ECE450 – Software Engineering II

Today: **Key Principles of Software Architecture and Design (III)**

Object-Oriented Design

- The object-oriented transformation of the 1980's and 1990's was particularly profound, but it wasn't easy
  - Object-oriented development salesmen took advantage of the wave of enthusiasm
    - Objects were supposed to improve your performance tenfold
    - Promise of reuse: Plug in your classes anywhere you need them
  - Many people struggled to "get it"
    - …and wrote object-oriented programs just like they used to write structure-oriented programs
    - I.e., programs -> classes; functions -> methods; or…
    - I.e., copy all of your program and put it in the main() method of your class.
- Object modeling in the 1990's
  - The best organization for a software system is one that is cohesive in the problem domain, not in the solution space
    - Tends to isolate changes
    - Tends to make the program easier to understand

Object-Oriented Principles

- Encapsulation
  - Which embodies one of our now-familiar principles (information hiding)
  - Modern languages allow us to *enforce* encapsulation through access declaration (example: public vs. protected attributes)
- Inheritance
  - Declare new classes by extending old ones
  - We inherit all of the old attributes and methods, but are free to modify/override any of them, and to add new ones
- Polymorphism
  - Substitute one type for another without the caller needing to know
    - We can make a *Student.getGrade()* call without worrying if we're dealing with an *UndergraduateStudent*, a *GraduateStudent*, or a generic *Student*.

Object Modeling Method

- How do we even come up with the classes we will use in our system?
- Step 1: Object-Oriented Analysis
  - Analyze the problem domain
    - Identify problem domain classes and relationships between classes
    - Identify attributes and methods
    - Identify states and transitions
    - Sample object structures and interactions
  - At this level we are not programming! We are abstracting the real world
- Step 2: Object-Oriented Design
  - Use the analysis as the core of a solution to:
    - User interface design
    - Database design
    - Program design

Can we model everything?

- I'd like to see you try…
  - But the world is too complex for us to model it completely
- A full model of you should include:
  - Your basic information (name, gender, etc.)
  - Your background
  - Your family background and a trace to your ancestors
  - Your medical history
  - Your record of marks
  - Your fingerprints and other bio-prints
  - Your financial information
  - A list of your friends, crushes, enemies, and acquaintances
  - Your DNA
  - …
- We only model that which is relevant to the problem domain that we face
  - Though there **is** such a thing as the CYC project…

Limitations of the Object-Oriented Paradigm

- Fluids
  - Need to be handled with amounts, but amounts vary with temperature/pressure
- Temporal concepts
  - If we represent them with objects, we end up with awkward processes to handle them
    - Is a year the accumulation of 365 day objects?
  - If we don't, we may lose other advantages of object-orientation (such as encapsulation)
- Abstract concepts
  - Can we reduce "preference" to an object or attribute?
- And sometimes objects just get in the way
  - You need a quick script and Java insists on a full object structure
  - Hence the rise of non-dogmatic object oriented languages (e.g. Python)

Trying it out: A meeting scheduler

- Here's a relatively concrete problem. We have the task of developing a meeting scheduler.
  - Meetings have an organizer that may or may not attend the meeting, and a number of attendees
  - Some of the attendees are essential to the meeting, others are optional
  - Everyone has some preferences about when to meet, and some constraints
  - Some of the attendees are Big Cheeses and we want the scheduler to satisfy their preferences over those of others
  - Meetings usually have an agenda. Often, somebody takes the role of secretary, and produces the meeting's minutes
  - Arrangements for the meetings should usually happen electronically. Some people are annoyed by this and take a long time to respond to invitations
  - Meetings require a space, a duration, and in some occasions equipment (such as a projector or a company-owned laptop)
- What are the classes in this problem domain?
  - What are their attributes and methods?

Some observations…

- There was some unnecessary information that should not be modeled
- Not all of our classes will become classes in the system
  - But many of them will
- So far we're dealing with the *business logic*
  - Later on we should also consider *how* the system is going to support the methods of these classes
    - User interface objects
    - Databases
    - …
- Most likely we got some classes wrong and we didn't plan for change
  - E.g. the meeting is a videoconference and we need to book rooms both in Toronto and in Vancouver
    - And we're on different timezones
  - This is where patterns will come in handy
- PRACTICE THAT EXERCISE IN OTHER DOMAINS
  - I guarantee you it's going to come up in your exam

Summary of design principles

- Decomposition
  - Avoid workflow-based decomposition
- Information hiding
  - Each module/class hides its own secrets (data representations, algorithms, formats, lower-level interfaces)
- Minimize coupling
  - If we talk/share information, it's because the problem demands it
- Maximize cohesion
  - Each module/class does (close to) one thing only
- Extensibility
  - Open for extension, closed for modification
  - No, we haven't talked about this one
    - We will get back to it