Topic 3: Object-Oriented Design Process

Part 2: Identifying Classes and Responsibilities Ch2.1-2.5

Class Design

Object Concepts

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

State: All information an object stores

Behavior: The methods or operations it supports for using and changing its state.

Identity: Able to differentiate two identical objects.

Class Concepts

- **Class**: Group of objects with:
  - same behaviors and
  - same set of possible states.
- An **instance** of a class: an object of the given class.

Object Class Identification

- No set process for object identification
- Relies on the skills experience and domain knowledge of the system designers
  - Unlikely to get it right the first time

- Typical Approaches:
  - Grammatical Approach
  - Use tangible things
  - Behavioral Approach
  - Use scenario based analysis

Identifying Classes

- Given a problem specification,
  - When customers call to report a product's defect, the user must record: product serial number, the defect description, and defect severity.

Classes could be

1. the nouns:
   - Class names should be singular
   - Try to avoid redundant object in names
   - Some nouns may be properties of other objects
2. Utility classes: stacks, queues, trees, etc.

Identifying Classes (2)

3. Other possible classes
   - Agents: Does a special task
     - Name often ends with “er” or “or”
   - Events & transactions: Ex: MouseEvent, KeyPress
   - Users & roles: Model the user.
   - Systems: Sub systems, or the controlling class for a full system
   - System interfaces/devices: Interact with the OS.
   - Foundational Classes: Date, String, Rectangle
     Use these without modelling them.

Identifying Responsibilities

- Responsibilities are the methods
  - Single Responsibility principle - Assign each responsibility to one AND ONLY one class
- Some examples are simple:
  - Set Person’s age
- Some examples are harder:
  - Add a Person to a Group
  - `person1.add(group1)`
  - `group1.add(person1)`
  - `groupCoordinator.add(person1, group1)`

Identifying Responsibilities (2)

- A general rule:
  - Avoid exposing the internals of an object just for access by another.

- Another example:
  - Adding a Page to a 3-ring Binder.

    ```
    myPage.addToBinder(myBinder);
    ```

    Must get access inside the Binder.

    ```
    myBinder.addPage(myPage);
    ```

    Does not need access to myPage

Identifying Responsibilities (3)

- Functionality often in the wrong class
  - Ask yourself:
    "How can this object perform its functionality?"
- Feature Envy - A "code smell" where a class uses methods of another class excessively. (FeatureEnvy.java)
- Warning sign:
  A method accesses the data of another object more than its own data.

- Solution: Move it to that other class.

Identifying Responsibilities (4)

Layering of Abstraction Levels

- Most large systems are divided into layers that do not interfere with each other.
  - Lowest levels are system level commands such as file access/manipulation, HCI, etc.
  - Highest levels (of abstraction) usually acts as a controller for the system
- Consider whether a responsibility can be handled by a lower (or higher) level of abstraction

Aside: String vs Enum

- Don't over use string!
  - Use only when a property is by nature a String
- Strings are problematic to compare and store.
- Even if going from string data (ex: text file) to string data (ex: screen output)
- Solution: Create classes or enums like Department, Course, or Model

(PersonDisplayer.java)

Class Relationships

… an Introduction

Class Design

- A Class Design typically include modules (components) and connectors
    - Good class design will consider both component coupling and cohesion

Cohesion Goals:

- High Cohesion desirable

Coupling Goals

- Low Coupling desirable

Class Relationships Overview

- Dependency
  - Class A “uses” class B.
- Aggregation
  - Class A “has-a” object of class B in it.
- Inheritance
  - Class A “is-a” sub-category of class B.

A ‘Uses’ B

- **Dependency**: Class A depends on class B whenever A may need to change if B changes.
  - Example: If class B changes class name or method definitions
  - Generally, if A knows B, then A depends on B.
- If a class A depends on class B, then we say that the two classes are **coupled**
  - Higher coupling makes it harder to change a system
  - General design goal is to reduce coupling whenever possible.

A 'Has a' B

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

- Foundational classes (String, Date, ...) are not considered part of aggregation

A 'Is a' B

- Class A inherits from class B whenever A is a sub-class (special case) of Y.
  - A has at least the same behavior as B (but could have more), and has a richer state
  - B is the superclass
  - A is the subclass
- Example:
  - Employee *is a* Person
    - Employee inherits from Person

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