CS193j, Stanford
Winter, 2002-03

Handout #10
Nick Parlante

OOP 3

Abstract Superclass

Factor Common Code Up

Several related classes with overlapping code
Factor common code up into a common superclass
Examples
- AbstractCollection class in java libraries
- Account example below

Abstract Method

The "abstract" keyword can be added to a method.
- e.g. public abstract void mustImplement(); // note: no { }, no code

An abstract method defines the method name and arguments, but there's no method code. Subclasses **must** provide an implementation.

Abstract Class

The "abstract" keyword can be applied to a class
- e.g. public abstract class Account { ...

A class that has one or more abstract methods is abstract -- it cannot be instantiated. "New" may not be used to create instances of the Abstract class.
A class is not abstract if all of the abstract methods of its superclasses have definitions.

Abstract Super Class

A common superclass for several subclasses.
Factor up common behavior
Define the methods they all respond to.
Methods that subclasses should implement are declared abstract
Instances of the subclasses are created, but no instances of the superclass

Clever Factoring Style

Common Superclass
- Factor common behavior up into a superclass. The superclass sends itself messages to invoke various parts of its behavior.

Special Subclasses
- Subclasses are as short as possible.
- Rely on the superclass methods for common behavior.
- Use overriding of key methods to customize behavior with the minimum of code

Rely on the "pop-down" behavior -- control pops down to the subclass for overridden behavior, and then returns to the superclass to continue the common code.

The java drawing class JComponent is an example of this sort of common superclass with lots of subclasses.

Danger:

Showing students this sort of example is a little dangerous. The engineering of it is neat and tidy, so it makes the whole concept appealing. However, opportunities for this sort of factoring are **rare**.

Polymorphism

Given an array of Account[] -- pointers with the CT type to the superclass

Send them messages like, withdraw() -- control pops down to the correct subclass depending on its RT type.

Account Example

The Account example demonstrates the clever-factoring technique.

Consider an object-oriented design for the following problem. You need to store information for bank accounts. For purposes of the problem, assume that you only need to store the current balance, and the total number of transactions for each account. The goal for the problem is to avoid duplicating code between the three types of account. An account needs to respond to the following messages:

- constructor(initialBalance)
- deposit(amount)
- withdraw(amount)
- endMonth()

  Apply the end-of-month charge, print out a summary, zero the transaction count.

There are three types of account:

Normal: There is a fixed $5.00 fee at the end of the month.

Nickle 'n Dime: Each withdrawal generates a $0.50 fee — the total fee is charged at the end of the month.

The Gambler: A withdrawal returns the requested amount of money, however the amount deducted from the balance is as follows: there is a 0.49 probability that no money will actually be subtracted from the balance. There is a 0.51 probability that twice the amount actually withdrawn will be subtracted. There is no monthly fee.

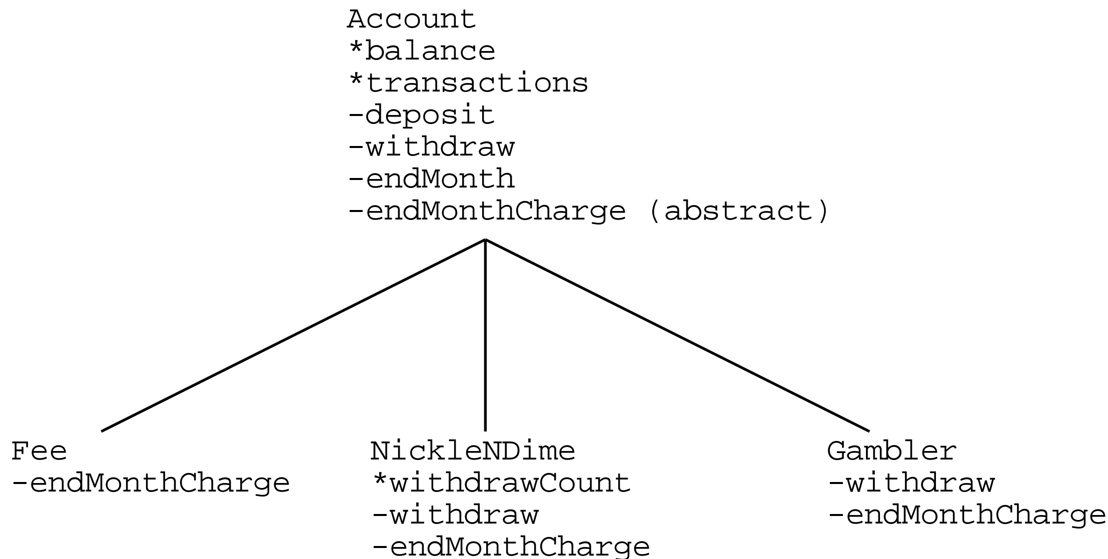
1. Factoring

The point: arrange the three classes in a hierarchy below a common Account class. Factor common behavior up into Account. Mostly the classes use the default behavior. For key behaviors, subclasses override the default behavior e.g. Gambler.withdraw(). This keeps the subclasses very short with most of the code factored up to the superclass.

2. Abstract Methods

A method declared "abstract" defines no code. It just defines the prototype, and requires subclasses to provide code. In the code below, the

endMonthCharge() method is declared abstract in Account, so the subclasses must provide a definition.



Account Code

```
// Account.java
/*
 The Account class is an abstract super class with the default
 characteristics of a bank account. It maintains a balance
 and a current number of transactions.
 There are default implementation for deposit(), withdraw(),
 and endMonth() (prints out the end-month-summary). However
 the endMonthCharge() method is abstract, and so must
 be defined by each subclass.

 This is a classic structure of using clever factoring to pull
 common behavior up to the superclass.
 The resulting sublclasses are very thin.
*/

import java.util.*;

public abstract class Account  {
   protected double balance;     // protected = avaiable to subclasses
   protected int transactions;

   public Account(double balance) {
      this.balance = balance;
      transactions = 0;
   }

   // Withdraws the given amount and counts a transaction
   public void withdraw(double amt) {
      balance = balance - amt;
      transactions++;
   }

   // Deposits the given amount and counts a transaction
```

```
public void deposit(double amt) {
   balance = balance + amt;
   transactions++;
}

public double getBalance() {
   return(balance);
}


/*
 Sent to the account at the end of the month so
 it can settle fees and print a summary.
 Relies on the endMonthCharge() method for
 each class to implement its own charge policy.
 Then does the common account printing and maintenance.
*/
public void endMonth() {
   // 1. Pop down to the subclass for their
   // specific charge policy (abstract method)
   endMonthCharge();


   // 2. now the code common to all classes

   // Get our RT class name -- just showing off
   // some of Java's dynamic "reflection" stuff.
   // (Never use a string like this for switch() logic.)
   String myClassName = (getClass()).getName();

   System.out.println("transactions:" + transactions +
      "\t balance:" + balance + "\t(" + myClassName + ")");

   transactions = 0;
}


/*
 Applies the end-of-month charge to the account.
 This is "abstract" so subclasses must override
 and provide a definition. At run time, this will
 "pop down" to the subclass definition.
*/
protected abstract void endMonthCharge();



//----
// Demo Code
//----

// Allocate a Random object shared by these static methods
private static Random rand = new Random();

// Return a new random account of a random type.
private static Account randomAccount() {
   int pick = rand.nextInt(3);
   Account result = null;
```

```
      switch (pick) {
         case 0: result = new Gambler(rand.nextInt(100)); break;
         case 1: result = new NickleNDime(rand.nextInt(100)); break;
         case 2: result = new Fee(rand.nextInt(100)); break;
      }

      /****
      // Another way to create new instances -- needs a default ctor
      try {
         Class gClass = Class.forName("Gambler");
         result = (Gambler) gClass.newInstance();
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      ****/

      return(result);
   }


   private static final int NUM_ACCOUNTS = 20;


   // Demo polymorphism across Accounts.
   public static void main(String args[]) {

      // 1. Build an array of assorted accounts
      Account[] accounts = new Account[NUM_ACCOUNTS];

      // Allocate all the Account objects.
      for (int i = 0; i<accounts.length; i++) {
         accounts[i] = Account.randomAccount();
      }

      // 2. Simulate a bunch of transactions
      for (int day = 1; day<=31; day++) {
         int accountNum = rand.nextInt(accounts.length);    // choose an account
         if (rand.nextInt(2) == 0) {   // do something to that account
            accounts[accountNum].withdraw(rand.nextInt(100) + 1);//Polymorphism Yay!
         }
         else {
            accounts[accountNum].deposit(rand.nextInt(100)  + 1);
         }
      }

      // 3. Have each account print its state
      System.out.println("End of month summaries...");
      for (int acct = 0; acct<accounts.length; acct++) {
         accounts[acct].endMonth(); // Polymorphism Yay!
      }
   }
// output
/*
End of month summaries...
transactions:1  balance:-1.0  (Fee)
transactions:5  balance:-84.0 (NickleNDime)
transactions:2  balance:43.5  (NickleNDime)
```

```
transactions:1  balance:90.0  (NickleNDime)
transactions:2  balance:89.0  (Fee)
transactions:1  balance:1.0   (Gambler)
transactions:1  balance:88.0  (NickleNDime)
transactions:1  balance:150.0 (Gambler)
transactions:6  balance:-19.5 (NickleNDime)
transactions:2  balance:-29.0 (Fee)
transactions:4  balance:226.0 (Gambler)
transactions:1  balance:86.0  (Gambler)
transactions:2  balance:70.0  (Fee)
transactions:2  balance:131.5 (NickleNDime)
transactions:4  balance:-42.5 (NickleNDime)
transactions:2  balance:-20.5 (NickleNDime)
transactions:3  balance:85.0  (Fee)
transactions:1  balance:-71.0 (Gambler)
transactions:2  balance:-175.0   (Gambler)
transactions:2  balance:-48.0 (Fee)
*/
}


/*
 Things to notice.

 -Because the Account ctor takes an argument, all the subclasses need a ctor
 so they can pass the right value up. This chore can be avoided if the superclass
 has a default ctor.

 -Suppose we want to forbid negative balance -- all the classes
 "bottleneck" through withdraw(), so we just need to implement something
  in that one place. Bottlenecking common code through one place is good.

 -Note the "polymorphism" of the demo in Account.main(). It can
 send Account obects deposit(), endMonth(), etc. messages and rely
 on the receivers to do the right thing.

 -Suppose we have a "Vegas" behavior where a person withdraws 500, and
 slightly later deposits(50). Could implement this up in Account..
 public void Vegas() {
   withdraw(500);
   // go lose 90% of the money
   deposit(50);
 }
 Depending on the class of the receiver, it will do the right thing.
 Exercise: trace the above on a Gambler object -- what is the sequence
 of methods that execute?
*/




// Fee.java

// An Account where there's a flat $5 fee per month.
// Implements endMonth() to get the fee effect.

public class Fee extends Account {
```

```
   public final double FEE = 5.00;

   public Fee(double balance) {
      super(balance);
   }

   public void endMonthCharge() {
      withdraw(FEE);
   }

}



// NickleNDime.java

// An Acccount subclass where there's a $0.50 fee per withdrawal.
// Overrides withdraw() to count the withdrawals and
// endMonthCharge() to levy the charge.

public class NickleNDime extends Account {

   public final double WITHDRAW_FEE = 0.50;

   private int withdrawCount;

   public NickleNDime(double balance) {
      super(balance);
      withdrawCount = 0;;
   }

   public void withdraw(double amount) {
      super.withdraw(amount);
      withdrawCount++;
   }

   public void endMonthCharge() {
      withdraw(withdrawCount * WITHDRAW_FEE);
      withdrawCount = 0;
   }

}



// Gambler.java

// An Account where sometimes withdrawals deduct 0
// and sometimes they deduct twice the amount. No end of month fee.
// Has an empty implementation of endMonthCharge,
// and overrides withdraw() to get the interesting effect.

public class Gambler extends Account {

   public final double PAY_ODDS = 0.51;
```

```
   public Gambler(double balance) {
      super(balance);
   }

   public void withdraw(double amt) {

      if (Math.random()<= PAY_ODDS) {
         super.withdraw(2 * amt);   // unlucky
      }
      else {
         super.withdraw(0.0);    // lucky (still count the transaction)
      }
   }

   public void endMonthCharge() {
      // ha ha, we don't get charged anything!
   }

}
```

1. Gambler.withdraw() -- super

Notice we use super.withdraw() to use our superclass code. Do not repeat code that the superclass can do. Be careful if you find yourself copying code from the superclass and pasting it into the subclass.

2. Account.endMonth() -- pop-down

Sends itself the endMonthCharge() message -- this pops-down to the implementation in each subclass.

3. Account.main() -- polymorphism

Constructs and Account[] array

Iterates through, sending the withdraw() message

Pops-down to the right implementation of withdraw() depending on the RT type of the receiver

Inheritance Issues...

Subclassing Examples

Animal : Bird ("bird isa animal")
Vehicle : Watercraft : Kayak
Collection : Stack
Drawable Thing : Drawable Thing which can also be clicked on : Button
Airplane : Engine NO ("engine isa airplane"? no)
Collection : Int NO
Stack : Collection NO (it's backwards)

Inheritance vs. Switch Statement

If different classes need to behave differently, then have them each implement the behavior and leverage the message/method resolution machinery. Never write code like the following...

```
switch (<type of object being dealt with>) { // probably a bad idea
   case <a>: <a behavior> break;
   case <b>: <b behavior> break;
   case <c>: <c behavior> break;
}
```

Use the message/method machinery to do the switch for you.
Switch logic still makes sense if it makes distinctions on some other run-time state, but if it has something to do with the class of the receiver, then using a method makes much more sense.

Inheritance vs. instanceof test

As with the switch case above, code that uses instanceof is a little suspect.
It can be necessary in some cases though.

if (obj instanceof Foo) { ...

Subclassing Relationship

Subclassing is used between classes with significant overlap. Subclassing establishes a significant constraint between the subclass and its superclass — the subclass "isa" specialized form of its superclass. The subclass must have every feature of the superclass. It must support every operation of the superclass. As a result, the subclass is capable of fitting in all the contexts where its superclass fits.

Subclassing vs. Encapsulation

As a practical matter, subclassing often breaks the encapsulation of the superclass. The subclass is very likely to pick up dependencies on the implementation of the superclass.

In this way, the super-sub relationship makes the subclass dependent on the implementation details of the superclass.

It is possible to keep the super/sub independent from each other, but it takes dedication. In particular, the superclass must declare its ivars private, and the subclass must go through get/set methods, like an ordinary client.

The relationship between a subclass and its superclass tends to involve tighter coupling than for a simple client/implementation relationship.

Subclass -- With Caution

When writing a subclass, realize there are significant couplings with the superclass. You need to understand the superclass so your subclass can fit in to its design.

Superclass -- Rare, Deliberate

When designing a class, do not include the "someone could subclass off here" feature as an afterthought. In general, you do not need to worry about subclassing -- declare ivars and utility methods private.

If you're going to support subclassing, the design, docs, etc. need to intentionally think about that case right from the start.

Inheritance Without Perfect Factoring

In the classical picture, behavior common to **all** subclasses is factored up to the common superclass.

However, not all cases are that tidy.

It may be acceptable to have behaviors that are 75% common up in the superclass, with some if logic to get around the cases that don't fit in.

This is where you might use an "if (obj instanceof Subclass) { ..." test to change the code path depending on the class of the receiver.

This goes against the no-switch rule, but may be reasonable in some cases. Not all problems fit the classic picture -- "it depends"

Java "interface"

Method Prototypes

- An interface defines a set of method prototypes.
- Does not provide code for implementation -- just the prototypes.
- Can also define final constants.

Class implements interface

- A class that implements an interface must implement all the methods in the interface. The compiler checks this at compile time.
- A Java class can only have one superclass, but it may implement any number of interfaces.

"Responds To"

- The interface is a "responds to" claim about a set of methods.
- If a class implements the Foo interface, I know it responds to all the messages in the Foo interface.
- In this sense, an interface is very similar to a superclass.

If an object implements the Foo interface, a pointer to the object may be stored in a Foo variable. (Just like storing a pointer to a Grad object in a Student variable.)

Lightweight

Interfaces allow multiple classes to respond to a common set of messages, but without introducing much implementation complexity.

Interfaces are lightweight compared to superclasses.

This is similar to subclassing, however...

Good news: A class an only have one superclass, however it can implement any number of interfaces. Interfaces are a more simple, more lightweight mechanism.

Bad news: An interface only gives the message prototype, no implementation code. The class must implement the method from scratch.

vs. Multiple Inheritance

C++ multiple inheritance is more capable -- multiple superclasses -- but it introduces a lot of compiler and language complexity, so maybe it is not worth it. Interfaces provide 80% of the benefit for 10% of the complexity.

e.g. Moodable Interface

Suppose you are implementing some sort simulation, and there are all sorts of different objects in the program with different superclasses.

However, you want to add a "mood ring" feature, where we can query the current color mood out an object.

Some classes will support mood and some won't

We define the Moodable interface -- any class that wants to support the Mood feature, implements the Moodable interface

```
public interface Moodable {
     public Color getMood();  // interface defines getMood() prototype but no code

}
```

If a class claims to implement the Moodable interface, the compiler will enforce that the class must respond to the getColor(); message.

Student implements Moodable

Here is what the Student class might look like, extended to implement the Moodable interface. The class must provide code for all the messages mentioned in the interface, in this case just getMood().

```
public class Student implements Moodable {

   public Color getMood() {
      if (getStress()>100) return(Color.red);
      else return(Color.green);
   }

   // rest of Student class stuff as before...
}
```

Client Side Moodable

Moodable is like an additional superclass of Student.
It is possible to store a pointer to a Student in a pointer of type Moodable.
The type system essentially wants to enforce the "responds to" rules. It's ok to store a pointer to a Student in a Moodable, since Student responds to getMood().

So could say...

```
Student s = new Student(10);

Moodable m = s;       // Moodable can point to a Student
m.getMood();          // this works
```