Introduction/Review of Object-Oriented Concepts

Jan 15, 2010
Objects

+ dateProduced(): long
+ color(): Color
+ calories(): int

return Jan 12, 2009;
return WHITE;
return 300;
Objects don’t need classes

typedef int Color;

unsigned long int _dateProduced() { return 862547416; }

Color _color() { return 0; }

int _calories() { return 300; }

struct {
    unsigned long int (*dateProduced) ();
    Color (*color)();
    int (*calories)();
} aWhiteCake = { _dateProduced, _color, _calories };
Several similar objects

+ dateProduced(): long
+ color(): Color
+ calories(): int

- return Jan 12, 2009;
- return WHITE;
- return 300;

+ dateProduced(): long
+ color(): Color
+ calories(): int

- return Jan 12, 2008;
- return WHITE;
- return 300;

+ dateProduced(): long
+ color(): Color
+ calories(): int

- return Jan 14, 2009;
- return WHITE;
- return 300;
class WhiteCakeLayer {
    private long date;
    public WhiteCakeLayer() {
        date = System.currentTimeMillis();
    }
    public long dateProduced() { return date; }
    public Color color() { return Color.WHITE; }
    public int calories() { return 300; }
}

"An object’s implementation is defined by its class. The class specifies the object’s internal data and representation and defines the operations the object can perform.” DP, pg 14
IT'S AMAZING HOW MUCH WE'VE CHANGED SINCE PREHISTORIC TIMES.

OUR ANCESTORS JUST SAT AROUND IN CAVES, GRUNTING AND DRAWING ON THE WALL.

NOT VERY PRODUCTIVE.

IT WAS JUST THIS, ALL DAY LONG. MMM YUP.

Scott Adams. ©1995, Universal Features Syndicate
What has changed?

:WhiteCakeLayer
+ dateProduced(): long  return Jan 12, 2008;
+ color(): Color  return WHITE;
+ calories(): int  return 300;

:WhiteCakeLayer
+ dateProduced(): long  return Jan 12, 2009;
+ color(): Color  return WHITE;
+ calories(): int  return 300;

:WhiteCakeLayer
+ dateProduced(): long  return Jan 14, 2009;
+ color(): Color  return WHITE;
+ calories(): int  return 300;
class WhiteCakeLayer {
    private long date;
    public WhiteCakeLayer() {
        date =
            System.currentTimeMillis();
    }
    public long dateProduced() {
        return date;
    }
    public Color color() {
        return Color.WHITE;
    }
    public int calories() {
        return 300;
    }
}

class YellowCakeLayer {
    public Color color() {
        return Color.YELLOW;
    }
    public int calories() {
        return 400;
    }
}

class ChocolateCakeLayer {
    public Color color() {
        return Color.BLACK;
    }
    public int calories() {
        return 500;
    }
}

class VelvetCakeLayer {
    public Color color() {
        return Color.RED;
    }
    public int calories() {
        return 450;
    }
}
Types

“The set of all signatures defined by an object’s operations is called the **interface** to the object. An object’s interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object’s interface may be sent to the object. A **type** is a name used to denote a particular interface.”
DP, pg 13

“It’s important to understand the difference between a object’s **class** and its **type**. An object’s class defines how the object is implemented. The class defines the object’s internal state and the implementation of its operations. In contrast, an object’s type only refers to its interface—the set of request to which it can respond. An object can have many types, and objects of different classes can have the same type.”
DP, pg 16
Subtyping

<<interface>>

CakeLayer

+ color(): Color
+ calories(): int

WhiteCakeLayer
YellowCakeLayer
ChocolateCakeLayer
VelvetCakeLayer
Liskov substitution principle

If for each object $o_1$ of type $S$, there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$, then $S$ is a subtype of $T$.


If a value of type $S$ can be substituted into any context where a value of type $T$ is expected, then $S$ is a subtype of $T$. 
Interchangeability

Cake

- layer1: CakeLayer
- layer2: CakeLayer
- fill: Filling
- frost: Frosting

+ calories(): int

<<interface>>

Frosting

<<interface>>

Filling

<<interface>>

CakeLayer

+ color(): Color
+ calories(): int

return layer1.calories() + layer2.calories() + fill.calories() + frost.calories()
Class extension

Cake
- layer1: CakeLayer
- layer2: CakeLayer
- fill: Filling
- frost: Frosting
+ calories() : int

WeddingCake

BirthdayCake
- candles: Candle()
- decoration : Sprinkles
+ age() : int
Class extension vs. interface implementation

“It’s also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object’s implementation in terms of another object’s implementation. In short, it’s a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

“It’s easy to confuse these two concepts, because many languages don’t make the distinction explicit.”

DP, pg 17
“Because inheritance exposes a subclass to details of its parent’s implementation, it’s often said that ”inheritance breaks encapsulation.” The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent’s implementation will force the subclass to change.”

DP, pg 19
Two principles

Program to an interface, not an implementation.
DP, pg 18
Favor object composition over class inheritance.
DP, pg 20
Suggestions from *Effective Java*

- 13: Minimize the accessibility of classes and members.
- 14: In public classes, use accessor methods, not public fields.
- 15: Minimize mutability.
- 16: Favor composition over inheritance.
- 17: Design and document for inheritance or else prohibit it.
- 18: Prefer interfaces to abstract classes
- 19: Use interfaces only to define types.
- 20: Prefer class hierarchies to tagged classes
- 21: Use function objects to represent strategies
- 22: Favor static member classes over nonstatic.