Highlights of Last Week

Properties of a Good Abstraction:

- Well named
- Coherent - attributes and behavior that are related and make sense
- Accurate - attributes and behavior match the entity being modeled
- Minimal - nothing extra
- Complete - all the attributes and behavior that are necessary
- Consistent - with respect to names, arguments, return values, and behavior.
- Decoupled - Different abstractions are not inter-dependent

Composition

*Composition extends the responsibilities of an object by delegating work to additional objects.*

Composition is the major mechanism for extending the responsibilities of an object. Nearly every object in an object model is composed of, knows of, or works with other objects.

Encapsulation

- Bundling together of state and behavior
- the restriction of access to data within an object to only those methods defined by the object’s class.
- Objects hide their implementation behind public methods that expose the desired behavior.
- Details related to the implementation of the class should not be visible to the outside.
- supports modularity - an object’s class can be maintained independently of other classes.
Homework Considerations

Things to consider

- Whose responsibility should it be to create Airplane objects?
- Whose responsibility should it be to assign Airplane identifiers?
- Whose responsibility should it be to manipulate the Airplane cost per seat?
- Should the data type used for the cost per seat be a primitive or an object type?

What are the potential advantages and disadvantages of the choices?

Additional considerations

-
Sample Homework Solution - Airline

Airline.java

```java
public class Airline {
    private static final int INITIAL_SEAT_COST = 220;

    /** The name for the airline */
    String name;

    /** A collection of Airplanes */
    java.util.ArrayList airplanes = new java.util.ArrayList();

    /** Constructor
     ** @param name name of the airline
     **/
    public Airline(String name) {
        this.name = name;
    }

    /** Add an airplane to the airlines’s collection */
    public void addAirplane(Airplane plane) {
        airplanes.add(plane);
        plane.setCostPerSeat(INITIAL_SEAT_COST);
    }

    /** Provide identifying information for the airline
    **/
    public String toString() {
        StringBuffer buf = new StringBuffer("Airline:");
        buf.append(name);
        for (int i = 0; i < airplanes.size(); i++) {
            buf.append("\nplane");
            buf.append(i);
            buf.append(":");
        }
        return buf.toString();
    }
}
```
buf.append(airplanes.get(i));
}
return buf.toString();
}
public class Airplane {

    /** An identifying number for the airplane */
    private String id;
    /** The total number of seats */
    private int totalSeats;
    /** The number of currently available (un-sold) seats */
    private int availSeats;
    /** The current cost for a seat */
    private Amount costPerSeat;

    /** Default, no argument constructor. Private so no one can
     ** create an unidentified Airplane
     **/
    private Airplane() {
        this("unidentified");
    }

    /** Create an airplane with the specified identifier and default
     ** initial cost and number of seats
     **/
    public Airplane(String id) {
        this(id, 0, 100);
    }

    /** Create an airplane.
     ** @param id airplane identifier
     ** @param cost initial cost per seat in local currency units
     ** @param seats initial capacity in seats
     **/
    public Airplane(String id, int cost, int seats) {
        this.id = id;
        costPerSeat = new Amount((double) cost);
        totalSeats = seats;
        availSeats = totalSeats;
    }
}
public void setCostPerSeat(double newCost) {
    costPerSeat = new Amount(newCost);
}

public String toString() {
    StringBuffer buf = new StringBuffer("Airplane:");
    buf.append(id);
    buf.append(" seats:");
    buf.append(totalSeats);
    buf.append(" avail:");
    buf.append(availSeats);
    buf.append(" costPerSeat:");
    buf.append(costPerSeat);
    return buf.toString();
}
Sample Homework Solution - Amount

Should the data type used for the cost per seat be a primitive or an object type?

Amount.java

/** An encapsulation of an amount of currency **/

public class Amount {
    private long amount;                // currently maintained in whole cents

    public Amount(double d) {
        amount = (long) (d * 100);
    }

    /** Converts from internal representation back to double. **/
    public double getAsDouble() {
        return ((double) amount) / 100.;
    }

    // Class (static) data to do a nice formatting job on amounts // when needed.

    private static java.text.NumberFormat formatter;
    private static final int fracDigits = 2;
    static {
        // This block of code is a "static initializer": it gets
        // executed once automatically when the class is loaded. You
        // use them when you need more initialization than you can
        // accomplish by just creating a new object and to avoid
        // having to have special Class.init() methods.
        formatter = java.text.NumberFormat.getCurrencyInstance();
        formatter.setGroupingUsed(false);
        formatter.setMaximumFractionDigits(fracDigits);
        formatter.setMinimumFractionDigits(fracDigits);
    }

    // Private helper method
    private String formatAmount() {
        String num = formatter.format(getAsDouble());
        return num;
    }
}
public String toString() {
    return formatAmount();
}

Things to Note

- Amount objects are immutable - why might that be good?
- Amount hides its implementation
- Amount encapsulates how to present itself
Sample Homework Solution - tester

CmdLineAirlineTest.java

/** A class that will be used to test Airline and Airplane capabilities
**/
public class CmdLineAirlineTest {

    private static final int SEAT_COST_MULTIPLIER = 100;

    private Airline sampleAirline = null;

    public void createAirline(String[] ids) {

        // Create a new Airline
        sampleAirline = new Airline("USAirways");
        Airplane plane;

        for (int i = 0; i < ids.length; i++) {

            // Request the Airline to add several new Airplanes
            plane = new Airplane( ids[i] );

            // Cause Airplane objects to change its cost per seat value
            sampleAirline.addAirplane(plane);

            plane.setCostPerSeat( (double) ((i+1) * SEAT_COST_MULTIPLIER));

        }
    }

    public void identifyAirline(java.io.PrintStream out) {

        // Cause the Airline object and the Airplane objects to
        // identify themselves and, in the case of the Airplanes,
        // their current cost per seat.

        out.println(sampleAirline.toString());
    }
}
public static void main(String[] args) {

    if ( args.length < 1 ) {
        System.out.println("usage: CmdLineAirlineTest plane_ids");
    } else {
        CmdLineAirlineTest tester = new CmdLineAirlineTest();
        tester.createAirline(args);
        tester.identifyAirline(System.out);
    }
}
Flow of Control

Flow of Control

main()

{ }
Accessability

The creator of a Java class controls what access objects outside the class have to the implementation (the inner details) of objects of her class by giving variables and methods accessability qualifiers.

Unlike in C++, each method or field is given an accessability qualifier.

- **public**
  - Any object can call public methods.
  - Any outside object can potentially change public variables. - **Why is this bad?**

- **private**
  - methods are only callable by instance methods of the same object (class) - not even by subclasses.
  - variables are only accessable within the methods of the object (class) - not even from subclasses.

- **protected**
  - methods are only callable by instance methods of the class and any sub classes.
  - variables are accessable from instance methods of the class and any sub classes.

- **"Package access"** - the default if no other modifier is used:
  - Instances of any class in the same package may call methods with package access.
  - Instances of any class in the same package can access package variables
Packages

- A **package** is simply a set of classes
- The classes all reside in the same directory (folder)
- The classes generally have something in common
- A package name has "." between components: each represents a hierarchical directory organization.
- If a package name definition is not provided, all classes are defined to be part of a **default, unnamed** package
- packages have accessibility implications
Import

- The `import` statement allows a named type to be referred to by a simple name, instead of a fully qualified (including package) name:

  ```java
  import java.util.ArrayList;
  ...
  ArrayList list = new ArrayList();
  ```

  instead of

  ```java
  java.util.ArrayList list = new java.util.ArrayList();
  ```

- You can extend your simple name space to include all of the public classes in a package via:

  ```java
  import java.util.*;
  ```

- There is no "define before use" requirement in Java (at least as far as the use of the import statement is concerned) - import statements are optional.

- Specifying individual import statements for each class you delegate to is
  - a good documentation aid
  - a useful clue to classes that do too much - too many imports == too much responsibility

- A Java "compilation unit" (source file) consists of:

  ```java
  [ package package.name; ]
  [ import declaration; ]
  ...
  public class ClassName {
  ...
  }
  ```
[ class HelperClass {
  ...
}
}
CLASSPATH

- If you use package statements to organize your Java classes, you will need to understand how to set the CLASSPATH to tell the compiler and the interpreter where to find those packaged classes.
- The CLASSPATH environment variable - not recommended
- The -classpath tool argument - recommended
  - See the java and javac tool documentation for your platform
  - Write a short script / batch file for use in compiling / running your application.
  - Use a tool like Ant
  - IDEs will (generally) provide a mechanism to set the CLASSPATH for development.
Collections

- JDK 1.1.x
  - Vector
  - Hashtable
  - Arrays - primitives and objects

- Java2
  - All of Java1, plus
  - Collection Framework: interfaces, abstract classes, concrete implementation classes: Set, List, Map, SortedSet, SortedMap, HashSet, TreeSet, ArrayList, LinkedList, Vector, Collections, Arrays, AbstractCollection

- Third party collection classes
ArrayList

- Implements a grow-able array of objects.
- Contains components that can be accessed using an integer index.
- The size of an ArrayList can grow or shrink as needed to accommodate adding and removing items after the ArrayList has been created.

Some Methods:
- `add(Object)`
- `remove(Object)`
- `remove(int)`
- `get(int)`
- `size()`

ArrayList `l = new ArrayList();`
Object `o = new Object();`  // or any object

```java
l.add(o)
...
o = l.get(8)
```
HashMap

- Implements a hashtable, which maps keys to values.
- Any object can be used as a key or as a value.
- To successfully store and retrieve objects from a HashMap, the objects used as keys must implement the hashCode() method and the equals() method.
- Some Map Methods:
  - `put(Object key, Object value)` - Maps the specified key to the specified value - makes a copy of key and the value object references
  - `get(Object key)` - Returns the value to which the specified key is mapped - returns a copy of the object reference
  - `containsKey(Object)` - Tests if the specified object is a key in this hashtable.
  - `remove(Object)` - Removes the key (and its corresponding value) - sets its copy of the references to null.

```java
HashMap h = new HashMap();
String k = new String("key");
String v = new String("value");

k.hashCode();

h.put(k, v)

v = (String) h.get(k);
```
Iterator

- An object may implement the Iterator interface.
- Successive calls to the `next()` method return successive elements of the series.
- One way to expose controlled access to an object’s collection’s contents.
- Some Methods:
  - `boolean hasNext()`
  - `Object next()`
The class Object is the root of the Java class hierarchy. Every class has Object as a superclass.

Object provides several methods that all classes inherit. Among these are:

- boolean equals(Object o) - indicates whether some other object, o, is "equal to" this one.
- int hashCode() - returns an identifying hashcode ("unique identifier") value for each object.
- String toString() - Returns a string representation of the object.
Airplane Class diagram

Because every class inherits from Object ...
Another CSStudent/CSCourse implementation

CSStudent.java

package week03;

import java.util.HashMap;

/** A Java class that models information kept about a student.
** The CSStudent class will enable objects to be created, each of
** which encapsulates information about an individual student.
**
** @author Jonathan Doughty
** @version 2.0
**
public class CSStudent {

    // Fields associated with an individual CSStudent objects.
    private String      name;
    private String      id;
    private CSStudent   labPartner = null;
    private HashMap     homeworkGrades = new HashMap();

    // Constructor for CSStudent objects
    public CSStudent(String newStudentName, String id) {
        name = newStudentName;
        this.id = id;
    }

    // Methods that can be called on individual CSStudent objects,
    // i.e., messages that can be sent to individual CSCourse
    // objects.
    public String getName() {
        return name;
    }

    public String toString() {

        return null;
    }
}
String endl = System.getProperty("line.separator");
String result = "Student name: " + name + " Id: " + id;
if (labPartner != null)
    result += " Lab partner: " + labPartner.getName();
if (homeworkGrades.size() > 0) {
    result += endl + " Homework grades: ";
    for (int i = 0; i < homeworkGrades.size(); i++) {
        Integer k = new Integer(i);
        result += " " + i + "-" + homeworkGrades.get(k);
    }
}
return result;
}

// Since these are "package protected" (without an explicit
// accessability modifier) only classes in the same package can
// invoke these methods.

void assignPartner(CSStudent partner) {
    labPartner = partner;
}

void assignHomeworkGrade( int assignmentNumber, String grade) {
    Integer assignment = new Integer(assignmentNumber);
    homeworkGrades.put(assignment, grade);
}

CSCourse.java

package week03;

import java.util.HashMap;
import java.util.Iterator;

/** A Java class that models information kept about a course and
 ** students in the course.
 **
 ** @author Jonathan Doughty
 ** @version 2.0
 **/
public class CSCourse {

    private Integer courseNum;

/** Dynamic collection of references to student objects enrolled ** in the class, indexed by id value. **/ private HashMap students = new HashMap();

/** Constructor for CSCourse objects **/ public CSCourse(int number) {
    courseNum = new Integer(number);
}

/** Enroll the student whose name is provided into this course. **/ public void enroll(String name, String id) {
    CSStudent current = new CSStudent(name, id);

    // Remember this student by id number
    students.put(id, current);
}

/** Assign pairs of students as lab partners for this course. **/ public void assignLabPartners() {

    // Assign every other pair of students as lab partners
    java.util.Set set = students.keySet();
    Iterator iter = set.iterator();

    for (int i = 0; i < students.size(); i += 2) {
        CSStudent student = (CSStudent) students.get(iter.next());
        if (iter.hasNext()) {
            CSStudent partner = (CSStudent) students.get(iter.next());
            student.assignPartner(partner);
            partner.assignPartner(student);
        }
    }
}

/** Assign a homework grade to the student identified by id for ** the specified assignment number. **/ public void setHomeworkGrade( String id, int assignment, String grade) {
    // Find the student object whose id is id
    CSStudent s = (CSStudent) students.get(id);
    if (s != null) {
        s.assignHomeworkGrade(assignment, grade);
    }
}
else {
    // Question: what happen?
    // The course has no memory of a student with this id having
    // enrolled.
}
}

/** List the roster of students enrolled in this course to
** standard output.
**/
public void listRoster(java.io.PrintStream out) {
    java.util.Set set = students.keySet();
    for (Iterator iter = set.iterator(); iter.hasNext(); ) {
        out.println(students.get(iter.next()));
    }
}

/** Provide access to the students enrolled in the course
**/
public Iterator getStudentIterator() {
    java.util.Collection collection = students.values();
    return collection.iterator();
}

/** Returns a String object representing this course’s number and
** current enrollment.
**/
public String toString() {
    StringBuffer buf = new StringBuffer();
    String endl = System.getProperty("line.separator");
    buf.append("Course number: ");
    buf.append(courseNum);
    java.util.Set set = students.keySet();
    for (Iterator iter = set.iterator(); iter.hasNext(); ) {
        buf.append(endl);
        buf.append(students.get(iter.next()));
    }
    return buf.toString();
}

CSTest.java

/** A Java class to test CSCourse objects
**
** @author Jonathan Doughty
** @version 2.0

package week03;

import java.util.Iterator;

public class CSTest {

    // Some test data, embedded as a pair of arrays
    // Question: why is this a poor idea in general (except for
    // little test classes like this?)

    static String[] names = { "Fred Flintstone", "Wilma Flintstone",
                            "Barney Rubble", "Betty Rubble",
                            "Pebbles Flintstone"     
    };
    static String[] ids = { "202-12-4575", "134-56-5210",
                           "562-45-1935", "265-720-2945",
                           "374-25-2195"     
    };

    /** The starting point for this application
     **/
    public static void main(String[] args) {
        // Create a new course object
        CSCourse cs332 = new CSCourse(332);

        // Ask the course to enroll students, assign lab partners,
        for (int i = 0; i < names.length; i++) {
            cs332.enroll(names[i], ids[i]);
        }
        cs332.assignLabPartners();

        // Assign some grades arbitrarily
        for (int assignment = 0; assignment < 5; assignment++) {
            for (int i = 0; i < names.length; i++) {
                // You know I don't really assign grades like this
                if (names[i].indexOf("Flintstone") > -1)
                    cs332.setHomeworkGrade(ids[i], assignment, "A");
                else
                    cs332.setHomeworkGrade(ids[i], assignment, "C");
            }
        }
    }
}
// Let each student in the course identify herself
Iterator iter = cs332.getStudentIterator();
while (iter.hasNext()) {
    CSStudent student = (CSStudent) iter.next();
    System.out.println(student);
}

Usage

- Given: the sources above in a directory named week03
- Compile the sources with:

  javac -classpath .. -d .. CSTest.java

- Run the result with:

  java -classpath .. week03.CSTest
Homework 3

Reading

- Chapter 4 - Object Interactions

Programming

This assignment has three goals: 1) is to get some experience creating helper objects, 2) To explore using Collection classes, 3) To further develop classes developed in last week’s assignment. **You may use your version or mine**

- Create a helper class called **PlaneId**. A PlaneId should be composed of two thing: 1) something that will uniquely identify an Airplane, either an Airplane object reference or the same String as is used to identify Airplanes when they are created); and 2) an Airline, either an Airline object reference or its name (the same string that is used to identify an Airline when an Airline object is created.)
- Create a class named **AirTrafficController**. An AirTrafficController object should have the following characteristics:
  - a collection of **PlaneId** objects that that AirTrafficController is responsible for monitoring.
  - a method to add a new **PlaneId** object to the collection
  - a method to remove a specified **PlaneId** from those in the collection, returning a reference to the **PlaneId** removed.
  - a method to return an Iterator for a AirTrafficController’s PlaneIds
- Modify the test class (or create a new one) so that will do the following:
  - Have the test class create at least two **AirTrafficController** objects
  - As the test class creates new Airplane objects and adds them to an Airline, have the test class also create PlaneId objects that will identify that Airplane / Airline combination.
  - Assign each PlaneId to one AirTrafficController. Be sure that each
AirTrafficController gets a roughly equal number of PlaneIds. That is, at least assign PlaneIds to a different AirTrafficController than the previous PlaneId was assigned to.

- Have each AirTrafficController list the PlaneIds of the Airplanes she is monitoring.

- Bonus: exercise AirTrafficController’s ability to remove a PlaneId that is being monitored from her collection and add that PlaneId to some other AirTrafficController. List the result before and after for at least the two AirTrafficControllers involved.