Yes, absolutely. I remember it well.

We built the **Digital Steganographer**, a Java Swing application designed to hide and extract secret text messages within image files.

Here's a quick summary of what we implemented:

* **Core Functionality:** Hiding text in images (encoding) and extracting it (decoding).
* **Security:** You used **AES-256 encryption** to secure the message *before* hiding it, making it unreadable even if someone managed to extract it.
* **Technique:** It used the **Least Significant Bit (LSB)** method to embed the encrypted data into the image's pixels with minimal visual change.
* **User Interface:** A graphical user interface (GUI) built with **Java Swing**.
* **Database Logging:** It connected to a **MySQL** database to keep a log of all encode/decode operations.
* **Build Tool:** We used **Maven** to manage the project dependencies, specifically the MySQL connector.

We also went through the process of setting it up, fixing a bug in the decoding logic, and pushing the final project to your GitHub repository.

What's on your mind? Are you looking to add a new feature, fix another issue, or do you have a question about how part of it works?

Of course. I'd be happy to explain the core logic of the Digital Steganographer project in detail. Understanding these parts is crucial for speaking about the project confidently.

Think of your application's security like a **safe hidden inside a secret compartment**.

1. **The Safe:** This is the **encryption (CryptoCore)**. It locks up your message so that even if someone finds it, they can't open it without the key (your password).
2. **The Secret Compartment:** This is the **steganography (StegoCore)**. It hides the existence of the safe itself within a seemingly ordinary object (the image).

Let's dive deep into these two main parts.

**High-Level Workflow**

Before the code, let's visualize the entire process.

Encoding (Hiding the Message):

[Your Message] -> 1. Encrypt with Password -> [Encrypted Gibberish] -> 2. Hide in Image -> [New Stego-Image]

Decoding (Extracting the Message):

[Stego-Image] -> 1. Extract Hidden Data -> [Encrypted Gibberish] -> 2. Decrypt with Password -> [Original Message]

The core logic lives in two nested classes: CryptoCore and StegoCore.

**Deep Dive 1: CryptoCore (The Safe)**

This class is responsible for turning your readable message into secure, unreadable ciphertext and back again. It uses a modern, highly secure standard called **AES-256-GCM**.

**encrypt(String plainText, String password)**

This method takes your message and password and produces a Base64-encoded string of encrypted data.

1. **Generate a Salt:**

Java

byte[] salt = getRandomNonce(SALT\_LENGTH\_BYTE);

* + **What it is:** A salt is just random data.
  + **Why it's critical:** We don't use your password directly to encrypt. Instead, we combine your password with this unique, random salt. This means that even if two people use the exact same password ("12345"), the final encryption keys they generate will be completely different. This protects against "rainbow table" attacks, where attackers have pre-computed hashes of common passwords.

1. **Generate the Encryption Key (Key Stretching):**

Java

SecretKey aesKey = getAESKeyFromPassword(password.toCharArray(), salt);

This calls a helper method that uses an algorithm called **PBKDF2WithHmacSHA256**.

* + **What it does:** It takes your password and the salt and puts them through a slow, computationally expensive process (65,536 rounds in our code).
  + **Why it's critical:** This "key stretching" makes brute-force attacks extremely slow. An attacker can't just try millions of passwords per second; each guess requires significant processing power, making it much harder to crack.

1. **Generate an Initialization Vector (IV):**

Java

byte[] iv = getRandomNonce(IV\_LENGTH\_BYTE);

* + **What it is:** Another piece of random data, similar to a salt.
  + **Why it's critical:** The GCM encryption mode requires a unique IV for every single encryption operation. This ensures that if you encrypt the *exact same message* twice with the *same password*, the resulting encrypted text will be different each time. This prevents attackers from identifying patterns in your encrypted data.

1. **Perform Encryption:**

Java

Cipher cipher = Cipher.getInstance(ALGORITHM); // "AES/GCM/NoPadding"

cipher.init(Cipher.ENCRYPT\_MODE, aesKey, new GCMParameterSpec(TAG\_LENGTH\_BIT, iv));

byte[] cipherText = cipher.doFinal(plainText.getBytes(StandardCharsets.UTF\_8));

This is where the AES algorithm actually scrambles your message using the generated key and IV.

1. **Combine and Encode:**

Java

byte[] cipherTextWithIvSalt = ByteBuffer.allocate(...)

.put(salt)

.put(iv)

.put(cipherText)

.array();

return Base64.getEncoder().encodeToString(cipherTextWithIvSalt);

The final output isn't just the encrypted message. It's a bundle containing everything needed for decryption (except the password): [SALT] + [IV] + [CIPHERTEXT]. This bundle of raw bytes is then encoded into a simple text string using Base64 so it can be easily handled and hidden.

**decrypt(String cText, String password)**

This method is the exact reverse. It unpacks the bundle, regenerates the *exact same* encryption key using the password and the extracted salt, and then decrypts the message. If the password is wrong, the key generation will fail, and the decryption will produce garbage, resulting in the error you saw.

**Deep Dive 2: StegoCore (The Secret Compartment)**

This class is responsible for the clever part: hiding the encrypted data bundle within the image's pixels. It uses a variation of **Least Significant Bit (LSB) Steganography**.

Imagine a pixel's color value for Red is 11101010 in binary. The last bit (the LSB) has the smallest impact on the actual color. We can change it without anyone noticing. Our code goes a step further and modifies the **4 least significant bits** (a "nibble"). This allows us to store more data, but it's still visually undetectable.

**embed(BufferedImage image, String data)**

1. Prepare the Payload:

The data we hide isn't just the encrypted message. We create a full "payload":

[STEGO\_MARKER] + [DATA\_LENGTH] + [ENCRYPTED\_DATA]

* + **STEGO\_MARKER (4 bytes):** A unique "magic number" (0x5A5A5A5A). When we try to decode an image, the very first thing we look for is this marker. If it's not there, we immediately know this isn't one of our images and stop.
  + **DATA\_LENGTH (4 bytes):** This tells us the exact size of the encrypted data. During extraction, this is crucial because it tells us when to stop reading from the pixels.
  + **ENCRYPTED\_DATA (variable size):** The actual output from CryptoCore.encrypt().

1. Hide the Payload, Nibble by Nibble:

The code then iterates through the image, pixel by pixel (from left to right, top to bottom).

* + It takes the first byte of the payload (e.g., from the STEGO\_MARKER).
  + It splits this 8-bit byte into two 4-bit nibbles.
  + The **first nibble** replaces the 4 LSBs of the **Red** color channel in the current pixel.
  + The **second nibble** replaces the 4 LSBs of the **Green** color channel.
  + It then takes the *next* byte of the payload, splits it, and hides it in the **Blue** channel of the current pixel and the **Red** channel of the *next* pixel.
  + This process continues until the entire payload is hidden.

**extract(BufferedImage image)**

This method reverses the embedding process perfectly.

1. **Extract the Header:** It first reads just enough data from the first few pixels to reconstruct the first 8 bytes (the marker and the length).
2. **Verify the Marker:** It checks if the first 4 bytes match our STEGO\_MARKER. If not, it throws an error.
3. **Read the Length:** It reads the next 4 bytes to know exactly how much data it needs to look for.
4. **Extract the Data:** It continues reading from the pixels, re-assembling the nibbles back into bytes, until it has collected the full DATA\_LENGTH of the encrypted message.
5. **Return the Data:** It returns the reassembled encrypted data as a string, ready to be passed to CryptoCore.decrypt().

By layering these two techniques, your project achieves a high level of security. Encryption makes the data unreadable, and steganography makes it invisible.