CSC2620 Object Oriented Programming — Spring 2024
Unit 5 – Object Oriented Design

5.1 The Unified Modeling Language (UML)

- The **Unified Modeling Language** (UML) is a graphical language that models the class organization of object oriented programs.
- Each class that you create gets its own box, called a **UML class diagram**:

Class name
+ Public Fields : `type` - Private Fields : `type`
+ Public Methods(`varName : type`) : `ReturnType` - Private Methods(`varName : type`) : `ReturnType`

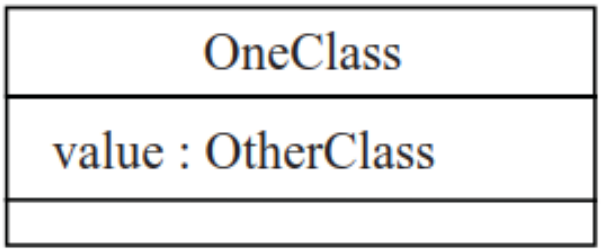
- Each class diagram has three compartments.
  1. The top compartment, holds the name of the class, boldface and centered. If the class is an interface, put `<<interface>>` above the class name.
  2. The middle compartment, holds the names and accessibility (`private` or `public`) of any fields of the class.
  3. The bottom compartment, holds the names and accessibility of any methods of the class, including their **parameters** (arguments).

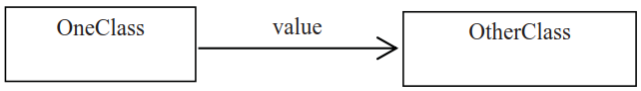
5.1.1 UML Syntax

- Attributes:
  `<visibility> <name> : <type> <multiplicity>`
- `<visibility>` categories:
  - `'+'` - public
  - `'-'` - private
  - `'#'` - protected
  - `'~'` - package
- `<multiplicity>` categories:
  - `'n'` - exactly n
  - `'*'` - zero or more
  - `'m..n'` - between m and n
- Methods:
  `<visibility> <name> ( <par1> : <type1>, <par2> : <type2> , ... ): <returntype>`
- Constructors (see grade book example):
  `<<Constructor>> ( <par1 > : <type1>, <par2> : <type2>, ... )`

5.1.2 UML Relationships

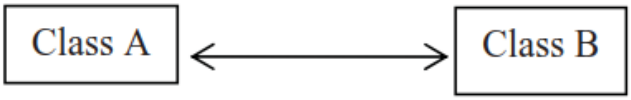
- As well as denoting individual classes, Class diagrams denote relationships between classes. One such relationship is called an 'Association'
- Other relationships:
  - Associations - One class uses instances of another class
    - Simple Association



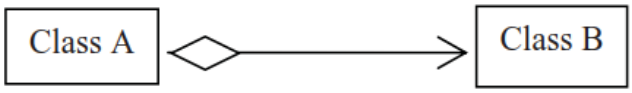


    - Association with multiplicity

    - Bidirectional Association



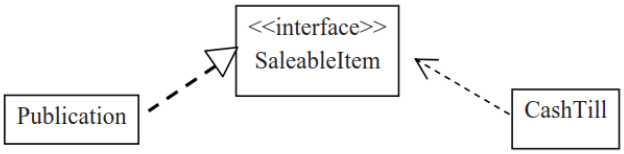
  - Stronger relationships
    - Aggregation - Class B *belongs to* Class A, but Class B can stand on it's own (small distinction from simple association)



    - Composition - Class B *is a part of* Class A, and is an integral part of Class A's utility

  - Inheritance - Class A *inherits from* Class B

  - Interface



- UML also has special diagram syntax for *packages* (outside scope of the class).
- UML also allows "object diagrams", which describe the data currently contained within objects. They are used to describe a moment in time in the execution of a program.

5.2 Other Misc. Rules

- `final` (and other constant) variable names should be written in `ALL_CAPS`.
- `static` methods and fields are indicated by underlining
- `abstract` methods and classes are *italicized*

5.3 An Example: Building a Grade Book

- Assume we have the following problem:

```
  Write a program that mimics the functionality of a grade book
which can, for a particular course, keep track of the students
within the course and their grades on various assignments.
```

1. Design a class structure.
2. Decide on an interface.
3. Design driver method (main).
4. Implement class(es).

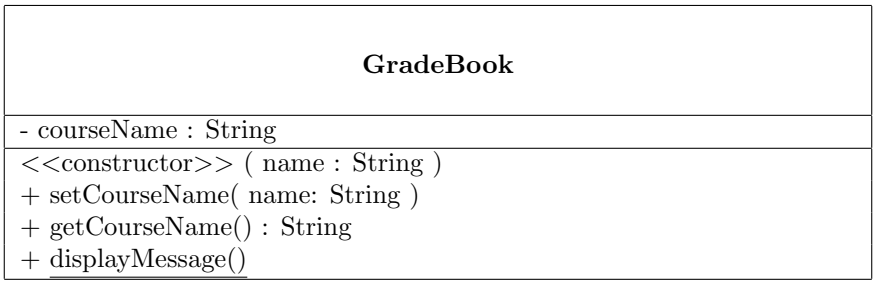
Testing class:

```
// A test program to create a class called GradeBook and test
//  its functionality
import java.util.Scanner; // We will need to take integer input

public class GradeBookTest
{
  public static void main( String[] args )
  {
    // Declare a Scanner to take input
    Scanner input = new Scanner( System.in );

    // Create an instance of the GradeBook class
    GradeBook myGradeBook = new GradeBook();

    // Output the initial value of the course name
    System.out.printf( "Initial course name is %s\n\n",
        myGradeBook.getCourseName() );

    // Prompt for and read in a new course name
    System.out.println( "Please enter the course name: " );
    String newName = input.nextLine();    // Read a line of text
    myGradeBook.setCourseName( newName ); // Use the set function
    System.out.println();                 // Output a blank line

    // Display the welcome message after specifying course name
    myGradeBook.displayWelcomeMessage();
  }
}
```

GradeBook class:

```
// The GradeBook class for maintaining a teacher's grade book
//   and manipulating student grades

public class GradeBook
{
  private String courseName;  // private field courseName of type String

  // constructor: method called when class instance is created
  //  Note: Constructors can take input arguments...which means
  //        that the instantiation (= new ClassName()) must be
  //        passed an argument...see: Scanner declaration.
  public GradeBook( String name )
  {
    courseName = name;
  } // end constructor

  // Method used to set the course name
  public void setCourseName( String name )
  {
    courseName = name;
  } // end setCourseName

  // Method to retrieve the course name
  public String getCourseName()
  {
    return courseName;
  } // getCourseName

  // Display welcome message
  public static void displayWelcomeMessage()
  {
    // Display a welcome message: Notice the call to the
    //  get method.
    System.out.printf( "Welcome to the grade book!" );
  } // get displayWelcomeMessage
} // end of class GradeBook
```

What would the UML class diagram look like for our GradeBook class?



5.3.1 Using UMLet

- Please extract your UMLet diagrams to a .pdf (preferably) or a .png. Do not turn in a UMLet file.
- I will post my solution to the practice online in both .pdf and UMLet file forms. Please review the syntax.
- Your UMLet diagrams must be planar (no arrows crossing each other) and be well organized and easy to read.

5.4 Practice

1. Find the `PizzaShoppe` project in the CS Examples repo.
2. Draw out the UML class diagrams for each class involved (including the `main` class). You do not need to worry about diagramming imported Java classes, just the user-created ones.
3. Add relationships between the classes.
4. Submit this to Classroom for 2 points.

5.5 Large Scale Object-Oriented Design

5.5.1 Identify Program Requirements

- Some software engineers specialise in the task of Requirement Analysis, which is the task of clarifying exactly what is required of the software. Often this is done by iteratively performing the following tasks

  1. Interviewing clients and potential user of the system to find out what they say about the system needed,
  2. Documenting the results of these conversations,
  3. Identifying the essential features of the required system,
  4. Producing preliminary designs (and possibly prototypes of the system),
  5. Evaluating these initial plans with the client and potential users,
  6. Repeating the steps above until a finished design has evolved.

- We'll focus on steps 3 and 4.
- Software design is a detailed and complex problem that takes practice to perfect.
  - A novice chess player may know all the rules but it takes experience to learn how to choose good moves from bad moves and experience is essential to becoming a skilled player.
  - Experience is essential to becoming skilled at performing user requirements analysis and in producing good designs.
- Starting with a *problem specification* we will work through the following steps to design a feasible and elegant solution to a problem.
  - Listing Nouns and Verbs
  - Identifying Things Outside The Scope of The System
  - Identifying Synonyms
  - Identifying Potential Classes
  - Identifying Potential Attributes
  - Identifying Potential Methods
  - Identifying Common Characteristics
  - Refining Our Design using CRC Cards
  - Elaborating Classes
- We will perform an example on a seemingly complex problem (this example is in your book, Chapter 6).

5.5.2 The Problem Specification

- The problem for which we will design a solution is 'To develop a small management system for an athletic club organising a marathon.'
- For the purpose of this exercise we will assume preliminary requirements analysis has been performed and by interviewing the club managers, and the workers who would use the system, the following textual description has been generated.

  The 'GetFit' Athletic Club is organizing their first international marathon in the spring of next year. A field comprising both world-ranking professionals and charity fundraising amateurs (some in fancy dress!) will compete on the 26.2 mile route around an attractive coastal location. As part of the software system which will track runners and announce the results and sponsorship donations, a model is required which represents the key characteristics of the runners (this will be just part of the finished system).

  Each runner in the marathon has a number. A runner is described as e.g. "Runner 42" where 42 is their number. They finish the race at a specified time recorded in hours, minutes and seconds. Their result status can be checked and will be displayed as either "Not finished" or "Finished in hh:mm:ss".

  Every competitor is either a professional runner or an amateur runner. Further to the above, a professional additionally has a world ranking and is described as e.g. "Runner 174 (Ranking 17)".

  All amateurs are fundraising for a charity so each additionally has a sponsorship form. When an amateur finishes the race they print a collection list from their sponsorship form.

  A sponsorship form has the number of sponsors, a list of the sponsors, and a list of amounts sponsored. A sponsor and amount can be added, and a list can be printed showing the sponsors and sponsorship amounts and the total raised.

  A fancy dress runner is a kind of amateur (with sponsorship etc.) who also has a costume, and is described as e.g. "Runner 316 (Yellow Duck)".

5.6 Working With the Problem

5.6.1 List Nouns and Verbs

- The first step of analysis of the problem is to list the nouns and verbs.
- List nouns in their singular form, and use nouns and verbs together in phrases when appropriate (e.g. "print receipt" is a better list item than simply saying "print").
- What nouns and verbs are prominent in our description above?

5.6.2 Identifying Those Things Outside the Scope of the System

- Almost equally as important as identifying what you want your system to do is to identify those things outside the scope of your system.
- Identify tasks in the project description that are performed by users while using the system and those performed by users while not using the system.
- K.I.S.S. (keep it simple, silly)

5.6.3 Identifying Synonyms

- Are there any synonyms among your list of nouns? We don't want to model the same attributes and behaviors more than once.

5.6.4 Identify Potential Classes, Attributes, and Methods

- Look through your list of nouns. Some will identify objects, while others identify attributes of objects.
- Start to tease out which nouns represent the objects that need to be modeled in your system.
  - Look for multiple behaviors (verbs) and attributes (nouns) that can be packaged under other nouns (classes).
  - Any noun which you think is associated only with a piece of data and not with behaviors can be an attribute of a larger class.

5.6.5 Identify Common Characteristics

- Does a natural inheritance hierarchy exist?
- Do more than one of your identified classes share common characteristics?
- If so, what are these characteristics? Which of the classes is "broader"?

5.7 Expansion and Elaboration

- Once you have your classes, attributes, behaviors, and relationships identified, use the UML to elaborate on your classes.

5.8 Practice

- Practice designing and implementing software based on general requirements descriptions. Follow the process described in these lecture notes. **You are given the following project description from a client:**

  We are the Andover Accountants Accounts Agency, a local company that helps clients maintain their banking information and stock portfolios. We'd like you design for us a piece of software to help us manage our clients and their accounts. We'd like to have several different clients managed by the system.

  Each client can have 0 or more bank accounts, and each account can have up to two clients' names on them. For each client we need to store her social security number and her name. Each client also has a portfolio, which is made up of stocks and bonds.

  We should be able to view a client's stock portfolio with ease. Each stock and bond should have a way of displaying the dividend payout from the previous quarter. Each bond should have a yield (the monetary return on the bond), and each stock should have a share price (the price for a single share). We should be able to get from the client how many shares of each stock she has.

  An account should have an associated bank, an account number, an account history, and a balance. A user should be able to view the account history for a given month and year, and the account's history should be displayed for that month and year, providing an account statement.

  The client can have two types of accounts: savings accounts, which have an interest rate, and a checking account, which keeps track of the current check number (which is the last check the client wrote). The client should be able to make withdrawls, deposits, and write checks through your system. All of these transactions should be part of the appropriate account's history.

  In addition, the software should allow us to add accounts, add clients, assign accounts to clients, build portfolios with stocks and bonds, and assign portfolios to clients (each client can have a portfolio, but a portfolio has only one associated client).

- This is supposed to be a big system. Use the steps in this unit to build a detailed UML class diagram of your system. You can work in groups to accomplish this task.

- Submit your design to Classroom for 2 points.

- Implement it.