OOD Process
Ch 2.1 – 2.5
Topics

1) What **phases** are used to **create software**?
2) How can we **identify** and **design classes**?
3) How can **classes** **work with other classes**?
Terminology

- **OOD**: ..
- **OOP**: ..
- **OOPS**: ..
- **Domain:**
  - **Ex**: Scheduling, accounting, vehicle control.
  - Encounter domain specific terminology. **Ex**: Bank, Pack, Battery, Module, Cell
Basic Software Creation Phases
Basic Software Creation Phases

- **Phases / Activities**
  1) Requirements
  2) Design & Implementation
  3) Verification
  4) Evolution
  - Done during any software development process such as Waterfall or Agile.

- **Evolution**
  - Change is inevitable for software.
  - OOD works well with software change because..
Phase 2: OO Design

- **Goal:** Identification of..

- **OOD Process**
  - An iterative process of *discovery* and *refinement*.

- **Product(s)**
  - of classes & relationships
  - Text description of classes

- **Time consuming, but a good design..**
  - "The sooner you start, the longer it takes"
Requirements Gathering

• **Goal**
  Create a complete description of..
  - Describes "what" not "how" (how is implementation).

• **End Product is a Functional specification**
  - completely describe the tasks to be performed
  - states constraints on development and operation

• **Software Developers must take a “spec” and then:**
  - Design the system
  - Implement a working system
Design is... [1]

- You need a good design to..
- You need to implement the system to know if..

- **Sloppy**: make many..
  - But cheaper during design than implementation!

- **Heuristic Process**
  - , vs fixed process
  - Use trial and error, analysis, refinement.
Implementation

● **Goal**
  Program, test, and deploy the software product.

● **Process Options**
  – **Skeleton Code**: Implement.. of full system first, then flush out code.
  – **Component Wise**: Implement one class/component at a time

● **Integration**
  – **Continual Integration**: Gradual growth of the system by continually integrating changes.
  – build parts separately, then..

(Fraught with peril!)
Class Design
Object & Class Concepts

- **Object**: A software entity with state, behaviours to operate on the state, and unique identity.

- **State**:..
  - **Ex**: pizza's size, car's colour, triangle's area

- **Behaviour**: The methods or operations it supports for..
  - Not all possible operations supported.
    - **Ex**: Pizza's don't support squaring their diameter.

- **Identity**: Able to..
  - **Ex**: same data, same operations, different copy.

- **Class**: .. of a set of objects with same behaviours and set of possible states.
Given a problem specification, how to find classes?

1. **Classes are often the..**
   
   When customers call to report a product's defect, the user must record: product serial number, the defect description, and defect severity.
   
   - Class names are..
     
     **Ex:** Customer, SerialNumber, ProductDefect
   
   - Avoid redundant "object" in names.
   
   - Some nouns may be properties of other objects.

2. **Utility classes: stacks, queues, trees, etc.**
   
   - **Ex:** MessageQueue, CallStack, DecisionTree
3. **Other possible classes**
   - **Agents:**...
     - Name often..  \(\text{Ex: Scanner}\)
   - **Events & transactions:** \(\text{Ex: MouseEvent, KeyPress}\)
   - **Users & roles:** Model the user.  \(\text{Ex: Administrator, Cashier, Accountant}\)
   - **Systems:** Sub systems, or the..
   - **System interfaces/devices:** Interact with the OS.  \(\text{Ex: File}\)
   - **Foundational Classes:**...
     Use these without modelling them.
The Evils of String

- Don't over use string!
  - ..
    (such as a name).
  - Strings are problematic to compare and store.
    Example: Spot the differences
    “CMPT 213” “cmpt 213” “CMPT213” “CMPT 213 ”
  - Even if going from string data (ex: text file) to string data (ex: screen output), ..
  - Suggestion: Create classes or enums like Department, Course, or Model
Enum Aside

- Imagine you are printing student names on paper. How to select horizontal vs vertical layout?
- *(Poor) idea for setting direction*
  ```java
  public const int HORIZONTAL = 0;
  public const int VERTICAL = 1;
  ```
  - May have other constants:
    ```java
    public const int NUM_PINK_ELEPHANTS = 0;
    ```
- *Use with functions*
  ```java
  public void printPage(int pageDirection);
  ```
  - The following generates..
    ```java
    printPage(NUM_PINK_ELEPHANTS);
    ```
Enum Aside

- Enums are better..

- Compiler enforces correct type checking
  ```java
  public void printPage(Direction pageDirection);
  Call it with:
  printPage(Direction.HORIZONTAL);
  ```

- Incorrect argument type generates error
  ```java
  printPage(NUM_PINK_ELEPHANTS);
  // Compiler error
  ```
Identifying Responsibilities

- **Responsibilities** (methods):
  Look for **verbs** in the problem description.
  - Assign each responsibility to..

  - **Easy Example**: Set the car's colour
    `myCar.setColour()`
  - **Harder Example**: Police comparing licence plates
    - `daCar.comparePlate(plate2)?`
    - `daPolice.comparePlate(plate1, plate2)?`
    - `daPlateComparator.compare(plate1, plate2)??`
Identifying Responsibilities (cont)

- **Responsibility Heuristic:**

  - **Example:** Adding a *Page* to a 3-ring *Binder*.
    - `myPage.addToBinder(daBinder);`
      Must get access inside the *Binder*.
    - `daBinder.addPage(myPage);`
      Does not need..
Identifying Responsibilities (cont)

- **Functionality often in the wrong class**
  - Ask yourself: “How can this object perform its functionality?”
  - ..

  - **A “code smell” where a class uses methods of another class excessively.**

- **Warning sign:**
  If a method..

  - **Solution:** Move it to that other class.
Relationships between Classes
Class Relations Overview

- **Dependency**
  - Where a class “uses” another class.
  - **Ex:** Any of our programs using `System`.

- **Aggregation**
  - Where a class “has-a” object of another class in it.
  - **Ex:** Car has-an Engine.

- **Inheritance**
  - Where a class “is-a” sub-category of another class.
  - **Ex:** Eagle is-a Bird.
“Use” (Dependency)

- **Dependency:**
  Class X *depends* on class Y if..
  - *Ex:* Changing Y's class name or methods.
  - If X knows of Y's existence, then..

- **Coupling:** Two classes are coupled if..
  - Coupling makes it harder to change a system because..
  - A design goal: Reduce coupling.

- *Ex: Which has lower coupling?*

```java
public String getName() {
    return name;
}
```

```java
public void printName() {
    System.out.println(name);
}
```
“Has” (Aggregation)

- **Aggregation**: When an object..
  - Usually through the object's fields.
- **Aggregation a special case of Dependency**:
  - If you *have* an object of type X, you must use *(depend on)* class X.
- **Multiplicity**:

  ```java
  class Person {
    private Car myCar;
  }
  
  class Album {
    private List<Song> songs;
  }
  ``
- **Foundational classes** *(String, Date, ...)* are..
"Is" (Inheritance)

- Class X inherits from class Y if..
  - X has at least the same behaviours (or more), and a richer state.
  - Y is the.. (base class)
  - X is the.. (derived class)

- Example
  - Car inherits from Vehicle.

- Heuristic
  - Use dependency (or aggregation) over inheritance when possible.
Summary

- **Terminology:** OOD, OOP, Domain
- **Phases:** Requirements, Design & implementation, Validation, Evolution
- **Class Design:** Object vs Class
  - Identifying classes via nouns.
  - Identifying behaviours via verbs.
- **Class Relationships:**
  - Dependency: uses, i.e., knows it exists.
  - Aggregation: has-a, usually through fields.
  - Inheritance: is-a