

# Design Specification

## For

# Paranoia

U-S-E  
CS169 – Software Engineering  
Oct 28 2003

Prepared by:

BERDAHL, EDGAR  
CHUNG, SANG  
GONG, GARY  
LE, TAM  
LEE, JOON YUL  
LI, SHENG  
MOZAFFARIAN, BEHRAD  
MURRAY, JASON  
NGUYEN, HIEP  
SHKOLNIKOV, YURIY

# Table of Contents

Table of Contents .....	2
1. Introduction.....	3
1.1 Abstract.....	3
1.2 Revisions to Functionality Specification .....	3
1.3 Project Schedule.....	4
2. Project Description.....	5
2.1 Platforms and Software Tools .....	5
2.2 Interaction of Main Objects .....	6
2.3 Objects and Methods.....	7
2.4 JPEG Stegonography Methods and Algorhythms .....	24
2.5 GIF Stegonography Methods and Algorhythms .....	30
2.6 Walkthrough Flowcharts .....	36
2.7 Security Features .....	39
3. Project Process .....	40
3.1 Code Review .....	40
3.2 Module Testing.....	41
3.3 Integration Testing .....	49
Appendix A: Glossary.....	50
Appendix B: JPEG Examples and Parameters.....	51

# 1 INTRODUCTION

## 1.1 ABSTRACT

Paranoia allows an average user to securely transfer text messages by hiding them in a digital image file. A combination of steganography and encryption algorithms provides a strong backbone for Paranoia's security. Paranoia features innovative techniques for hiding text in a digital image file or even using it as a key to the encryption.

## 1.2 REVISIONS TO THE FUNCTIONALITY SPECIFICATION

- Our group would like to clarify its commitment to produce images “visibly similar to the original images”. In the case of JPEG images, we will attempt to achieve this goal by only making slight alterations to the luminance coefficients. This will increase the likelihood the output will retain its visual similarity. In the case of GIF images, increasing the color map on GIF images is an option at this point.  
However, we cannot not guarantee the user that any image will store any given text and maintain its visual similarity. Our help file will describe the types of files that are ideal for our steganography process.
- For every image, the steganographic capacity will be calculated and compared to the volume of text data to be stored. If the volume of that data exceeds the capacity the user will not be able to use that image to store the text. The user will always be able to repeat the process with another image file if the initial results are unsatisfactory or cannot be completed.
- Due to the different steganography methods for different file formats, the ‘Save As’ function on the ‘File Menu’ will allow the user to change the name and/or location of the file but not the format. Otherwise the ‘Save As’ function could possibly change the appearance of the image while converting between file formats.

## 1.3 SCHEDULE AND MILESTONES

### 1.3.1 GUI Group – Gong, Li, Murray, Shkolnikov

10/23	Revise data structures
10/25	Start Coding
11/1	Done Coding
11/5	Modular Testing
11/10	Begin Integration Testing

### 1.3.2 Security Group – Le, Nguyen

10/23	Revise data structures
10/25	Start Coding
10/28	Dummy class for JPEG & GIF people
11/5	Done Coding
11/7	Modular Testing
11/10	Begin Integration Testing

### 1.3.3 JPEG Group – Berdahl, Lee

10/26	JPEG files can be opened and the contents properly interpreted. JPEG files can be opened, converted to a <code>BufferedImage</code> and then saved in a new file.
11/2	Information can be embedded in and extracted from grayscale JPEG files.
11/7	The module will work with color JPEG files, even though the information will only be hidden in the luminance values. Modular testing.
11/10	Begin Integration Testing.

### 1.3.4 GIF Group – Chung, Mozaffarian

10/30	Open GIF file, convert into <code>BufferedImage</code> , read or write something from it, extract and modify RGB values
11/3	Create internal data structures for <code>colormap[]</code> , <code>neighbormap[]</code> , and <code>pixelmap[]</code> .
11/7	Write out GIF file with the GIF 89a standard. Modular testing.
11/10	Begin Integration Testing.

# 2 PROJECT DESCRIPTION

## 2.1 PLATFORMS AND SOFTWARE TOOLS

Paranoia will be developed using the Java Development Kit (JDK) 1.4.2. Any device that can support a Java Virtual Machine for JDK 1.4.2, along with the necessary memory and disk storage space will be able to execute our application.

For construction of the GUI we would like to use JBuilder by Borland.

Revision and code management will be done using CVS.

Furthermore, in order to aid in the final documentation process, we will comment our code such that it can be parsed by JavaDoc. For more information see, <http://java.sun.com/j2se/javadoc/>.

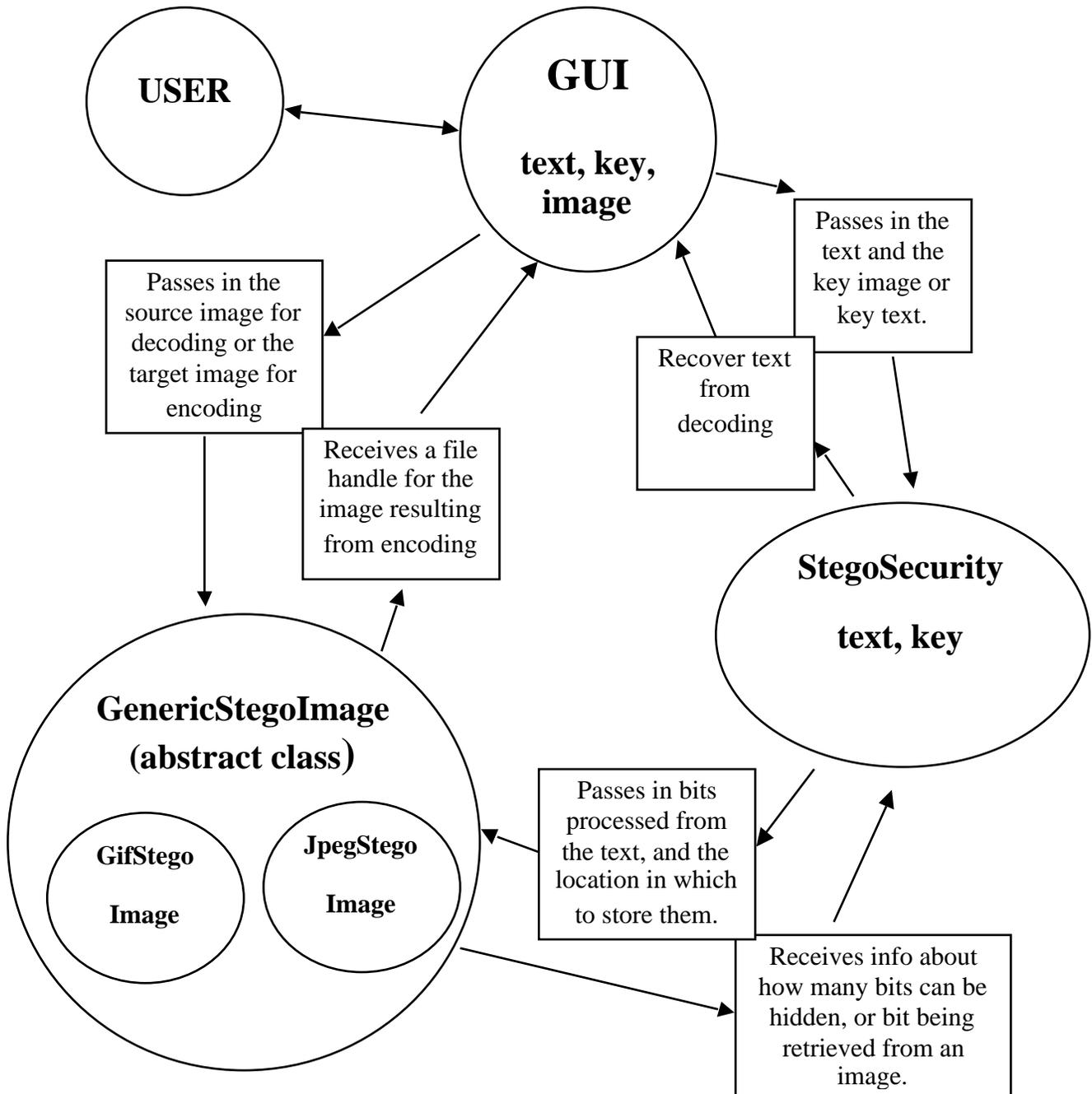
## 2.2 INTERACTION OF MAIN OBJECTS

The project team will split into four groups. Each group will work on one of the major objects in our implementation. Here is a general overview of how these objects interact.

*GUI*: The graphical user interface.

*StegoSecurity*: A package of compression, encryption and bit placement algorithms used in conjunction with our steganography.

*GifStegoImage* and *JpegStegoImage*: Packages that handle steganography for particular file formats.



## 2.3 OBJECTS AND METHODS

### Class GUI

[java.lang.Object](#)

└ Huffman

**All Implemented Interfaces:**

none

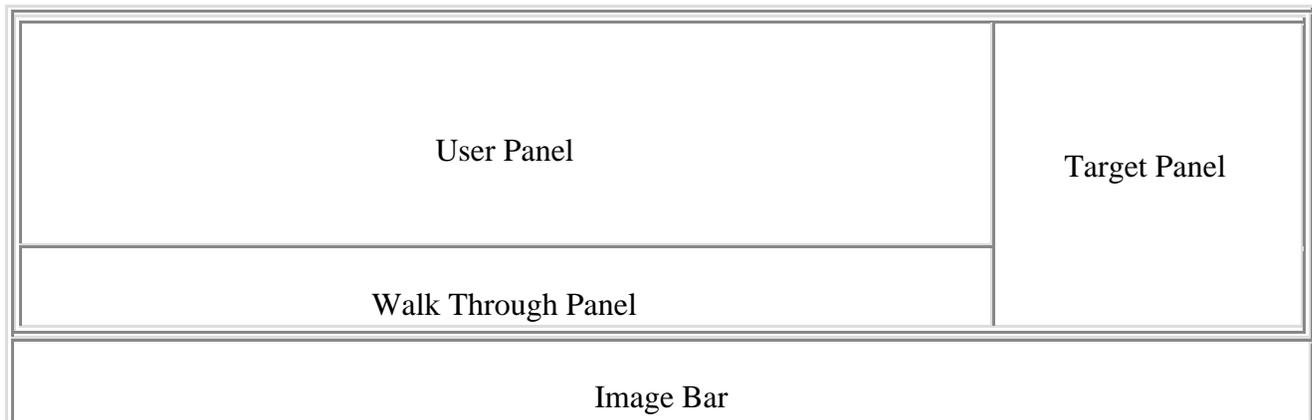
**Direct Known Subclasses:**

none

---

```
public class GUI
  extends JFrame
  implements WindowConstants, Accessible, RootPaneContainer
```

The GUI has a menubar and is separated into 4 parts.



The GUI will contain a `StegoSecurity` object, and will instantiate `GifStegoImage` and `JpegStegoImage` objects as needed to encode and decode images. `User` will respond to all of the user's events and set the progress bar to visible when an encryption process is started.

---

Field Summary	
integer	WalkThroughState Contains information regarding the current state of the walkthrough.
JFrame	progressPopUp A dialog box for the progress bar.
JProgressBar	codingProgress Shows the user the status of the coding process.
User Panel	
JPanel	userPanel The main panel of the application contains a text tab and an image tab as described below.
JTabbedPane	userTabStrip This strip contains the text box for entering the user's text and an image box to display the result image.
JTextPane	userText This is the main text area into which the user types the text to be hidden within an image. In the case of extracting text from an image, the text would be loaded into this same tab.
BufferedImage	resultImage This tab is also located within the main panel. Its sole purpose is to display an image, which has been produced as a result of embedding text within another image.
Target Panel	
JPanel	targetPanel The target panel contains the source and target images as well as buttons for loading them.
JTabbedPane	keyTabStrip This strip contains the key image and a text box for entering the key manually.
BufferedImage	keyImage This is the image that will be used a key.
JTextField	keyText This text box will be where users can manually enter keys.
JButton	loadKey This button will load the image, related to the currently selected thumbnail in the image bar, into the key image.

BufferedImage	targetImage This is the image that will be used the target for encoding or source for decoding.
JButton	loadKey This button will load the image, related to the currently selected thumbnail in the image bar, into the target image.
<b>Walkthrough Panel</b>	
JPanel	walkthroughPanel This panel contains the necessary components to perform the user walkthrough.
JButton	nextButton This button will initially have caption “Encode” which will change to “Next” as the user begins/follows the walkthrough.
JButton	backButton This button will initially have caption “Decode” which will change to “Back” as the user begins/follows the walkthrough.
JButton	cancelButton This button will initially have caption “Cancel” which will be greyed out until the user starts the walkthrough.
JTextPane	walkthroughInfo This box will contain instructions to the user on what steps to perform next.
<b>Image Bar</b>	
ImageBar	imageBar A scrollable pane with thumbnail images.
<b>Constructor Summary</b>	
GUI() Instantiates, adds, and sets listeners for all interactive components. Instantiates StegoSecurity object and JProgressBar dialog.	

<b>Method/Menu Summary</b>	
void	walkThrough (integer step) Will process integers indicating the button presses by the user and update the walkthrough button captions and the information text displayed.
<b>File Menu</b>	
	<i>Start Over:</i> Frees current work so user can restart without any loaded documents, or work in progress.

	<i>Open Text:</i> This menu item simply opens a file dialog, which allows the user to open a text file into the main text box.
	<i>Save Text:</i> Allows user to save the text in the main text box; usable if text is present in main text box.
	<i>Save Text As...:</i> Allows user to save the text in the main text pane as a different file than what is was opened from.
	<i>Open Image:</i> Opens an image into the Target/Source Image Pane, which can be used to extract text from or to embed text into.
	<i>Save Image:</i> Allows user to save generated Image with embedded text, located in main image box;
	<i>Save Image As...:</i> Allows user to save generated Image with a different file name than it already has.
<b>Edit Menu</b>	
	<i>Cut:</i> Removes the selected text from the document and puts it into the clipboard, replacing previous clipboard contents.
	<i>Copy:</i> Puts the selected text into the clipboard, replacing previous clipboard contents.
	<i>Paste:</i> Inserts the text on the clipboard into the document, beginning from the current position of the cursor.
<b>Tools Menu</b>	
	<i>Embed text:</i> Embeds the text in the Main Panel into the selected image in the Target.
	<i>Extract text:</i> Extracts text from an image and displays it in the Main Panel. Click.
	<i>Open image key:</i> Opens a file browser to select an image to use as a key. When the user selects an image, it will appear in the Key Pane.
	<i>Open text key:</i> Opens a file browser to select a text file to use as a key. When the user selects a text file, its contents will appear in the Key Pane.
	<i>Save text key:</i> Saves the text key in the Key Pane by opening a file browser to select a location to save the text file. It is usable when text has been entered in the Key Pane.
	Submenu: Tools <b>à</b> <b>Options</b> <i>Encryption ON/OFF:</i>
<b>Help Menu</b>	
	<i>Help:</i> Links the user to HTML help documents.

# Class ImageBar

[java.lang.Object](#)

└ ImageBar

**All Implemented Interfaces:**

none

**Direct Known Subclasses:**

none

---

```
public class ImageBar
    extends JPanel
    implements Accessible
```

A scrollable pane with thumbnail images.

It uses a smaller class:

//A placeholder object for thumbnail images. We need this because an Image cannot be directly added to a Container, but a JPanel can.

Thumbnail extends JPanel

```
// Holds the thumbnail.
```

```
// Image thumbnail;
```

```
// Holds the image's filename.
```

```
// String fileName;
```

```
// Overrides the JComponent method for painting the component.
```

```
// void paintComponent(Graphics g)
```

```
// use g.drawImage() to draw the thumbnail, with this object as
```

```
// the ImageObserver
```

```
// Outlines the image with a border to indicate that it is selected.
```

```
void selectImage();
```

```
// Removes the border from an image when it is deselected.
```

```
void deselectImage();
```

---

Field Summary	
String	directory Current directory for browsing images.
Thumbnail	currentThumbnail Currently selected thumbnail.
Constructor Summary	
<b>GUI()</b> Instantiates, adds, and sets listeners for all interactive components. Instantiates StegoSecurity object and JProgressBar dialog.	

Method Summary	
Component	<p>getComponent() Returns the Component for this image bar, so that it can be added to the main pane.</p>
void	<p>changeDirectory(String pathName) Changes current directory to pathName and repaints the imagebar.</p> <pre>// clear the image bar // open pathName directory // for each file in the directory: //   if file is GIF or JPEG //     call addThumbnail(file)</pre>
void	<p>clearImageBar() Removes all thumbnails from the image bar.</p>
void	<p>addThumbnail(String fileName) Adds a thumbnail-sized image to the ImageBar.</p> <pre>// if fileName is a JPEG // create a FileInputStream object for fileName // create a JPEGImageDecoder object from the input stream // create an Image object from the image decoder // scale the image to fit the image bar // create a Thumbnail for the image // register a MouseListener with the Thumbnail // add the thumbnail to the JScrollPane imageBar // adjust the preferredSize of the JScrollPane to // accommodate the new thumbnail // else if fileName is a GIF // similar to JPEG case above</pre>
	<p>void setSelectedThumbnail(Thumbnail t) Changes the currently selected thumbnail.</p>
	<p>Thumbnail getSelectedThumbnail() Returns the currently selected thumbnail.</p>

# Class StegoSecurity

[java.lang.Object](#)

└ StegoSecurity

## All Implemented Interfaces:

none

## Direct Known Subclasses:

none

---

```
public class StegoSecurity
  extends
  implements
```

This class works with the following three modules:

Huffman class

cipher class

Md5 class (standard java class)

---

## Field Summary

Cipher	cipherFunction A class with encryption and decryption functions.
Random	randomLocations Generate random location of pixels/blocks.
Random	randomBits Generate random bits.
byte[]	cipherText Storing the cipherText in here, plainText isn't stored.
hashtable	oldRandomLocations Storing the random numbers that have been generated to make sure there's no repeat.
byte[]	encryptionKey Generated when setKey is invoked.
bool	encryptionStatus; Encryption on/off.
long	randomNumber Generate a long pseudo-random number for any use.

## Constructor Summary

Security (progressBar, canceled);

## Method Summary

bool	isEncryptionOn(); Get encryption status.
void	turnOnEncryption(); Turn Encryption On.
void	turnOffEncryption(); Turn Encryption Off.
void	setKey(BufferedImage key); Set key.
void	setKey(String key); Set key.
byte[]	getKeyHash() Return the hashed value of the key, could be used to check if key is set returns NULL if key is not set.
void	setPlainText(String text) Feed the plain text for decryption, key must be set first.
String	getPlainText(); Throw 1 exception that could mean any of the following data is corrupted, wrong key, image has no data hidden.
void	setMaxCapacity(long max); Set the maximum capacity (max # of bits that can be hidden).
long	getNextLocation() Return the next location: where the next bit should be hidden given a number in this range, the coordination of the pixel or the block should be easily obtained. There will be no repeated output.
void	locationReset(); Reset to start from the beginning of the random sequence.
byte	getRandomBit() Return a 0 or a 1 pseudo-randomly for JPEG algorithm.
void	randomBitReset() Reset the random bit stream to the beginning.

long	getCipherLength() Return the entire length in BITS of the cipherText, including the first 24 bits that are used to store this length will throw exception if one of the first 24 bit is undefined.
void	setBit(byte bit,long location); Set the bit of the ciphertext, use in decoding location is the bit position in the whole ciphertext.
byte	getBit(long location); Get the bit of the ciphertext, use in encoding location is the bit position in the whole ciphertext.

## Class Huffman

[java.lang.Object](#)

└ Huffman

### All Implemented Interfaces:

none

### Direct Known Subclasses:

none

```
public class Huffman
  extends
  implements
```

A static class that provides compression features.

The Huffman algorithm builds the tree  $T$  corresponding to the optimal code in a bottom-up manner.  $C$  is a set of  $n$  characters and each character  $c$  belongs to  $C$  is an object with a defined frequency  $f[c]$ .  $Q$  is a priority queue, keyed on  $f$ , is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

Two smaller classes are used:

```
// an entry is an object that contains a character, its frequency,
// and the form it will be represented in 0's and 1's. (called codeword)
class Entry {
char symbol;    //example: character "t"
double freq;    //frequency is: 0.235
String codeword; //codeword is: 0101
```

```
// store data used as input and output for the Huffman class
class HuffmanData {
    HuffmanData(long bitlength, byte[] data);
public:
long bitlength; //length of the data in bit
byte[] data; //the actual data (could be compressed or uncompressed)
}
```

---

### Field Summary

### Constructor Summary

### Method Summary

HuffmanData	zip(String plaintext); return compressed text.
String	unzip(HuffmanData); return normal text.

## Class Cipher

[java.lang.Object](#)

└ **Cipher**

#### All Implemented Interfaces:

none

#### Direct Known Subclasses:

none

---

```
public class Cipher
    extends
    implements
```

A static class that provides two functions for encryption and decryption using AES algorithm; at least 128-bit strong.

---

## Field Summary

--	--

## Constructor Summary

--	--

## Method Summary

byte[]	encrypt(byte[] key, String plainText); Return cipherText encrypted using key.
String	decrypt(byte[] key, byte[] cipherText); Return normal text decrypted using key.

## interface GenericStegoImage

[java.lang.Object](#)

└ GenericStegoImage

### All Implemented Interfaces:

none

### Direct Known Subclasses:

**GifStegoImage, JpegStegoImage**

---

```
public interface GenericStegoImage
    extends
    implements
```

### Interface GenericStegoImage

This object is created and persists throughout the entire lifetime of the *'Paranoia'* application. The parameters are passed to the specific method calls except a handle to detect the `ProcessCancelledException`.

## Exceptions .

Errors may occur under certain circumstances. In order to handle them as gracefully as possible, exceptions will be used. Often the exception will be thrown by an image or security class. Such exceptions will be caught by the GUI class. The GUI will then present the user with a window explaining what has happened.

- TooMuchInformationException (not within the capacity)
  - NumberOfColorsException (too few, too many, other)
  - ImageSizeException (image too small, too large, other)
  - Invalid Key (unexpected format)
  - ProcessCancelledException (user clicking on the cancel button)
  - InvalidFormat (non-complying image format)
  - Other Security reasons
- 

Field Summary		
long	location;	// hot spot in raster image
byte	bitValue	// hot item
Image	Target	// target image or encrypted image
Image	keyImage;	// image to be hashed for key
String	keyText;	// text key
long	capacity;	// max bits that can be hidden in the image
Constructor Summary		
<b>void GenericStegoImage(void);</b>		

## Method Summary

fileHandle

encode(Image target, stegoSecurity S);

Encode method is invoked with 4 parameters to initialize the security module and eventually to write out the embedded image. *Target* refers to the image used to hide textual information. *Key* can be either an image or string, and StegoSecurity module handles them accordingly. *Text* refers to the plain text. *Security* refers to the StegoSecurity class instantiated in GUI. This method writes encrypted, compressed and bit-placed information to the target image and returns the file handle of the produced image.

1. Calculate the *Capacity* with the dimension of the target image given – **Format Specific**
2. Initialize the security module - Pass off (*Text, Key, Capacity*) to security module
3. Get the final *cipherlength* in bytes from security module (this number is used to calculate the number of iteration) – getCipherLength()  
  
\* important to notice that cipherlength actually refers to the size of the ciphertext + 3 bytes (header portion representing the size of the ciphertext)
4. Iteration for ( $index = 0; I < (cipherlength * 8); index++$ )
  - i.  $bitValue = getBit(index)$  – get a bit from the security
  - ii.  $location = getNextLocation()$  – location to modify the bit
  - iii.  $writeBit(bitValue, location)$  – **Format Specific**

Save the file and return the file handle back to the GUI

fileHandle

decode(Image source, stegoSecurity S);

Decode method is invoked with 3 parameters to again initialize the security module and eventually to extract the plain text. *Target* refers to the image that contains embedded text. *Key* can be either an image or string. *Security* refers to the StegoSecurity class instantiated in GUI. When job is completed, method returns decrypted, uncompressed, and assembled plain text.

1. Calculate the *Capacity* with the dimension of the encrypted image – **Format Specific**
2. Initialize the security module - Pass off (*Key, Capacity*) to security module

3. First 3 bytes contain the size of the ciphertext. (Purpose of the first iteration is to obtain the size of the hidden text. 3 bytes = 24 bits)

Iteration for ( $index = 0; index < 24; index++$ )

- i.  $location = getNextLocation()$  – location to fetch the bit
- ii.  $bitvalue = extractBit(location)$  – **Format Specific**
- iii.  $setBit(bitvalue, index)$  – passes fetch bit to security module

4.  $cipherlength = getCipherLength()$  – from the 3 bytes passed, size can be decrypted from the security module and be returned to Codec module

5. From the  $cipherlength$  subtract 3 which was used as the header to contain the size of the length of text only.

Iteration for ( $index = 0; index < ((cipherlength - 3) * 8); index++$ )

- i.  $location = getNextLocation()$  – location to fetch the bit
- ii.  $bitvalue = extractBit(location)$  – **Format Specific**
- iii.  $setBit(bitvalue, index)$

6.  $text = getPlainText()$

7. Pass  $text$  to GUI to be displayed.

# class GifStego

[java.lang.Object](#)

└ GenericStegoImage

└ GifStego

## All Implemented Interfaces:

none

## Direct Known Subclasses:

none

```
public class GifStego
    extends
    implements GenericStegoImage
```

A class that implements steganography for GIF files.

## Field Summary

int	colorMap[]	Colors in a given image in java's TYPE_INT_ARGB format.
int	neighborMap[]	Closest matching color's index that has different least significant bit from referencing index.
byte	redValue[][]	Red component of the color.
byte	greenValue[][]	Green component of the color.
byte	blueValue[][]	Blue component of the color.
byte	alphaValue[][]	Alpha component of the color.
int	pixelMap[][]	Two-dimensional array, dimension of the image.
BufferedImage	b	Interface to the GIF reading and writing libraries.

## Constructor Summary

```
void GifStego(void);
```

## Method Summary

see GenericStegoImage

# class JpegStego

[java.lang.Object](#)

└ GenericStegoImage

└ JpegStego

## All Implemented Interfaces:

none

## Direct Known Subclasses:

none

---

```
public class JpegStego
  extends
  implements GenericStegoImage
```

A class that implements steganography for JPEG files.

## Field Summary

BufferedImage	b	Interface to the JPEG reading and writing libraries.
float	bf[][]	For the internal high-precision representation.
StegoTable	st	To help represent the stego-tables.

## Constructor Summary

```
void JpegStego(void);
```

## Method Summary

	see GenericStegoImage
--	-----------------------

## 2.4 JPEG Steganography and Algorithms

JPEG/JFIF Format:

Strictly speaking, JPEG refers only to a family of compression algorithms; it does not refer to a specific image file format. The JPEG committee was prevented from defining a file format by turf wars within the international standards organizations. JFIF has emerged as the de-facto standard on Internet, and is what is most commonly meant by "a JPEG file". Most JFIF readers are also capable of handling some not-quite-JFIF-legal variant formats.<sup>1</sup>

JPEG/JFIF works by extracting coefficients describing 8x8 pixel blocks and then compressing these coefficients. The blocks can be revealed by saving a JPEG of a gradient pattern with the lowest possible compression quality. The blocks, which start in the upper left-hand corner, are shown in Fig. J.1. Partial blocks will be included on the right and lower edges if the dimensions are not divisible by 8 pixels.

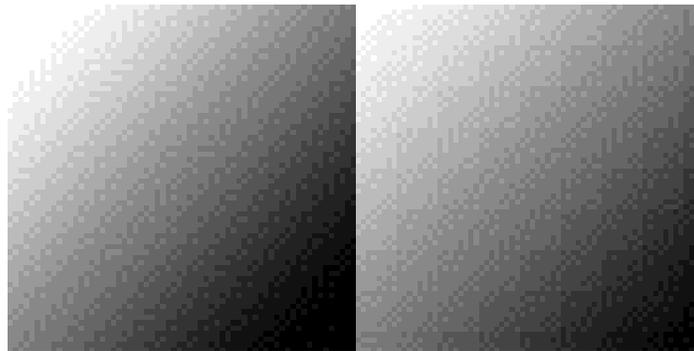


Fig. J.1. Left: gradient with high quality compression  
Right: gradient with low quality compression

Encoding:

A flowchart describing typical JPEG encoding is shown below in Fig. J.2.

- 1) The input is a BufferedImage object, which contains a ColorModel and a matrix representing the image with pointers aimed at indices of the ColorModel. The RGB values of the uncompressed input image are converted into three components: one luminance component and two chrominance components (YUV). The luminance component is considered more important.
- 2) The image is separated into 8x8 pixel blocks starting from the upper left-hand corner (see Fig. J.1).
- 3) The component signals for each 8x8 block are transformed into the frequency domain by using the two-dimensional discrete cosine transform (DCT). This transformation is similar to the two-dimensional fast Fourier transformation, except that the coefficients are real instead of complex. The perceptually most important information is contained in the first coefficients, and the later coefficients are generally small (redundance reduction).
- 4) While the coefficients closest to 0 are eliminated, the remaining coefficients are quantized

---

- <sup>1</sup> JPEG/JFIF FAQ: <http://www.faqs.org/faqs/jpeg-faq/>

using various degrees of accuracy (irrelevance reduction). This can be modified by changing the quantization tables. The DC luminance coefficients are the most important and are quantized with the most accuracy.

5) Finally, the quantized coefficients are compressed using a Huffman encoder.

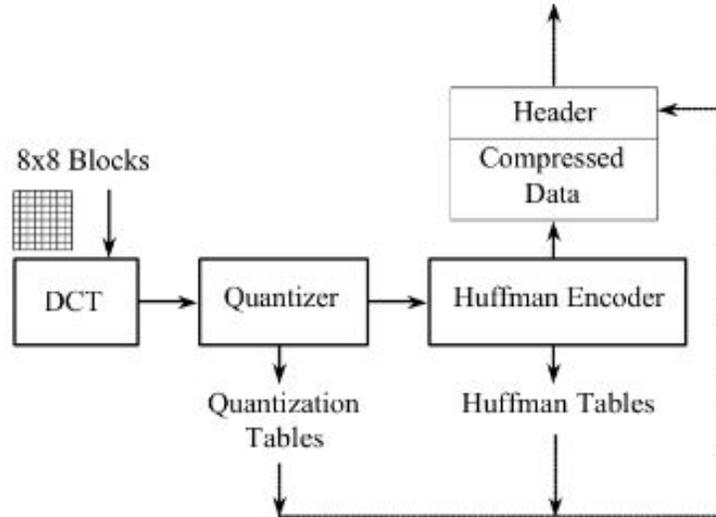


Fig. J.2. JPEG/JFIF encoding scheme

Since the DC luminance coefficients are quantized with the most accuracy, we will hide the information inside them. We will simply shift the values up or down by certain amounts into areas that are defined as being 0's and 1's. While this could be implemented by modifying the code of a JPEG encoder manually, we will simply modify the image beforehand so that the DC luminance coefficients will correspond to those that we seek. This method is possible and not especially processor intensive because the DC luminance coefficient corresponds to the average luminance values for each 8x8 block. It works as follows:

- 1) Convert the RGB signals for the image into YUV using the following formula. Store the result in a temporary float[][] to reduce quantization effects that occur before the compression itself.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.159 & -0.332 & 0.050 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- 2) Add a small amount of white zero-mean Gaussian noise with standard deviation  $\sigma$  to reduce blocking effects. An appropriate value for  $\sigma$  will be determined empirically during the implementation stage.

---

<sup>2</sup> Taken from „ **Operation of the JPEG Algorithm**“  
<http://www.ctie.monash.edu.au/emerge/multimedia/jpeg/impl03.htm>

- 3) Carry out a small amount of preprocessing to reduce the likelihood of the rare cases described in step 6. Shift the values of the input  $f(x,y)$  to a compacted version  $f'(x,y)$ .  

$$f'(x,y) = p + f(x,y) (128 - p) / 128$$
- 4) Determine which blocks should be modified according to the bit-placement algorithm.
- 5) Cycle through all of those blocks. Call the current one block B, and call b the bit that is supposed to be transmitted with block B.
- 6) Calculate the average value  $y_a$  of the luminance signal Y for block B.
- 7) Find the target average value  $y_t$  for the luminance signal for block B using the stego-table shown in Fig. J.3. The target value is the value in the middle of the region closest to  $y_a$  that is labeled with the bit b. The target value will get changed slightly during compression because JPEG/JFIF is a lossy format.
- 8) Shift the luminance values for block B by adding  $y_t - y_a$  to them. (As long as  $p < k$ , there are rare cases in which this will not work due to the fact that certain pixels have luminance values that are bunched up near the edges of the luminance spectrum. That is, they would get shifted out of the valid range [0 255]. In those cases, a new target value  $y_t$  will be chosen, which is further from the boundaries. Then step 8 will be repeated.)
- 9) Convert the YUV signals back into RGB signals using the inverse of the transformation above in step 3. (The inverse transformation exists because the matrix is invertible.)

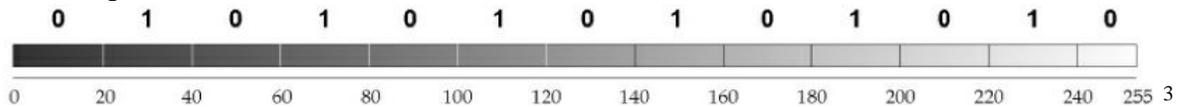


Fig. J.3. Simple stego-table with  $k = 20$ , where  $k$  is the width of each region.

Finally, we can simply save the image using the default JPEG encoder for Java called `javax.imageio.plugins.jpeg`. A block diagram for the combined processes is shown in Fig. J.4.

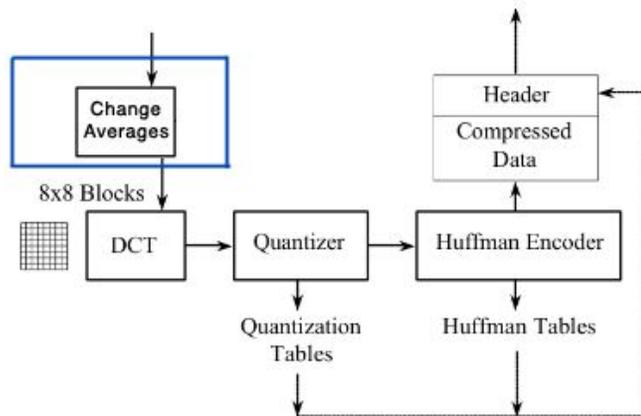


Fig. J.4. JPEG/JFIF encoding scheme for steganography

<sup>3</sup> p. 7 of Lee and Chen.

To make it more difficult to detect the presence of embedded data, the bit-values along the top of the stego-table will actually be randomized as shown in Fig. J.5. The randomization of the location of the 1's and 0's will simply cause a histogram of the resulting DC coefficients to look more random. If the values were simply pseudorandom, then it would be possible to get multiple consecutive 0's or 1's, so 0's will simply be expanded into 01's and 1's expanded into 10's. So that the same bits will be used for decoding, these  $k$  random bits that are dependent on the key will be received from the StegoSecurity object, where  $k$  will be empirically determined such to criteria that will be mentioned later. Therefore, no DC coefficients will be changed by more than  $3k/2$  before compression.

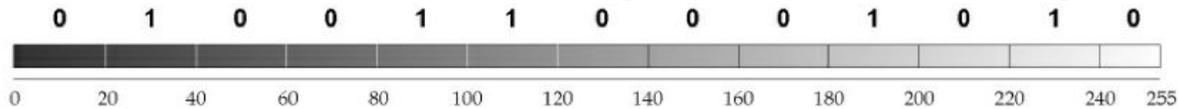


Fig. J.5. Randomized stego-table.

#### Possible Extensions:

- Multiple stego-tables could be used to make it more difficult to detect steganography on the output file using statistical analysis. Table  $i = 0$  would be as shown above, and tables  $i = 1$  to  $p - 1$  would be the same except shifted to the right by  $i$  indices. As each bit is extracted or embedded, a table to use would be chosen via pseudorandom numbers returned by the StegoSecurity object. This method would ensure that a histogram of the DC coefficients would look be approximately evenly distributed.
- Information could be stored in the chrominance as well as the luminance coefficients. The compression quality might have to be quite high for this to work properly.
- Changing the averages of the 8x8 blocks could be carried out not only using additive techniques but also multiplicative techniques depending on the variance of the block. Additive (uniform) techniques are better for larger variances, while multiplicative (proportional) techniques are better for smaller variances.<sup>4</sup>
- By traversing through the bits to change sequentially in terms of the image dimensions  $x$  and  $y$  instead of in terms of the encoding order, it would be possible in some circumstances to choose slightly better entries in the stego-table minimizing the total change of the image.<sup>5</sup>

#### Decoding:

The decoding scheme is much simpler than encoding. The averages of the luminance of the 8x8 blocks just need to be calculated and converted back to bits while paying attention to the current stego-table and encoding order.

<sup>4</sup> p. 10 of Lee and Chen.

<sup>5</sup> p. 9 of Lee and Chen.

Useful References:

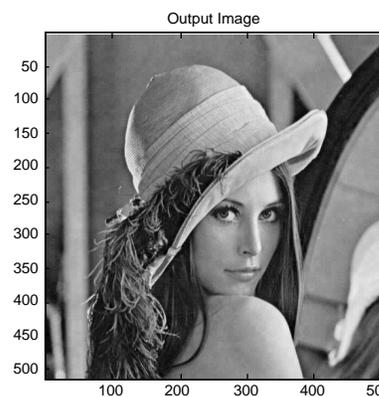
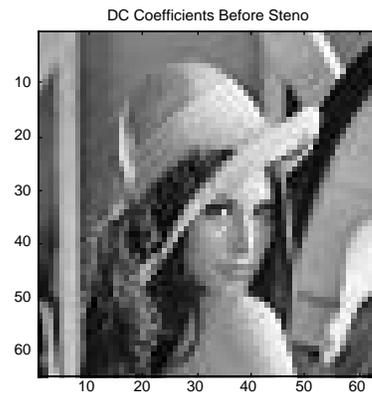
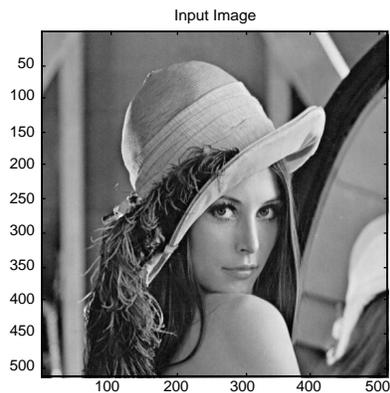
- JPEG/JFIF FAQ: <http://www.faqs.org/faqs/jpeg-faq/>
- Yeuan-Kuen Lee and Ling-Hwei Chen. „Secure Error-Free Steganography for JPEG Images.“ Dept. Of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Rd., Hsinchu 30050, Taiwan, R.O.C.

Proof of JPEG Concept:

MATLAB was used to implement a simplified version of the algorithm explained above. 4096 random bits were hidden in „lena.jpg“ using the following settings. The compression quality is maximum, and a relatively large number of regions in the stego-table  $z \approx \text{floor}(255/k)$  are used.

z: # of entries in the stego-table  
81

Compression Quality  
100



Numeric results:

BER	Size of resulting JPEG file	SNR in dB
0	162 kB	37

The bit error rate (BER) is good—that is, all of the bits were extracted properly. The signal-to-noise ratio (SNR) seems fairly good, even though it is also limited due to JPEG compression, while the size of the resulting JPEG file is rather large. Most importantly, the output image looks just like the input image when compared by the human eye. For more examples, see Appendix A: JPEG Examples and Parameters.

NOTE: Although we calculated SNR measurements with MATLAB, our final program will probably not carry out such an analysis, since its performance is limited in comparison with subjective tests.

## 2.5 GIF STEGANOGRAPHY AND ALGORITHMS

### GIF Format:

GIF (graphics interchange format) files have become very widely used on the Internet. GIF became popular because it used Lempel-Ziv-Welch (LZW)<sup>6</sup> data compression, thus, fairly large images could therefore be downloaded in a reasonable amount of time, even with very slow modems.

An image is structured as a grid of cells, called pixels (picture elements). Each pixel has a color, which is represented in GIF as RGB-triple. Each color component is 8-bit long, so there are  $2^{(3*8)} = 2^{24}$ , or about 16.7 million possible colors. This would quickly increase file size, so GIF uses compression.

The first kind of compression that GIF uses is called a `colormap`. Instead of allowing the image to contain all 16 million colors, GIF restricts the image to a maximum of, say, any 256 out of the 16 million (the number of colors in the `colormap` can be varied). Thus, while there is no loss of richness of possible colors, no more than 256 distinct colors can be used simultaneously. The second kind of compression that GIF includes is a sophisticated variation of run-length coding of mentioned LZW.

### Simple Encoding and Decoding Algorithm:<sup>7</sup>

Based on these alternatives, we chose to embed the data into the pixels. Here we present an algorithm used by EZ Stego. The `colormap` is first sorted by luminance, so that neighboring palette entries are near to each other in the color space. We run the following algorithm for each pixel that is selected for embedding by the bit placement module.

#### Encoding

- 1) Find the index of the pixel's RGB color in the sorted palette.
- 2) Get one bit from the input file. Replace the least significant bit of the index.
- 3) Find the new RGB color that the index now points to in the sorted palette.
- 4) Find the index of the new RGB color in the original palette.
- 5) Change the pixel to the index of the new RGB color.

#### Decoding

The least significant bit of the index came from the input file. Write it to the output file.  
Optional interlaced

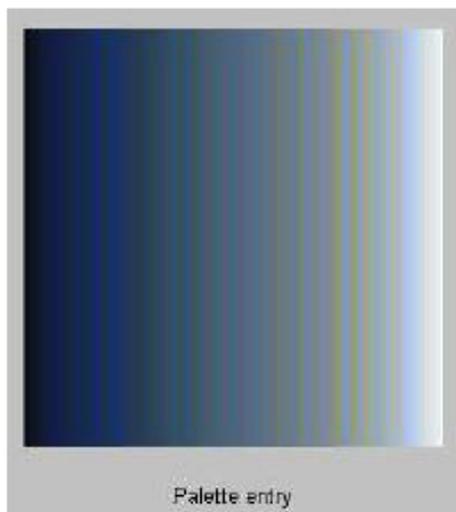
### Sample Images

---

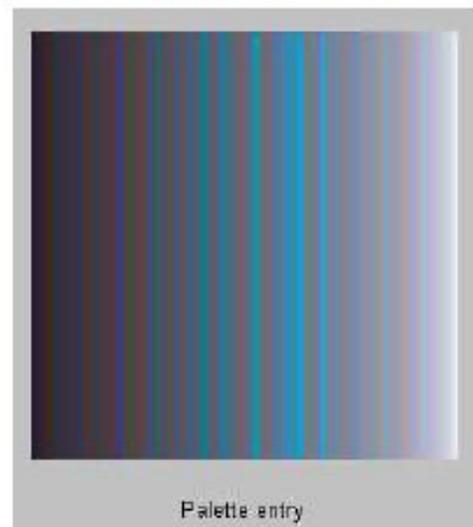
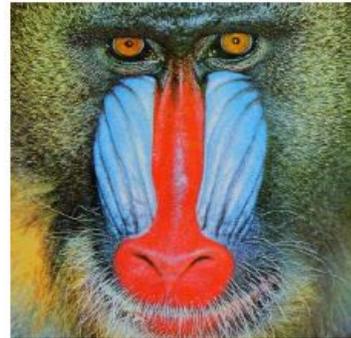
<sup>6</sup> On June 20, 2003, the United States patent on the LZW algorithm expired.

<sup>7</sup> Implementation of EzStego: <http://www.stego.com/>

The EZ Stego solution works with varying degree of success, depending on the input image. To illustrate how one image can be a better input than another, here are shown two images with their respective luminance-ordered palettes. Figures 1 and 2 show the image “Fox” and its palette; Figures 3 and 4 show image “Madrill” and its palette. Both “Fox”, with 240x320 pixels, and “Madrill”, with 512x512 pixels were truncated to 256 colors. We can clearly see that the luminance-ordered palette of “Fox” contains visibly fewer and less severe discontinuities in color than that of “Madrill”. Therefore, if EZ Stego would be used in this case, there would be a much greater chance of failure with “Madrill” than with “Fox”.



*Figure 2 Luminance-ordered palette.*



*Figure 4 Luminance-ordered palette.*

## Analysis:

During the process of mapping 3-dimensional RGB colors to one-dimensional luminance rendered colormap, this may lead to visible color discontinuity among the colors that have same luminance yet so far away in the color spectrum. Since luminance is mere sum of three RGB values, luminance ordering does not guarantee the visibly similarity.

## Improvement over Simple Algorithm:<sup>8</sup>

For each pixel to contain the data, the set of the closest colors is calculated. This is done by calculating the Euclidian distance between the color of the pixel from each `colormap` entry and sorting the result. The distance between colors  $(R1, G1, B1)$  and  $(R2, G2, B2)$  is

$$\sqrt{(R1 - R2)^2 + (G1 - G2)^2 + (B1 - B2)^2}$$

For each data bit to be embedded, we know the pixel where it will be embedded. We first have to check whether embedding the data will actually change the LSB. In the best case, it will not be changed, so we will not have to do anything. Otherwise, we start to search for the closest color till we find a match between the bit to be encoded and the parity of the color. The parity of the color is defined as  $R+G+B \bmod 2$ . Once the color is found, the index for the pixel is changed to point to the new color.

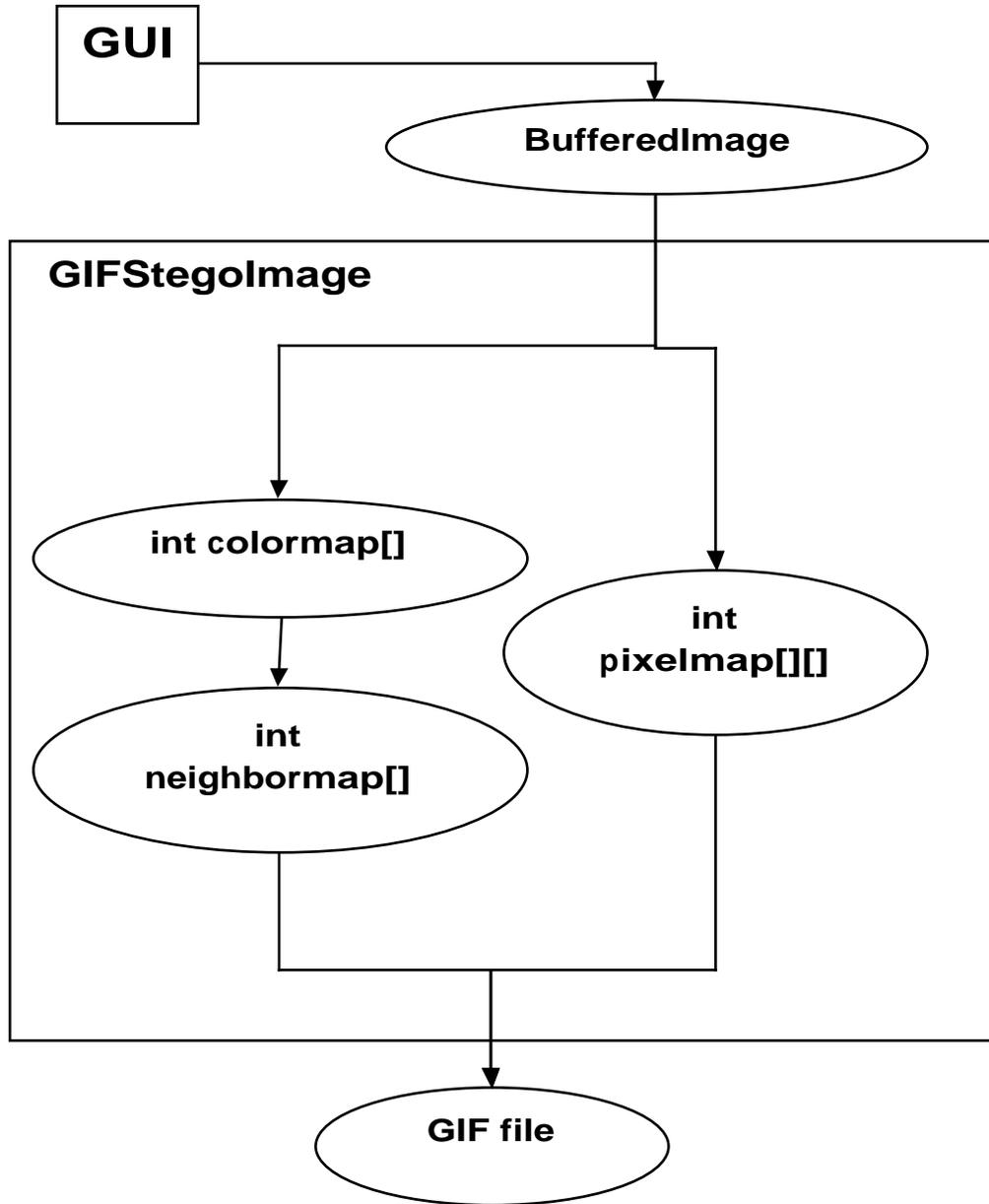
Since this match making step has to be repeated for each bit modification. `GIFStegoImage` will perform a batch job of creating `NeighborMap[]`. Based on the LSB of the `colormap` index, `NeighborMap[]` will contain closest matching color that has different LSB. Thus `GIFStegoImage` only need to reference `NeighborMap[]` for each replacement.

## Description of the following flowchart:

1. GUI passes `BufferedImage` to `GIFStegoImage`.
2. In `GIFStegoImage`, `pixelmap` is formed by extracting each `int rgb` value in corresponding index in the 2 dimensional array called `pixelmap[][]`.
3. From the `pixelmap[][]`, `colormap` is generated by passing through and adding countered new colors to `colormap[]`.
4. Extracting and applying each red, green and blue color to Euclidean distance formula to find matching color that has different least significant bit of `colormap` indexes and store them to `neighormap[]`.
5. Using `colormap[]`, `neighormap[]`, and `pixelmap[][]` to write out a file in GIF 89a standard format.

---

<sup>8</sup> “A new Steganographic Method for Palette Based Images”: <http://www.ws.binghamton.edu/fridrich/Research/pics99.ps>



**Flowchart of image processing by GIFEncoder**

### Java's representation of RGB colors

- TYPE\_INT\_RGB : 8 bit RGB color components packed to an integer pixel without alpha
- TYPE\_INT\_ARGB : 8 bit RGB color components packed to an integer pixel with alpha
- TYPE\_INT\_ARGB\_PRE : 8 bit RGB color components packed to an integer pixel with alpha. The color in the image is pre-multiplied with alpha

## Useful methods:

### Class BufferedImage

- setRGB(int x, int y, int rgb) : sets a pixel in specified RGB value
- getRGB(int x, int y) : returns a pixel of RGB integer

## Structure of GIF Format:

- Header – 6 bytes
- Logical Screen Descriptor (dimension, background) – 7 bytes
- Global Color table – 768 bytes
- Image Descriptor – 10 bytes
- Optional Local Color Table – 768 bytes
- Graphic Control Extension – 10 bytes
- Data Block – dependent on the dimension and compression

## Possible Extensions:

If the image has significantly fewer colors than 256, the change of the pixel's colors may be visibly detectable. There are a few things that can be done regarding this issue.

- Adding more colors to the colormap. Based on the colors already in the colormap, plus on the pixels chosen to contain data, we can decide what colors are needed more than others, and add those colors to the colormap. This complicates things because we have to create a histogram showing which colors are needed more than others. This also requires manipulating the colormap, in order to insert the colors in the correct places.
- Substituting colors from the colormap as needed. Based on the pixels chosen to contain data, we can determine that some colors are under-represented in the colormap, and that may impair the visual similarity of the resulting image. If some other group of similar colors is well-represented, and visual similarity will not be impaired by decreasing the size of that group, we may choose to do so. Then, we will add the more needed colors. This option also complicates our task, since we will have to calculate the statistics of how much one color is needed compared to another.
- Analyzing the neighboring pixels. We may want to look at the colors of the neighboring pixels, before changing the value of a given pixel. This may be useful in the areas of uniform color.
- Possible addition of noise or randomness in adding colors to colormap
- Adding a color assurance by checking the mean distance. If the distance exceeds the predetermined limit, different remedying method can be applied.

We have decided to first implement the simple scheme, without adding or deleting colors from the colormap. The extensions given above will be optional, and they will be done only after the basic scheme is implemented.

## Useful References:

GIF image format: <http://ptolemy.eecs.berkeley.edu/eecs20/sidebars/images/gif.html>

EZ Stego: <http://www.stego.com/howto.html>

Steganography using color map shuffle:

<http://www.darkside.com.au/gifshuffle/description.html>

A new Steganographic Method for Palette Based Images:

<http://www.ws.binghamton.edu/fridrich/Research/pics99.ps>

## 2.6 THE USER WALKTHROUGH

A portion of the main GUI is devoted to the user walkthrough. This walkthrough will lead first time users through the process of Encoding and Decoding images and in its final step initiate the Encoding/Decoding of the users text. The user may prepare for and perform encoding and decoding at any time without the use of the walkthrough.

Consult the following flow chart and corresponding description of states to understand the Walkthrough process.

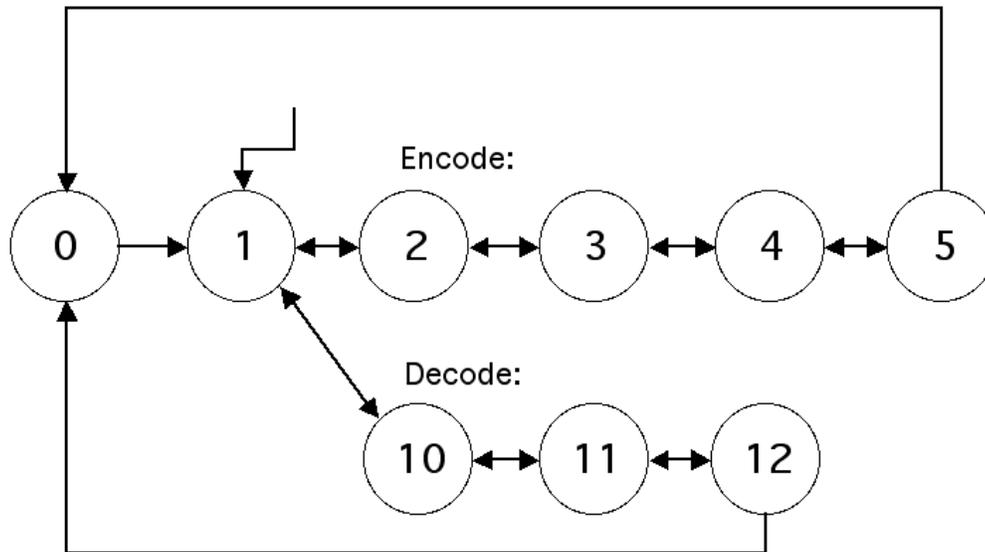


Figure 1. Flowchart for the walkthrough

The box below explains how each state will be described. Each has a unique state number and three buttons, some of which may be grayed out from time to time.

State n:	Button <sub>1</sub> (f <sub>1</sub> )	Button <sub>2</sub> (f <sub>2</sub> )	Button <sub>3</sub> (f <sub>3</sub> )
<p>n is the index of the state being described. Button<sub>1</sub>, Button<sub>2</sub>, and Button<sub>3</sub> are the names of the top, middle, and bottom buttons, respectively. f<sub>1</sub>, f<sub>2</sub>, and f<sub>3</sub> describe the following states if the button is clicked, respectively. If the value is gray, then the button should be grayed out and thus unclickable. Text similar to the text in this area should show up on the right side of the walkthrough in a text box.</p> <p>Text in the Courier font consists of general notes about the state. Users should not be able to see this text in the real walkthrough slides.</p>			

The following boxes detail each state.

State 1:	Embed (2)	Extract (10)	Cancel (gray)
----------	-----------	--------------	---------------

This Walk-through can help you learn how to use Paranoia. Please select a process: Embed means hide text within a picture, and Extract means extract hidden text from a picture. You can return to this slide at any time by clicking on Cancel, and you can reset everything by selecting Start Over from the File menu.

This is the initial state of the walkthrough when the program is originally loaded and after “File|Start Over” has been chosen. Once the walkthrough knows which process is to be carried out, it can offer hints.

State 2:	Next (3)	Back (1)	Cancel (1)
----------	----------	----------	------------

Choose an image to use as a key by selecting it on the image bar below, and clicking the load button on the Key Image pane. Other people will be able to extract the text from the picture if they use the same Key Image. If you want to use an image from another directory, you can use Set Image Directory from the Tools menu.

The simplest technique for loading a key is described.

State 3:	Next (4)	Back (2)	Cancel (1)
----------	----------	----------	------------

Please pick out the Target Image that you would like to hide the text in selecting it on the image bar and clicking the load button on the Target/Source Image pain. Again, for images from another directory, select Set Image Directory from the Tools menu.

The simplest technique for loading a target image is described.

State 4:	Embed (5)	Back (3)	Cancel (1)
----------	-----------	----------	------------

Enter text into the text box on the Text tab of the main panel. You can also paste text from the clipboard with the Paste command in the Edit menu, or you can choose Open Text from the File menu. When you are done, click the Next button to hide the text in the source image.

The user is instructed to input text, using combinations of three different methods: typing, pasting, and importing.

If the text cannot be embedded for some reason, an appropriate dialog will appear informing the user what he or she can do to alleviate the problem.

State 5:                      Next (0)                      Back (4)                      Cancel (1)

Click on the Image tab on the main panel to see the resulting image. If you are satisfied with its appearance, choose Save Image or Save Image As... from the File menu.

After viewing the image, the user saves it if satisfied. Otherwise, he or she can click cancel. After Save or Save As... is selected, the Back and Cancel buttons should preferably be grayed out.

State 10:                      Next (11)                      Back (1)                      Cancel (1)

Choose an image to use as a key by selecting it on the image bar below, and clicking the load button on the Key Image pane. If you want to use an image from another directory, you can use Set Image Directory from the Tools menu.

The simplest technique for loading a key is described.

State 11:                      Extract Text (12)                      Back (11)                      Cancel (1)

Please pick out the image from which the hidden text should be extracted. Selecting it on the image bar and click the load button on the Target/Source Image pane. Again, for images from another directory, select Set Image Directory from the Tools menu.

The simplest technique for loading a source image is described.

State 12:                      Next (0)                      Back (12)                      Cancel (1)

The text can be exported to another application via the clipboard by selecting Copy from the Edit menu. Alternatively, the text can also be exported to a text file by selecting Save Text from the File menu. Otherwise the text can be read as it stands by the eyes it was intended for!

The user is instructed how the extracted text can be exported.

State 0:                      Next (gray)                      Back (gray)                      Cancel (gray)

To Embed or Extract again, select Start Over from the File menu.

After a process has been completed, the walkthrough is at an end. All of the walkthrough buttons are grayed out, until it is restarted by choosing Start Over.

## 2.7 SECURITY FEATURES

Paranoia will employ several security mechanisms that ensure the confidentiality and integrity of messages being hidden into photos:

- Because each bit of the message to be hidden is stored in one pixel (or one 8x8 block in case of JPEG) of the photo, this application generates a pseudo-random selection of which pixels are used to store these bits. This selection is based on a key extracted from a key image chosen by the user. Since hiding messages into photos are very likely to produce minor changes, this random selection of pixels helps make the destination image even more indistinguishable from the original image.
- Because scrambling pixel selection alone is not proven to have any strong encrypting effect, messages are compressed and then encrypted using Advance Encryption Standard, with 128-bit keys generated from the key image. The compression helps hide more text in to an image and enhance the encryption by reducing the amount of data to be encrypted and the repetition of bit patterns. As of now, there are no known methods of breaking this encryption standard, except for exhaustively searching through  $2^{127}$  possible key values. The efforts spent to break this encryption far outweigh the importance of these hidden messages. Thus, we can consider this application secured.
- Paranoia also guarantees the integrity of the messages by performing Cyclic Redundancy Check on messages after they are decrypted and decompressed.

# 3 PROJECT PROCESSES

## 3.1 CODE REVIEW:

We will have meetings where we'll do the following checking:

- The review will be an oral presentation of a portion of each subgroup's code.
- Each subgroup chooses the portions to be reviewed based on which portions they want to get the most input from people in the group.
- The code to be reviewed should have been compiled to remove obvious bugs in the code. Some simple testing may be appropriate before reviewing.
- Each subgroup is responsible to distribute a copy of the code to be reviewed some time in advance of the review.
- All subgroups will talk to each other checking if they all follow agreements on interfaces, which were agreed in the early time of design phase.

### Coding Standard:

#### **Comments:**

- We expect there are reasonable amount of comments in the codes. Each file and each function should have a comment at the start, explaining what the code does. Each document's comment is set inside the comment delimiters `/**...*/`. Code should be as self-documenting as much as possible.
- Variable names are self-explained, otherwise comment on it as needed. Comments are necessary to explain complex code/algorithm.

#### **Naming Conventions:**

- Class names should be nouns, in mixed case with the first letter of each internal word capitalized.
- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
- The names of variables declared class constants should be all uppercase with words separated by underscores.
- Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters.

In short, we will apply Java coding standard to our codes. The reference can be found here: <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

### Project Management:

We also use CVS to manage our codes because it allows multiple people to edit files at the same time. This feature helps us when we want to code on different machines without worrying about transferring the source code between machines manually. CVS also guarantees that we won't be using old versions of files.

## 3.2 MODULE TESTING

### Overall testing strategy

Once working on our main modules, we will unit test each module. Once each module had been thoroughly and successfully tested, we will start integrating the modules one by one and test them working together (integration tests). Once interoperability of the modules has been tested, we will do system testing on the entire program. If a need of incorporation of a new feature or module should arise, then we will repeat the entire testing process.

### Unit Testing

For all the main units of the application (GIF steganography, JPEG steganography, image to key conversion, compression, and user interfaces) unit testing is conducted. Unit testing involves white box testing – testing with access to the internals of the software – generally with full access to the unit code. In most cases it involves an input that will be provided for each component tested and an expected output that is calculated beforehand to check if the module is functioning correctly. If the given input to a module results in an output that does not match the expected output, the module will be checked for possible coding errors. Instead of development of test harnesses involving design of drivers (input stimulus generators) and monitors (output behavior verifiers), the JUnit testing framework will be utilized to provide drivers and monitors. However, given the specificity of the application, some function output will have to be checked by a human being. (For instance, to see whether the original image and the container image with hidden text appear similar to a human eye).

## 3.2.1 GUI TESTING

<p>Testing User Interface</p> <p>Test1: Opening / saving file</p> <p>Test Approach Through File -&gt; Open Text (Image), File-&gt;Save Text (Image), etc. try to open file, save file, open non-existing file.</p> <p>Goal Check if we can open the files or allowed format only (pure text, GIF, and JPEG). Check that the application consistently handles numerous exceptions such as absence of a file, saving file in a folder where a file with the same file name already exists, trying to open a file with valid file extension but of invalid format (i.e. rename a PDF file extension to TXT and try to open the file), try to open a file of a size exceeding the application's size limitations, etc.</p> <p>Scope This test is applied to Open Text, Open Image, Save Text, Save Text As, Save Image, Save Image As... in File Menu; Open Image Key, Save Text Key, Open Text Key of Tools Menu, and the Image Bar (all the possible places where file open is required).</p>
<p>Test2: Setting test</p> <p>Test Approach Set Image Folder via Tools -&gt; Options -&gt; Set Image Directory Toggle Encryption On/ Off</p> <p>Goal Check if The Application Keeps the Image Directory Settings. Check if the application correctly "memorizes" Encryption Settings.</p> <p>Scope This test applied to Tools -&gt; Options Menu of GUI</p>
<p>Test3: Image bar functionality</p> <p>Test Approach Copy some images in the image directory set through Tools-&gt;Options Set Image Directory. Click on the next and previous buttons on the Image Bar.</p> <p>Goal To make sure the image bar functions according to the specifications. Only valid images get loaded to the image bar. No exception is thrown if the image directory contains other files (not image files), the files that pretend to be images (i.e. have image file extension, but are not in the correct format). To make sure that we can move to other images (not shown on the bar) by clicking Next and Prev.</p>

Scope

This test applies to Image Bar part of the GUI only

Test4: Switching tabs

Test Approach

Click to different tabs in the GUI of the application to switch between the text tab and image tab.

Goal

Make sure that when the tab is clicked, the view switches to other tab window

Scope

This test is applied to Main Panel (where there are Text and Image Tab for Text to be hidden and container image) and to Key Pane (where there are Text Tab and Image Tab for Text or Image Keys)

Test5: Responding to click events

Test Approach

Either use JUnit or write proprietary drivers and monitors to check how the buttons and menu items respond to user clicks or button presses.

Goal

To test if all the components correctly respond to user clicks or button presses.

Scope

This test applies to all of the Menu items (Excluding Help) not mentioned previously in the document and to Encode, Decode, and Load Image buttons.

Test6: Cut, copy, and paste

Test Approach

Copy, Cut, and Paste passages of text to and from all the GUI text containers.

Goal

To make sure that Copy, Cut, and Paste functions are correctly implemented.

Scope

This test is to be applied to the Text Box on the Key Pane and the Text Box on the Main Panel.

Test7: Loading help

Test Approach

Open Help Topics in a system default browser by clicking Help Topics item from Help Menu.  
Open About window by clicking about from Help Menu.

**Goal**

To test that Help Topics loads correctly to a system browser and About window is viewable to a user

**Scope**

Help Menu. Limitations: The application uses a default browser, so the correct functionality of Help Topics clicking depends on the correct settings and functionality of a system browser.

## 3.2.2 SECURITY TESTING

<p>Encryption Unit</p> <p>Test 1: Long plaintext</p> <p>Approach: Convert text files of size 0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5 Mb in size.</p> <p>Goal: To test how well this unit deals with large file sizes. Does it throw exceptions appropriately? (induce errors such as invalid keys, corrupt data.) Does it throw an exception when the cancel button is clicked.</p> <p>Scope: This test is applied to the function that converts plain text into ciphertext.</p>
<p>Image to Key Conversion Unit</p> <p>Test 1: Deterministic conversion</p> <p>Test Approach Take a pool of 100 images (different color set, size, and detail intensity) and convert each of these images to text key 100 times.</p> <p>Goal Check if every time we convert an image to a text key, we obtain the identical text key.</p> <p>Scope This test is applied to the function that takes an image and returns the text key representing an image.</p>
<p>Test 2: Variation in keys produced</p> <p>Test Approach Generate keys from 100 different images and compare the keys produced to see if they have acceptable amount of variation.</p> <p>Goal: To measure the obscurity ramifications of using hashed images for keys</p> <p>Scope: This test is applied to the function that takes an image and returns the text key representing an image.</p>
<p>Test 3: Large key image</p> <p>Approach:</p>

Convert images of size 1, 5, 10, 15, 20 Mb in size.

Goal:

To test how well this unit deals with large file sizes. Does it throw exceptions appropriately? (induce errors such as invalid keys, corrupt data.) Does it throw exception when the cancel button is clicked.

Scope:

This test is applied to the function that takes an image and returns the text key representing an image.

Huffman Unit

Test1: Compression / Decompression (Extractability)

Test Approach

Take 100 passages of text of different size, compress and decompress them.

Goal

Check if every time after we compress a passage of text we can decompress it. Make sure that the text we receive after decompression is 100% identical to the text that was compressed

Scope

This test is applied to the function that takes a passage of text and returns the text that is a compressed version of the text fed to the function.

## 3.2.3 GIF AND JPEG TESTING

### GIF

Due to the licensing issue, Java does not provide built-in GIF encoder, thus we must provide our own encoder. The `GifStegoImage` module must be broken down to 4 sub-modules. The first module constructs `colormap` and neighboring `colormap` from the `BufferedImage`. The second module handles the actual manipulation of the color in bit levels. The third module writes out the file in standard GIF format. The fourth module assures visual similarity and statistical soundness. While it will be easy to visually inspect the output images to determine if they have been visibly changed, it will be difficult to determine whether or not the method is statistically undetectable. We will do some testing on our own but will rely chiefly on the batch method implemented by the GUI group to try encoding hundreds of images at once in search of rare errors.

### JPEG

While it will be easy to visually inspect the output images to determine if they have been visibly changed, it will be difficult to determine whether or not the method is statistically undetectable. This somewhat dubious claim is made in the paper by Lee and Chen; however, it seems hard to believe that the method is truly undetectable, and it would be hard for us to measure without asking all of the security experts in the world to try breaking it. We will try to at least produce some histograms of the luminance values using MATLAB to show that the output images do not look suspicious.

Furthermore, the hardest part about testing the `JpegStegoImage` module will be ensuring that the bit error rate (BER) is zero. We will do some testing on our own but will rely chiefly on the batch method implemented by the GUI group to try encoding hundreds of images at once in search of rare errors.

## GIF and JPEG steganography unit tests

### Test 1: Bit error rate zero

#### Test Approach (Part 1)

100 container images, 100 **key images** (the images are of different size, color set, and detail intensity) and 100 passages of text of different size will be fed to a GIF steganography function for testing encryption / decryption<sup>9</sup> function.

#### Test Approach (Part 2)

100 container images (the images are of different size, color set, and detail intensity), 100 **text keys** of different size and 100 passages of text of different size will be fed to a GIF steganography function for testing encryption / decryption<sup>10</sup> function.

#### Goal

To make sure that after hiding the text in an image it could be successfully extracted and the extracted text 100% identical to the text to be encoded.

### Test 2: minimal picture distortion

#### Test Approach

Test 1 and Test 2 are performed on the smaller subset (on the scale of 10) of container, key images and text passages. A human being visually compares the empty container image and the container image containing text

#### Goal

To make sure that after hiding the text in an image, the image does not appear much different to a human eye.

#### Scope

The tests will be applied to:

- 1 . GIF and JPEG steganography functions that take a key (an image or a text), a container image and a text to hide and hide the text in a container image.
2. GIF and JPEG steganography functions that take a container image with hidden text and a key (either in the form of an image or text) and extract the hidden text from the container image.

---

<sup>9</sup> Here encryption and decryption means steganographic text hiding.

<sup>10</sup> Again encryption and decryption means steganographic text hiding.

## 3.3 INTEGRATION TESTING

Integration testing will be conducted after the integration the main units of the application described above. Integration testing should focus on ensuring that combining the parts of the application into larger units maintains the correct behavior; it should also ensure that the interfaces between the units are implemented correctly and that the new units' behavior is compatible with the behavior of the lower level units.

Integration Tests
<p>Integration Test 1</p> <p>Test Approach We test the new module with test1 and test2 of GIF and JPEG steganography unit section combined with test1 of compression unit section. Instead of supplying text to be encoded to GIF and JPEG functions directly, through integration these functions are supplied with the compressed text.</p> <p>Goal To make sure that after text is compressed and hidden to images it could be extracted from images and uncompressed, i.e. to ensure that the modules interoperate correctly</p> <p>Schedule: After integrating text compression unit with GIF and JPEG steganography functions</p>
<p>Integration Test 2</p> <p>Test Approach We test the new module with test1 and test2 of GIF and JPEG steganography unit section combined with test1 of compression unit section and combined with test 1 of image to key conversion unit. Instead of supplying text to be encoded to GIF and JPEG functions and key to be used to encode text directly, through integration these functions are supplied with the compressed text and the key converted from key images.</p> <p>Goal To make sure that after text is compressed and hidden to images it could be extracted from images and uncompressed, i.e. to ensure that the modules interoperate correctly.</p> <p>Schedule After integrating text compression unit with GIF and JPEG steganography functions, and with the function that converts key images into text</p>

# Appendix A: GLOSSARY

## Key Image

An image used by Paranoia to generate keys used to implement its security features. This image is not altered in the process.

## Source Image

An image, which is believed to contain hidden text. It is indistinguishable from a regular image, until an Extraction is performed.

## Target Image

An image from which the user would like to produce a copy, or output image, which looks similar and contains hidden text.

## Output Image

An image which has had text embedded into it, and can be exported or save to the user's hard drive.

## Keys

The input to Paranoia's security algorithms. They are in one of three forms; and image, a text string or an internal default key used if the user does not supply one.

## Pane / Panel

An area of the screen pertaining to a particular control or button.

## Extract

The process of finding and recovering text hidden in an image file.

## Embed

The process of hiding text in an image file.

## Appendix B: JPEG Examples and Parameters

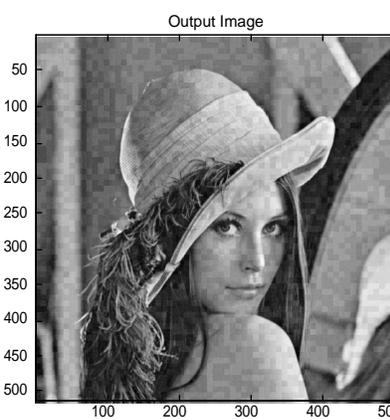
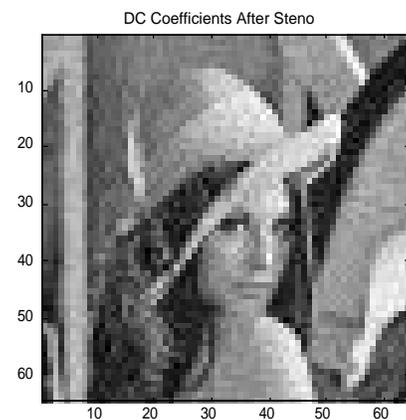
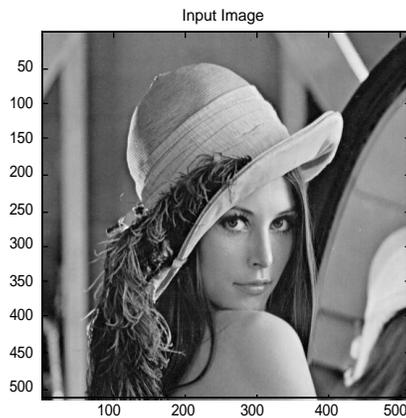
The following test shows that there is a lower limit on  $z$  (and thus an upper limit on  $k$ ). As will be shown, if  $z$  is too small, then the user will be able to notice the presence of steganography by simply viewing the output image.

**$z$ : # of entries in the stego-table**

21

**Compression Quality**

100



Numeric results:

<b><u>BER</u></b>	<b><u>Size of resulting JPEG file</u></b>	<b><u>SNR in dB</u></b>
0	162 kB	26

Here the output image looks rather bad because the stego-table contained too few entries. Thus, the DC coefficients had to be shifted further and introduce blocking effects. This also explains why the SNR is worse. Finally, the resulting file is just as big as before in the JPEG section of the design document because the same output compression level was used.

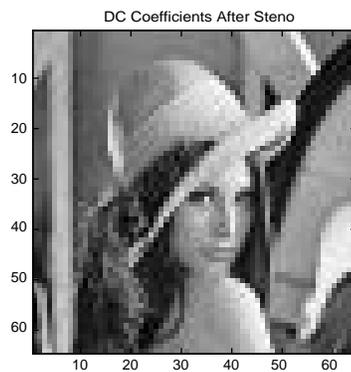
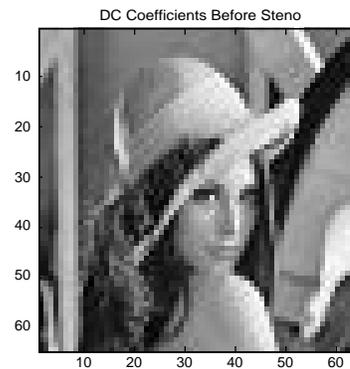
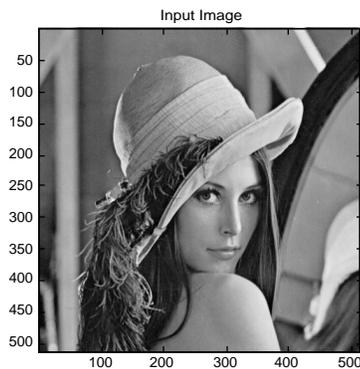
The following example shows what can go wrong if the compression quality is too low. While the paper from Lee and Chen purports that the technique works down to a compression quality of 25, our implementation in MATLAB began to break down around 40. This difference is probably due to some of the prototyping shortcuts that we took.

**z: # of entries in the stego-table**

81

**Compression Quality**

30



Numeric results:

<b><u>BER</u></b>	<b><u>Size of resulting JPEG file</u></b>	<b><u>SNR in dB</u></b>
0.15	15 kB	28

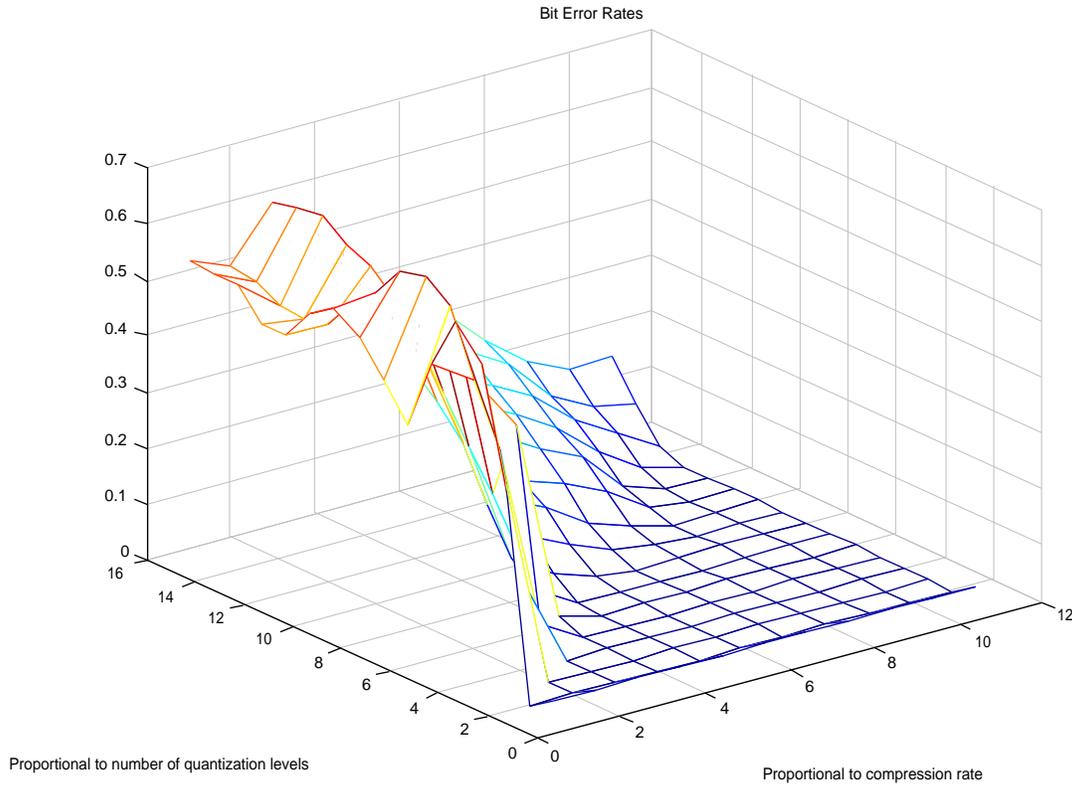
In this case, the output image looks good despite the small size of the JPEG file because the stego-table contained a medium number of entries. However, the compression rate was so low that it caused on average, 15 out of 100 bits to be extracted (and therefore also embedded) incorrectly. This shows that we need to be careful about not compressing too much, even though the output may appear correct when viewed with a program such as Photoshop.

In fact, when using Java, we can probably get bit error rates (BER's) always equal to zero because we can set the quantization levels used by Java for encoding the DC coefficients of the blocks. Setting the quantization levels to unusual values may however arouse suspicion.

The following tests were performed using many different compression levels and number of entries  $z$  in the stego-tables.

NOTE: The number of entries in the stego-tables  $z$  means the same thing as the number of quantization levels in the plots below.

For our application, we need a bit error rate equal to zero, so we need to choose the parameters such that they are on the right-half side of the plot.



For the especially curious reader: due to problems with indices on 3D plots in MATLAB, the indices here are only roughly proportional to the actual values used in the tests. The real values corresponding the vertices shown above and on the plot on the next page are given below:

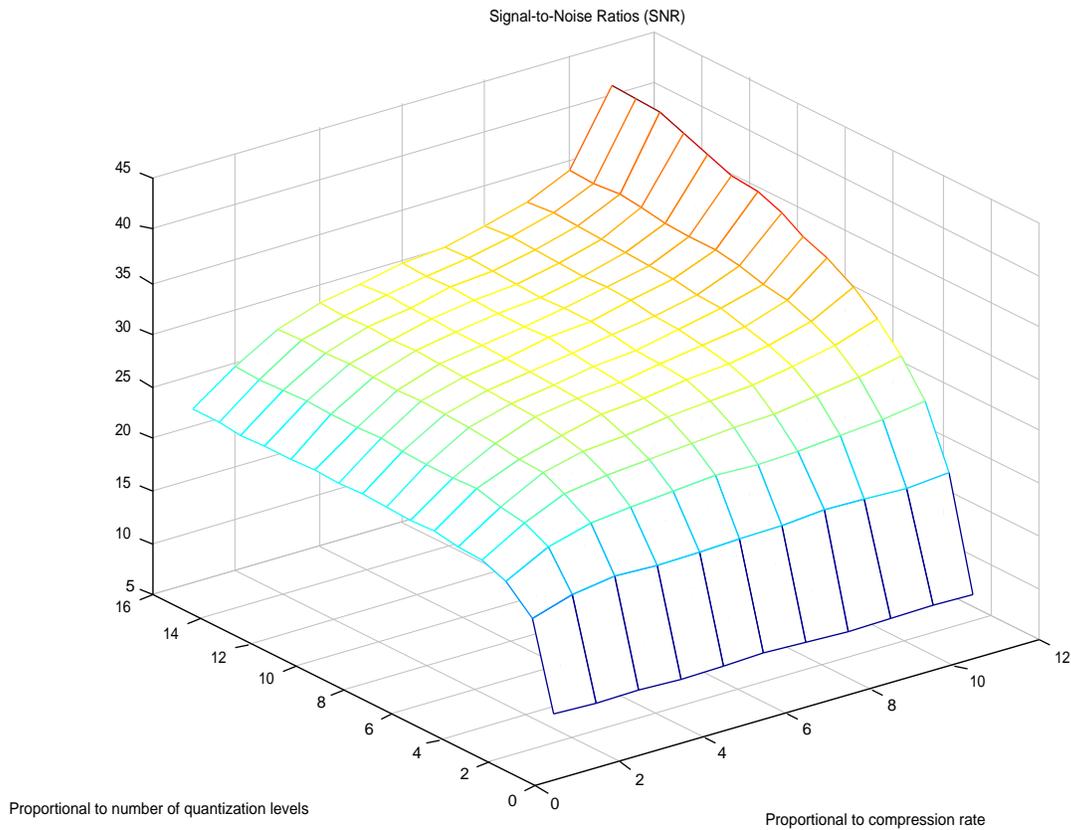
Number of quantization levels ( $z$ ):

5 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151

Compression quality ( $c$ ):

5 10 20 30 40 50 60 70 80 90 100

The signal-to-noise ratio (SNR) is an objective measure that is not always directly correlated with the subjective measure of whether or not the input and output images can be distinguished by the eye. However, very low SNR's indicate very high amounts of noise, so we need to avoid SNR's that are very low. It follows that neither of the parameters should be chosen to be particularly small.



Thus, a good guide for a compression rate would be 80% while using about 60 or 50 quantization levels. That implies, that  $z \approx 55$ .