Lecture 2 - The Object Oriented Paradigm

CSCI4448 - Object Oriented Analysis and Design

Dana Hughes

University of Colorado - Department of Computer Science

January 16, 2014

Introduction

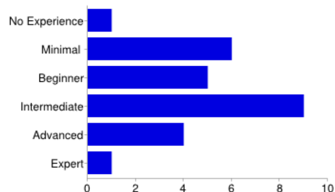
Announcements

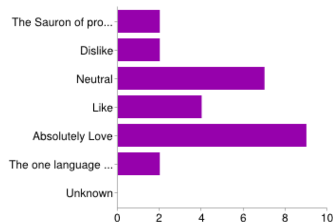
- If you haven't signed up for Piazza, please do so
  - If you haven't received an invite in your colorado.edu email, please email me
  - If you want me to use a different email account, please sent it to me
- If you know you can't make an exam, please email me as soon as possible
  - This applies to quizzes as well
- Expect the first homework tomorrow (on Piazza and website)

Objectives

- Lecture Goal: Introduce the Object-Oriented Paradigm
  - Contrast this with functional decomposition
  - Introduce and define object oriented concepts
    - Inheritance, abstraction, polymorphism, encapsuluation, delegation, etc.
  - Discuss how objects can deal with changes during and after software development

PL Survey Response - Java

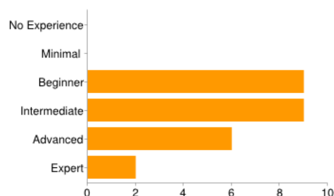


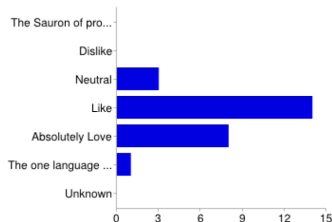


Java

- I was expecting experience to be more universal (i.e., fewer Beginner / Minimal)
- Shame, since Java's syntax reinforces many of the concepts
- And, I'm pretty familiar with implementing patterns with Java

PL Survey Response - C++

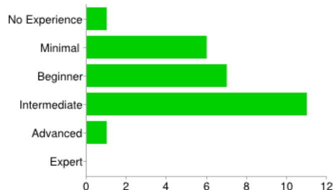


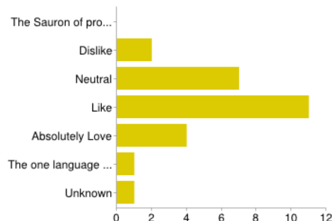


C++

- Seems to be the universal choice
- Apparently, no one's really having trouble with pointers
- One main issue is C++ supports multiple inheritance (most other languages support only single inheritance)
- Also, interfaces == pure virtual classes, might be confusing

PL Survey Response - Python

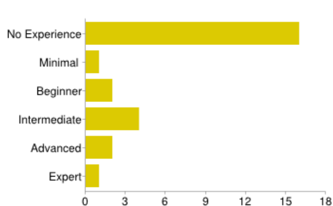


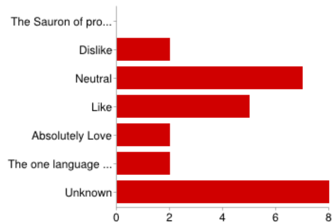


Python

- No Python experts?
- Probably don't need to have examples in this language, but the dynamic typing does introduce interesting issues

PL Survey Response - C#

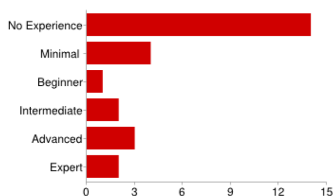


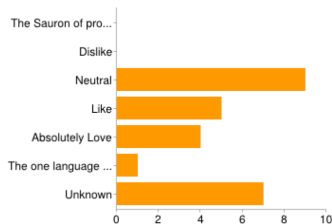


C#

- So, a perfectly normal distribution with experience
- But, preference is skewed a bit towards dislike

PL Survey Response - Objective-C

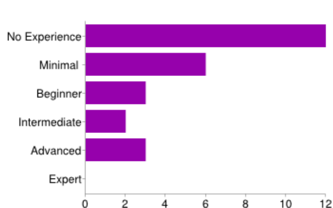


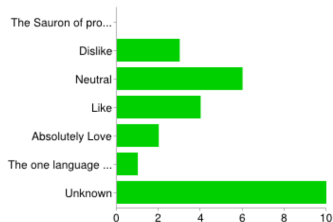


Objective-C

- Again, not too much universal experience
- But, no one *dislikes* the language
- Also, previous C experience applies here

PL Survey Response - Ruby

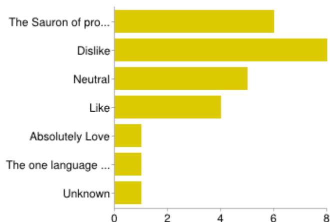




Ruby

- A little more experience here than Objective-C, but still not widely adopted

PL Survey Response - Others



- Scala doesn't seem to be some people's favorites
- Smalltalk, Perl, PHP, Eiffel unknown (one Eiffel expert?)
- Requests for Rust, IDL and CoffeeScript
- My leaning
  - Have examples in the slides in Java and C++
  - Have the same examples available in C#, Objective-C on the web page
  - Use Ruby and Python for complimentary examples

Design Methods

- **Structured Design / Programming**
  - Iteratively decomposing large problems into smaller steps
  - Implementing small steps as procedures / routines
- **Functional Programming**
  - Developing a set of functions to perform some evaluation
  - Creating larger functions through composition of functions
- **Object Oriented Programming**
  - Creating a set of objects which do things
  - Developing large systems through object interaction

Object Oriented Paradigm

Big Picture View

- An object-oriented system views software as a system of communicating objects
- Each object is an **instance** of a **class**
- Classes defined the **features** of an object
  - **Attributes**
  - **Methods**
- Classes can be **specialized** by subclasses
- Objects communicate through message passing

Procedural Paradigm

Compare this with the procedural paradigm...

- Functional decomposition breaks a problem down into several subproblems
  - Decompose subproblems into functional steps
- The goal is to subdivide the problem to a level of granularity which is easily solved in a few steps
- Then, assemble the steps in the correct order to solve the subproblems until the main problem is solved
- Very natural way of looking at things–approach used in introductory CS courses

Procedural Paradigm

- There are two main problems with this approach
  1. The resulting program is centralized around a “main program”
     - This program is aware of all the details of the data structure
     - This program is responsible for coordinating with subroutines
  2. This creates designs which are not well suited to change
     - Programs are not well **modularized**
     - Subroutines are very **static**, changes in some context requires changes to the routine itself
     - Minor changes to the data structure may cause impacts throughout the entire main program

Object Oriented Paradigm

- What causes these problems?
  - Poor use of **abstraction**
  - Poor **encapsulation**
  - Poor **modularity**
- If you have poor abstractions in your code, modification can become difficult
  - Consider a data structure implemented as an array
  - The required amount of data increases to an unknown amount, so switching to a Linked List makes sense
  - Now, need to create routines to insert, delete, etc., and find all spots in code which directly indexes the array...
- With poor encapulation and modularity, there is nothing preventing dependencies from forming in the code
- Changes to one part may percolate through the code

Properties of Robust Code

To design software which is resilient to change, we want these desireable properties

- Good **abstraction**
- Good use of **encapsulation**
- Strong **cohesion**
- Loose **coupling**

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
  - "This phonebook needs to be stored so that I can access a phone number using a person's name. What classes in Java provide this functionality?"

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
  - "This phonebook needs to be stored so that I can access a phone number using a person's name. What classes in Java provide this functionality?"
  - **Abstraction** - Looking at Java's Map API, we can see that this class does what we want, but don't need to know the concrete details.

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
    - "This coordinate needs to be limited to the dimensions of the GUI window. The client of this code should not be able to modify these values to set the coordinate outside of the window."

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
  - "This coordinate needs to be limited to the dimensions of the GUI window. The client of this code should not be able to modify these values to set the coordinate outside of the window."
  - **Encapsulation** - We want to make sure that there's no external access to the coordinate data.

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
  - "I may want to use an ArrayList or a LinkedList here, but it depends on the size of my dataset. I won't know how big the data is until runtime. Is it possible to be able to make this decision at run time, but still finish this code?"

Abstraction and Encapsulaton

- What's the difference between **abstraction** and **encapsulation**?
- Consider the following two observations of a program (sorry for the Java...)
    - "I may want to use an ArrayList or a LinkedList here, but it depends on the size of my dataset. I won't know how big the data is until runtime. Is it possible to be able to make this decision at run time, but still finish this code?"
    - **Abstraction** - We can think of this decision at an even more abstract level–all we need is a List object, exact details of *which type* of list is used is immaterial during development.

Abstraction and Encapsulation

Definitions

- **Abstraction** refers to the set of concepts that some entity provides in order to achieve a task or solve a problem
  - What features does a class provide to it's user?
  - What services does a class perform?
  - What's the programming interface for the class?
- **Encapsulation** refers to hiding of data or behaviors from the rest of the application
  - Limits access to sensitive data from clients of the class
  - Prevents misuse of the data

Cohesion

- **Cohesion** refers to how related the elements in a class (or module, framework, etc.) are.
  - A strongly cohesive class implies that the methods in the class all relate to some common task
  - Said another way, strong cohesion implies a class has a single purpose, and makes use of all its features
  - A weakly cohesive implies implies that the class performs many unrelated tasks
- Strong cohesion in classes is desired, weak cohesion should be avoided
  - Consider a change in the requirements of the software - some object(s) is/are *responsible* for that requirement
  - If the classes have strong cohesion, then only one (or a few) classes need modification
  - If the classes have weak cohesion, then many more will need to be modified to account for the change

Coupling

- **Coupling** refers to how strongly related to two or more classes (or methods, routines, etc.) are.
    - Tight coupling implies a strong relationship between two objects
        - Tightly coupled objects depend on many other classes to perform a task
    - Loose coupling implies a weaker relationship between objects
        - Loosely coupled objects have few dependencies on other classes
- Cohesion and coupling are typically complimentary

Object Oriented Analysis and Design

The goal is to create a set of objects with the following desirable properties

- A well defined set of features and services (abstraction)
- Implementation details hidden from clients (encapsulation)
- Internal integrity (strong cohesion)
- Flexible relations to other objects (loose coupling).

Objects - Levels of Perspective

Martin Fowler describes three levels of perspective for objects

- **Conceptual Level** - An object has a set of responsibilities. At this level, *implmentation* is not considered.
- **Specification Level** - An object is a set of behaviors with a well defined interface.
  At this level, an method signatures are defined, but *how* they are implemented are not.
- **Implementation Level** - An object consists of a set of data and methods which implmement the behavior of the object.

Object - Implementation Level

At the implmementation level, objects consist of data and methods.

- **Data**
  - Often called *Attributes*, *Data Members*, *Instance Variables*, etc.
  - Collectively, indicate the *state* of an object
- **Method**
  - A procedure associated with with the object
  - Indicates what a object can *do*
- **Identity**
  - Every object is unique and distinguishable
  - Objects can refer to themselves (self, this, etc.)

Approaches

- Class Based
  - Structure and behavior of objects are defined by a class
  - Individual objects are created from classes
  - Java, C++, C#, Objective-C, Python
- Prototype Based
  - Objects are defined directly, classes are not used
  - Objects can be defined by using other objects as a **prototype**
  - Javascript, Lua
- We'll focus on class-based languages

Object Oriented Paradigm

- What do objects offer to assist analysis and design
  - Classes
  - Inheritance and Polymorphism
  - Abstract classes and methods
  - Interfaces
  - Accessibility of Features

Classes

- Classes define a *type* of object, based on an object's responsibilities.
    - Behavior - as defined by methods
    - State - as defined by attributes
- Classes may be though of a blueprints or templates for an object.
- An individual object is an *instance* of the class.
- *Instantiation* is the process of creating an individual object.
- Classes also define two special methods for object creation and descruction
    - **Constructor** Used during object creation to initialize the object before use
    - **Descructor / Finalizer** Used for cleanup when an object is deleted

Inheritance

- Classes can exhibit an **inheritance** relationship
  - A **subclass** inherits its features from a **superclass**
  - Behaviors and data associated with the superclass are passed down to the subclass
  - The subclass can add new behaviors or data, or may modify inherited behavior or data to account for it's particular situation
  - Inheritance relationships are often refered to as "IS-A" relationships (e.g., a Banana IS-A Fruit)
    - We'll compare this with compositional ("HAS-A") relationships next week...
  - Often, a superclass will be called a **generalization** of a subclass, and a subclass will be called a **specialization** of a superclass

Polymorphism

- Polymorphism derives from greek, meaning “many forms”
- An instance of a subclass can be treated as if it were a instance of the superclass
  - After all, it contains the same methods and data as the superclass
- If need be, we can treat an instance of the subclass as if it were an instance of the superclass
- So, code built to process the superclass can be used to process the subclass
- From *Design Patterns Explained*, polymorphism is defined 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.”

Polymorphism

- Polymorphism allows a program to **dynamically** use different objects in different situations at runtime
- Consider a superclass *Quadrilateral*, which has two subclasses, *Rectangle* and *Rhombus*
- *Quadrilateral* has methods to *calculateArea*, *calculatePerimeter*, and *draw* it on a screen
  - Therefore, so does *Rectangle* and *Rhombus*, though the implementation details may vary
- Now consider a program where a user creates several instances of all three classes to create a layout for a backyard
- The program needs to calculate the total area of *all* of the shapes to determine how much sod to purchase
- If every object is simply treated as a *Quadrilateral*, we can iterate through a list of *Quadrilaterals*, calculating the area of each

Abstract Classes

- As superclasses become more generalized, being able to concretely define some specifications may no longer be possible
    - Our program may also need to have a *Circle* class, which has a very different formula for calculating area than a *Quadrilateral*
    - However, *Circles* and *Quadrilaterals* have the save behavior and state, namely, they are *Shapes*
    - A *Shape* class may specify that *Shapes* have a *calculateArea* method, but provide no default implementation
- Abstract classes define **generic** behavior, but doesn't provide all implementation details
- Abstract classes are classes which cannot be instantiated, whereas concrete classes can
    - My program cannot create a new *Shape* object
- Abstract classes *do* allow for polymorphism
    - My program can iterate through a list of *Shape* objects, calculating the area

Interfaces

- Interfaces are similar to a class, except that it only provides specification for its members
  - It provides method signatures, but no implementation or data
  - *Traits* in Scala, *Protocols* in Objective-C
- Interfaces are used to describe some characteristic which may be common among unrelated classes
- Classes can implement (or be extended with) an interface
  - Many different kinds of things can spin - tops, gears, wheels, etc.
  - These are all different classes of objects, but we may want to calculate angular momentum for each
  - Create a *Spin* interface, which specifies *angularMomentum* method
- In most languages, a class can only inherit from one other class, but may implement many interfaces

Accessibility

- Classes can control the accessibility of the features of their objects
- Attributes and methods are specified as
  - **Public** - A feature which can be accessed by anyone (including itself)
  - **Protected** - A feature which can only be accessed by itself and its subclasses
  - **Private** - A feature which can only be accessed by the object itself
  - Other levels may exist (e.g., *Package* in Java)
- This ability to hide features of a class (by setting its accessibility to private) is referred to as **information hiding** or **encapsulation**
  - Encapsulation isn't limited to just data hiding, though

Accessibility

- Consider three different objects
  - Object *Alice*, an instance of class *Communicator*
  - Object *Bob*, an instance of class *Listener*, which is a subclass of *Communicator*
  - Object *Eve*, an instance of class *Criminal*, which is unrelated to *Communicator* or *Listener*
- Every *Communicator* has a set of encryption keys - a *public* key and a *private* key for RSA encryption, and a shared *protected* key for 3DES

Accessibility

- Eve is able to access both Bob and Alice's public keys.
- Bob is able to access Alice's protected (3DES) key.
- Neither Bob nor Eve can access Alice's prive key.

Summary

- **Class** - defines a type of object
- **Object** - an individual member of a class
  - **Attribute** - data associated with an object
  - **Method** - behavior associated with an object
- **Abstract Class** - class which cannot be instantiated
- **Interface** - a specification for some behavior
- **Inheritance** - the ability for one class to specialize another class
- **Polymorphism** - the ability to treat several different subclasses as the superclass from which they derive
- **Encapsulation** - hiding information from clients of a module or class
- **Cohesion** - the degree to which a class performs related tasks
- **Coupling** - the degree to which one class is relies on another

For Next Time

- Check the website and Piazza for the first homework assignment.
  - Will be due two weeks from Friday
  - I will make an announcement via Piazza, which should be emailed directly to you
  - So if you haven't signed up, you may want to
- Topic for next time
  - Object Composition and Interaction
  - Introduction to analysis and design with objects
  - By end of next week, hopefully refactor the Tic-Tac-Toe code!