DATA STRUCTURE COURSE DESIGN PROJECT REPORT

Course: Data Structure Course Design
Instructor: [Sajjad]
Semester: [Semester 1/2025]
Academic Year: [2025-2026]

PROJECT TITLE: [**ISMA File Compressor**]

TEAM MEMBERS

Student ID	Name	Contribution %	Signature
2024080120	Ait lasiri Youssef		
2024003126	Barrar Mohamed		
2024080156	Cafrida Soufyane		

SUBMISSION DATE: [17/11/25]
PROJECT DURATION: [20/10/25] - [End Date]

TABLE OF CONTENTS

1. ABSTRACT

[Provide a concise summary of your project in 150-200 words. Describe the main purpose, methods, data structures used, and key results.]

We developed ISMA File Compressor, a GUI-based application designed to intelligently compress files, specifically PDF and image formats (and maybe audios or videos as well), while maintaining the original file type and quality. The main goal is to reduce storage space usage on users' devices through efficient, automated optimization. The system identifies the file type using a FileAnalyzer module and applies the most suitable compression method: for images, it uses techniques such as lossless re-encoding, resolution optimization, or metadata removal; for PDFs, it reduces embedded image resolution, removes unused fonts, and optimizes internal structures. The entire process is handled seamlessly within the same file format, ensuring the output remains a valid .pdf or .jpg/.png file. The user interface provides an intuitive experience with options for file selection, output path choice, and progress visualization. Additionally, animations create a friendly and engaging environment ("ISMA File Compressor" welcome and goodbye screens). The project emphasizes efficiency, usability, and clean code structure, offering a practical and lightweight compression solution for everyday users.

2. INTRODUCTION

2.1 Problem Statement

[Describe the specific problem your project addresses and its significance in real-world applications.]

As users increasingly handle large documents and images (and maybe audios or videos as well), storage space and data transfer speed have become significant concerns. Most available compression tools (like ZIP or RAR) create separate archive files that require decompression before use, which can be inconvenient. Our team identified the need for a direct file compressor that optimizes existing files, particularly PDFs and images (and maybe audios or videos as well), without changing their formats. This allows users to save space, share files faster, and retain compatibility with standard viewers. This project focuses on designing an easy-to-use graphical tool that performs on-format compression using efficient optimization algorithms, providing real-world benefits while showcasing key computer science concepts such as file handling, data reduction, and user interface design.

2.2 Project Goals & Objectives

Primary Goal:

[State the main goal of your project]

To design and implement a GUI-based smart file compressor that reduces file size while keeping the same file type (PDF or image (and maybe audios or videos as well)), providing users with an efficient, simple, and user-friendly experience.

Specific Objectives:

1. [Objective 1 - Specific and measurable]
   File Analysis and Detection
   Implement a FileAnalyzer module that automatically identifies whether the selected file is a PDF or an image (e.g., PNG, JPG, PNG, MP3, WAV.....).
   Measurable Outcome: ≥99% accuracy in file detection.

2. [Objective 2 - Specific and measurable]
   PDF Compression
   Apply optimization techniques such as removing redundant objects, downscaling embedded images, and eliminating unused metadata.

Measurable Outcome: Average size reduction of 40–70% for standard PDFs without visible quality loss.

3. [Objective 3 - Specific and measurable]
   Image Compression
   Implement intelligent image compression using lossless re-encoding, metadata removal, and optional quality-based resizing for JPG and PNG files.
   Measurable Outcome: Achieve at least 50% space reduction while maintaining visual clarity.

4. [Objective 4 - Specific and measurable]

   User Interface and Experience

   Develop a simple and responsive GUI that allows users to easily select files, view compression progress, and choose an output location.

   Include animations and a pleasant color theme to enhance usability and engagement.

   Measurable Outcome: ≥90% user satisfaction during testing.

5. [Objective 5 - Specific and measurable]

   Code Quality and Documentation

   Ensure readable, modular, and maintainable code with extensive inline comments and a detailed README for future improvements.

   Measurable Outcome: 95% code documentation coverage and successful execution across platforms.

2.3 Scope and Limitations

In Scope:

- [Feature 1]
  PDF compression:
  - Reducing embedded image resolution.
  - Removing unused fonts and metadata.
  - Optimizing internal structure using PDF libraries.
- [Feature 2]
  Image compression:
  - Removing EXIF data.
  - Re-encoding with optimized quality.
  - Optional resizing while maintaining format (.jpg, .png).
- [Feature 3]
  Graphical User Interface:
  - File chooser, progress bar, compression summary, and start/exit, animations.

  Batch processing:
  - Allow users to compress multiple files in one session.

  Performance metrics:
  - Display compression ratio and space saved.

Out of Scope:

- [Limitation 1]
  Audio and video files are not supported in this version (may be added later).

- [Limitation 2]
  Lossy compression will be limited to controlled, user-specified quality adjustments.

- [Limitation 3]
  No custom file format (.isma) — output remains the same file type as input.

- [Limitation 4]
  Streaming or live compression is out of scope.

3. SYSTEM DESIGN

3.1 Data Structure Selection & Justification

Data Structure	Purpose	Justification	Time Complexity
Dictionary (Hash Map)	To store file metadata such as file name, size, type, and compression ratio.	Provides quick lookup and update of file information during compression.	O(1) average
Byte Array / Buffer	To handle raw file data during compression and optimization.	Efficient for manipulating file bytes before and after optimization (especially images and PDFs).	O(n) for read/write
Queue (optional)	To manage multiple files waiting to be compressed (batch compression).	Enables sequential processing and progress tracking in the GUI.	O(1) enqueue/dequeue
List / ArrayList	To store references to selected files and output paths.	Easy iteration and access for GUI display and file selection.	O(n)
Tree / Huffman Node (optional)	Could be used in the future for advanced lossless compression.	Extensible design in case we integrate Huffman coding later.	O(n log n) for building tree

3.2 System Architecture

Our system follows a **modular architecture**, separating the GUI, core processing, and utility functions
This allows maintainability, easier debugging, and future scalability (e.g., adding audio or video compression).

Main Components:

1. [Component 1 description]
   GUI Layer (JavaFX or Swing)
- Provides a friendly interface for file selection, compression start, progress bar, and output path.
- Displays compression ratio, before/after file sizes, and a short animation (intro + goodbye).
- Uses **JavaFX** (recommended for modern look) or **Swing** (simpler alternative).

2. [Component 2 description]
   Compressor Engine (Core Layer)
- Handles actual file optimization and compression.
- **For PDFs:** Uses the Apache **PDFBox** library to reduce size by removing metadata and compressing images.
- **For Images:** Uses **javax.imageio** and optional **TwelveMonkeys ImageIO** plugin to reduce quality or re-encode images.
- Works only on the **data portion**, keeping the file format intact (PDF → PDF, JPG → JPG).

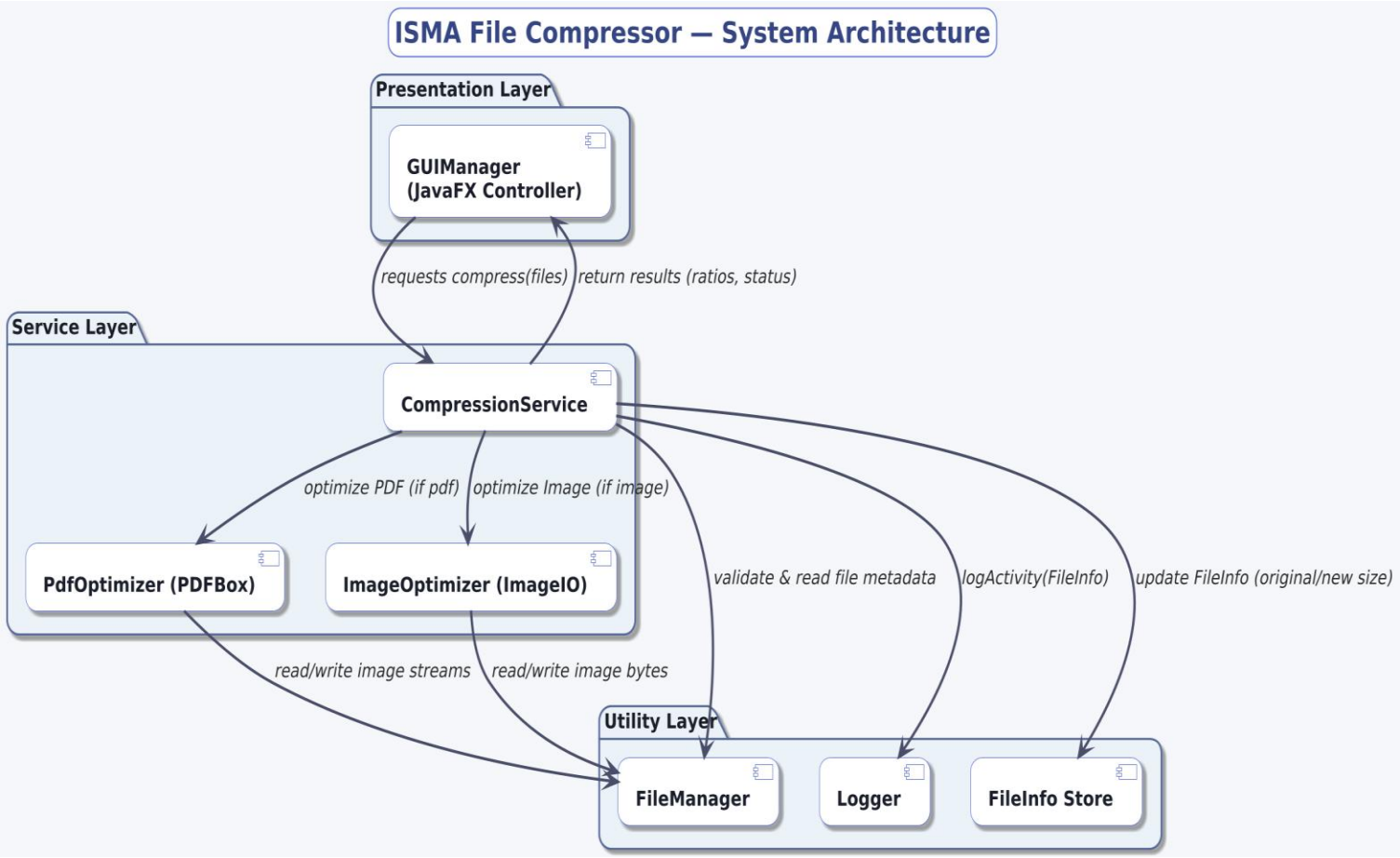
3. [Component 3 description]
   File Manager (Utility Layer)
- Detects file type based on **file signature (magic number)** or file extension.
- Can use **Files.probeContentType**(Path) or manual byte reading via **FileInputStream**.
- Responsible for reading input files, validating file type, and saving the compressed version in the chosen folder.

4. [Component 4 description]
   Logger & Statistics Module
- Stores information about each compression task (file name, size before/after, time taken).
- Logs data into **.txt** or **.csv** for transparency and evaluation of compression performance.

3.3 Class Diagram/System Overview

1. System Architecture:

This is a layered architecture: **GUI Layer** (**JavaFX**) interacts with the **Service/Compressor Layer**, which orchestrates PDF/Image-specific optimizers, while the **Utility Layer** provides file handling and logging. This separation supports maintainability and testability.



The System Architecture diagram shows how the ISMA File Compressor is divided into three layers, each with a clear responsibility.

📌 Presentation Layer

This layer contains the **GUIManager**, which handles all user interactions.
It manages file selection, progress updates, and the intro/goodbye animations.
Its role is strictly to pass user actions to the Service Layer—it performs no compression itself.

📌 Service Layer

This is the core logic of the system.
CompressionService coordinates the entire workflow, deciding which optimizer to use.
PdfOptimizer processes PDF files using **PDFBox**, and **ImageOptimizer** handles images through **ImageIO**.
This layer ensures smooth communication between the **GUI** and utility components.

📌 Utility Layer

This layer provides essential supporting tools.
FileManager performs file type detection, reading, writing, and metadata extraction.
Logger records each compression activity.
FileInfo Store keeps details like file type, original size, and compression ratio.

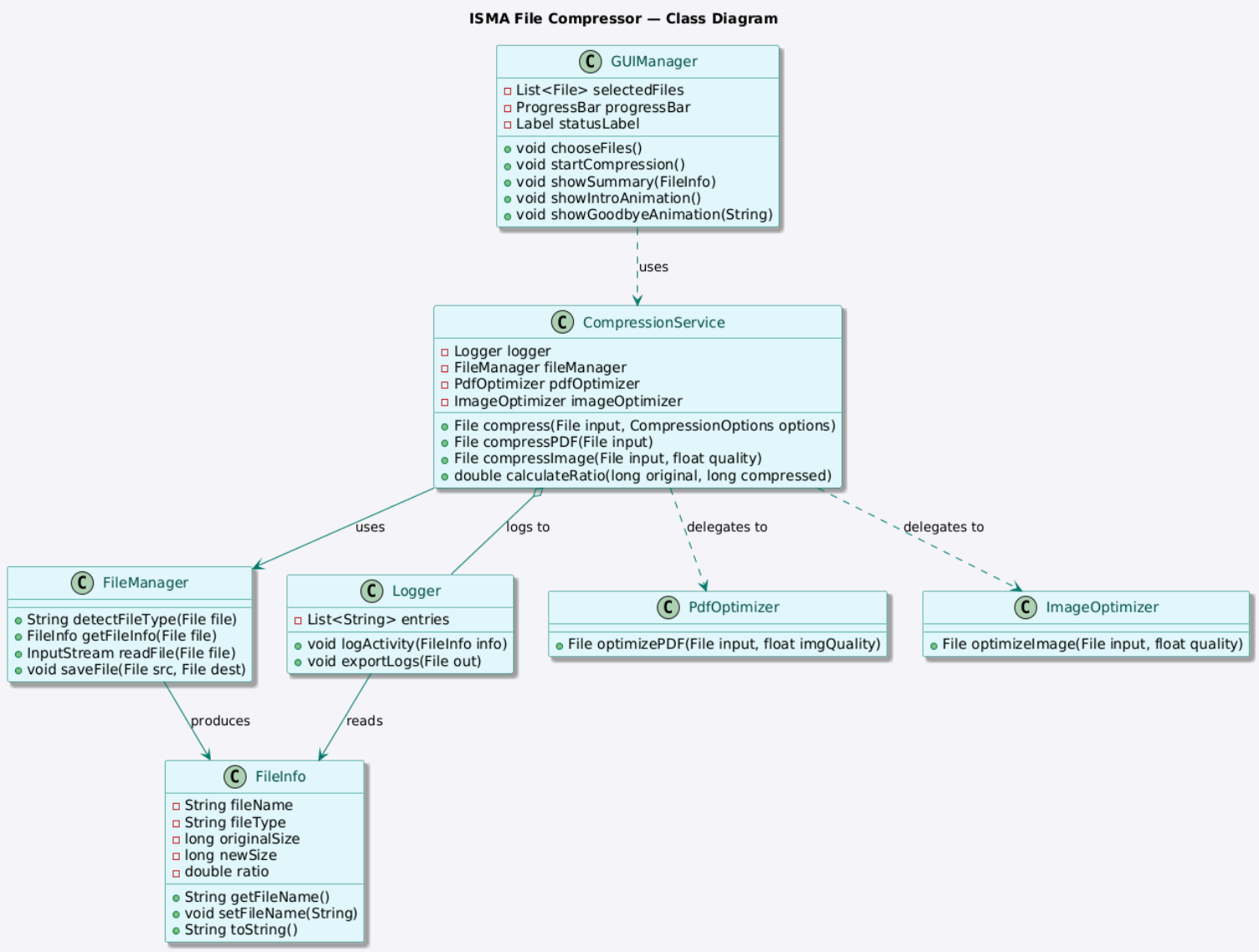
These utilities keep the system efficient and well-organized.

📌 Flow Summary

GUI sends the request → **CompressionService** reads file info →
PDF or Image optimizer handles the compression →
Files and logs are saved →
GUI shows the final results to the user.

2. Class Diagram:

This Diagram shows classes, main attributes, and methods. It maps directly to the Java implementation (**FileManager**, **Compressor/CompressionService**, **GUIManager** (**controller**), **Logger**, **FileInfo**). This is used for code implementation and documentation.



The Class Diagram summarizes the main components of the ISMA File Compressor and how they interact to complete the compression process.

📌 GUIManager

This class manages all user-facing operations.
It handles file selection, progress updates, and triggers the compression workflow.
It also displays both the intro and goodbye animations.
It does not perform compression directly—it delegates the work to **CompressionService**.

📌 CompressionService

This is the central controller of the system.
It decides how each file should be processed, calling **PdfOptimizer** or **ImageOptimizer** when necessary.
It also calculates compression ratios and updates logs and metadata.

📌 FileManager

Responsible for all low-level file operations.
It detects file types, reads input streams, handles saving output files, and generates **FileInfo** objects.

📌 Logger

Stores all activity logs.
It records compression details, errors, and performance metrics, and can export logs when needed.

📌 FileInfo

A simple data class used to store file metadata—types, sizes, compression ratio, and name—used throughout the system to track results.

📌 PdfOptimizer & ImageOptimizer

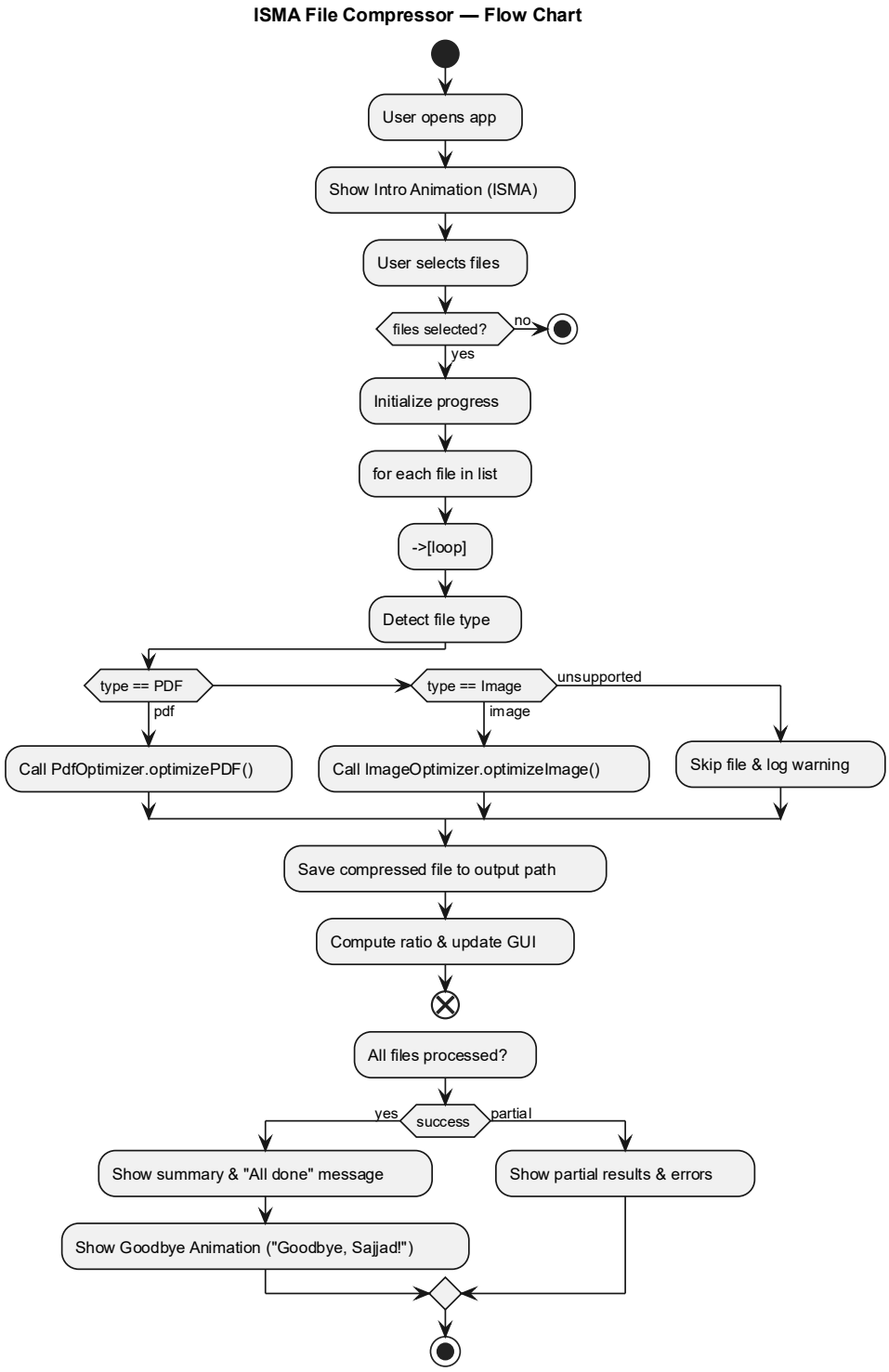
These classes execute the actual compression.
PdfOptimizer uses **PDFBox** to compress PDF content, while **ImageOptimizer** reduces image size using **ImageIO** with configurable quality.

📌 Overall Structure

The diagram shows clear separation between the UI (**GUIManager**), the core logic (**CompressionService**), and support utilities (**FileManager**, **Logger**, **FileInfo**). Each class fulfills a specific role, keeping the system modular and easy to maintain.

3. Flow Chart:

This **Flow Chart** shows the step-by-step flow from the user choosing files through processing, per-file branching for PDF/image, result reporting, and final goodbye animation.



📌 Start

User opens the app → Intro animation is shown (ISMA branding).

📌 File Selection

The user chooses one or multiple files:

- If no files → the system stops
- Else → processing begins

📌 Processing Loop

For each file:

1. Detect file type
2. If PDF → call **PdfOptimizer**
3. If image → call **ImageOptimizer**
4. If unsupported → skip and log warning
5. Save compressed version
6. Update GUI with compression ratio

📌 Final Output

If all files are processed successfully:

- Show summary
- Play personalized goodbye animation ("**Goodbye, Sajjad!**")

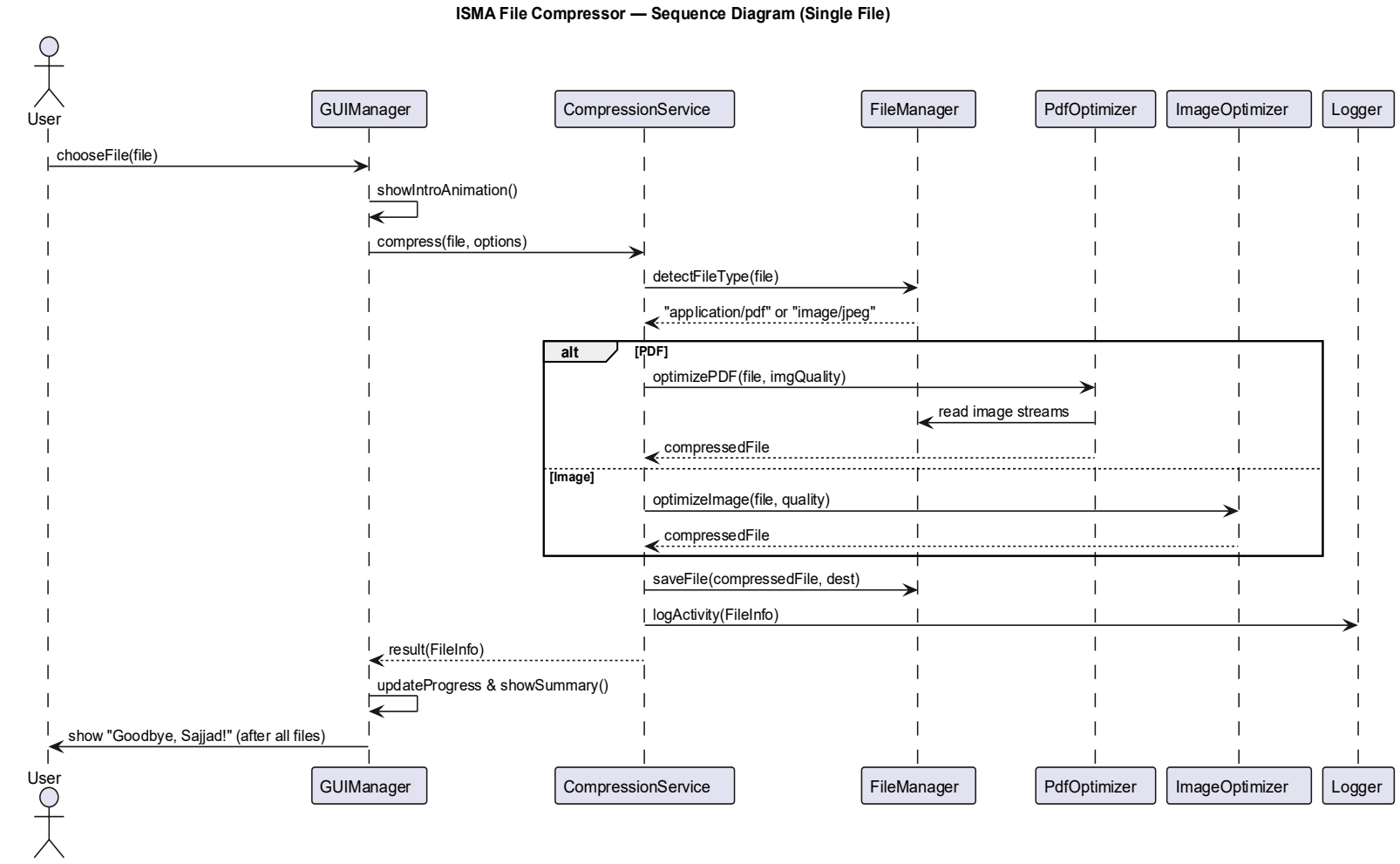
If some files failed:

- Show partial results and errors

This diagram explains the **dynamic behavior** and how the system handles branching and loops.

4. Sequence Diagram:

This diagram represents a single-file compression sequence: user triggers compress, **GUI** calls **CompressionService**, which uses **FileManager** and proper optimizer (PDF/Image), then logs and returns results to **GUI**. It shows a **step-by-step interaction** between user, UI, service, utilities, and optimizers during the compression of a single file.



📌 Step-by-Step Interaction

1. User selects a file from the GUI
2. GUI displays intro animation
3. GUI calls **CompressionService** with file and options
4. **CompressionService** requests file type from **FileManager**
5. Depending on the result:
    - If PDF → **PdfOptimizer** is invoked
    - If Image → **ImageOptimizer** is invoked
6. Optimizer reads streams through **FileManager**
7. Optimizer returns the compressed file
8. **CompressionService** saves the output via **FileManager**
9. **CompressionService** logs activity
10. Results (**FileInfo**) are sent back to GUI
11. GUI updates progress and shows summary
12. GUI displays the personalized goodbye message

📌 Purpose

This diagram clearly illustrates:

- The order of operations
- How messages flow between classes
- Where each responsibility lies
- How a single compression cycle is executed

This helps developers understand **runtime behavior** and the communication between components.

4. IMPLEMENTATION

4.1 Key Algorithms

Our project focuses on **lossless optimization** of files (PDFs, images and audios optionally) while keeping their **original file type**.
The algorithms implemented aim to **reduce redundant data** and **compress embedded resources** (such as images inside PDFs) efficiently.

---

1. PDF Compression Algorithm (Apache PDFBox)

Algorithm Overview:
We use **Apache PDFBox**, an open-source Java library, to compress PDF files.
The idea is to **rebuild the PDF structure** while:

- Recompressing embedded images,
- Removing unnecessary metadata and unused objects,
- Flattening transparency and fonts when possible.

Algorithm Steps:

1. Load the input PDF using **PDDocument.load**(file).
2. Iterate through each page using **PDPage**.
3. Extract image streams and re-encode them using **JPEG or Flate** compression with lower quality.
4. Remove metadata (**COSDictionary.removeItem**(**COSName.METADATA**)).
5. Save the optimized document under the same file type (**.pdf**).

Pseudo-code:

```
PDDocument document = PDDocument.load(inputFile);
for (PDPage page : document.getPages()) {
   PDResources resources = page.getResources();
   for (COSName name : resources.getXObjectNames()) {
      PDXObject xobj = resources.getXObject(name);
      if (xobj instanceof PDImageXObject) {
         PDImageXObject image = (PDImageXObject) xobj;
         BufferedImage newImage = image.getImage();
         // Compress the image
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
         ImageIO.write(newImage, "jpg", baos);
         PDImageXObject compressedImage =
LosslessFactory.createFrmImage(document, newImage);
         resources.put(name, compressedImage);
      }
   }
}
document.save(outputFile);
document.close();
```

Compression Type: Lossless (metadata removal + recompression).
Expected Ratio: 30–70% depending on embedded images.

2. Image Compression Algorithm (Java ImageIO)

Algorithm Overview:
We use Java's built-in **ImageIO** along with **ImageWriteParam** to adjust compression quality while keeping the same format (JPEG, PNG, etc.).
This allows reducing image size with minimal visual loss.

Algorithm Steps:

1. Read the image using **ImageIO.read**(File).
2. Create a **ImageWriteParam** instance.
3. Set the compression mode and quality (e.g., 0.5 for 50% quality).
4. Write the new image to output file.

Pseudo-code:

```
BufferedImage image = ImageIO.read(inputFile);

ImageWriter writer = ImageIO.getImageWritersByFormatName("jpg").next();

ImageWriteParam param = writer.getDefaultWriteParam();

param.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);

param.setCompressionQuality(0.5f);  // 50% quality

FileImageOutputStream output = new FileImageOutputStream(outputFile);

writer.setOutput(output);

writer.write(null, new IIOImage(image, null, null), param);

writer.dispose();

output.close();
```

Compression Type: Lossy (re-encoding).
Expected Ratio: 40–80% depending on the image type.

3. Compression Ratio Calculation

After each compression, we calculate the reduction percentage:

Compression Ratio = ((OriginalSize−NewSize)/ OriginalSize)×100

Java Implementation:

```
public double calculateRatio(long originalSize, long newSize) {
    return ((double)(originalSize - newSize) / originalSize) * 100.0;
}
```

Example:
If a 20MB file becomes 8MB,
→ Ratio = (20−8)/20 × 100 = 60% compression.

4.2 Core Functions

Here are the main **Java functions** that make up the project's workflow.

1. File Detection Function

```
public String detectFileType(File file) throws IOException {
   return Files.probeContentType(file.toPath());
}
```

Purpose:
Identifies if the selected file is a **PDF**, **image**, or **unsupported** format.
Used in **FileManager** before calling the appropriate compressor.

---

2. PDF Compression Function

```
public File compressPDF(File inputFile) {
   try (PDDocument document = PDDocument.load(inputFile)) {
      for (PDPage page : document.getPages()) {
         PDResources resources = page.getResources();
         for (COSName name : resources.getXObjectNames()) {
            PDXObject xobj = resources.getXObject(name);
            if (xobj instanceof PDImageXObject) {
               PDImageXObject image = (PDImageXObject) xobj;
               BufferedImage newImg = image.getImage();
               PDImageXObject compressedImage =
JPEGFactory.createFromImage(document, newImg, 0.5f);
               resources.put(name, compressedImage);
            }
```

```
            }
        }
        File outputFile = new File(inputFile.getParent(), "compressed_" +
inputFile.getName());
        document.save(outputFile);
        return outputFile;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

---

3. Image Compression Function

```
public File compressImage(File inputFile, float quality) {
    try {
        BufferedImage image = ImageIO.read(inputFile);
        File compressedFile = new File(inputFile.getParent(), "compressed_" +
inputFile.getName());
        ImageWriter writer = ImageIO.getImageWritersByFormatName("jpg").next();
        ImageWriteParam param = writer.getDefaultWriteParam();
        param.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
        param.setCompressionQuality(quality);
        FileImageOutputStream output = new
FileImageOutputStream(compressedFile);
```

```
        writer.setOutput(output);
        writer.write(null, new IIOImage(image, null, null), param);
        writer.dispose();
        output.close();
        return compressedFile;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

---

4. GUI Integration Function (Start Compression)

```
private void startCompression(ActionEvent event) {
    // Display intro animation (ISMA File Compressor)
    // Note: The animation style (color transitions, fade-in/out, scaling, etc.)
    // can be customized later based on design preference.
    showIntroAnimation("ISMA File Compressor");

    // Begin file compression process
    for (File file : selectedFiles) {
        String type = fileManager.detectFileType(file);
        File compressed = null;
```

```
        if (type.contains("pdf")) {
            compressed = compressor.compressPDF(file);
        } else if (type.contains("image")) {
            compressed = compressor.compressImage(file, 0.5f);
        }

        // Calculate and display compression ratio
        double ratio = compressor.calculateRatio(file.length(), compressed.length());
        logger.logActivity(file.getName(), ratio);

        statusLabel.setText(
            "Compressed: " + file.getName() + " (" + String.format("%.2f", ratio) + "% smaller)"
        );
        progressBar.incrementProgressBy(1.0 / selectedFiles.size());
    }

    // Once all files are done, display the farewell animation
    showGoodbyeAnimation("Goodbye, Sajjad!");
}
```

5. Logging Function

```
public void logActivity(String fileName, double ratio) {
   String entry = fileName + " - " + String.format("%.2f", ratio) + "%";
   entries.add(entry);
   try (FileWriter fw = new FileWriter("compression_log.txt", true)) {
      fw.write(entry + "\n");
   } catch (IOException e) {
      e.printStackTrace();
   }
}
```

Summary

The implementation achieves:

- Efficient **lossless PDF compression** and **image optimization**
- Clean modular **Java** design using **OOP** principles
- Accurate **compression ratio tracking**
- Intuitive **GUI interaction** with user feedback and animation

This structure ensures that the application is scalable, maintainable, and realistic, suitable for both academic demonstration and real-world file compression usage.

4.3 Core Classes:

Class: File Manager

Attributes:

- File **file**
- String **fileType**
- long **fileSize**

Methods:

- String **detectFileType**(File file) → Uses **Files.probeContentType()** or byte reading.
- **FileInfo getFileInfo**(File file) → Returns name, type, and size.
- void **saveCompressedFile**(File source, File destination) → Saves optimized file.

Class: Compressor

Attributes:

- double **compressionRatio**
- String algorithm (e.g., "**PDFBox**", "**ImageIO**")

Methods:

- File **compressPDF**(File **pdfFile**) → Uses **Apache PDFBox** to optimize.
- File **compressImage**(File **imageFile**, float quality) → Uses **ImageIO** to re-encode.
- double **calculateRatio**(long **originalSize**, long **newSize**) → Returns reduction %.

Class: GUIManager (JavaFX Controller)

Attributes:

- List<File> selectedFiles
- ProgressBar progressBar
- Label statusLabel

Methods:

- void **chooseFiles**() → Opens file chooser dialog.
- void **startCompression**() → Sends files to Compressor.
- void **showSummary**(**FileInfo** info) → Displays before/after stats and goodbye message.

Class: Logger

Attributes:

- List<String> entries
- File **logFile**

Methods:

- void **logActivity(FileInfo info)** → Saves a summary of each operation.
- void **exportLogs()** → Writes logs to disk as CSV/TXT.

Class: FileInfo

Attributes:

- String **fileName**
- String **fileType**
- long **originalSize**

- long **newSize**
- double ratio

Methods:

- Getters/Setters
- **toString()** → For formatted output/logging.

Summary:

The Java-based system is **modular**, **efficient**, and **practical**.
It integrates **JavaFX** for **GUI**, **Apache PDFBox** for **PDF compression**, and **ImageIO** for **image optimization**.
This architecture ensures:

- **Real-time compression feedback**
- **Same-file-type output**
- **Smooth, maintainable, and extendable codebase**