COMS W1007x: Object-Oriented Programming & Design
Fall, 2011
Assignment 2: Due Tuesday, October 18, 2011

Part 1: Theory (45 points)

The following problems are worth the points indicated. They are generally based on the book's exercises at the end of Chapter 3, under the section heading "Exercises". They cover issues in the design of classes.

"Paper" programs are those which are written on the same sheet that the rest of the theory problems are written on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work beyond the design and coding necessary to produce on paper the objects or object fragments that are asked for, so computer output will have no effect on your grade. The same goes for the theory portion in general: clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

1) (10 points) Suppose we define a class Equilateral, which can represent an equilateral triangle whose base is aligned with the horizontal directions of the display, using three parameters: x, y, s, where (x,y) are the coordinates of the center of the base, and s is the length of the base. Suppose we define the relationship "A contains C" to indicate that all of the area or perimeter of Equilateral C is in or on the area or perimeter of Equilateral A.

a) Show that this is a partial order (that is, show that it obeys each of the first three axioms on page 91).

b) Show that it is not a total order.

c) Using exercise 3.2 as a hint, define a total order for Equilaterals, and explain why this is not a very useful total order.

2) (10 points) Do Exercise 3.6, except write it to compute the number of days since Justin Bieber's birthday (3/1/94). Write a paper program.

3) (15 points) Do Exercise 3.10, except that the time is represented as 12:00:00a through 11:59:59p, where day and night is indicated by a short string. This means that the class stores three *int*s and one *String*. Make sure you can handle whatever happens if any of these parameters is outside their appropriate bounds; look up IllegalArgumentException. Write a paper program. You might find the answer given at http://www.horstmann.com/oodp2/solutions/solutions.html for Exercise 3.11 helpful.

4) (10 points) This exercise is based on Exercises 3.21 and 3.22. Critique the class ColorSpace, as it is given in the API, according to each of the "five C's" of section 3.5. That is, give an answer for each of: cohesion, completeness, convenience, clarity, consistency. You can use the answer to Exercise 3.21 in the on-line companion as a guide. Make sure you comment on its constructors and/or factory methods.

Part 2: Programming (55 points)

This assignment is intended to explore an alternative design to a class, just as Chapter 3 has alternative designs for the class called Day. Except, here we will use the class RLESequence ("Run Length Encoded Sequence") based on some of the ideas on the book's class of Matrix, introduced in Exercise 3.12, extended in 3.13 (with answers in the on-line companion, which should be examined!), and further elaborated in 3.14.

Please recall that all programs must compile, so keep a working version of each part before you proceed to the next. Please also note for this assignment, only digital (but not hardcopy) Javadoc output is required for all classes that you write. However, the Javadoc should be complete, including class comments, constructor comments (with @param), and method comments (with @param and @return, as appropriate). You also need to provide some test examples, particularly those cases that are "at the borders", but you do not have to use JUnit (which appears to have gone somewhat out of favor since the book was written).

1) Step 1 (15 points): Getting started.

Write a class called RLESequence. This class will demonstrate a compression technique actually used as part of the JPEG and MPEG encodings of images and videos, particularly for videos consisting of cartoons or video games consisting of animations, because those tend to have large areas of solid colors. You can use the on-line code for Exercise 3.13 on sparse matrices as an general example, even though you won't be able to take much re-usable Java from it. (Go to http://www.horstmann.com/oodp2/solutions/solutions.html and select Chapter 3 and Exercise 13; note that the solution has three parts.) You should also imitate the general design philosophy of the Day class in the book, particularly with respect to the encapsulation of the internal representation that you will using.

An RLESequence squeezes data storage by noting those values which repeat. It represents a sequence of numbers such as [5 5 5 5 1 1 3 3 3 3] as a sequence of "field code pairs", the first code of the pair giving a count, and the second code of the pair giving a value. So, the above example becomes something like: [4 5 2 1 4 3]. Since many regions of an image, such as the sky, or even skin in close-up, have many identical values next to each other in any given row of pixels, this encoding usually results in a great space saving. This is especially true of graphics and animations like video games or cartoons, where it can lead to compression factors of over 100. (As Homer Simpson would say, viewing a row of 500 pixels all identically zero: "D'oh!" which is really best understood using Roman numerals; the real answer is [500 0].)

In particular, this assignment should demonstrate how it is possible to write a class so that a user *doesn't know*--and in fact, *can't know*--how a sequence is stored internally, since on the surface it will still *operate* as if it were not compressed at all. Anything you can do to a sequence--for example: initialize, store, modify, retrieve, toString, etc.--is already provided by means of methods. But these methods hide from the user that the class has encapsulated all the secret codes and algorithms that make the implementation very efficient. It also permits the class to *modify* any of its internal representations and techniques at any time, without the user knowing that that has happened.

So, first start by considering a one-dimensional array of ints; this one dimensional array can capture the visual information on a single line of image pixels, where 0 usually means pure black and higher values

represent lighter shades of gray, usually stopping at 255 for pure white. Your job is to write the class so that it has some of the same methods as shown in 3.13, except applied to this new class. In particular, for this starting step, you will need to write: (a) at least two constructors, (b) at least one additional private method that establishes the internal data structures, (c) a method useful for outputting debugging information, and possibly (d) a helping class called Entry.

For (a), you need to write the constructor RLESequence(length), which establishes a RLESequence (in internal compressed form) that can hold "length" number of values. You need to decide if you should preload it with anything. In particular, what would an empty RLESequence look like? And, if you want, you can also establish a default constructor, RLESequence(), for a reasonable default length. It should be easy to write the accessor method length(). Further, you need to write the constructor RLESequence(input), which takes a standard one-dimensional array of ints (that is: "int[] input"), and which creates its compressed form.

For (b), to do the conversion of an array into the internal representation, you will probably need an internal method, compress(). The method compress() takes input, iterates over it looking for runs of same values, and creates and fills whatever data structure that you have decided will best represent a sequence of runs.

For (c), you need to write a method, internalToString(), that takes the internal representation of [4 5 2 1 4 3] and produces the string "[4 5 2 1 4 3]", unless you decide on a helper class, below.

For (d), depending on how you choose to implement the compressed form, you may or may not also need to have a helper class, Entry, as exercise 3.13 did. If you do use it, then the way to think of the internal representation is not so much as "[4 5 2 1 4 3]" in the above example, but more like "[(4,5),(2,1),(4,3)]". You should consider that Entry can simplify your life significantly if it has its own toString() method that can dump the internal form of an Entry into a string.

For all of Step 1, you should consider that it is difficult to predict how much compression will happen; again, see exercise 3.13, which uses ArrayList<> because it is easy to add and remove elements of an AL. Note that sometimes an array can be compressed to a single (count, value) pair, but at other times, the so-called compressed form may actually take up *more* space: consider [6 8 6 8 6 8 6 8 . . .].

Test your class by creating some RLESequences and printing them out in their internal form, that is, like "[4 5 2 1 4 3]" or "[(4,5),(2,1),(4,3)]".

2) Step 2 (15 points): Adding class functionality.

Complete the class by writing five more usual methods that make it more useful.

First, write toString(), which returns something the user expects, instead of the debugging information; that is, it should return something like " 5 5 5 5 1 1 3 3 3 3".

Then, write equals(myOtherRLESequence), which returns the correct boolean saying if two instances are the same in contents. Please consider that if the sequence is "7 7 7" can be encoded as [3 7], or [2 7 1 7], or [1 7 1 7 1 7], then all three of those are equal: but are two of these in error?

Then, write toArray(), which returns the standard one-dimensional array form of the sequence that the

instance has compressed and stored internally.

Finally, write the standard accessor and mutator methods get(i), where i is an index value into the original array, and set(i,v), where i is an index and v is the incoming value. For example, j = myRLES.get(3) in the above example should return 5. Indices should work like Java indices, starting at 0 as their base.

The biggest challenge will be with the method set(i,v). Consider the following cases, all based on the idea that the RLESequence is initialized to [5 5 5 5 1 1 3 3 3 3]

a) Starting from the initial values, someone says: myRLES.set(2, 4). The internal representation becomes [(2,5),(1,4),(1,5),(2,1),(4,3)], which is correct, but not much of a savings in space.

b) Starting from the initial values, someone says: myRLES.set(4, 5). The internal representation should become [(5,5),(1,1),(4,3)]. If this is then followed with myRLES.set(5,5), the internal representation should collapse to [(6,5),(4,3)].

c) Starting from the initial values, someone says: myRLES.set(6, 3). The internal representation should be unchanged.

d) Starting from the initial values, someone says: myRLES.set(3, -9). Depending on the ranges that your application can take, this may or may not be an error, since many graphics allow only non-negative integers within a restricted range. How should you handle this? Does this mean anything for your constructor?

e) Starting from the initial values, someone says: myRLES.set(10, 1). This is probably an indexing error, as there are only ten slots allocated, and their indices are from 0 to 9. How should you handle this?

There are a number of ways to write set(i,v). Consider that you have already written both toArray() and compress(). One obvious but expensive way is to fully uncompress the RLESequence into a standard one-dimensional array called temp; then execute temp[i]=v; then compress temp again. But, is there a better way?

If you do this Step, you do not have to submit separate code and testing for Step 1, as long as you test Step 2 as thoroughly as you tested Step 1, since this is an extension of it.

3) Step 3 (10 points): Adding two useful graphics functions.

Now, augment the class so that it can implement two other functions useful for graphics applications. Implement a method tailReplace(p, myOtherRLESequence), that takes the values in myOtherRLESequence and uses them to replace the values in the current instance (that is, in the implicit parameter), starting at p. Note that this can increase, decrease, or leave unchanged the size of the internal representation: for example, if [9 9 9 9 2 2 2] has [9 9 9 9] tail-replaced at position 4, then the internal representation should become simply [(8,9)]. There are a number of ways to design this method; you must defend in a comment whichever method you select.

Also, implement increment(byWhat). This method says that every value v in the sequence is converted to v+byWhat, that is, it is increased by byWhat. Can byWhat be zero? or negative? Can the result be too

big? Document your decisions. If you used a helper class Entry, this should be relatively clean and easy.

For both of these methods (and, in fact, for the entire class), you should consider whether it is a good idea to make your internal representations to be immutable, like Strings are. You must justify your decision.

If you do this Step, you do not have to submit separate code and testing for Steps 1 and 2, as long as you test Step 3 as thoroughly as you tested Steps 1 and 2, since this is an extension of them.

4) Step 4 (15 points): Creativity.

Pick one, and only one:

1) In real life, images have color. For humans, each image value ("pixel") has three values, one each for red, green, and blue.

Rework your entire design so that it works for color. This means that each constructor and method, including all internal methods and any helper classes, have to be extended so that a pixel is considered not just an integer, but a collection of three integers. There are a number of ways to do this; pick an approach and defend it. In particular, what does it mean now that "two pixels are the same"? You may find that the proper use of a helper class in your original design would make things considerably easier and would minimize the number of changes you have to make. Make sure you do adequate testing.

2) In real life, based on extensive experience, pixels are represented as byte values in the range of 0 to 255. Count codes are also represented as bytes, so the most you can compress at any given time is 256 pixels. (Or is it 255? Fencepost time! What does a count of 0 mean?) So, if you have a string of "9"s that are 500 long, this means you would have to say [(250,9),(250,9)] or something equivalent. But note that this still would only take up 4 bytes, which is the size of a single int; the standard representation of [(500,9)] takes two ints.

Rework your entire design so that your internal representation only uses bytes. Note in particular that this might affect the method increment(byWhat). Would it also affect your method equals(myOtherRLESequence)? Note also that this redesign may also mean that you have to do some additional exception handling. Make sure you do adequate testing.

3) In real life, images and videos are two-dimensional. Design a new class, RLEImage which uses RLESequence to build a two-dimensional array of pixels. It should also use run length encoding, except on the columns of a two-dimensional assembly of numbers. That is, if two rows are equal, then the representation notes that event, and then compresses them column-wise. For example, if an image looks like: [[0 0 0 0 0] [0 0 0 0 0] [1 1 1 1 1]] then you should have an internal representation that looks something like: [(2,rowA),(1,rowB)], where rowA is [(5,0)] and rowB is [(5,1)]. Note that "length" of an image takes on a new meaning here.

If you choose to do this creativity choice, you only have write: a constructor that takes a 2D int array and converts it to an internal representation; a toString method (plus an internalToString if you choose); an equals method; and a to2DArray method. That is, you just have to write a "codec", a compressor/decompressor. You still need to do all of RLESequence, though, since any get and set

operations will have to be done using a single row at a time.

If you do this Step, you do not have to submit separate code and testing for Steps 1, 2, and 3, as long as you test everything thoroughly.

General Notes:

For each step, design the system using whatever CRC, UML, or other technology you find helpful; we do not require you to submit the results of these tools, but you should find that they make the Javadoc easier. Then, write the documentation and the system, and text edit its classes into files. Compile them, execute them on your own test data, and debug them if necessary. When you are ready, submit your classes, your Javadoc output, and your testing; the classes and the testing have to also be in hard copy. Make sure that the source code is clear and its user interface is comprehensive and informative.

Write your classes clearly, with comments at the beginning describing what the class does, how it does it (i.e. in some form, talk about its CRC and UML), and how it will be tested. Put all the comments about testing in a Tester class. Show some of the test data in the comment, also. Document each constructor and method. Clear programming style will account for a substantial portion of your grade for the programming part of this assignment.

CHECKLIST FOR SUBMISSION:

For theory: Hard copy, handed in to Prof or TA by the beginning of class. No extra points for using a text editor. Neatly stapled! Name and UNI attached.

For programming: Hard copy, handed in to Prof or TA by the beginning of class. All code, all testing runs (cut and pasted from console, and/or captured screenshots). No hard copy javadoc is required this time. Neatly stapled, separately from the theory! Name and UNI attached.

Also for programming: Soft copy, submitted to the Course Files Shared Folder on courseworks by the beginning of class, in a tarball created by CUIT Unix tar command ("tar -cvzf uni_HW2.tar.gz whateverMyDirectoryIs"). *Please use "uni_HW2" as the name of your Courseworks submission.* Include softcopy of javadoc html. Name and UNI should be included as comments in every class. The code must compile. The last submission before deadline is the official one that will be executed if needed.