Object Oriented Programming

Chapter 6 and 7 in the book

Abstract data type

- Consider a datatype called *stack*.

  The operations such as $push$, pop, is_null etc... are common to stacks of various types (i.e., do not depend on whether it is a stack of integers or a stack of booleans or so forth).

  We abstract the datatype *stack* to *'a stack* (SML syntax - 'a can be instanciate by any type).

  We define the operations of this datatype as:

```
push: 'a * 'a stack -> 'a stack;
pop:  'a stack -> ('a * 'a stack);
top:  'a stack -> 'a;
is_null: 'a stack -> bool;
```

  We make a separation between the product (the ADT and its operations) and the process (the implementation of the ADT itself).

- What about some aspirin!

- The implementation of the operations of the abstract data type is like the process by which the 'a stack is made (definitions of the operations based on recursion). The end user does not have to worry about how it is implemented.

- This separation between use and implementation is necessary for maintainable programs: one implementation of an ADT can always be modified/replaced by another, and the rest of the system will behave (compute the results) the same way as before.

- Many ideas of ADTs have been taken and melded into the paradigm of "Object Oriented Programming" (OOP). OOP essentially supports the discipline of ADTs, enabling programmers to hide the implementation of data structures while exposing only the interface functions.

- What about databases?

Object oriented paradigm

- A program has to be built in such a way that it can be modified.

- **ISA and HASA**

  Example: Consider the New York Yankee organization and the database of its employees (players, managers,...).

  We can recognize an entity $employee$. Each employee has attributes: $name$, $address$ and $salary$. Each player is an employee. In addition to the generic attributes of an employee, a player also has some statistics such as RBI, HR, $Errors$, etc. Some players (pitchers) have even more statistics: ERA etc. Some non-players employees such as managers simply have different statistics altogether: the winning percentage. Other non-playing staff may have other attributes.

  We started with the broad generalization of employee to find that there are different kinds of employees and each has a different set of attributes.

```
employee HAS_ATTR name, address, salary
player ISA  employee
player HAS_ATTR  RBI, HR

pitcher ISA  player
pitcher HAS_ATTR  ERA, Saves

manager ISA  employee
manager HAS_ATTR  winning_pct

office-worker ISA  employee
office-worker HAS_ATTR  beeper
```

- **ISA hierarchy:**

  Employee is the root node. It has three children: $player$, $manager$ and $office - worker$. The node $player$ has a child $pitcher$.

  A node has an attribute if it is either defined directly using a HAS_ATTR, or can inherit the attribute from its ancestor. Thus a pitcher has attribute ERA, directly from HAS_ATTR definition. In addition a pitcher has attribute address inherited from its grandparent node employee.

- **Why is this hierarchy useful?**

Classes, Instances, Instance Variables and Methods

- A **class** describes a set of objects.
- Each object in a class is called an **instance** of that class.
- Each object is associated with a set of **instance variables** (attributes of the class).
- We define a class for $Circle$. Each circle which is an object in this class has the following attributes: $center\_x$, $center\_y$, and $radius$. We will assume that each instance variable of $Circle$ is of type $integer$.

  JAVA code:

```
class Circle {
        int center_x, center_y;
        int radius;
      }
```

  Let c be an instance of $Circsle$. Then the center coordinates and radius of c can be accessed as: $c.center\_x$, $c.center\_y$, and $c.radius$ respectively.

- We can define a method called $draw$() that returns the area of a circle.

```
        void draw() {
        // method to draw circle on the screen...
        }
```

  The type void here means that the method draw which does not return any value.

- Consider the problem of determining, for a given circle, whether it overlaps with another circle.

  This can be written as a method $overlaps$ that takes the second circle as a parameter. We can now modify the definition of class $Circle$ by adding the following method to the class:

```
boolean overlaps(Circle other) {
return ((center_x - other.center_x) *
(center_x - other.center_x)
+ (center_y - other.center_y) *
(center_y - other.center_y))
< ((radius + other.radius) *
(radius + other.radius));
}
```

- **Exercise:** d and c are circles. Can we also do d.overlaps(c)? Will that be different from c.overlaps(d)?

Inheritance

- Let's describe the set of colored circles: circles whose interior is filled with some color. Note that each colored circle is also a circle. Thus, we can define a class of colored circles $ColoredCircle$ that is a subclass of $Circle$. In Java syntax, this can be written as:

```
      class ColoredCircle extends Circle{
         Color color;
      }
```

  The subclassing means that each instance of $ColoredCircle$ is also an instance of $Circle$. In addition, $ColoredCircles$ have an extra attribute, i.e. their color. Thus, if d is an instance of ColoredCircle, then d.color is valid, as well as d.center_x, d.center_y, and d.radius. Also, we can apply the methods applicable to any instance of Circle to d. For instance, d.area() is a valid invocation that will compute the area of the colored circle d.

Overloading

- The class $ColoredCircle$ may implement its own $draw()$ method (since colored circles will be filled with the corresponding color).

  ```
  void draw(Color color) {
     // method to draw the circle in the given color
     }
  ```

  When we say $c.draw()$, we mean the colorless draw we defined earlier.

  Let r be an instance of class $Color$. When we say $c.draw(r)$, we mean the draw method we have just now defined.

- Note that these are **two different methods that implement two different functions**. We just called them using the same name, $draw$. We can distinguish between the two methods based on the **number of parameters** they take.

  This phenomenon is called **overloading**: we've overloaded the name $draw$ to mean two different methods. Which of the two we mean can be determined by its context: in general, the number and types of arguments.

Polymorphism versus overloading

- **Overloading is often confused with polymorphism that we saw in SML.**
- The simplest way to remember the distinction is:
  – a name may be overloaded,
  – but a function (or procedure) may be polymorphic.
- Overloading is the phenomenon by which we can associate more than one function/procedure with the same name.
- Polymorphism is the phenomenon by which the same function/procedure may be applied to data structures of different types.
- Note that the $draw$ methods above are all different, yet they are called by the same name.
- In contrast, a single function/procedure may be polymorphic. For instance, the $append$ and $reverse$ function we defined in SML are polymorphic: the very same function is used to append two integer lists, two real lists, two lists of lists etc etc.

Method resolution

- Let d be an instance of $ColoredCircle$, and c be a variable of type $Circle$. What do we get if we do $c.draw()$ after we do $c = d$ (i.e., assign d to c)?
- The techniques used to find out which method (or field) to use for an object is called **method (or field) resolution**.

Listing

```
class Circle {
  int center_x, center_y;
  int radius;
  int window_number;

  float circumference() {
     return 2 * 3.141529 * r;
  }

  float area() {
     return 3.141529 * r * r;
  }

  boolean overlaps(Circle other) {
     return ((center_x - other.center_x) *
     (center_x - other.center_x)
     + (center_y - other.center_y) *
     (center_y - other.center_y))
     < ((radius + other.radius) *
     (radius + other.radius));
  }
```

```
  void draw() {
  // method to draw circle on the screen...
  }

  void draw(Color color) {
  // method to draw circle on the screen with
  // given color
  }

   void draw(TextureMap tm) {
   // method to draw the circle and fill it
   // with given texture map
   }
}

class ColoredCircle extends Circle{
   Color color;
   int window_number;

   void draw() {
   // method to draw the colored circle
   }
}
```