Stego my Ego: In which our hero hides information in waffles...

Test Program
Due: 09 May 2007 - 17.00 (5PM)

Exordium
A test program is a laboratory that you must complete on your own without the help of others. It is like a take-home exam. You may consult your text, your notes, your lab work, or our on-line examples and web pages (including all starter projects), but use of any other source for code is forbidden. You may not discuss this program with anyone aside from the course instructors. You may only ask the TAs for help with hardware problems or difficulties in retrieving your program from a disk or network.

Note that while you should not ask other students (including TAs) questions about this lab, you are allowed to ask either of the instructors any question you want while you are working on this assignment. In some cases, we will decline to answer your questions, just as an instructor might during an in-class examination. Do not, however, hesitate to ask. We can and will answer many of your questions during this assignment.

As we have done for all of your labs this semester, this handout provides suggestions for how to organize the code required to implement the program described below into classes and methods. While we strongly urge you to follow these suggestions, they are just suggestions, not requirements. If you find some other way to implement the desired functionality is easier for you to understand, you should not feel bound to follow our design and implementation suggestions. Your final code, however, should be clear, reasonably efficient, well documented, and follow the guidelines for good programming style expected throughout the semester.

Steganography
Let’s start off with a definition. The Wikipedia entry for steganography provides the following:

> Steganography is the art and science of writing hidden messages in such a way that no one apart from the intended recipient knows of the existence of the message; this is in contrast to cryptography, where the existence of the message itself is not disguised, but the content is obscured.

Generally, a steganographic message will appear to be something else: a picture, an article, a shopping list, or some other message. This apparent message is the covertext. For instance, a message may be hidden by using invisible ink between the visible lines of innocuous documents.

The advantage of steganography over cryptography alone is that messages do not attract attention to themselves, to messengers, or to recipients. An unhidden coded message, no matter how unbreakable it is, will arouse suspicion and may in itself be incriminating, as in countries where encryption is illegal.

In our final laboratory project this semester you will implement a very basic steganographic system. The program you complete will implement two pairs of data transformations.

First it will provide the ability to hide a secret image within the pixels of a cover image and to reveal such secret images. You have already seen an example illustrating how this can be accomplished. Remember how we hid a picture of the Pentagon inside a picture of Griffin Hall? In that case we used the low 3 bits of each pixel to encode the image of the Pentagon while the high order 5 bits were used to encode the image of Griffin Hall. The variation in the brightness of a pixel that results when only its low 3 bits are changed is so small that we could use those bits to record the brightness values of the Pentagon image without perceptibly changing the image of Griffin. On the other hand, when we extracted the low 3 bits and expanded the range of those bits, a clear picture of the Pentagon emerged, even though only 8 distinct shades of gray could be encoded for each pixel.

While we replaced the lowest 3 bits of each pixel of our cover image with values from the secret Pentagon image, we want you to use even fewer bits for your secret images. In particular, you should replace just the
lowest bit in each pixel brightness value in the cover image with the high bit of a brightness value from the secret image. That means you will only preserve two color levels for each brightness value of the secret image. If you try to hide a grayscale image, it will be reduced to a black and white image. If you try to hide a color image, only 8 shades of color will be preserved in the hidden version since you will only have two levels of red, two levels of green and two levels of blue.

Once you have completed the code to hide one image within another, you will implement another pair of transformations that will make it possible to efficiently hide text within an image. First, you will write code to convert a String of text into a black and white image. Each pixel of the image produced will correspond to one bit from the ASCII representation of the String being encoded. If the bit is a 1, then the pixel will be bright. If the bit is a 0, then the pixel will be black. The first bit of the first character of the text will determine the color of the pixel in the upper left corner of the image. The second bit of this character will determine the color of the next pixel in the first column. The remaining bits of the first character and of succeeding characters will be placed in pixels moving down the leftmost column of the image until the end of the column is reached. Then you will continue the process starting with the top bit in the second column and so forth. A sample of an image produced in this way is shown to the right.

The pattern of 1s and 0s used to encode a piece of text can be pretty random. That's why the encoded image looks like the matrix (take the red pill and stay in wonderland). As a result, encoding text in this sort of image would not make sense in most contexts. However, if you have a program that can hide black and white images under cover images steganographically, converting text into random looking images is quite handy. You can then hide the image itself under a cover image providing a way to hide quite a bit of text in a small image (the entire Gettysburg Address is encoded in the image shown on the right).

Accordingly, as the second component of this assignment, we want you to incorporate in your program the ability to convert text into images as described above and vice versa. Your program's interface will be designed so that you can combine the text to image conversion with the ability to hide secret images under cover images. This will make it possible to encode text within a cover image.

**User Interface**

An example of the user interface is given below. It is similar to the dual image viewer you’ve seen in previous labs. This time, however, the window is divided into three sections displaying three images. The left image is the *cover* image, the middle image is the *secret* image and the right image is the *encrypted* image. We will use these names throughout the exposition so familiarize yourself with them now.
We will provide a simplified version of the ImageViewer class to help you construct this interface. This version of ImageViewer includes only a Load Image button and a JLabel used to display an image. Each of the three sections of your program’s window will hold an ImageViewer.

The left panel holds a cover image. The rightmost panel holds an image formed by combining a cover image with a secret image. The middle panel is further subdivided into two regions. The bottom holds an ImageViewer designed to display a secret image. The top of the center panel holds a JTextArea placed within a JScrollPane. The image displayed in this panel can be obtained in three ways. It can be loaded from a file using the Load Image button. It can be extracted from the image on the right using the decrypt button. Finally, it can be formed from text entered in the JTextArea located above the image by pressing the Encode button.

The nicest way to divide the center panel into two regions is to use a JSplitPane. We discuss JSplitPanes below. The left and right ImageViewers along with the JSplitPane are placed in a JPanel using a GridLayout with 1 row and 3 columns. This JPanel is placed in the center of a contentPane using a BorderLayout. The six buttons are placed in a JPanel which is added to the south of the contentPane. The JSplitPane is similar to a GridLayout with 2 rows and 1 column except you can move the center divider. Since we want a JSplitPane that splits its two components vertically, you should create it using:

```java
JSplitPane split = new JSplitPane(JSplitPane.VERTICAL_SPLIT)
```

You can set the top component with the setTopComponent method and the bottom component with the setBottomComponent method. Additionally, use setDividerLocation(0.5) to set the divider halfway between the top and bottom components. This will not work if you do it before all of the window components are placed in the contentPane since Java cannot determine the actual size of the JSplitPane until it and all your other components have been placed in the contentPane. Accordingly, place the invocation of setDividerLocation at the very end of your StegoDisplay constructor.

**Overview and Functionality**

**Encrypting**

The Encrypt button encrypts the secret image inside the cover image. The result is an encrypted image. We encrypt the secret image into the cover image by first clearing the low bit of every pixel of the cover image. This has the side-effect of making every pixel value even. For example, 253 becomes 252 but 68 remains the same since its lowest bit is already 0. Next we re-scale the secret image down to 0 / 1 brightness values. Finally, we add the brightness values for the scaled version of the secret image to the bit-cleared cover image. That is, for each pixel in the encrypted image, we add the pixel at that point in the cover image to the pixel at that point in the secret image – *hiding* the secret image in the cover image. The result of this addition is the displayed on the right as the encrypted image. Since we only alter the lowest bits, it’s essentially indistinguishable from the original cover image.

**Decrypting**

The Decrypt button takes the image in the rightmost ImageViewer and extracts the low order bit from each pixel to obtain an image where each pixel has value either 0 or 1. To do this, we make a copy of the encrypted image, clear the low order bit for each brightness value, and subtract this copy from the original encrypted image. Finally, we scale the pixel values of the image by multiplying each value by 128 because looking at black images isn’t that much fun. This should yield a two-tone version of the secret image.

**Encoding**

The Encode button takes the text from the JTextArea and produces a secret image. The encoded image should have the same width and height as the cover image. Therefore, there must be a cover image present for encoding to work. Trying to encode text when no cover image is present should have no effect.

The details of how to encode text as images are outlined later, but in general it means computing an array of ASCII values for each character in the text field and converting those ASCII values to binary. This results in
a large array of bits. Every 8 bit block encodes some ASCII value between 0 and 255 which in turn encodes
some character. We could encode the array of bits as an image with two pixel values: 0 and 1. However,
both 0 and 1 appear black on the screen so we will instead use a brightness value of 128 to encode each 1 and
a brightness value of 0 to encode each 0.

Decoding
The Decode button assumes that the secret image displayed in the center panel was constructed by encoding
text and consequently, attempts to extract this text. Extraction can be done by first dividing each brightness
value by 128 to obtain the original 0/1 values used to encode the text in ASCII. Then each block of 8 bits
can be converted to the corresponding ASCII character and these characters are joined together to form a
String. Since the encoded text may not have been long enough to provide values for all of the values of the
image, many of the 8 bit blocks will have only zero bits. We ignore such blocks when decoding the image
back into text.

Implementation
Begin this lab by implementing the user interface. Call your top-level class StegoDisplay. Make sure it
extends GUIScreen. You can leave your buttonClicked method empty for now except for the Clear
Text and Clear Image button code. Clearing text from the text field means setting the
JTextArea to the
empty string. Clearing the encoded image is similar – just set the image to be a small all-white SImage.
Recall that SImage has several constructors, one of which takes a width, a height, and a brightness level and
returns an SImage of appropriate brightness and dimensionality. We always create an image with width 100
and height 100. Test your Clear Text and Clear Image button code to make sure it works. We’ll fill in the
code for the other buttons later.

Encrypting Images
Encrypting messages has three steps. First we take a cover image and clear the low order bit of each pixel.
That is, we make it zero. This is equivalent to requantizing the cover image so that it uses only 128 distinct
levels of brightness. Then we reduce the range of brightness values used in the secret image from 0-255 to 0-
1. This can be done by dividing all pixel values by 128. Finally, add the rescaled secret image to the requa-
nitized source image.

Bit Clearing
Recall that integer division in Java always produces another integer. As a result, consider what happens if we
first divide a number by 10 and then immediately multiply the result by 10. The final result will always be
the same as the original value with the rightmost digit replaced by a 0. For example, 354 divided by 10 is 35
and 35 times 10 is 350.

Division by 2 in binary behaves the same way as division by 10 does in decimal. If we divide a binary num-
ber by 2 and then multiply by two, the result will be the same as simply replacing the last digit of the initial
value with 0. For example, the number 27 represented using 8 bits in binary is 00011011. We will always
write the bits from most significant to least significant. Dividing 27 by 2 yields 13 which is 0001101 when
encoded as an 8 bit binary number. Notice that 0001101 is exactly 00011011 shifted to the right one bit
(with the final bit chopped off and a 0 bit append to the front). Similarly, multiplying by 2 shifts the binary
number to the left by one bit and fills in the low order bit with a 0. For example, if we shift 0001101 to the
left by one bit we get 00011010 which is 26. In other words, we can clear the low order bit of a pixel by first
dividing by 2 and then multiplying by 2.

Implementation
Create a class called BitClearer that extends ImageFilter. You can approach the implementation of this
class in two ways. The easiest way is to take advantage of the Scaler class we provided in an earlier lab.
Recall that Scaler takes an image and first multiplies each brightness value by a given factor and then di-
vides each value by another factor. You can create one Scaler to divide all of the brightness values in an
image by 2 and another to multiply all the values by 2 and then use these two Scalers to do all the work. If
you take this approach, you should override the filter method of the ImageFilter class in your defini-
tion of BitClearer. Alternately, if you are more comfortable writing loops to process all of the pixels yourself, you can define BitClearer by mimicking the code found in our Scaler class. The only big differences will be that a) Scaler multiplies first and then divides while you will want to perform the operations in the opposite order and b) the values to multiply and divide by are passed as parameters to Scaler while your class can simply use 2. If you take this second approach, you will override the pixelFilter method. Note that regardless of which approach you choose, BitClearer does not need a constructor since implementing filter only requires an SImage.

Scaling Images
The easiest way to rescale the secret image so that all of its brightness values are either 0 or 1 is to divide all of the original brightness values by 128.

Implementation
You should find that the Scaler class provides just the tool you need to rescale the secret image.

Adding Images
For our purposes, adding two images together is as simple as adding two pixel values together. Given two images of the same dimensions, we wish to create a new image where each pixel in the new image is exactly the sum of the pixels in the original images.

Implementation
Create a class called Adder that extends ImageFilter. The constructor for Adder takes a single parameter of type SImage called left. Override the layerFilter method so that given another SImage right and an RGB layer, you extract the appropriate RGB layers from left and right, add the pixels together and return a new two-dimensional array of pixels.

Use the Adder, Scaler and BitClearer classes to implement the correct functionality of the Encrypt button. That is, clear the low bits of the cover image and add it to the scaled secret image to form the encrypted image.

Decrypting Images
Decrypting images also has three steps although you’ve already completed one of them – bit clearing – and the functionality of the others – differencing and range expansion/rescaling – is already provided. Recall that decrypting the encrypted image means creating a copy of the image, clearing the low order bits from each brightness value in the copy, differencing the copy from the original, and finally rescaling the brightness values from 0s and 1s to 0s and 128s.

Implementation
Since you already implemented BitClearer and since Differencer and Scaler are provided in the starter project, implement the Decrypt button functionality by creating a Differencer with the encrypted image as the original. Then clear the bits of the encrypted image using the BitClearer. Notice you don’t actually need to make a copy of the encrypted image since BitClearer constructs a new SImage. Now use the filter method of Differencer to extract the secret image. Make sure you scale the secret image up by a factor of 128 before displaying it. You can now test your Encrypt and Decrypt functionality by loading an image file which came in the starter project. First load test-encrypt.png into the ImageViewer of the secret image. Next load the same image into the ImageViewer of the cover image. test-encrypt.png is comprised of pixels with value 0 and pixels with value 128. It serves as a simulacrum for an image produced by encoding text in an image. We need an image with the same dimensions as our source so that you can actually perform the encryption. Encrypt the secret image, clear the secret image, and then decrypt the encrypted image. You should see the secret image reappear. It’s worth noting that this secret image almost surely corresponds to a beautiful sonnet. Alright, it probably doesn’t.
Encoding Strings as Images

Encoding strings as images is a two step process. First we convert a string of text into an array of bits. Then we form an image by treating each bit as a pixel.

Converting Strings to Arrays of Bits

Early in the semester we talked about how ASCII provides a way to encode characters as 8-bit binary strings. For example “A” has ASCII code 65 so it corresponds to the 8-bit binary string 01000001. You can determine the numeric value of the ASCII code for any character in a `String` very conveniently in Java. You can extract a character from a `String` by providing the position of the desired character as a parameter to the `String.charAt` method. This method returns the character at that position as a value of type `char`, which Java views as a slightly special form of integer value. For example, we can grab the ASCII value of “A” using the following lines of Java

```java
String s = "ABC";
int val = s.charAt(0);  // val is 65
```

This, however, is only the first step if you want to set the brightness values of an image’s pixels based on the individual bits used to encode some text in ASCII. We now need a way to convert `val` to its binary representation.

Converting an integer into its binary representation is based on the same observations about binary arithmetic that we used to clear the low order bit of a brightness value. We can first check if the low order bit of an ASCII code is 1 by checking if the number is odd. Then, we can shift the number to the right by one bit by dividing it by 2. Repeating this procedure 8 times while recording whether the low order bit was a 0 or a 1 yields the 8 digit binary representation of an ASCII value.

Implementation

Rather than converting text directly into an image, we will first convert a `String` into an array of integers in which each integer in the array holds the value of a single bit of the ASCII representation of the `String`. For example, if a `String` `text` has length 10, then each character in `text` requires 8 bits in its binary representation. Therefore the array of bits for `text` will have length 80.

The key to converting Strings to arrays of bits is carving up the steps correctly.

Create a class called `StringConverter`. The class should have two private methods and two public methods. The two public methods:

- `public int[] convertFromString(String text)`
- `public String convertToString(int[] bits)`

will be used to convert a `String` into an array holding the individual bits of its ASCII representation and vice versa. These methods will be implemented using the two private methods

- `private int[] convertFromString(int x)`
- `private int convertToInteger(int[] bits)`

which will perform similar conversion operations for single characters rather than for complete `Strings`.

We’ll concentrate on `convertFromString` and `convertFromInteger` methods for now. `convertFromString` takes the integer value of an ASCII code and returns an array of 8 bits where the first bit represents the most significant bit. This means that as you convert your integer into its binary format, you’ll need to fill in the array from back to front. *Going from front to back is super bad!*
convertFromString begins by allocating an array of integers of appropriate size to hold the 8-bit binary representation of text. It iterates over the characters of text, using charAt to extract the ASCII value of each character in the String. It then calls convertFromInt and copies the result into the final array of integers. Note that when converting the character appearing at index 3 of text, we will copy the 8 bits representing that character into the final array starting at position 8*3=24. For example, convert(“AX”) should return a 16 bit array of integers with contents [0 1 0 0 0 0 1 0 1 0 1 1 0 0 0] since the ASCII code for A is 01000001 and the ASCII code for X is 01011000.

Converting Arrays of Bits to Images
Earlier in class we talked about how two dimensional arrays are really just arrays of arrays. It’s also possible to think of two-dimensional arrays as a single array where each column array is just concatenated end-to-end with its neighbor. In this way, we can copy the bits of our bit array into the consecutive columns of an image.

Implementation
For the process of converting between a 1-dimensional array of bits and a two dimensional image composed of pixel values, you should define two private methods within your StegoDisplay class:

- SImage convertToImage(int[] bits, int width, int height)
- int[] convertFromImage(SImage image)

We’ll address the first one now and leave the second until later. convertToImage takes an array of integers (each of which represents a single bit) together with two integers specifying the dimensions of an image and returns a greyscale SImage of the desired dimensions such that the brightness of the pixel at location (x,y) is equal to 128 times the value at position height*x + y in the bit array. We can accomplish this by iterating through the image and array of bits simultaneously. For example, given the array [0 1 0 0 0 0 1 0 1 0 0 0 1 1], an integer width 4, and an integer height 4, we’d return an SImage with pixel values

```
0    0    0    0
128    0    128    0
0    0    0    128
0    128    0    128
```

If your array of bits runs out first, make the remaining pixel values 0. For example, given the array [0 1 0 0 0 0 1 0 1 0 0 0 1 1], an integer width 4, and an integer height 5, you should return an SImage with pixel values

```
0    0    0    128
128    0    0    0
0    128    0    0
0    0    0    0
0    128    128    0
```

(where the added zeroes are shown in italics). If the image pixels run out first, then stop encoding the array of bits – decoding the image just means the original text will be truncated.

You can now implement the Encode button. Begin by converting the text in the JTextArea to its binary representation using the convertFromString method of StringConverter. Next use the convertToImage method of StegoDisplay to generate an SImage.

Decoding Images into Strings
It’s now time to reverse the process of encoding. While the details may not yet be clear, the main idea is to take a two-dimensional array of pixels and make it linear by copying it, one column at a time, into an appro-
appropriately sized array of integers. Once we have the array of integers, we can read consecutive blocks of 8 bits to decode the ASCII values into characters and recover the original text.

**Converting Images to Arrays of Bits**

Given an image with dimensions width and height, we should create an array of length width*height to store our bits. Then we can simultaneously iterate through the image (column by column) and the bit array filling in 1s for pixel values greater or equal to 128 and 0s for smaller pixel values.

**Implementation**

We begin by writing the `convertFromImage` method of the `StegoDisplay` class. This method takes an `SImage` and returns an array of integers representing bits. These bits correspond to the binary ASCII values of the original text. Since your bit array is sized according to the dimensions of the image, you should have one bit for every pixel in the image. We assume that the encoded image is grayscale so extract its pixel array using the default `getPixelArray()` method without any parameters.

**Converting Arrays of Bits into Strings**

Converting arrays of bits into strings is, like many dances, done in two-steps. We begin by iterating over the bit array in 8-bit blocks, extracting each block, and converting that 8-bit array of bits into its corresponding decimal ASCII value. Next we convert this ASCII value into a `String` and append it to our string buffer.

Converting an array of bits into an integer is quite natural given that we’ve already discussed how to do the converse. Notice that shifting an array of bits to the left by one bit is the same as multiplying its decimal value by 2. For example, if 110 is our 3-bit binary number with decimal value 6 then 1100 is a 4-bit binary number with decimal value 12. If the bit we acquire on the right is a 1 instead of a 0, we multiply by 2 and then add 1. That is, we always double the total and add the new bit. We can use this idea to convert our array of bits into its decimal value. We begin by keeping an integer value `total` that corresponds to the current value of our 8-bit binary number. Then we shift our bit array to the left, and acquire the next bit in our bit array. Likewise, we double `total` and add the newly acquired bit. For example, say our bit array is [ 0 0 0 0 1 0 1 ] and we want to convert it to its decimal value. We initialize `total` to 0. Starting with the bit at position 0, we double `total` and add 0 since the bit at position 0 is 0. This means `total` has value 0. The steps for positions 1, 2, 3, and 4 will be similar since all of the array elements at these positions are 0. Once we get to position 5, however, the array value found is 1. Therefore, we will again double `total` but this time we will add one leaving the value 1 in `total`. Next we move to the bit at position 6. We double `total` and add 0. This means `total` has value 2. Finally we move to the bit at position 7. We double `total` and add 1. This means `total` has value 5 which is exactly the decimal value of 101.

Once we have the decimal value of our bit array, we need to convert it into its corresponding ASCII character, create a `String` from the character, and append that `String` to our current buffer. In Java, one converts an integer ASCII value into its corresponding character value by casting the integer as a `char`. Then, we can concatenate this `char` onto a `String` much as we can concatenate two `Strings` together. For example

```java
String text = "HAH";
int x = 65;   // ASCII FOR A
char c = (char) x;

String text = text + c; // text equals "HAHA"
```

**Implementation**

Start by writing the private `convertToInt` method in `StringConverter`. This method takes an array of integers (representing bits) and returns an `int` representing its decimal value. Use the doubling plus adding procedure outlined above to accomplish this implementation.

Next, implement the `convertToString` method of `StringConverter`. This method takes an array of bits and returns a `String` representing the ASCII characters of the consecutive 8-bit blocks of binary numbers in
the bit array. You’ll probably want to do this with two for loops. If there are length bits in the array, the outer for loop will perform length/8 iterations. The inner for loop will always perform 8 iterations. Most likely your inner for loops will copy the 8-bit block into an array of 8 integers and then use convertToInt along with the Java code above to recover the ASCII String. You’ll append this String to your overall buffer and return the buffer after the iteration is finished. There is one caveat. If the converted integer value is 0, then the original 8-bit block had 8 consecutive 0s. This corresponds to the NULL character in ASCII and not surprisingly, to where we stopped encoding our text in the image. We do not want to include the NULL characters in our decoded text so don’t convert them to ASCII or add their values to the final String.

You can now implement the Decode button. Begin by grabbing the secret image and converting it to an array of ints using the convertFromImage method of your StegoDisplay class. Next, convert this array of bits to a String using the convertToString method from StringConverter. Now set the JTextArea to the recovered String. Test your decode method by using the test-decode.png image file located in the startup package.