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

- OOP:.. Object Oriented Programming
  - Object-Oriented building blocks like fields, methods, inheritance, encapsulation, polymorphism, etc.
- OOD:.. Object Oriented Design
  - Applying design principles to construct an object-oriented system which meets the needs of the user in a flexible and maintainable way.
- Domain:

  .. the industry or area of the system
  - 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
    classes represent stable problem domain concepts

Requirements Gathering

- Goal
  Create a robust description of..what the software product is to do
  - Describes "*what*" not "*how*" (how is implementation).
- Agile or Plan Driven
  - May be a backlog of user stories:
    descriptions of tasks that the user needs to do
  - May be a functional specification:
    completely describe the features
- Software Developers must take a “spec” and then:
  - Design the system
  - Implement a working system

OO Design

- Goal: Identification of..

  classes, their resposibilities, and relationships among them

- OOD Process
  - An iterative process of *discovery* and *refinement*.
- Product(s)
  - diagram of classes & relationships
  - Text description of classes
- Time consuming, but a good design..

  speed up implementation

  - "The sooner you start, the longer it takes"

OO Design – Challenges

Design is... [1]

- a wicked problem
  - You need a good design to.. implement the system
  - You need to implement the system to know if..
    you have a good design
- Sloppy: make many.. mistakes and mis-steps
  - But cheaper during design than implementation!
- Heuristic Process
  - use rules of thumb , vs fixed process
  - Use trial and error, analysis, refinement.

[1]: Code Complete 2, McConnell, 2004

Implementation

- Goal
  Program, test, and deploy the software product.
- Process Options
  - Skeleton Code: Implement.. minimal parts/features 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.
  - Big Bang implementation build parts separately, then.. assemble at once

    (Fraught with peril!)

Class Design

Object & Class Concepts

- Object: A software entity with state, behaviours to operate on the state, and unique identity.
- State:..all information an object stores
  - Ex: pizza's size, car's colour, triangle's area
- Behaviour: The methods or operations it supports for..using and changing its state
  - Not all possible operations supported.
    Ex: Pizza's don't support squaring their diameter.
- Identity: Able to..differentiate two identical objects
  - Ex: same data, same operations, different copy.
- Class: ..the type of a set of objects with same behaviours and set of possible states.

Identifying Classes

Given a problem specification, how to find classes?

1. Classes are often the... nouns

these things could be considered to be classes

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... singular
  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

Identifying Classes (cont)

3. Other possible classes

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

The Evils of String

- Don't over use string!
  - ..only use if your data type is by nature a 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),
    ..convert to non-string type internally
  - 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

  ```
  public const int HORIZONTAL = 0;
  public const int VERTICAL = 1;
  ```

  - May have other constants:

    ```
    public const int NUM_PINK_ELEPHANTS = 0;
    ```

- Use with functions

  ```
  public void printPage(int pageDirection);
  ```

  - The following generates..no compiler warning / error!

    ```
    printPage(NUM_PINK_ELEPHANTS);
    ```

Enum Aside

- Enums are better..

```
public enum Direction {
 HORIZONTAL,
 VERTICAL
}
```

can not do “new Direction”
always have to do Direction.HORIZONTAL

  - Compiler enforces correct type checking
    public void printPage(Direction pageDirection);

    Call it with:
    printPage(Direction.HORIZONTAL);
  - Incorrect argument type generates error
    printPage(NUM_PINK_ELEPHANTS);   // Compiler error

Identifying Responsibilities

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

  avoid exposing the internals of an object just for access by another

- Example:
  Adding a *Page* to a 3-ring *Binder*.
  - myPage.addToBinder(daBinder);

    Must get access inside the Binder.
  - daBinder.addPage(myPage);

    Does not need. internal access to page

Identifying Responsibilities (cont)

- 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.
- Warning sign:
  If a method..
  calls methods on another object more than the this object
  - 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..
  X may need to change if Y changes
  - Ex: Changing Y's class name or methods.
  - If X knows of Y's existence, then.. X depends on Y
- Coupling: Two classes are coupled if.. one depends on the other
  - Coupling makes it harder to change a system because..
    more parts need to change at once
  - A design goal: Reduce coupling.
- Ex: Which has lower coupling?

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

coupled to System, and PrintStream (System.out)

```
public void printName() {
    System.out.println(name);
}
```

“Has” (Aggregation)

- Aggregation: When an object.. contains the other 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:

1:0..1

```
class Person {
      private Car myCar;
}
```

1:*(a collection)

```
class Album {
      private List<Song> songs;
}
```

- Foundational classes (String, Date, ...) are..

not usually considered part of aggregation

"Is" (Inheritance)

- Class X inherits from class Y if..
  X is a sub-class (special case) of Y
  - X has at least the same behaviours (or more), and a richer state.
  - Y is the.. superclass (base class)
  - X is the.. subclass (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