Undergraduate Events

Events this week

Public Speaking 101
Date: Mon., Jan 25
Time: 5 – 6 pm
Location: DMP 101

CS Co-op Info Session
Date: Wed., Jan 27
Time: 5:00 – 6:30 pm
Location: DMP 201

OR
Date: Thurs., Jan 28
Time: 12:30 – 2 pm
Location: DMP 201

Undergrad Research Poster Competition
Date: Thurs., Jan 28
Time: 3:30 – 5:30 pm
Location: Atrium, ICICS/CS Building

FOOD WILL BE PROVIDED!

Events next week

EADS Info Session
Date: Mon., Feb 1
Time: 3:30 – 5 pm
Location: CEME 1202

RIM Info Session
Date: Thurs., Feb 4
Time: 5:30 – 7 pm
Location: DMP 110

Administrivia

- Lecture slides (day by day) are on the web:
- Reminder: Assignment #1 is due this week
  - Due Thursday, January 28th, 10:00pm
Class Design II: Class Diagrams

You should be able to:

• interpret UML class diagrams to identify relationships between classes
• draw a UML class diagram to represent the design of a software system
• describe the basic design principles of low coupling and high cohesion
• design a software system (expressed in UML) from a given specification that adheres to basic design principles (low coupling and high cohesion)
• identify elements of a given design that violate the basic design principles of low coupling, high cohesion

Reading:
2nd Ed:
Chapter 9: 9.1, 9.2
Chapter 17: 17.2, 17.3, 17.4
3rd and 4th Ed:
Chapter 8: 8.1, 8.2
Chapter 12: 12.2, 12.3, 12.4

Some ideas in this section come from:
“Object-Oriented Software Development Using Java”, Xiaoping Jia, Addison Wesley, 2002

Where are we?

• We have seen
  • how to design a single class
    • define attributes, methods, invariants, pre/post-conditions
    • how to implement a class robustly using
      • exceptions
      • unit testing
• Now we are going to discuss
  • how to identify the classes we need in order to provide a solution to a problem
  • what relationships exist between these classes
  • good design principles
Where are we?

- The overall roadmap of the course…

<table>
<thead>
<tr>
<th></th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Design and implementation of a single class</td>
</tr>
<tr>
<td>Now</td>
<td>Design of multiple classes</td>
</tr>
<tr>
<td>Feb</td>
<td>Collections</td>
</tr>
<tr>
<td>Feb-Apr</td>
<td>Implementation techniques GUI Threads Streams</td>
</tr>
</tbody>
</table>

Software Design

- Difficult, interesting, and important phase of software development
- Based on the requirements we have defined for a given problem, we need to identify and define:
  - classes and their relationships
  - the attributes of each class
  - the behaviour of each class
  - the interactions between classes
- We focus on the functionality and static relationships, *not* on implementation details
Representing Design: UML

• To represent the structure of a software system we need to show:
  • its classes
  • the relationships between the classes
• UML (Unified Modelling Language) is graphical modelling language that is used to describe these.
• UML allows a user to describe different views (aspects) of a software system
  • static view of the components, how components are deployed to different machines, etc.
• We will focus on one type of UML diagram which is called a **Class Diagram** and describes the static structure (logical view) of the system

Class Diagram

• Describes the static structure of a system
  • its classes
  • relationships between classes
• Example of a class:

<table>
<thead>
<tr>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: string</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>+ addSale(s : Item) : void</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Relationship 1: Association

- **Association:**
  - a *structural* relationship that describes a connection between objects: each object of one type contains references to objects of the other type
  - Example: Unidirectional association
    - employee stores a reference of a department

```
| Department          | 1          | Employee
|---------------------|------------|----------------|
| - name: string      |            | - name: string,
| ...                 |            |   - empId: integer
| + addSale(s: Item) : void |            | ...            |
```

Bidirectional Association

- Indicates that both classes reference each other
- Shown with a line without arrows
- Example:

```
| Department          | 1          | Employee
|---------------------|------------|----------------|
| - name: string      |            | - name: string,
| ...                 |            |   - empId: integer
| + addEmployee(Employee) : void |            | ...            |
| ...                 |            | + getDept() : Department |
```

- 1 and * in the previous examples are called *multiplicities*
  - indicate the number of objects of this side that are associated by each object of the other side
  - * means any number
Relationship 2: Aggregation

- **Aggregation**:
  - a special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part)

- Example:

```
Inventory

... + addItem() : void ...
```

```
Item

- name : String
- description : String
... + getName() : String ...
```

Relationship 3: Composition

- **Composition**:
  - a form of aggregation, where the composite (whole) strongly owns the parts
  - when the whole is deleted (dies) the parts are also deleted (die)
  - a part is in exactly one whole (implicit multiplicity of 1)

- Example:

```
ChessBoard

... + movePiece() : void ...
```

```
ChessSquare

- colour : String
... + getColour() : String ...
```
Relationship 4: Dependency

- **Dependency**: A relationship describing that a change to the target element may require a change in the source element.
- **Example**:

```
Player
- name : String
... 
+ move() : boolean
...

Joystick
- position : int
... 
+ changePos() : void
...
```

Relationship 5: Generalization

- **Generalization**: An inheritance relationship, where a subclass is a specialized form of the superclass.
- **Example**:

```
Person
- name : String
- age : int
- SIN : int
+ toString() : String

Student
- major: String
- year: int
+ enrollInCourse() : boolean
+ toString() : String
```
Heuristics for Finding Classes

- We usually start with the problem description and map each *relevant* word as follows:
  - nouns → classes or attributes
  - is/are → inheritance
  - has/have → aggregation or association
  - other verbs → methods
  - must → constraint
  - adjective → attribute, relation

- This is called Abbott’s heuristics for natural language analysis
- This is not always very accurate but it provides a good start

Simple Design Example

- Problem Description:
  We want to simulate a simple betting game. In this game, a player with money to gamble makes a bet and then rolls a single die. If a 1 is rolled, the player wins an amount equal to the bet, otherwise (s)he loses the bet.

- Let us try to identify the classes and their behaviour …
- Nouns:
  - game, player, money, bet, die, amount, bet
- Verbs:
  - gamble, makes (a bet), rolls, wins, loses
Putting it Together

Player
money: double
name: string
make bet(): double

Game
bet: double

Die
roll(): int