<< Lecture 15 >>

Summary

- I started with a quick review of last lecture's main points: [very basic] class diagrams, understanding polymorphism through sets, and covariance-contravariance
- I spent some time of this lecture connecting class diagrams and object modeling to commercial software (i.e., *show me the money!!!!*)
- I started covering **linked data structures**, starting with the **node of a linked list**

Polymorphism: more examples

- We have mainly seen polymorphism and abstraction through the Animal hierarchy, but we also made a case for GUIs (graphical user interfaces)
- Geometrical shapes are another great example of polymorphism (one that has been used throughout the discussions)
    - Main issue: formulas are all different
        - Can't compare the perimeter of a circle, a triangle, and a rectangle, uh?
- Analytic functions
    - Analytic functions as classes and objects is a topic that arises in **symbolic computation** (a.k.a., symbolic algebra)
        - There is an ACM group dedicated to this: http://www.sigsam.org/
    - Analytic functions are the ones that are *differentiable* (i.e., we can get their *derivative*)
    - Main issue: arbitrary composition of functions
        - Addition, multiplication, functional composition, all lead to new analytic functions
        - Derivative: $(f + g)' = f' + g'$
        - Derivative: $(f \times g)' = f' \times g + f \times g'$
        - Derivative: $(f(g))' = f'(g) \times g'$
    - Software:
        - Wolfram's Mathematica and Maplesoft's Maple are the most popular commercial symbolic computation applications
        - The open source community offers quite a few decent packages, including Maxima (modern sequel to older Macsyma), SageMath, SymPy

Parenthesis: games

- Let us look at actual examples of applications using class/object diagrams
- Snowdrop engine: https://www.youtube.com/watch?v=dORoTIEOyEg
- Text RPG class diagram: http://members.gamedev.net/emmanuel_deloget/images/text_rpg_rule_system.png
- Space invaders: http://www.c-sharpcorner.com/UploadFile/mgold/SpaceInvaders06292005005618AM/Images/SpaceInvadersUML.jpg

Parenthesis: domain modeling

- A usual application can be split into three layers:
  - the **model**, which contains the data model
  - the **view**, which contains the user interfaces
  - the **controller**, which contains the
- The model can represent the world, as in the *domain* of the application
- There is a big community for domain modeling and design/model driven development
  - http://dddcommunity.org/
  - http://www.infoq.com/domain-driven-design/
  - http://www.agilemodeling.com/essays/amdd.htm
- You can describe a business model and develop an application for it
  - E.g., restaurant
    - https://www.youtube.com/watch?v=flzTsUw5zFc
    - http://retailprofitsystems.com/kitchen-nightmares-and-gordon-ramsay-recommend-dinerware/
    - https://www.lavu.com/how-ipad-pos-works
  - E.g., game http://www.adom.de/jade/javadoc-jade-007/index.html

Parenthesis: Homework 2 as MVC

- MVC (model-view-controller) can be seen in homework 2
- Can you draw a [simple] class diagram of the second problem of the homework?
- Some modeling will be part of the midterm
  - *Diagram* ⟶ *code* and *code* ⟶ *diagram* as well

- Wildfire simulators are advanced… and pricey!
  - http://www.simtable.com/purchasing-and-funding/

< Pointers and linked data structures >

Introduction to pointers

- In Java, variables whose type subtypes Object are *pointers*
- *Pointer*: an indication of a memory address, to an object or value that resides in memory
- Why do they matter? Objects can link objects

Example: Node in a list

```
class Node{

   // Fields: next is a pointer, size is some int
   public Node next;
   public int size;

   // Constructor: field next is not pointing to another Node
   public Node() {
      next = null;
      size = 1;
   }

   // Constructor: field next points to another Node
   // --the other Node might be point to other Nodes as well!
   public Node( Node nxt )   {
      next = nxt;
      size = nxt.size + 1;
   }

}
```

- **Interpretation**:
  - The empty constructor creates a Node that does not point to other nodes, and assigns size to **1**
  - The non-empty constructor creates a Node that points to another Node, and assigns size to 1 + next's size
  - This can create *chains* of Nodes that point to other Nodes in the chain
  - We actually refer to such chains as **list**s
  - Pointers are also called links, because they link one object to another
    - Thus, this data structure (well, more like the evolved version) is called **Linked List**

- Now, let us use Node:

```
// We construct a list of size = 4 nodes
Node a = new Node();
Node b = new Node(a);
Node c = new Node(b);
Node d = new Node(c);

// We traverse the list starting from the Node pointed at by variable d
for( Node j=d; j!=null; n=n.next ) {
   System.out.println( "Size: " + j.size );
}
```

- *Lesson*: we can create “big” stuff
- *Observation*: changing the fields of Node, we can have something like an Array or a Map (i.e., like Python's **dict**)
- We constructed a simplified version of a linked list
  - We will discuss how to *properly* construct one later
  - Accessing the Nodes to create a list breaks the principle of *encapsulation*!!!

Thinking about the linked list

- Linked lists store items much like an array
  - But with flexible length, unlike array
    - No need to preempt the size of the array! We can just make the list grow...
- Problem: where to put a new node?
  - At the head? At the end? In the middle?
  - Do we need objects sorted???
- Problem: how to retrieve data?
  - *i*-th possition? Or retrieve position by object?

<< Lecture 16 will resume from here >>