Object-Oriented Modeling Using UML

Dr. Yvon Feaster

Objects and Classes

In this document we will discuss characteristics of Objects and Classes. We will discuss UML notations for various OO concepts. We will discuss a number of important principles in the OO development approach.

Characteristics of Objects and Classes

Object

- has a unique identity, state, and behavior
- state is composed of fields or attributes
- a field has a name, type, and value
- behavior defined by methods (methods access or manipulate the state of an objects )
- values of fields of an object are mutable (changable)
- objects can be mutable or immutable (non changable)
- mutators are methods that can modify the state of a field (a.k.a., setters)
- accessors are methods that do not modify the state of the object (a.k.a, getters)
- often referred to as an instance

Class

A class characterizes the structure of states and behaviors that are shared by all its instances.

- features of objects are defined in a class that instantiates the objects.
- defines:
  - the names and types of all fields
  - the names, types, and implementations of all methods
- values of the of fields are not defined or fixed in the class definition.
- values of fields are mutable
- each instance of a class has its own state.
- implementations of methods are defined in the class definition and are therefore fixed for a given object

Simple example of a class:

class Point{
   int x, y; // fields
   public void move(int dx, int dy); // an attribute or method as we know it
}
### UML Notation for Classes

The UML notation for classes is a rectangular box, as shown in Table 1 with as many as three compartments.

#### Syntax for fields

- **Java**
  
  \[ \text{[visability]} \ [\text{type}] \ \text{Name} \ [\text{[multiplicity]}] \ [-\text{Initial value}] \]

- **UML**
  
  \[ \text{[visability]} \ \text{Name} \ [\text{[multiplicity]}] \ [:\text{Type}] \ [-\text{Initial value}] \]

#### Syntax for methods

- **Java**
  
  \[ \text{[visability]} \ [\text{Type}] \ \text{Name} \ (\ [\text{Parameter(1)}, \ \text{Parameter(2)}, \ ..., \ \text{Parameter(n)}]\] \)

- **UML**
  
  \[ \text{[visability]} \ \text{Name} \ (\ [\text{Parameter(1)}, \ \text{Parameter(2)}, \ ..., \ \text{Parameter(n)}]\) \ [:\text{Type}] \]

#### Visability

<table>
<thead>
<tr>
<th>Visability</th>
<th>Java Syntax</th>
<th>UML Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>public</td>
<td>+</td>
</tr>
<tr>
<td>protected</td>
<td>protected</td>
<td>#</td>
</tr>
<tr>
<td>package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>private</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Multiplicity specification of a field

specifies whether an object may have multiple occurrences of the field.

#### Examples of field and method declarations:

**Field declarations**

**Java Syntax**

- Date birthday
- public int duration = 100
- private Student students[0 - MAX_SIZE]
UML syntax

birthday:Date
+duration:int = 100
-students[0 - MAX_SIZE]:Student

Method declarations

Java Syntax

void move(int dx, int dy)
public int getSize()

UML syntax

move(dx:int, dy:int)
+getSize():int

UML representation of the Point class

<table>
<thead>
<tr>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int x</td>
</tr>
<tr>
<td>private int y</td>
</tr>
<tr>
<td>public void move(int dx, int dy)</td>
</tr>
</tbody>
</table>

Table 3: Java Syntax

<table>
<thead>
<tr>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x:int</td>
</tr>
<tr>
<td>-y:int</td>
</tr>
<tr>
<td>+move(dx:int, dy:int)</td>
</tr>
</tbody>
</table>

Table 4: UML Syntax

UML Notation for Objects

The UML notation for objects is a rectangular box with one or two compartments.

<table>
<thead>
<tr>
<th>objectName:ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>field1 = value1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>field2 = value2</td>
</tr>
</tbody>
</table>

Table 5: UML Notation for Objects and Example

The name and class is underlined in the top row of the table with the fields and values listed in the bottom.

Principles of Object-Oriented Development Approach

This section will touch on several important concepts in OO software development, including:

- Modularity
Modularity

Modularity is defined by Techopedia [http://www.techopedia.com/definition/24772/modularity] as follows: “In software engineering, modularity refers to the extent to which a software may be divided into smaller modules. ... Modularity is successful because developers use prewritten code, which saves resources. Overall, modularity provides greater software development manageability.”

They also say, “software engineering modularity allows typical applications to be divided into modules, as well as integration with similar modules, which helps developers use prewritten code. Modules are divided based on functionality, and programmers are not involved with the functionalities of other modules. Thus, new functionalities may be easily programmed in separate modules.”

Our book defines the Principle of Modularity as follows:
A complex software system that should be decomposed into a set of highly cohesive but loosely coupled modules. What that heck does that mean? Let’s define cohesive and coupled. The book says “Cohesion refers to the functional relatedness of the entities within a module.” It says, “coupling refers to the interdependency among different modules.” So basically, modules (I think of these as classes) need to be related but individual.

Advantages of modularity:

• makes software easier to maintain
• allows reuse of software
• makes testing software easier

The concept of modularity is not new. With procedural languages you see it in the form of small routines and functions. In OO it takes the form of classes and packages.

Abstraction

Our book says “abstraction means separating the essential from the nonessential characteristics of an entity. The result is simpler but sufficiently accurate approximation of the original entity, obtained by removing or ignoring the nonessential characteristics.”

The principle of Abstraction provided in our book is as follows:
“The behaviors, or functionalities, of a module should be characterized in a succinct and precise description known as the contractual interface of the module. In other words, the contractual interface captures the essences of the behavior of the module. The contractual interface is an abstraction of the module.”

Are you sufficiently confused yet?

Here are a few examples that may help you understand.

1. From our book: Let us consider the telephone. The mechanism for providing telephone service is a rather complex one. It involves routing and connecting calls, converting voice to electronic signals and back to voice, etc. However, telephone users don’t need to understand how the phone system work. They only need to understand how to use the phone, such as dialing, speaking, hanging up. The user manual in this case is the contractual interface of the telephone service, and it serves as an abstraction of the telephone service from the user’s perspective.

2. From [http://whatis.techtarget.com/definition/abstraction](http://whatis.techtarget.com/definition/abstraction) Consider yourself as an object. You are arranging to meet a blind date and are deciding how to describe yourself so you will be recognizable to your date. You would include information concerning where you will be located, your height, hair color, maybe even what you will be wearing. This is all data that will help the procedure (your date finding you) work smoothly. You should include the relevant information necessary. However there are various bits of information, about you, that is not relevant to this situation since it will not help your date find you, such as: your SSN, your hobbies, and what you
took to “show and tell” in the fifth grade. This information can be abstracted since you do not need to reveal this in this situation. If the date goes well you may eventually share what you took to “show and tell”. :-)

3. From http://programmers.stackexchange.com/questions/230401/confused-about-the-definition-of-abstraction-in-oop. You as a person have different relationships in different roles. When you are at school, then you are a “Student”. When you are at work, you are an “Employee”. When you are at government institution, you can be viewed as a “Citizen”. So it boils down to in what context are we looking at an entity/object. So if I am modeling a Payroll System, I will look at you as an Employee(PRN, Full Time/Part Time, Designation). If am modeling a Course Enrollment System, then I will consider your aspects and characteristics as a Student(Roll Number, Age, Gender, Course Enrolled). And if I am modeling a Social Security Information System then I will look at your details as a Citizen(like DOB, Gender, Country Of Birth, etc.) Abstraction focuses on relevant or necessary details only.

Encapsulation
Encapsulation (a.k.a, information hiding) is closely related to abstraction. Remember Abstraction focusses on only necessary details. Encapsulation is hiding details from the outer world. Encapsulation means hiding the details of the object and providing a decent interface for the entities in the outer world to interact with that object or entity. It reduces coupling among modules.

Simpler put from http://www.tutorialspoint.com/java/java_encapsulation.htm, “encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.” Below is an example depicting encapsulation, also from tutorialspoint.com.

/* File name : EncapTest.java */
public class EncapTest{
    private String name;
    private String idNum;
    private int age;

    public int getAge(){
        return age;
    }

    public String getName(){
        return name;
    }

    public String getIdNum(){
        return idNum;
    }

    public void setAge( int newAge){
        age = newAge;
    }

    public void setName(String newName){
        name = newName;
    }

    public void setIdNum( String newId){
        idNum = newId;
    }
}
The public “getter” and “setter” methods are the access points to the class fields from the outside world.

Before we move on, consider the telephone system again. (This is a continued example from the book.) “The telephone service is an example of an application in which the contractual interface and implementation are separated. In the past, signals were transmitted in analog mode. Over time, telephone services have been upgraded until nowadays, the signals can be transmitted in digital mode with encryption. Although the implementation of telephone service has changed, the contractual interface remains the same. (in other words the user still makes a call the same way) The only effects on telephone users are that they enjoy better sound quality and greater security.”

A couple advantages of encapsulation:

- improves maintainability, flexibility, and re-usability
- fields can be made “read-only” by not using setter methods, or “write-only” by not defining getter methods
- user does not know what is happening behind the scene. Any Java object that can pass an IS-A test is considered to be polymorphic.

Polymorphism

In CPSC 1020 and (hopefully) 2100 we learned about Polymorphism, so we will not spend a great deal of time on it. Basically, it is defined by the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Any Java object that can pass an IS-A test is considered to be polymorphic. Below is an example:

```java
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

The Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object
Modeling Relationships and Structures

In this section we introduce the UML class diagrams for modeling object-oriented software systems and various types of relations among the classes. Class diagrams, for example, consist of nodes that represent classes and interfaces and links that represent relationships among classes.

The relationships that we will look at are inheritance, association, and dependency. We saw graphical notations for classes earlier, we will now look at notations for relationships.

Inheritance

Inheritance is one of the most important relationships in OO modeling. Inheritance can be defined as the process where one object acquires the properties of another. The keyword associated with inheritance is extends and implements. Inheritance implies “IS-A” relationship.

Various forms of inheritance relations are as follows:

- extension relationship between 2 classes – class C2 extends class C1, class C2 is known as a subclass of class C1, and C1 is known as a superclass of class C2
- extension relationship between 2 interfaces – when interface I2 extends interface I1, interface I2 is known as a subinterface of interface I1, and interface I1 is known as a superinterface of interface I2
- implementation relation between a class and an interface – when class C2 implements interface I1, class C2 is known as an implementation of interface I1, and interface I1 is known as an interface of class C2

Figure 1 is a graphical depiction of the above.

![Figure 1: UML Notation for Inheritance Relationships](image)

Extension is represented by a solid link, implementation by a dashed link, and the superclass and superinterface is represented by a hollow triangle pointing toward the superclass.

![Figure 2: Class Diagram: Inheritance Relation Among Various Student Groups](image)
Figure 2 is an example of inheritance relations among classes that represent various student groups. The Student class represents students in general. This is the superclass of subclasses, Nondegree, Undergraduate, and Graduate. Graduate is the superclass of Master and PhD. In other words, Nondegree, Undergraduate, and Graduate each have an “is-a” relationship with Student. Just as Master and PhD both have a “is-a” relationship with Graduate. (note: OO in general supports multiple inheritance relationships, however Java only allows one extends)

Levels of Abstraction  Class and interfaces represent abstractions. The inheritance relationship organizes the classes and interfaces into different levels of abstraction. The higher the class is in the tree or hierarchy the more general the abstraction. The lower levels of the tree are considered more specialized. Look at Figure 2 again. The Student class is very general. However, the Master and PhD classes (leaf classes) are highly specialized.

Association  Association models the various relationships between classes. The graphical notation for association is a solid line between the two classes with an optional label and optional adornment attached to either end. Figure 2 is an example of association between Student, Faculty, and Course classes. The teach and enroll keywords are the names of the association between Faculty – Course and Student – Course classes. The solid arrow with no tail indicates the direction of the association with respect to the name. As an example, the arrow next to the enroll means that “a student enrolls in a course”. Unless otherwise specified, the default association is left to right top to bottom. Hence, “Faculty teaches Course”. The keywords “advisor and advisee” are the role names associated with Faculty and Student. The multiplicity specification is a comma-separated sequence of integers values. The integer interval can be one of the following:

- $l .. u$ specifies a closed, that is, inclusive, range of integers from the lower bound to upper bound. If the upper bound is an (*), this indicates an unlimited upper bound
- $i$ specifies a singleton range that contains integer $i$, which is an integer literal.
- * specifies the entire nonnegative range: 0,1,2,3,...

In Figure 3, the enroll association is many-to-many – a student may enroll in many courses and a course may have any number of students enrolled. The teach association is one-to-many – each course has only one Faculty. Also, each student only has one advisor but an advisor can have many advisees.

Aggregation and Composition  Aggregation is an association that represents a has-a or part-whole relationship with another class. The stronger form of aggregation is composition which implies exclusive ownership of the component
class by the aggregate class. Figure 4 depicts the UML notation for the aggregation relationship. Aggregate is indicated by a hollow diamond pointing toward the aggregate class and composition is indicated by a solid diamond pointing toward the composite class. Figure 5 shows an example of the aggregation relationships among the University, College, Department, Faculty, and Student classes. All relationships shown in the diagram are aggregation relationships:

- a student is part of a department
- a faculty member is a part of a department
- a department is a part of a college
- a college is a part of a university

From the first two items we see a student or a faculty member may exist without being a part of a department. The lifetimes of the students and the faculty members are independent from the lifetime of the department to which they belong. In the last two items we see the department exclusively belongs to a college, and a college exclusively belongs to a university.

The distinction between aggregation and composition is entirely conceptual. There is not difference in the implementation.

![Figure 4: UML Notation for Aggregation Relationships, Aggregation on Left – Composition on Right](image)

![Figure 5: Class Diagram: Aggregation Relationship](image)

**Dependency**

Dependency is a directed relationship between named elements, classes, interfaces, components, artifacts, and packages.

The following is directly from the our book (Object-Oriented Software Development Using Java, by Xiaoping Jia) “Dependency is a relationship between entities such that the proper operation of one entity depends on the presence of the other entity, and changes in one entity would affect the other entity. A common form of dependency is the use relation among classes. In other words, class C1 depends on class C2 if C1 uses C2 in places such as the parameters, local variables, or return types of its methods. The graphical notation for dependency relationships is a dashed line with an arrow pointing in the direction of the dependency, as show in Figure 6.

The class diagram shown in Figure 7 illustrates the dependency relationship. The Registrar class has a number of methods for adding and removing courses in the course schedules and enrolling and dropping students in courses. These methods take parameters that are instances of CourseSchedule, Course, and Student classes. Therefore, the Registrar class uses, that is, depends on, CourseSchedule, Course and Student classes.”
Figure 7 shows the Registrar class uses CourseSchedule, Course and Student classes.