse, object interactions and responsibilities.
4. Object-Oriented Design (OOD)
   Fundamental software design principles, concepts and applications of software design patterns.