AP® COMPUTER SCIENCE AB
2006 SCORING GUIDELINES

Question 2: Packs & Bundles (Design)

Part A: Pack **3 1/2 points**

+1/2 `class Pack implements Product`
+1/2 declare both private fields (`int` and `Product`)

+1 constructor
- **+1/2** `public Pack(int ?, Product ?)` (`Item` OK if matches field)
- **+1/2** initialize fields

+1 1/2 `getPrice`
- **+1/2** `public double getPrice()`
- **+1/2** access product's current price and pack's current number
- **+1/2** calculate and return price

lose if don't call getPrice on a product

Part B: `Bundle` **5 1/2 points**

+1/2 `class Bundle implements Product`
+1/2 private collection field

+1 constructor (*if collection is initialized when declared, no constructor is needed*)
- **+1/2** `public Bundle()`
- **+1/2** initialize collection

+1 add
- **+1/2** `public void add(Product ?)` (*No penalty if returns reasonable value*)
- **+1/2** add parameter to collection

+2 1/2 `getPrice`
- **+1/2** `public double getPrice()`
- **+1/2** declare & initialize sum (must be added to)
- **+1/2** loop over every element in collection
- **+1/2** sum is updated with `getPrice` for each element
- **+1/2** return sum

lose if accum. price in add()

Common Usage: **-1** for extraneous code with compile-time errors
-1/2 missing `public` on `getPrice`

AP® COMPUTER SCIENCE A/AB
2006 GENERAL USAGE

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

default constructor called without parens; for example, `new Fish;`

use keyword as identifier

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= instead of == (and vice versa)

`length`/`size` confusion for array, `String`, and `ArrayList`, with or without `()`

`private` qualifier on local variable

extraneous code with no side-effect, for example a check for precondition

common mathematical symbols for operators (x • ÷ ≤ ≥ <> ≠)

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing “new” for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

Minor Errors (1/2 point)

confused identifier (e.g., `len` for `length` or `left()` for `getLeft()`)

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example `int x; …x.equals(val)`

`[] – get` confusion if access not tested in rubric

assignment dyslexia, for example, `x + 3 = y;` for `y = x + 3;`

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, e.g., `a = method(int x)`

missing `public` from method header when required

"false"/"true" or 0/1 for boolean values

"null" for `null`

Major Errors (1 point)

extraneous code which causes side-effect, for example, information written to output

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

`aMethod(obj)` instead of `obj.aMethod()`

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g., by using root reference to a `TreeNode` for traversal of the tree)

use class name in place of `super` either in constructor or in method call

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

Question 2: Packs & Bundles (Design)

PART A:

```
public class Pack implements Product
{
  private int numProducts;
  private Product prod;

  public Pack(int num, Product p)
  {
    numProducts = num;
    prod = p;
  }

  public double getPrice()
  {
    return prod.getPrice() * numProducts;
  }
}
```

PART B:

```
public class Bundle implements Product
{
  private ArrayList productList;

  public Bundle()
  {
    productList = new ArrayList();
  }

  public void add(Product newProd)
  {
    productList.add(newProd);
  }

  public double getPrice()
  {
    double totalCost = 0.0;
    for (int i = 0; i < productList.size(); i++)
    {
      totalCost += ((Product)productList.get(i)).getPrice();
    }

    return totalCost;
  }
}
```

AB2 A1

Write the `Pack` class below.

```
public class Pack implements Product {
    private int numItems;
    private Item ItemType;

    public Pack(int n, Item i) {
        numItems = n;
        ItemType = i;
    }

    public double getPrice() {
        return (itemType.getPrice() * numItems);
    }

    public String getDescription() {
        return "" + numItems + " x " + 
                                  itemType.getDescription();
    }
}
```

Part (b) begins on page 12.

GO ON TO THE NEXT PAGE.

AB2 A2

Write the Bundle class below.

```
public class Bundle implements Product {
    ArrayList items;

    public Bundle (){
        items = new ArrayList();
    }

    public void add(Product p){
        items.add(p);
    }

    public double getPrice(){
        double sum = 0.0;
        Iterator i = items.iterator();
        while(i.hasNext())
            sum +=((Product) i.next()).getPrice();
        return sum;
    }

}
```

GO ON TO THE NEXT PAGE.

AB2 B1

(a) Write the `Pack` class that implements the `Product` interface. A `Pack` is used to represent multiple occurrences of a given product. Its constructor should have two parameters: the first represents the number of occurrences of a product in the pack, and the second represents the product.

The price of a `Pack` is the price of the product times the quantity. If the price of the product changes after the `Pack` has been constructed, the result of a call to `getPrice` for the `Pack` should reflect the updated product price.

The following code segment shows an example of how a `Pack` object can be declared and initialized.

```
Product toaster = new Item("NeverBurn Toaster", 20.0);
Product toasterPack = new Pack(4, toaster);
```

```
class Pack implements Product
{
    double packPrice;
    Item item;
    int occurences;
    pack(occur, itemobj
      { occurences=occur;
        item=itemobj;
        packPrice=item.getPrice()*occurences;
      }
   double getPrice()
   {
    return(item.getPrice()*occurences);
   }



}
```

GO ON TO THE NEXT PAGE.

AB2 Bz

Write the `Bundle` class below.

```
Class   Bundle
{
 ArrayList items;


 Bundle()
  { items=new ArrayList}

 void add(product toAdd)
 {
  items.add(toAdd)
 }
 double getPrice()
 { double total_price;
  for(int i=0;i<items.size();i++)
      total_price+=items.get(i).getPrice();
  return total_price;
 }
```

```
}
```

GO ON TO THE NEXT PAGE.

AB2 C1

Write the Pack class below.

```
public class Pack implements Product
{ private int numItems;
  private Item myItem;
  private double unitPrice;
  public Pack (int Num, Item item)
  {
    numItems = Num;
    myItem = item;
  }

  public double getPrice()
  {
    return unitprice*numItems;
  }
  public void setPrice (double price)
  { unitprice = price*numItems;
  }
}
```

Part (b) begins on page 12.

GO ON TO THE NEXT PAGE.

AB2C2

Write the `Bundle` class below.

```
public class Bundle  implementsProduct
{
   private int unitprice;

   public Bundle()
   {
   }
   public add (Item item)
   {
   }
    public get Price()
    {
      return unitprice+unitprice 2;
     }

   }
```

GO ON TO THE NEXT PAGE.

Question 2

Overview

This question focused on students' ability to design a hierarchy of classes using inheritance. A Product interface was provided, along with an Item class that implemented the interface. The Item class contained private data fields for storing a product description and a unit price, a constructor for initializing the fields, accessor methods for retrieving the description and unit price, and a mutator method for changing the price. In part (a) students were required to design and implement the Pack class, which also implemented the Product interface. A Pack could store an arbitrary quantity of a Product and so needed private fields for the Product and its quantity. The class also required a constructor (to initialize the fields) and the getPrice method (to calculate the price of the Pack by multiplying the Product price by its quantity). In part (b) students were required to design and implement a Bundle class, which also implemented the Item interface. A Bundle could store an arbitrary collection of Products and so needed a private field for the collection. The class also required a constructor (to initialize the field), the getPrice method (to iterate through the collection and calculate the total cost of all Products), and an add method (to add a new Product to the collection). In addition to testing whether the students could design and implement classes from scratch, this question measured their understanding of polymorphism. Each class implemented the Product interface and contained fields of type Product, which allowed for arbitrary structures such as Bundles of Bundles containing Packs of Bundles.

Sample: AB2A
Score: 8

This response earned almost all the points in part (a). The only deduction was a ½ point for declaring itemType as an Item rather than a Product. Since the constructor header is consistent with this declaration, the student earned the ½ point for the constructor header. The getPrice method earned all three ½ points. The extraneous method getDescription is also consistent with the student's use of Item and was not penalized.

In part (b) the solution is very similar to the canonical solution. There was a ½ point deduction, however, for not declaring the ArrayList items as a private field.

Sample: AB2B
Score: 6

The Pack class correctly implements Product, earning the first ½ point from the scoring guidelines. None of the fields are declared to be private, so the second ½ point was lost. Because the constructor is missing parameter types, the student lost the first ½ point for the constructor. The initializations are correct, earning the second ½ point for the constructor. The header for getPrice is missing the public qualifier, so while the first ½ point of getPrice was earned, a ½ point was deducted for usage (since the header does not explicitly follow the interface). The body of the method is correct, earning the last two ½ points.

The Bundle class does not implement Product, so the first ½ point on part (b) was deducted. As in part (a) the student does not make the collection field private, and so the second ½ point was lost here as well. The constructor and add methods are correct (the lack of the public qualifier on either was counted as unpenalized usage). The getPrice method is also missing public contradicting the interface, but since usage for this error was deducted on part (a), it was not deducted again in part (b). The ½ point for the getPrice header was given. There is no initialization of total_price, costing that ½ point. The remaining three ½ points were awarded.

Question 2 (continued)

Sample: AB2C
Score: 3

The header for the class Pack is correct. The variable myItem is declared as an Item rather than as a Product, losing the second ½ point. However, since Item is used consistently in the constructor, the student earned the ½ point for the constructor header. The initializations correctly assign the parameters to the fields. The header for getPrice is correct and matches the interface. The last two ½ points of the getPrice scoring guidelines were not awarded since getPrice is never called on a product.

The header for the Bundle class is also correct, earning the first ½ point from the part (b) scoring guidelines. The only other ½ point was earned for the correct header for the default constructor. No collection is declared or initialized. The add and getPrice methods do not have a return type. There is no variable to which prices taken from Product objects are added and no loop over a collection, so no points were given for the getPrice method.