Lecture 4: Fundamentals of Object Technology

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2003

Credit where Credit is Due

- Some material presented in this lecture is taken from section 2.1 of Maciaszek's "Requirements Analysis and System Design". © Addison Wesley, 2000

Goals for this Lecture

- Introduce fundamental object-oriented concepts
  - classes, objects, attributes, methods, associations, encapsulation, inheritance, polymorphism, etc.
- to lay the foundation for using these concepts in object-oriented analysis

"Real-World" Objects

- The world consists of objects in a particular state
  - a *full* coffee mug
  - a *tired* athlete
- Some objects exhibit behavior
  - a dog *barks*
- All objects have identity
  - a property that allows us to distinguish one object from another

More on identity

- If we examine two cups from the same set of china, we may say that the cups are equal but not identical
- They are equal because they have the same set of attributes (size, shape, color, …) and they have the same values for each of their attributes
  - but they are two distinct cups and we can select among them; no two objects can have the same identity (otherwise they would be the same object)

Artificial and Natural Systems

- Since natural systems consist of objects and can exhibit complex behavior…
- …perhaps we can construct artificial systems by emulating the structure and behavior of natural systems
- The object-oriented approach to software development is based on this premise;
  - we model the real world using objects
    - the behavior and state of these objects capture the information of real-world tasks
  - we implement systems with these objects to support the automation of the modeled tasks

Instance Object

- An object is an instance of a class
  - A class is a generic description of all of its possible instances
    - in particular it specifies the set of attributes its objects possess and the set of behaviors its objects can perform
- Note: we sometimes need an object that represents a class, or a class object
  - this enables reflection
    - the ability for objects to ask questions about their own characteristics, such as “how many operations do I have?”
  - and attributes and operations that are not associated with any particular instance of a class but with the class itself

Object Notation

- The UML notation for an object is a rectangle with two compartments
  - the upper compartment specifies an object’s name and its class separated by a colon; both are underlined
    - Ken : Employee -- an object Ken of the Employee class
    - Ken : -- an object whose type is not known
    - : Employee -- an unnamed object
  - the lower compartment lists the values the object has for its attributes

Object Notation, continued

Ken : Employee
Name = "Ken" Start_Date = "Fall 1998"

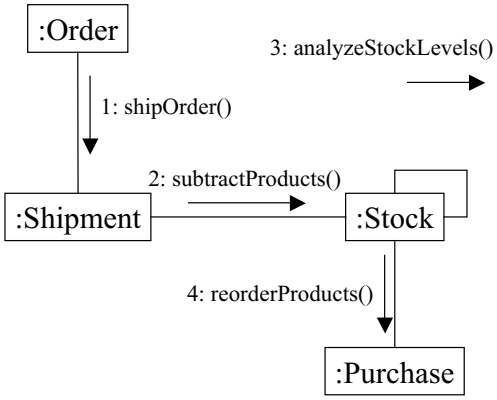
Object Collaboration

- The number of instances for a class can be very large
  - as such objects are rarely drawn
- Typically, they are only drawn to demonstrate how certain objects collaborate over time to accomplish a task
- Tasks are performed by objects invoking operations (behavior) on each other
  - an operation is invoked by passing the target operation a message, aka message passing

Object Collaboration, cont.

- A message specifies the name of an operation and provides values for that operation's parameters (if any)
- Once invoked, the operation may result in a change of state or…
  - a change in an object's attribute values
- …it may lead to the target object invoking operations on additional objects

Object Collaboration, cont.



An Order object requests a Shipment object to ship its products. To do so, the Shipment object instructs the Stock object to decrease its inventory; later the Stock object may analyze its inventory levels and reorder any required products

Identifying Objects

- How does an object learn the identity of an object to which it wants to send a message?
  - How does the Order object find the Shipment object in order to send the shipOrder message?
- Each object is given an object identifier (OID) when it is created
  - this identifier is a unique number that remains with the object for its entire life (from the time it's created until it's destroyed)
- So, the OID of the Shipment object must be passed to the Order object; how is this done?

Identifying Objects, continued

- A link provides a means for one object to know the OID of another object
  - links can be persistent or transient
- The type of link used in a given situation depends on the longevity of objects
  - transient objects live only as long as a program executes
  - persistent objects outlive the execution of the program; they are stored persistently to be accessed on subsequent program runs

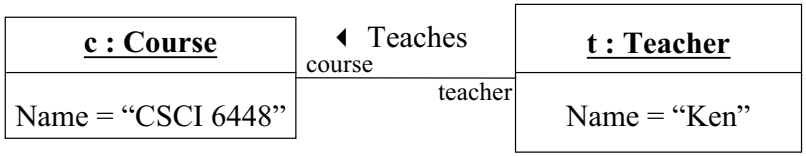
Identifying Objects, continued

- A persistent link is an object reference in a persistent object to another persistent object
  - Hence, to persistently link a Course object to a Teacher object, both objects must be persistent and the OID of the Teacher object must be stored in the Course object as the value of a link attribute

A Persistent Link

The Course object has an attribute named teacher that stores a reference to the OID of its instructor. Both the Course and Teacher objects are persistent. This is indicated by placing their OIDs in the upper right hand corner of their object representations. When these objects are written to disk, the reference in Course is saved too. When these objects are read from disk, the reference in Course is once again set to the OID of the Teacher object; In this particular set-up, only the Course object can invoke operations on the Teacher object, the Teacher object is unaware of the existence of the Course object.

A Persistent Link, continued



During analysis and design, we rarely care about the specific OIDs of objects; instead we care about the relationships that exist between objects; As such a persistent link is represented in the UML as an instance of an association that exists between the object's classes; Here we show that Teacher and Course objects participate in a "teaches" association with two roles, than can be used to navigate the association from one end to the other. Note, in the previous slide only the Course object could call on the Teacher object; on this slide, we have delayed that decision, this notation allows either object to call upon the other (there are ways to specify the direction of an association which we will cover later in the semester)

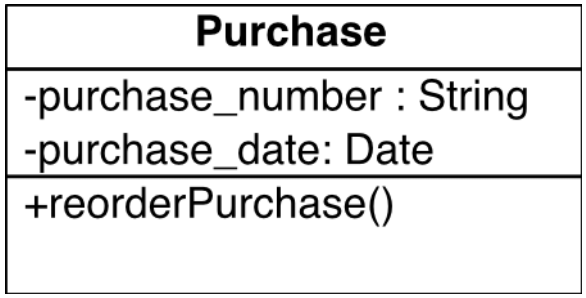
Transient Links

- Transient links are similar to persistent links, except that the link is established at run-time and is not made persistent
  - Thus, if I have multiple Course objects and I loop over them
  - Then, the loop variable is temporarily storing the OID for each Course object and establishing a *transient link* to a new Course object for each iteration of the loop

Classes

- A class is a generic descriptor for a set of objects that have the same attributes and operations
  - It serves as a template for object creation
  - Each object created from the template has space allocated for each attribute specified in the template and can invoke the operations defined in its class
- The notation for a class is a rectange with three compartments; one for the name, one for attributes, and one for operations
  - the UML actually defines four compartments; we will discuss the fourth compartment later in the semester

Class Notation



Attributes

- An attribute is a type-value pair
  - Classes define attribute types
  - Objects contain attribute values
- An attribute can be a built-in primitive type (that is supplied by an object-oriented programming language) or it can be another class
  - In objects, an attribute with a class-based type contain OIDs to objects of the specified class
  - In UML, class-based attributes are not listed in the attribute compartment, instead they are represented as associations

Attribute Visibility

- Not all attributes (and operations) are available to objects who have a link to another object
  - The visibility of an attribute determines who can access and manipulate the value of the attribute
    - private visibility (indicated by prefixing the attribute name with a minus sign) means that the attributes can only be manipulated by instances of that class
    - public visibility (indicated by prefixing the attribute name with a plus sign) means that the attributes can be manipulated by instances of all classes
    - there is also protected visibility, which we will discuss later

Attribute Visibility, continued

- Most operations are public, but most attributes are private
  - Operations are said to encapsulate attributes; protect them from being changed in unauthorized ways
  - This is important since a class may have a set of attributes that need to be updated in tandem
  - if an object of another class changes an object's attribute, it may not know to update the values of the other associated attributes
    - This can cause an object's values to get out of synch and potentially lead to system failures

Operations

- An operation is an algorithm that acts on attributes
- Operations are implemented by methods
- Operations are invoked by messages or events
  - the name of the message and the name of the invoked operation are the same
  - a message can pass parameter data to an operation and an operation can return a value to the object that sent the message
- The name of an operation together with a list of its parameter types is called the operation's signature
  - A signature must be unique within a class; this means that a class can have multiple operations with the same name just as long as their parameter types are different

Operation Visibility and Scope

- Operation visibility is the same as attribute visibility; it determines if objects of other classes can see and invoke an object's operations
- Operation scope is a distinct concept that determines if an operation acts on objects (instance scope) or class objects (class scope)
  - Example in textbook: The age of an employee vs. the average age of all employees

Associations

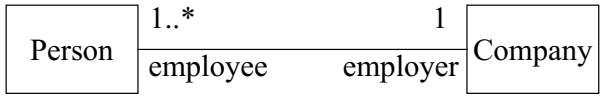
- An association is a type of relationship between classes
  - other types of relationships include generalization, aggregation, dependency, etc.
- As we have seen, associations govern the linkage between objects of particular classes

Associations, continued

- Association degree defines the number of classes that can be connected by an association
  - A binary association is the most common, although unary and ternary associations can be specified (see page 38)
- Association multiplicity defines how many objects may fill the position identified by a role name
  - The multiplicity states how many objects of the target class can be associated with a single object of the source class

Multiplicity Example

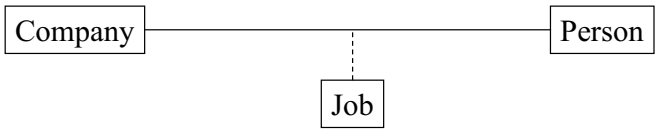
- To interpret a multiplicity always assume a "1" is at the opposite end of the association, for example,
  - a person may have only one employer
  - a company may have one or more employees



- The multiplicity is shown as a range of integers n1..n2. n1 defines the minimum number of connected objects, while n2 defines the maximum number; page 39 shows common multiplicities

Association Class

- Sometimes an association has attributes and/or operations of its own
  - Such an association must be modeled as a class (because attributes can only be defined in a class)



Aggregation and Composition

- Aggregation is a whole-part relationships between a class representing an assembly of components and the classes representing the components themselves
  - The containment property can be strong (containment by value) or weak (containment by reference)
  - In the UML, containment by value is known as composition

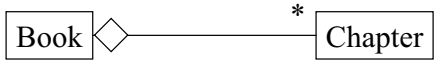
Aggregation/Composition, cont.

- In modeling, aggregation is a special kind of association which is transitive and asymmetric
  - Transitive: If A contains B and B contain C, then A contains C
  - Asymmetric: If A contains B, then B cannot contain A
- Composition has an additional constraint of existence dependency
  - If the container is deleted, its contents are deleted as well; the contents cannot exist without being contained

Examples

Aggregation:

Composition:



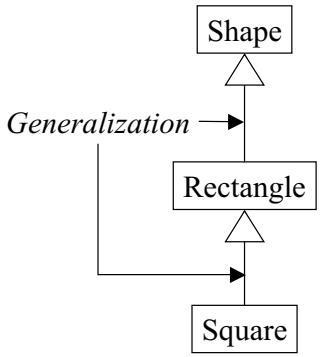
Generalization

- Generalization is a kind-of relationship between a more generic class (superclass or parent) and a more specialized class (subclass or child)
  - Since a subclass is a particular type of the superclass, an object of the subclass can be used anywhere an object of the superclass is allowed

Generalization, continued

- Subclasses may reuse any of the attributes and operations defined by its superclass (it is said to inherit these features from the superclass)
  - A generalization is notated with a hollow triangle on a line connecting the superclass and subclass

Example



An instance of Square will contain all the attributes and operations defined for Square, Rectangle, and Shape;

Any place a Shape object can be used in a program, a Square object can be substituted (since it has the same attributes and operations that Shape does)

Polymorphism

- An inherited operation from a superclass can be overridden (modified) in a subclass to correspond to semantic variations of the subclass
  - For instance, calculatePerimeter() will be implemented differently in Square than it would be for Circle()
  - This is indicated by placing a superclass operation with the same name and parameters in the subclass
  - Calls to such methods are polymorphic (many forms) in that the most specific implementation of the method will be invoked based on the class of the target object

Polymorphism, continued

- Thus, if we have an array of Shape objects, in which some objects are Squares and some are Circles, the following code can correctly calculate the sum of all their perimeters
  - for (i = 1; i < shapes.length; i++) {
    - Shape s = shapes[i];
    - totalPerimeter += s.getPerimeter()
  - }
- Note that we call getPerimeter() on a Shape object, but if the shape is a Square then it is Square's getPerimeter() method that executes and likewise if the shape is a Circle then Circle's getPerimeter() executes

Multiple Inheritance

- Some object-oriented programming languages will allow a subclass to inherit from multiple superclasses
  - A "diamond" in the inheritance structure can cause problems; consider Figure 2.19 on page 44 of the textbook
  - An instance of Tutor inherits the attributes and operations of Person twice; these situations are typically difficult to handle

Multiple Inheritance, continued

- In addition, a problem arises if a superclass is specialized into multiple orthogonal hierarchies;
  - because with multiple inheritance while a class can have multiple superclasses; an object must be an instance of a single class
  - so if I want an instance of a Person, who inherits from the Child, Female, and Student classes, I need to create a single class called ChildFemaleStudent
  - if the number of orthogonal hierarchies is large, an explosion in the number of classes can result

Multiple Classificiation

- Multiple classification is a solution to this problem; it simply states that objects are allowed to be instances of more than one class;
  - Unfortunately, not many object-oriented programming languages support multiple classification
  - Some languages provide the concept of an interface, which partially addresses this problem without resorting to multiple classification, except that with interfaces, no method implementations can be shared among objects that implement the same interface

Dynamic Classification

- Dynamic Classification allows an object to switch classes during its life-time
  - again, unfortunately, most object-oriented programming languages do not support dynamic classification
- This concept is useful in tracking changes to a long-lived object
  - For instance a Programmer object may need to change into a Manager object

Abstract Class

- An abstract class is a parent class that cannot be directly instantiated
  - the reason for this is that all or some of its operations have undefined methods
- An abstract class is often used to specify a contract that subclasses must follow for a particular task
  - When a subclass is defined, it provides method implementations for each of the parent's abstract methods that help it fulfill the contract
- For instance, a Shape object may define many abstract methods that allow Shapes to be moved and manipulated; a shape subclass cannot be instantiated until it provides methods for each abstract method

What's Next

- This lecture: fundamental object-oriented concepts
- Next lecture: Introduction to Analysis
- After that
  - Structured Analysis Techniques
    - to show how they contrast with OO approaches
  - Object-Oriented Analysis Techniques
  - Use Cases