The Object-Oriented Paradigm

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 2 — 08/25/2011

Happy Birthday to me!

Praise for Last Spring's Class

l1m5 Ben Limmer

@kenbod wow - your OO class has been incredibly helpful the past few weeks at my new gig!

Jul 11, 12:26 PM via web

Lecture Goals

- Introduce the object-oriented paradigm
  - Contrast it with functional decomposition
  - Discuss important concepts of object-oriented programming
- Discuss the difference between abstraction and encapsulation
  - This is VERY important
- Address the problem of requirements and the need to deal with change

Design Methods

- Ways of solving problems
  - Structured Design/Programming (a.k.a. functional decomposition)
    - "Think in terms of steps"
  - Functional Programming (a.k.a. declarative programming)
    - "Think in terms of functions and their composition"
  - Object-Oriented Design/Programming
    - "Think in terms of objects that do things"

Simple Problem: Display Shapes

- **Functional decomposition: break problem into small steps**
  - Connect to database
  - Locate and retrieve shapes
  - Sort the shapes (perhaps by z-order; draw background shapes first)
  - Loop through list and display each shape
    - Identify shape (circle, triangle, square?)
    - Get location of shape
    - Call function to display the shape at the given location

Functional Decomposition

- Decompose big problems into the functional steps required to solve it
  - For a very big problem, simply break it down to smaller problems
    - then decompose smaller problems into functional steps
- Goal is to slice the problems up until they are at a level of granularity that is easy to solve in a couple of steps
- Then arrange the steps into an order that solves all of the identified subproblems and, presto, the big problem is solved along the way
- Extremely natural approach to problem solving; we do this almost without thinking about it

Functional Decomposition: Problems

- There are two main problems with this approach to design
  - It creates designs centered around a "main program"
    - This program is in control and knows all of the details about what needs to be done and all of the details about the program's data structures
  - It creates designs that do not respond well to change requests
    - These programs are not well modularized and so a change request often requires modification of the main program; a minor change in a data structure, for example, might cause impacts throughout the entire main program

With respect to change…

- A process-based approach to solving problems does not lead to program structures that can gracefully react to change
  - And change in software development often involves a variation on an existing theme
    - display new types of shapes
    - change the way shapes are rendered
    - add new functionality to the program such as being able to move the shapes after they have been displayed
- In "main programs," these types of changes typically cause complexity to increase and require that lots of files have to be recompiled

Why do these problems exist?

- These problems occur with the functional decomposition approach because the resulting software exhibits
  - poor use of **abstraction**
  - poor **encapsulation** (a.k.a. **information hiding**)
  - poor **modularity**
- If you have poor abstractions and you want to add another one, it's often not clear how to do it (easily)
- If you have poor encapsulation and poor modularity, changes tend to percolate through the code since nothing shields dependencies from forming throughout the code

Why should we care?

- As the book says
  - "Many bugs originate with changes to the code"
- and
  - "Things change. They always do. And nothing you can do will stop change [from occurring to your software system]."
- We need to ensure that we do not get overcome by change requests; that we create designs that are resilient to change;
- Indeed, we want software designs that are "designed" to accommodate change in a straightforward manner; that is what OO A&D provides!

Start of a Journey (I)

- **What is the difference between abstraction and encapsulation?**
  - Any takers?
- How would you interpret the following statements if you heard them in casual (admittedly nerdy) conversation?
  - "That sound processing package offers a great set of abstractions!"
  - "Wow, that Employee class is horrible! There is no encapsulation!"

Start of a Journey (II)

- Identify which concept applies to the following statements
  - "I wonder if Java's Map class will do what I need?"
  - "I wonder if I can prevent users of my library from finding out that MyClass.id is implemented as a floating point number?"
  - "I like how I can decide at run time whether my List variable will point at an instance of LinkedList or ArrayList! I mean List's API is fine but it's nice to know that I have the flexibility of picking the more efficient implementation when my list size is small"
- I'm going to drill the definitions of these two terms and the difference between them into your head this semester! Why are these concepts so important?

Analysis

- Analysis is the phase of software development that occurs
  - before design when starting from scratch
  - that occurs first when responding to a change request during the maintenance of an existing system
- Its primary goal is to answer the following question
  - **What is the problem that needs to be solved?**
- Design is the phase that comes after analysis and its goal is:
  - **How am I going to solve the problem?**

Requirements

- Requirements for a software system are initially generated during the analysis phase of software development and are, typically:
  - simple statements of desired functional capabilities
    - "the system should allow its users to sort records by priority"
  - statements of non-functional capabilities
    - "the system should support 10,000 simultaneous users"
  - statements of constraints that must be met
    - "the system will comply with regulation XYZ at all times"

The Problem of Requirements (I)

- The problem? Experienced developers will tell you that
  - Requirements are incomplete and do not tell the whole story
  - Requirements are typically wrong
    - factually wrong or become obsolete
  - Requirements and users are misleading
- In addition, users may be non-technical and may not understand the range of options that could solve their problem
  - their ill informed suggestions may artificially constrain the space of solutions

The Problem of Requirements (II)

- The other problem with requirements is
  - "requirements **always** change"
- They change because
  - a user's needs change over time
  - as they learn more about a new problem domain, a developer's ability to generate better solutions to the original problem (or the current problem if it has evolved) will increase
  - the system's environment changes
    - new hardware, new external pressures, new techniques

The Problem of Requirements (III)

- Most developers view changing requirements as a bad thing
  - and few design their systems to be resilient in the face of change
- Luckily, this view is changing
  - agile software methods tell developers to welcome change
    - they recommend a set of techniques, technologies and practices for developers to follow to remove the fear of change
  - OO analysis, design and programming techniques provide you with powerful tools to handle change to software systems in a straightforward manner

The Problem of Requirements (IV)

- However, this does not mean that we stop writing requirements
  - They are incredibly useful despite these problems
  - The lesson here is that we need to improve the way we design our systems and write our code such that change can be managed
- Agile methods make use of "user stories"; other life cycle methods make use of requirements documents or use cases (dressed-up scenarios that describe desired functional characteristics of the system)
  - Once we have these things, and the understanding of the problem domain that they convey, we then have to design our system to address them while leaving room for them to change

The Problem with Functional Decomposition

- The book highlights a problem with code developed with functional decomposition
  - such code has **weak cohesion** and **tight coupling**
    - translation: "it does too many things and has too many dependencies"
- Example
  - void process_records(records: record_list) {
    - // sort records, update values in records, print records, archive records and log each operation as it is performed …

Cohesion

Code Complete by Steve McConnell; Microsoft Press, 1993

- Cohesion refers to "how closely the operations in a routine are related"
  - A simplification is to say "we want this method to do just one thing" or "we want this module to deal with just one thing"
- We want our code to exhibit **strong cohesion** (a.k.a. highly cohesive)
  - methods: the method performs one operation
  - classes: the class achieves a fine-grain design or implementation goal
  - packages: the package achieves a medium-grain design goal
  - subsystems: this subsystem achieves a coarse-grain design goal
  - system: the system achieves all design goals and meets its requirements

Coupling

Code Complete by Steve McConnell; Microsoft Press, 1993

- Coupling refers to "the strength of a connection between two routines"
  - It is a complement to cohesion
    - weak cohesion implies strong coupling
    - strong cohesion implies loose coupling
- With strong or tight coupling, a single change in one method or data structure will cause **ripple effects**, that is, additional changes in other parts of the system
- We want systems with parts that are **highly cohesive** and **loosely coupled**

Ripple Effects

- Ripple effects cause us to spend a long time doing debugging and system understanding tasks
  - We make a change and unexpectedly something breaks
    - This is called an **unwanted side effect**
  - If we have tightly coupled code we discover that many parts of the system depended on the code that changed
    - It takes time to discover and understand those relationships
- Once understanding is achieved, it often takes very little time to actually fix the bug

Transitioning to the OO Paradigm

- Rather than having a main program do everything
  - populate your system with objects that can do things for themselves
- Scenario: You are an instructor at a conference. Your session is over and now conference attendees need to go to their next session
  - With functional decomposition, you would develop a program to solve this problem that would have you the instructor do everything
    - get the roster, loop through each attendee, look up their next session, find its location, generate a route, and, finally, tell the attendee how to get to their next class
      - You would do everything, attendees would do (almost) nothing

Transitioning to the OO Paradigm

- The book asks
  - **Would you do this in real life?**
- And the answer is (hopefully) NO!
  - What would you do instead?
    - You would assume that everyone has a conference program, knows where they need to be next and will get their on their own
    - All you would do is end the session and head off to your next activity
    - At worst, you would have a list of the next sessions at the front of the class and you would tell everyone "use this info to locate your next session"

Compare / Contrast

- In the first scenario,
  - you know everything, you are responsible for everything, if something changes you would be responsible for handling it
  - you give very explicit instructions to each entity in the system
- In the second scenario,
  - you expect the other entities to be self sufficient
  - you give very general instructions and
  - you expect the other entities to know how to apply those general instructions to their specific situation

Benefits of the second scenario

- The biggest benefit is that entities of the system have their own responsibilities
  - indeed this approach represents a **shift of responsibility** away from a central control program to the entities themselves
- Suppose we had attendees and student volunteers in our session and that volunteers needed to do something special in between sessions
  - First approach: the session leader needs to know about the special case and remember to tell volunteers to do it before going to the next session
  - Second approach: the session leader tells each person "Go to your next session"; volunteers will then automatically handle the special case without the session leader needing to know anything about it
    - We can add new types of attendees without impacting the leader

As an aside...

- OO main programs tend to be short
  - On the order of create an object and send a message to it
  - See next slide…

```
 1 import wx
 2
 3 from ACE.GUI.Managers.RepositoryManager import RepositoryManager
 4
 5 class ACEApp(wx.App):
 6
 7     def OnInit(self):
 8
 9         bmp = wx.Image("images/ace_logo.png").ConvertToBitmap()
10         wx.SplashScreen(bmp, wx.SPLASH_CENTRE_ON_SCREEN | wx.SPLASH_TIMEOUT, 500, None, -1)
11         wx.SafeYield(None,True)
12         self.repoman = RepositoryManager()
13         return self.repoman.HandleAppStart(self)
14
15     def OnExit(self):
16         self.repoman.HandleAppQuit()
17
18 if __name__ == '__main__':
19     app = ACEApp(redirect=False)
20     app.MainLoop()
21
```

iOS Main Program

- Here's the main program of every iOS application in existence

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Android Main Program

- The main program of an Android app is even shorter in that it is **completely hidden** from the Android developer
- Instead, in the application manifest, you specify your initial activity and when your application launches, your activity's onCreate() method is automatically called

Revisiting the Shape system

- Recall our "display shapes" program from earlier in the lecture
  - How would you rearrange it to follow this new approach?

Foreshadowing

- The benefits we've been discussing are inherent in the OO approach to analysis, design and implementation that will be learning this entire semester
  - self-sufficient entities ➡ objects
  - "give general instructions to" ➡ code to an interface
  - "expect entities to apply those general instructions to their specific situation" ➡ polymorphism and subclasses
  - "add new attendees without impacting session leader" ➡ code to an interface, polymorphism, subclasses
  - shift of responsibility ➡ functionality distributed across network of objects

Perspectives in Software Development
(UML Distilled, Martin, Fowler, Addison-Wesley, 1999)

- Conceptual — What are the main concepts of the problem domain and what are each of those concepts responsible for? "A conceptual model should be drawn with little or no regard for the software that might implement it."
- Specification — What are the interfaces of the objects (derived from the concepts above) in the system. How are these objects used?
- Implementation — How do the objects fulfill their responsibilities?
- "Go to your next class" : a conceptual instruction that shields the requestor from how this command is actually carried out (implementation). The fact that we know that objects can respond to this command comes from the specification level.

The Object-Oriented Paradigm

- OO Analysis & Design is centered around the concept of an object
  - It produces systems that are networks of objects collaborating to fulfill the responsibilities (requirements) of the system
- Objects are conceptual units that combine both data and behavior
  - The data of an object is referred to by many names
    - attributes, properties, instance variables, etc.
  - The behavior of an object is defined by its set of methods
- Objects inherently know what type they are. Its attributes allows it to keep track of its state. Its methods allow it to function properly.

a.k.a. **features**

Return to Slide 34

Object Responsibilities

- In OO Analysis and Design, it is best to think of an object as "something with responsibilities"
  - As you perform analysis (What's the problem?), you discover responsibilities that the system must fulfill
  - You will eventually find "homes" for these responsibilities in the objects you design for the system; indeed this process can help you "discover" objects needed for the system
    - The problem domain will also provide many candidate objects to include in the system
    - This is an example of moving from the conceptual perspective to the specification and implementation perspectives

Objects

- Conceptual — a set of responsibilities
- Specification — a set of methods
- Implementation — a set of code and data

- Unfortunately, OO A&D is often taught only at the implementation level
  - if previously you have used OO programming languages without doing analysis and design up front, then you've been operating only at the implementation level
    - as you will see, there are great benefits from starting with the other levels first

Objects as Instances of a Class

- If you have two Student objects, they each have their own data
  - e.g. Student A has a different set of values for its attributes than Student B
- But they both have the same set of methods
  - This is true because methods are associated with a **class** that acts as a blueprint for creating new objects
  - We say "Objects are instances of a class"
- Classes define the complete behavior of their associated objects
  - what data elements and methods they have and how these features are accessed (whether they are public or private)

Classes (I)

- The most important thing about a class is that it defines a type with a legal set of values
- Consider these four types
  - Complex Numbers ➡ Real Numbers ➡ Integers ➡ Natural Numbers
- Complex numbers is a class that includes all numbers; real numbers are a subtype of complex numbers and integers are a subtype of reals, etc.
  - in each case, moving to a subtype reduces the set of legal values
- The same thing is true for classes; A class defines a type and subclasses can be defined that excludes some of the values from the superclass

Classes (II)

- Classes can exhibit inheritance relationships
  - Behaviors and data associated with a superclass are passed down to instances of a subclass
  - The subclass can add new behaviors and new data that are specific to it; it can also alter behaviors that are inherited from the superclass to take into account its own specific situation
- It is extremely desirable that any property that is true of a superclass is true of a subclass; the reverse is not true: it is okay for properties that are true of a subclass not to be true of values in the superclass
  - For instance, the property isPositive() is true for all natural numbers but is certainly not true of all integers

Classes (III)

- Inheritance relationships are known as **is-a** relationships
  - Undergraduate IS-A Student
- This phrase is meant to reinforce the concept that the subclass represents a more refined, more specific version of the superclass
- If need be, as we shall see, we can treat the subclass as if it IS the superclass. It has all the same attributes and all the same methods as the superclass and so code that was built to process the superclass can equally apply to the subclass

Classes (IV)

- Classes can control the accessibility of the features (see Slide 27) of their objects
  - Assume object A is an instance of class X; object B is an instance of class Y which is a subclass of X; object C is an instance of class Z which is unrelated to X and Y.
  - **Public** visibility of a feature of class X means that A, B and C can access that feature
  - **Protected** visibility of a feature of class X means that A and B can access the feature but C cannot.
  - **Private** visibility of a feature of class X means that only A can access the feature
- This ability to hide features of a class/module is referred to as **encapsulation** or **information hiding**; encapsulation is a topic that is broader than just data hiding

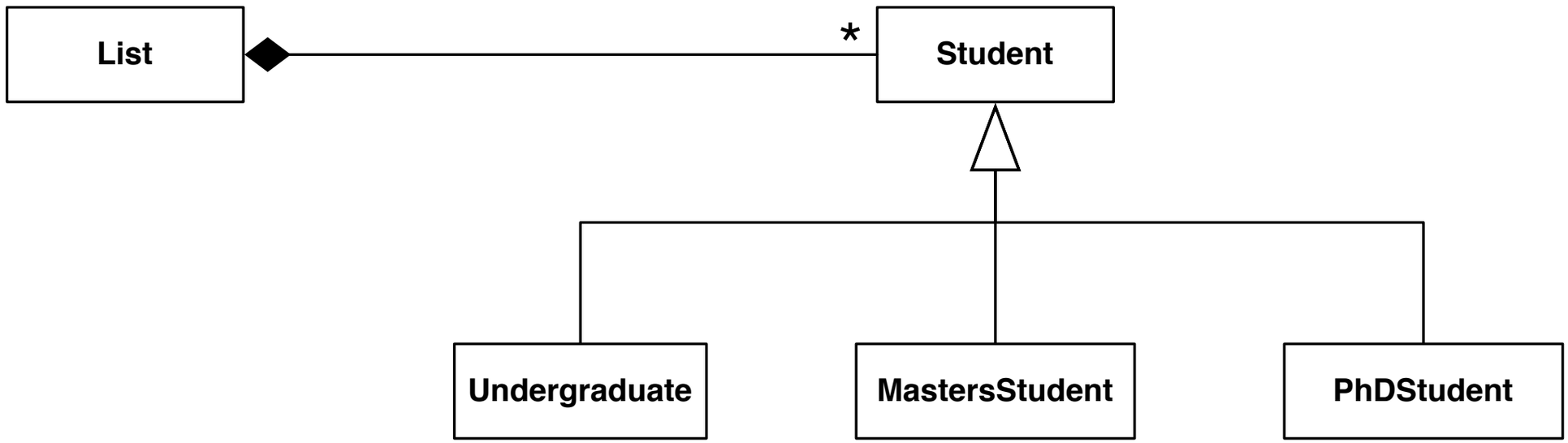
Classes (V)

- Classes can control how their objects are created and destroyed
  - OO Programming languages will (typically) provide "special methods" known as **constructors** and **destructors** (a.k.a. **finalizers**) to handle these two phases in an object's life cycle
    - Constructors are useful for ensuring that an object is properly initialized before any other object makes use of it
    - Destructors are useful for ensuring that an object has released all of the resources it consumed while it was active
      - Destructors can be tricky; in languages with garbage collection, an inactive object might hang around for a significant amount of time before the garbage collector gets around to reclaiming its space

One benefit of superclasses

- Treat all instances of a superclass-subclass hierarchy as if they were all instances of the superclass even if some are instances of subclasses
- Example
  - Suppose we have the classes, Undergraduate, MastersStudent and PhDStudent in a software system
  - We may have a need for acting on all instances of these three classes at once, for instance, storing them all in a collection, sorting by last name and displaying a roster of the entire university
  - Solution: **Make all three of these classes a subclass of the class Student;** You can then add all of the students to a single collection and treat them all the same

Example rendered in UML



Note: UML Notation will be discussed in Lecture 3

Another benefit of superclasses

- Not only can you group all instances of an object hierarchy into a single collection, but you can apply the same operations to all of them as well
  - In our example, any method defined in the superclass, Student, can be applied to all instances contained in our collection (the List of Students)
- On the following slide:
  - Student has a method called saySomething() which is overridden by each subclass to say something different
  - Yet look how clean the code is…

```java
import java.util.LinkedList;
import java.util.List;

public class Test {

   public static void main(String[] args) {

      List<Student> students = new LinkedList<Student>();

      students.add(new Undergraduate("Bilbo Baggins"));
      students.add(new MastersStudent("Aargorn"));
      students.add(new PhDStudent("Gandalf the White"));

      for (Student s: students) {
          System.out.println("" + s);
      }

      System.out.println();

      for (Student s: students) {
          s.saySomething();
      }

   }

}

```

The True Power: Clean Code!

- The most powerful code in the previous example was

```
for (Student s: students) {
    s.saySomething();
}
```

- Why?
  - You can add as many subclasses to the Student hierarchy as you want and this code **never has to change**!
  - **It doesn't even have to be recompiled**
  - Indeed, given the right techniques, a server running this code doesn't even need to be "brought down"; the new subclass can be dynamically loaded and this code will recognize instances of that subclass and do the right thing

Polymorphism (I)

- The previous example demonstrated polymorphism
  - which literally means "many forms"
  - in OO A&D it means that we can treat objects as if they were instances of an abstract class but get the behavior that is required for their specific subclass
    - The "many forms" refers to the many different behaviors we get as we operate on a collection of objects that are instances of subclasses of a generic, abstract class
- We will see many examples of polymorphism as we move forward in the semester and you will get a chance to try it out for yourself in Homework 1

Polymorphism (II)

- In the book, polymorphism is defined specifically as
  - ***"Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to"***
- As you can see, it is not an easy thing to define! But, it is very powerful and the "clean code" example should show why we as designers should strive to design OO hierarchies that allow us to write polymorphic code
  - There are other variations on polymorphism to learn, we will get to those in future lectures

Abstract Classes

- The classes that sit at the top of an object hierarchy are typically **abstract classes** while the classes that sit near the bottom of the hierarchy are called **concrete classes**
- Abstract classes
  - **define a set of generic behaviors** for a related set of subclasses;
  - act as **placeholders** for other classes defining method signatures that they must implement, defining method bodies for behaviors that should be the same across all subclasses, defining data that will be useful for all subclasses
  - In OO programming languages, abstract classes **cannot be instantiated**
    - instead you instantiate concrete classes but access them via the interface defined by the abstract class

Summary

- In this lecture, we have touched on a variety of OO concepts
  - Functional Decomposition vs. the OO Paradigm
  - Requirements and Change in Software Development
  - Objects, Classes (Abstract and Concrete)
  - Polymorphism and Encapsulation

Coming Up Next

- Homework 1: To be assigned today
- Lecture 3: UML
  - Read Chapter 2 of the Textbook
- Lecture 4: More review of fundamental OO A&D concepts