Building your application as a DLL in C++ for integrating **GStreamer** and **OpenCV** is a great idea, but the question of organizing your code into one class versus multiple classes depends on **modularity**, **maintainability**, and **scalability**. Here's a structured approach to help you decide.

**Key Considerations**

1. **Complexity of Your Pipeline**:
   * If your pipeline is simple (e.g., source → processing → sink), a single class can work, but it's generally not ideal.
   * For complex pipelines with multiple stages (e.g., branching, multi-threading, error handling), using multiple classes provides better separation of concerns.
2. **Separation of Responsibilities**:
   * Keep **GStreamer-specific logic** (e.g., pipeline creation, event handling) and **OpenCV-specific logic** (e.g., image processing) separate.
   * This separation makes the code easier to debug, extend, and reuse.
3. **Reusability and Testing**:
   * Breaking functionality into smaller, focused classes allows for easier unit testing and potential reuse in other projects.
4. **Flexibility for Future Changes**:
   * If you plan to add features (e.g., support for new sources, filters, or outputs), a modular design makes such changes easier.

**Recommended Class Design**

**1. Core Pipeline Class**

* A single class to manage the overall pipeline.
* Responsible for:
  + Initializing GStreamer.
  + Connecting GStreamer and OpenCV (using appsink and appsrc).
  + Managing the flow of data through the pipeline.

**2. Separate Classes for Key Responsibilities**

Break down the functionality into logical units. For example:

| **Class Name** | **Responsibility** |
| --- | --- |
| PipelineManager | High-level management of the GStreamer pipeline (e.g., start, stop, error handling). |
| FrameProcessor | OpenCV-based image processing (e.g., transformations, object detection, or filtering). |
| GStreamerSource | Handles GStreamer source elements (e.g., video files, live streams, or cameras). |
| GStreamerSink | Manages the GStreamer sink elements (e.g., display, file output, or network streaming). |
| Config | Encapsulates configuration settings for the pipeline (e.g., resolution, formats, processing parameters). |

**Example Code Structure**



**How It Works Together**

1. **PipelineManager**:
   * Initializes GStreamer.
   * Sets up the pipeline by combining GStreamerSource, FrameProcessor, and GStreamerSink.
   * Starts and stops the pipeline.
2. **GStreamerSource**:
   * Responsible for setting up GStreamer source elements like v4l2src or filesrc.
3. **FrameProcessor**:
   * Handles OpenCV-based frame processing logic.
   * Takes raw frames from appsink and applies processing.
4. **GStreamerSink**:
   * Manages sink elements like autovideosink, filesink, or rtmpsink.

**Advantages of Multiple Classes**

1. **Modularity**:
   * Each class has a single responsibility, making the codebase easier to understand and extend.
2. **Testability**:
   * You can test FrameProcessor, GStreamerSource, and GStreamerSink independently.
3. **Reusability**:
   * Classes like FrameProcessor can be reused across different pipelines.
4. **Easier Debugging**:
   * If something goes wrong, it's easier to locate the issue in a specific class.

**When to Use a Single Class**

If the pipeline is **very simple** and you are confident that the requirements will not grow, using a single class might be acceptable. In this case, ensure:

* The class is well-organized with clear methods (e.g., initPipeline, processFrame, handleOutput).
* Comments and documentation explain the different sections of the code.

**Conclusion**

For a scalable and maintainable design, it's better to use **multiple classes**:

* **PipelineManager** for orchestration.
* Specific classes for **GStreamer sources, sinks, and OpenCV processing**.

This modular approach ensures your pipeline is robust, extensible, and easier to debug or adapt in the future. Using a single class might be suitable for simpler prototypes but is not recommended for production-quality code. Let me know if you'd like a more detailed implementation!