Facade Pattern: Media Converter

 **Name:** Yermakhanbet Balgyn

 **Group:** SE-2404

# Introduction

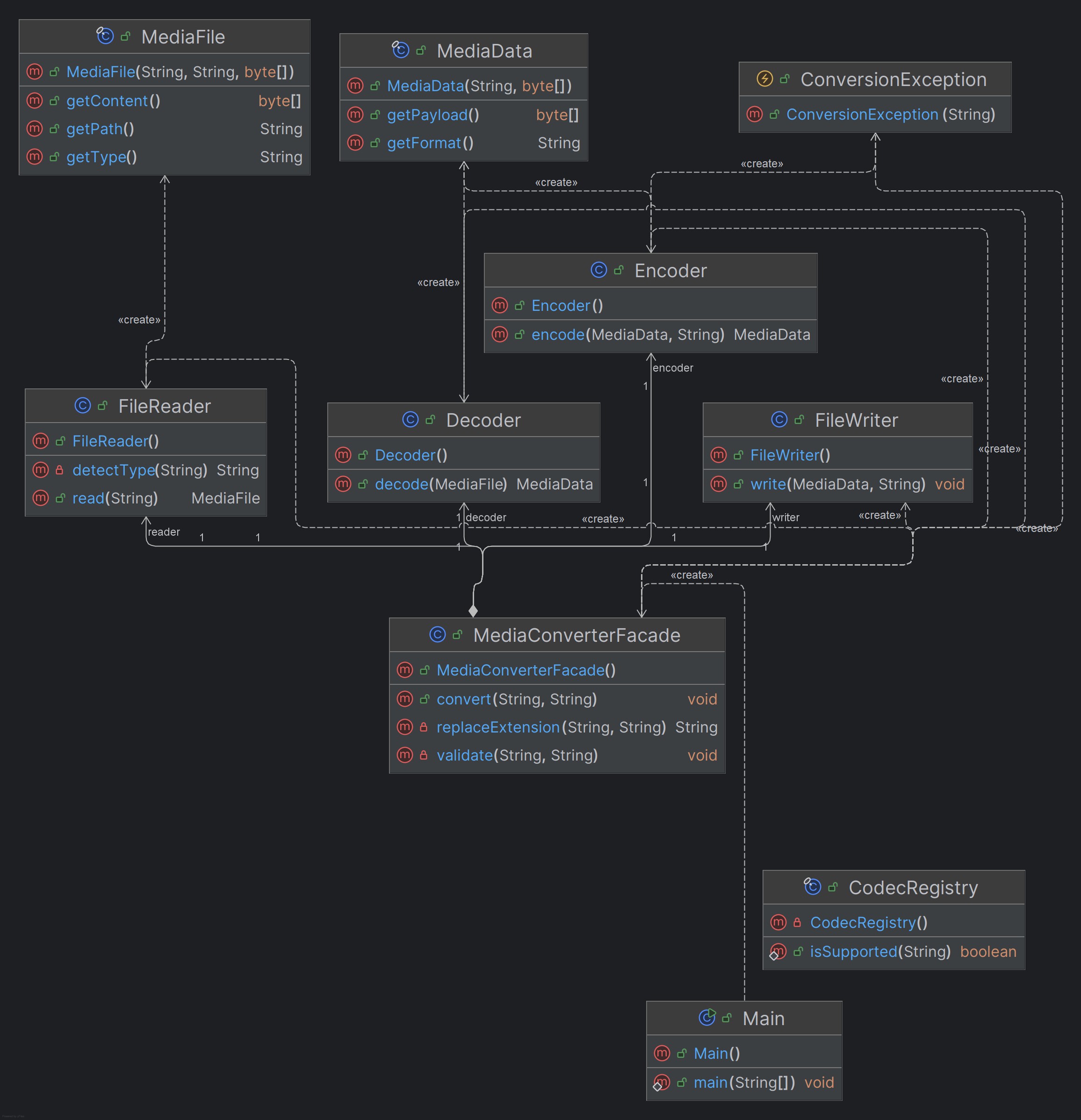
The Facade pattern is a structural design pattern that provides a simplified interface to a complex subsystem.

This project demonstrates the pattern through a media file converter that handles video, audio, and image format conversions.

The key benefit: clients interact with one simple interface instead of managing multiple complex components.

# System Architecture

UML Diagram



# Core Implementation

## The Facade Class

public class MediaConverterFacade {

private final FileReader reader = new FileReader(); private final Decoder decoder = new Decoder();

private final Encoder encoder = new Encoder();

private final FileWriter writer = new FileWriter();

public void convert(String inputPath, String outputFormat) throws ConversionException {

validate(inputPath, outputFormat);

MediaFile raw = reader.read(inputPath); MediaData decoded = decoder.decode(raw);

MediaData encoded = encoder.encode(decoded, outputFormat);

writer.write(encoded, replaceExtension(inputPath, outputFormat));

System.out.println("Converted '" + inputPath + "' -> '" + replaceExtension(inputPath, outputFormat) + "'");

}

}

The facade hides four subsystem components (FileReader, Decoder, Encoder, FileWriter) behind a single

convert() method.

## Client Usage

public class Main {

public static void main(String[] args) {

MediaConverterFacade facade = new MediaConverterFacade();

try {

facade.convert("sample.mov", "mp4"); facade.convert("song.wav", "mp3"); facade.convert("photo.heic", "jpg");

} catch (ConversionException e) {

System.err.println("Conversion failed: " + e.getMessage());

}

}

}

Simple and clean — no need to understand the internal pipeline.

## Subsystem Components

public class FileReader {

public MediaFile read(String path) {

System.out.println("[FileReader] Reading " + path); String detectedType = detectType(path);

return new MediaFile(path, detectedType, new byte[]{});

}

}

public class Encoder {

public MediaData encode(MediaData source, String targetFormat) throws ConversionException {

System.out.println("[Encoder] Encoding to " + targetFormat); if (!CodecRegistry.isSupported(targetFormat)) {

throw new ConversionException("Unsupported output format: " + targetFormat);

}

return new MediaData(targetFormat, new byte[]{});

}

}

Each component has a single responsibility: reading, decoding, encoding, or writing.

Format Registry

public final class CodecRegistry {

private static final Set<String> SUPPORTED = Set.of("mp4", "mp3", "jpg", "png", "avi", "wav");

public static boolean isSupported(String format) {

return SUPPORTED.contains(format.toLowerCase());

}

}

# Key Benefits

 **Simplified Interface**: One method call instead of coordinating four components

 **Loose Coupling**: Clients depend only on the facade, not internal subsystems

 **Easy to Extend**: Add new formats or components without changing client code

 **Single Responsibility**: Each class has one clear purpose

# Output Example

[FileReader] Reading sample.mov

[Decoder] Decoding sample.mov (type=mov) [Encoder] Encoding to mp4

[FileWriter] Writing to sample.mp4 (format=mp4) Converted 'sample.mov' -> 'sample.mp4'

Conclusion

This implementation demonstrates how the Facade pattern effectively simplifies complex subsystems. By

providing a single entry point, the pattern reduces coupling, improves code maintainability, and makes the

system easier to use. The media converter shows that clients can perform sophisticated operations through a simple interface without understanding the internal conversion pipeline.