Concrete Architecture of OpenCV

Authors

Gabriel Rincon - 218624734- gabrielenrrincon@gmail.com
Arash Saffari - 218791632 - arashrt@my.yorku.ca
Nargis Rafie - 220785903 - nargis.rafie00@gmail.com
Joshua Zuker - 218135574 - joshz@my.yorku.ca
Leonard Schickedanz - 222634661 - leon.schickedanz@gmail.com
Shaheer Lone - 217277807 - shaheerlone@gmail.com

Abstract

This report demonstrates a concrete architecture analysis of OpenCV (Open Source Computer Vision Library), building upon the previously developed Conceptual architecture that was developed and analysed through the use of online articles, documentation, and manual source code analysis. The primary objective is to validate our theoretical architecture patterns that were identified when studying conceptual architecture. This includes the repository, layered, and pipe-and-filter styles. Through the use of powerful software architecture analysis tools, such as Scitools Understand and LSEdit (Landscape Editor), the level of detail was increased significantly.

Scitools Understand was used for static code analysis and fact extraction, which generated dependency data from OpenCV's source code. This raw data was then transformed into structured formats and merged with manually created containment files, which set logical groupings of files into understandable and accurate modules and subsystems. LSEdit was used to visualize this data through graphs that consist of entities and nodes that showed the different interactions between subsystems. This approach allowed us to thoroughly examine OpenCV's design at a deeper level and made it easier to compare the data with our conceptual theory.

The concrete architecture examined and extrapolated ties closely with the conceptual architecture originally discovered and analysed. It confirmed that OpenCV does in fact use the repository, layered, and pipe-and-filter styles. The Core Manager acts as the central repository that almost all of the other subsystems depend on for various tasks such as storage, data structures, algorithms, etc. The layered architecture shows itself in the way higher level modules depend on lower level ones for operations, with the HAL (Hardware Abstraction Layer) acting as the base for optimization. The pipe-and-filter pattern shows itself in different modules where data flows through different stages and has many operations performed within processing pipelines.

Detailed use cases, including face detection and automated architectural feature extraction, are used to show how these different patterns present themselves in real life applications. The dependency diagrams generated from LSEdit provide evidence of different subsystem interactions that were found in the conceptual analysis. This analysis demonstrates architecture derivation through the use of software tools, while still requiring lots of manual effort in different

aspects. Overall, this report provides a solid validation of the theoretical conceptual architecture of OpenCV and goes into deeper, more accurate details of its nature.

Introduction and Overview

The conceptual architecture of OpenCV previously studied was extrapolated primarily through the use of manual analysis of source code, online articles, and documentation. The conceptual architecture focused on the modules, structure, and architectural styles that best fit OpenCV, such as the repository, layered, and pipe-and-filter styles. In this report, we dive deeper into the system structure and code using tools made for exactly this kind of task, and aim to turn our theoretical analysis into a solid, verifiable architecture style and design. Transitioning from conceptual to concrete architecture is a crucial part of the process of validating the initial observations made and seeing how they compare with OpenCVs actual design principles and implementation.

Specialized tools for software architecture extraction and analysis were employed to complete this task. These tools were Scitools Understand and LSEdit. Scitools understand is a static analysis tool that allows us to extract file-level dependencies, understand source code on a deeper level, and view and document the codebase with precision. The LSEdit software was used in tandem with the analysis provided by Understand, generating graph visualizations and displaying entities and edges that represent the subsystem interactions and dependencies. Through the use of these tools, we made many comparisons between the generated concrete architecture and our original conceptual architecture, with a focus on a specific subsystem - Calib3d, to demonstrate dependency patterns. Combining automated fact extraction, containment file definition, and interactive visual diagrams allowed us to determine the validity of our original conceptual diagrams with higher certainty. As a result, we got a much better understanding of OpenCVs modular structure, dependencies, and its architectural patterns.

Architecture

The derivation process of the concrete architecture involves systematically extracting, organizing, and visualizing the actual structural elements of OpenCV based on its source code. It begins with **fact extraction**, where the system's codebase is analyzed using *SciTools Understand* to identify file-level dependencies and relationships between components. This extracted data is exported as a CSV file, which represents the raw, tool-generated understanding of OpenCV's structure. Next, this data is transformed into a structured format (.raw.ta file) using scripts such as transformUnderstand.pl, converting the raw dependency information into machine-readable form suitable for architectural analysis.

The next stage is **containment definition**, where the engineer manually defines logical groupings of files into meaningful subsystems within a containment file (CustomizedFileDependencies.contain). This step is critical because it reflects the human understanding of the system's design intent, mapping low-level source code files into higher-level architectural units like subsystems, modules, or layers. After defining these relationships, the containment file is merged with the extracted data using scripts in createContainment.sh providing a consolidated representation of the system's structure (the .con.ta and .ls.ta file).

Finally, this derived data is loaded into a visualization tool (LSEdit) which allows the engineer to examine the architecture graphically. Through this process, the abstracted relationships between subsystems, classes, and modules become visible, forming the concrete architecture. The derived architecture can then be analyzed to identify architectural styles, design patterns, and dependency structures, providing valuable insights into maintainability, modularity and system evolution.

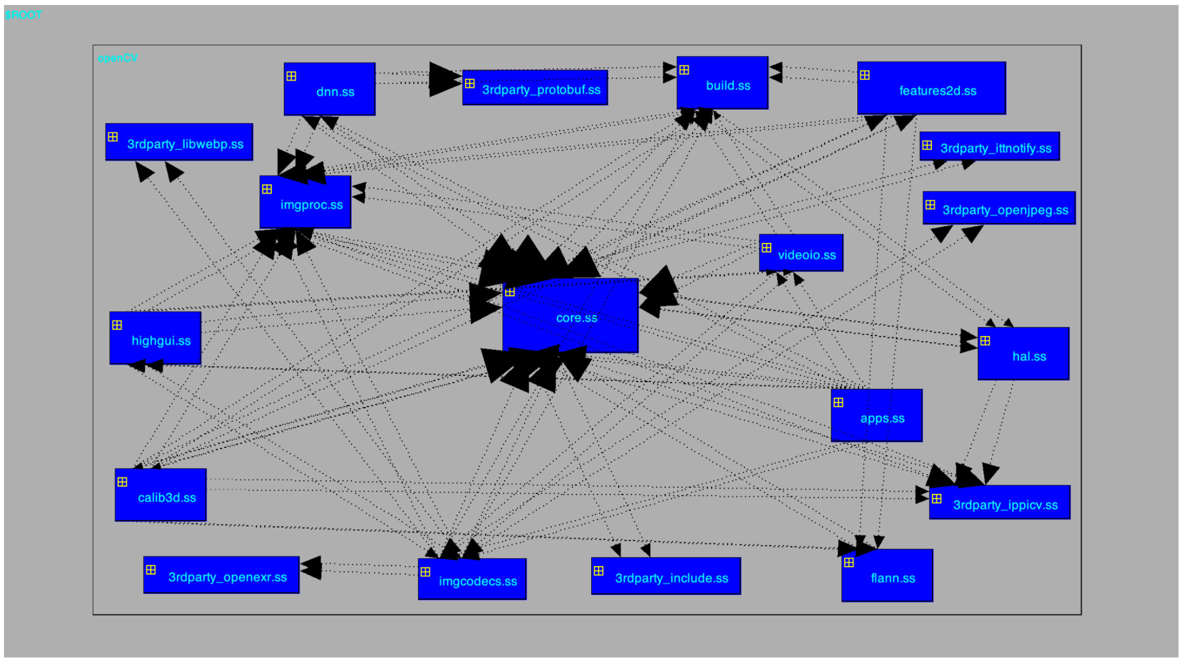

Figure 1 - OpenCV dependencies

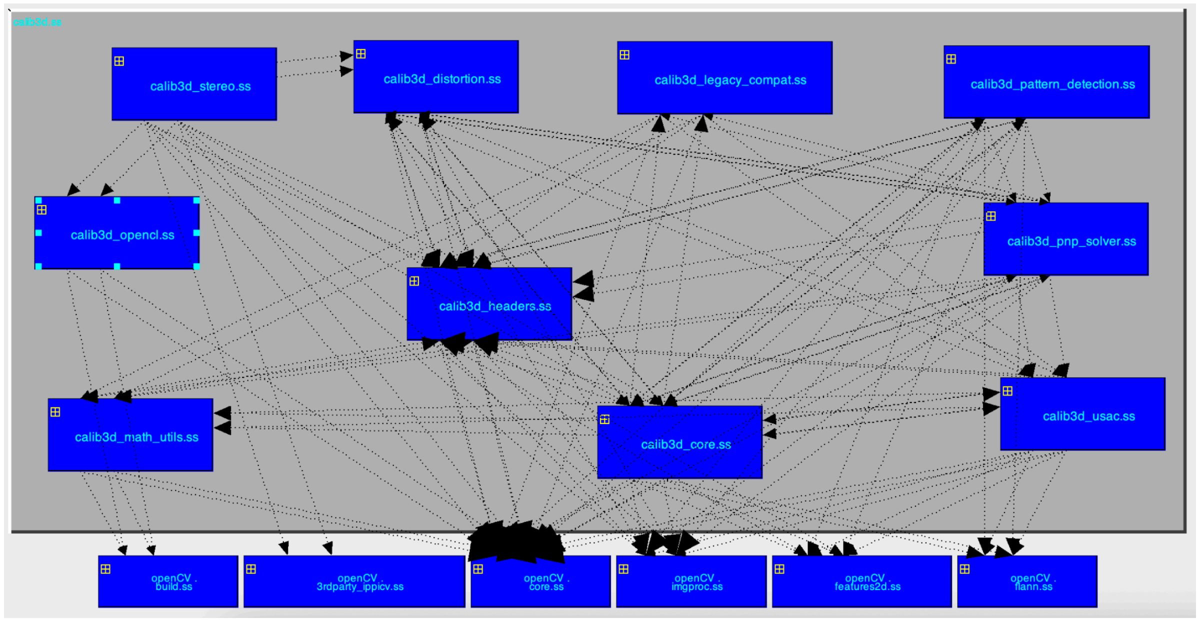

Figure 2 - calib3d dependencies

Diagrams

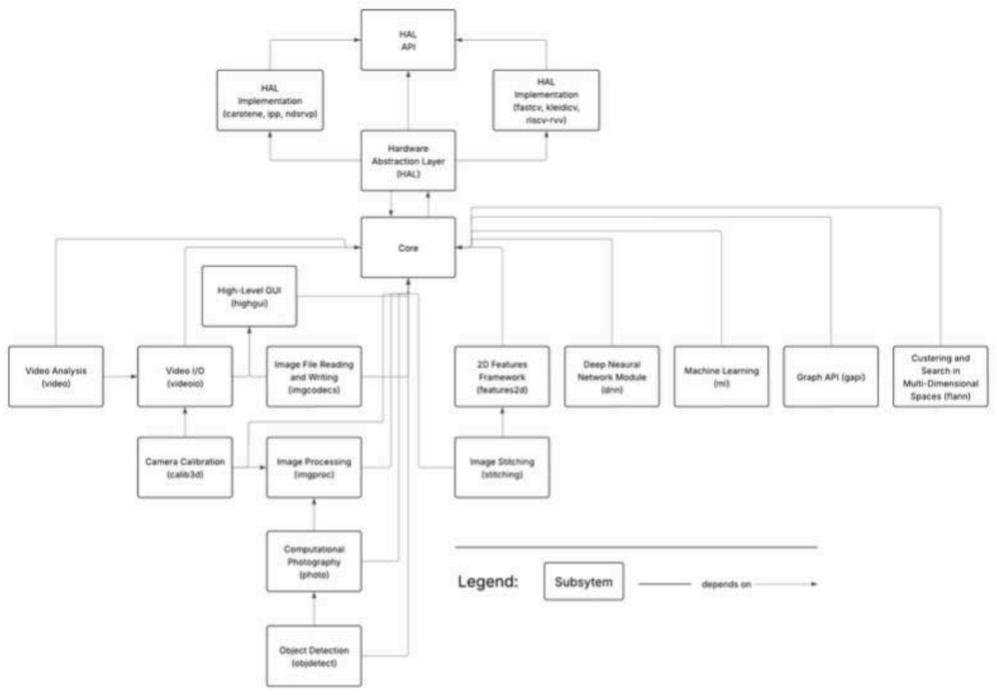
HAL API
HAL Implementation (carotene, ipp, ndsrvp)
HAL Implementation (fastcv, kleidicv, riscv-rvv)
Hardware Abstraction Layer (HAL)
Core
High-Level GUI (highgui)
Video Analysis (video)
Video I/O (videoio)
Image File Reading and Writing (imgcodecs)
2D Features Framework (features2d)
Deep Neaural Network Module (dnn)
Machine Learning (ml)
Graph API (gapi)
Custering and Search in Multi-Dimensional Spaces (flann)
Camera Calibration (calib3d)
Image Processing (imgproc)
Image Stitching (stitching)
Computational Photography (photo)
Object Detection (objdetect)
Legend:
Subsytem
depends on

Figure 3 - Conceptual architecture

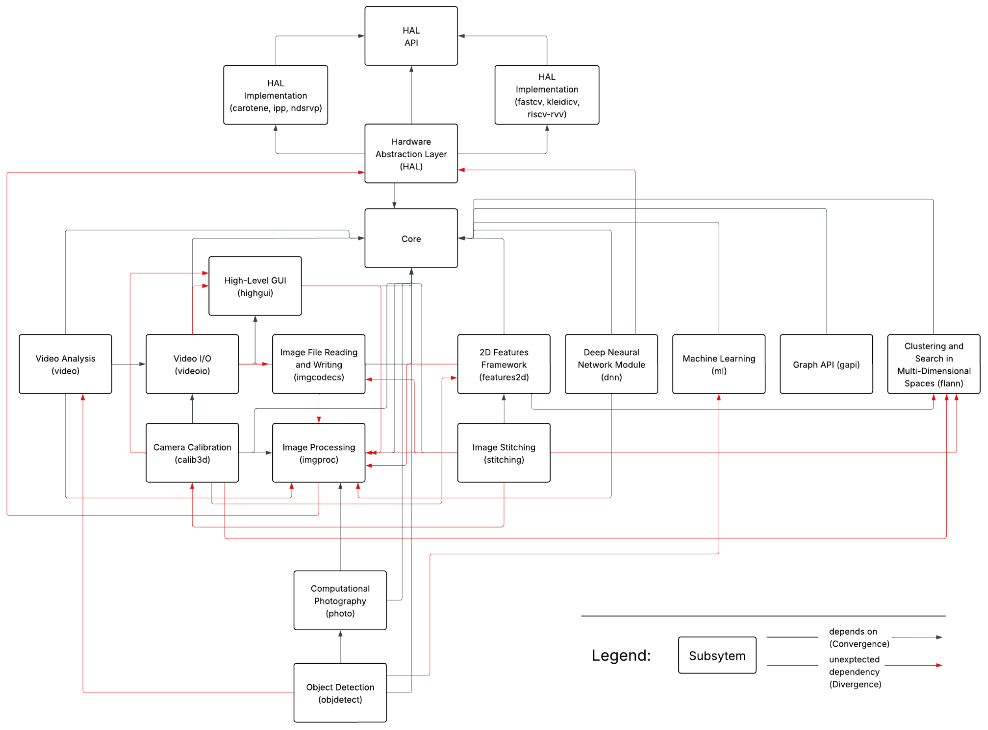
HAL API
HAL Implementation (carotene, ipp, ndsrvp)
HAL Implementation (fastcv, kleidicv, riscv-rvv)
Hardware Abstraction Layer (HAL)
Core
High-Level GUI (highgui)
Video Analysis (video)
Video I/O (videoio)
Image File Reading and Writing (imgcodecs)
2D Features Framework (features2d)
Deep Neaural Network Module (dnn)
Machine Learning (ml)
Graph API (gapi)
Clustering and Search in Multi-Dimensional Spaces (flann)
Camera Calibration (calib3d)
Image Processing (imgproc)
Image Stitching (stitching)
Computational Photography (photo)
Object Detection (objdetect)
Legend:
Subsytem
depends on (Convergence)
unexpected dependency (Divergence)

Figure 4 - Concrete architecture

Architecture Styles and Design Patterns

As discussed in the conceptual architecture, we see a very clear repository structure with core.ss acting as the center of the repository with all other modules directly or indirectly relying on it in figure 1. In fact, this reliance is high enough that it is hard to discern a pipe and filter train longer than 3 due to the chains ultimately leading back into core.ss. We do see some smaller chains however indicating that there is some pipe and filter architecture in this system such as the chain core.ss → dnn.ss → 3rdparty_protobuf.ss → build.ss. The layered architecture style is also visible however as opposed to being present top to bottom, we see it running left to right. Higher level subsystems like highgui.ss are seen on the left, core.ss being in the middle and hal.ss (hardware abstraction layer) on the right. The other subsystems are similarly positioned with dnn.ss being further left and build.ss being further on the right.
Design Patterns are harder to identify from a generated concrete architecture due to the abstraction from implementation. However with some analysis, we can see evidence of some in use. For example, 3rdpart_protobuff.ss is a type of adapter, likely taking the results produced by the deep neural network and packaging it into a useful format for build.ss. Similarly, due to the strong evidence of the repository architecture style, we can also identify the mediator design pattern with core.ss mediating the function of all other subsystems.

Subsystems and Interactions

As seen in figures 1 and 5, the concrete architecture derived from the use of Understand gave a better understanding of the subsystems and how they interact with each other. The dependency chart showed nineteen main subsystems as well as eight third party subsystems that are important to the functions of OpenCV. The substems interactions are fairly linear, solidifying the Pipe and Filter architecture. Many of the subsystems can be grouped together by interactions, such as Video based subsystems, AI and Machine Learning, the Hardware Abstraction Layer and the Core.

- Core
    - The OpenCV Core module serves as the fundamental backbone of the entire OpenCV library. It provides the essential building blocks and core functionalities upon which all other modules rely.
    - Interacts with all of the main modules in order to perform its functions. The cv::Mat is the core data structure, representing a dense multi-dimensional array designed for efficient memory management and mathematical operations.
- Video
    - All the video subsystems can be grouped together as they all mainly interact between each other as well as the core. They consist of the High Level GUI, Video I/O, Video Analysis, Camera Calibration, Image Processing, Computational Photography, and Object Detection.
    - OpenCV's video functionality is primarily built around two core classes: cv2.VideoCapture and cv2.VideoWriter. The cv2.VideoCapture class is responsible for reading video streams, either from various file formats like AVI

and MP4 or from live camera feeds, where the default webcam is accessed with the integer 0. It relies on backend libraries like FFmpeg for broad codec support and uses its read() method for essential frame-by-frame processing. Conversely, the cv2.VideoWriter class handles writing video, allowing processed frames to be saved to a file by specifying parameters such as the FourCC codec, frames per second, and frame size, and then using the write() method to compile the individual frames into a new video.

- AI and Machine Learning
    - These subsystems provide the algorithms and deep neural networks required for computer vision tasks, again being highly connected to each other. They consist of the 2D Features Framework, Deep Neural Network Module, Machine Learning Module, Graph API, Cuttering and Search in Multi- Dimensional Spaces, and Image Stitching.
    - The interactions between the modules begin in the core module with data handling, where image or video data is initially read and manipulated. It then moves to preprocessing, often using core functions alongside specialized DNN utilities like cv::dnn::blobFromImage(), to format the data into a 4D blob suitable for model input. Model execution is handled by either the ML module for classical algorithms or the DNN module for deep learning inference. Finally, post-processing relies again on the core module to convert the model's raw output into standard OpenCV structures for visualization, such as drawing bounding boxes.
- Hardware Abstraction Layer
    - Consisting of the HAL API, HAL Main Module, and the HAL Implementation, HAL is a crucial component designed to optimize performance by enabling the library to leverage hardware-specific optimizations and accelerate certain image processing and computer vision functions.
    - The Hardware Acceleration Layer (HAL) in OpenCV is an internal, private interface designed for use within the library itself, meaning general users do not interact with it directly. Instead, when a user calls a standard function like cv::resize(), the library employs a layered design. It first checks if an accelerated implementation for that specific operation exists within the HAL. If a supported HAL function is available, it is executed for optimal performance. If not, OpenCV seamlessly falls back to its own default, generic C++ implementation, ensuring the function always completes successfully.

Camera Calibration (calib3d)

A closer examination of the calib3d module was carried out in order to obtain a deeper comprehension of how OpenCV's modules function. Seen in figure 2, the calib3d module is fundamentally dependent on OpenCV's core module, which acts as the foundational library providing essential data structures and operations. This reliance is evident as calib3d extensively uses the core's cv::Mat class for passing data like images, camera matrices, and 3D point sets. Furthermore, it leverages the core's basic matrix operations for complex linear algebra, such as geometric transformations and camera parameter calculations. The module also builds upon core-defined data types like cv::Point and cv::TermCriteria, and utilizes the core's underlying mechanisms for error handling, making it an integral component for all 3D reconstruction and camera calibration tasks. This solidifies the repository architecture of OpenCV overall.

Furthermore, with all the linear transformations calib3d performs on images and videos, this demonstrates that the modules internal architecture follows a pipe and filter design, as its interactions form a pipeline that transforms raw image data into structured 3D information. Since the photos and videos may be of different formats, calib3d must also change its code depending on the type of video or photo it revives, thus another architecture design it follows is Adaptor.

Comparison of Conceptual and Concrete Architecture

Throughout the analysis of the conceptual architecture, it has been concluded that OpenCV makes use of three different architectural patterns in order to develop a successful program. These three styles are pipe and filter, repository, and layered architecture. These conclusions were came to due to the organization of the various modules that have been developed. They will be brought up against the concrete architecture that has been uncovered with the usage of Understand. In addition, these new dependencies that this program has displayed will allow for further comparison.

The first architectural style that has been determined was the pipe and filter style. When it comes to this style, it has been observed that inputs are taken, transformed, and used throughout each of the modules. After observing the concrete architecture, it appears that this behaviour persists, but not to the extent originally thought. Though pipe and filter chains can be spotted, they are not as prevalent. The longest chain identified was only the length of 3 modules according to figure one, which was from dnn.ss to build.ss. This reveals that this style, though still present, does not dominate the architecture of OpenCV.

Regarding the repository style, it was clear that the core module was the host for many important functions and tools such as mat. Within every other module of the program, it would appear that they would have a connection to the core module in some fashion, by making use of the tools it offers. By viewing the concrete architecture, the dependencies show that this deduction is correct. Many of the modules have a dependency with the core module as seen in the diagram, reaffirming that this program makes use of the repository architectural style.

The last style covered was the layered architectural style. In the conceptual architecture phase, it appeared as though the architecture was split into three layers, the modules, the core layer, and the hardware abstraction layer. From the concrete architecture that has been formed, it appears that this style still holds up. Each of the higher level modules are dependent on the core module which is a lower level module. It can be seen that the core module also makes use of the hardware abstraction layer, which is the lowest level layer in the architecture. The difference this has from the conceptual model is that the modules from the higher level to the lower level are organized from left to right as opposed to top to bottom.

By comparing the styles mentioned in the conceptual architecture with the concrete architecture, it is clear that many of the deductions made during that phase still hold true. There is still a heavy emphasis on module dependencies to the core module, and the core module still makes use of attributes of the hardware abstraction layer. However, some aspects of the conceptual architecture such as the pipe and filter style were shown to be more uncommon than initially thought.

Use Cases

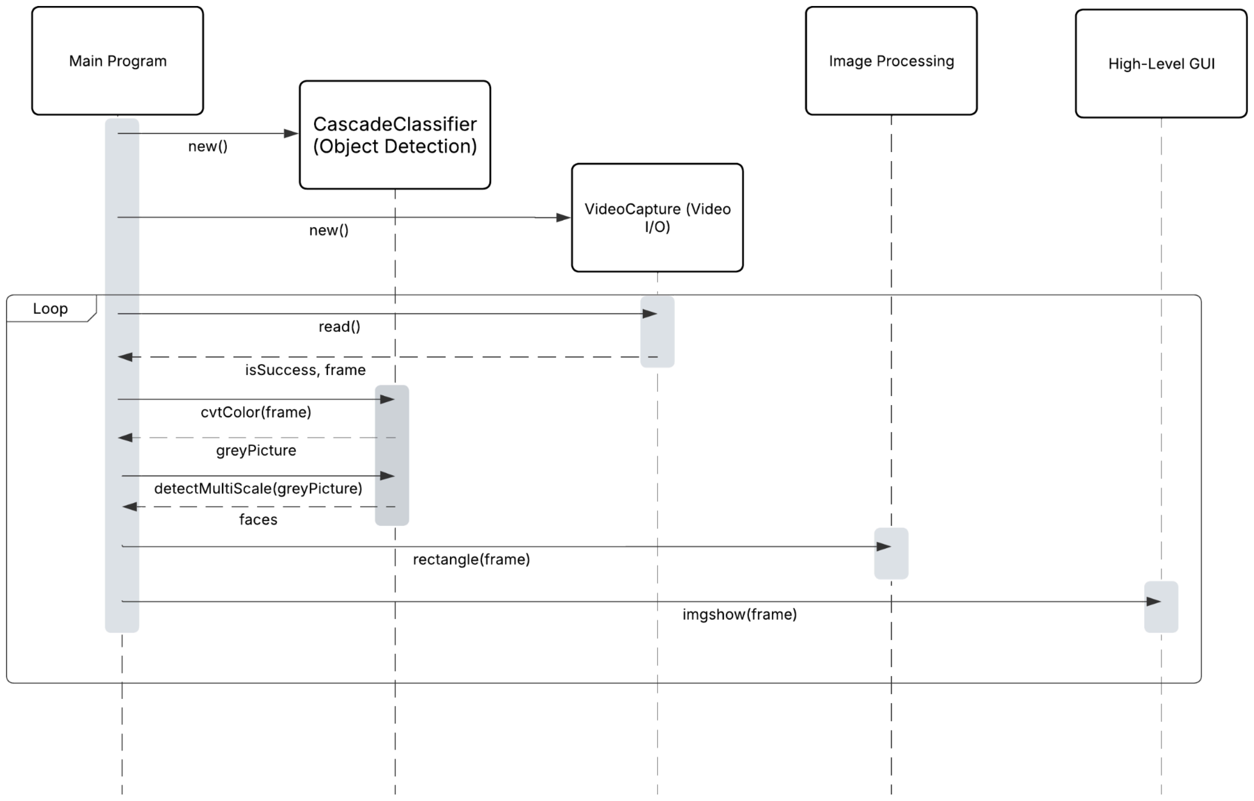

Figure 5 - Face Detection Workflow

The workflow, illustrated in Figure y, details the dynamic sequence of interactions between the main components of a real-time face detection system built with OpenCV. The process is centrally controlled by the Main Program which orchestrates the entire operation.

The setup phase begins with the Main Program creating two essential objects: it first instantiates the CascadeClassifier class from the module Object Detection and subsequently the VideoCapture class from the module Video I/O to connect with the live camera feed. This read-out object is then ready for the continuous processing phase, the Loop.

Within this Loop, the Main Program initiates the video stream by calling the read() method on the VideoCapture object, which returns the success status and the current frame. Upon successful capture, the processing chain starts. The frame is immediately passed to the CascadeClassifier, which internally executes the cvtColor(frame) function to convert the image data into a grayscale format (greyPicture). This preparation is necessary for efficient algorithmic processing. The Main Program then uses this greyPicture to call the key detection method, detectMultiScale(greyPicture), on the CascadeClassifier object, which returns a list of coordinates for the recognized faces.

Following the detection, the focus shifts to visualization. The faces coordinates are sent to the Image Processing module's rectangle(frame) method, where bounding boxes are drawn onto the original color frame. Finally, to complete the loop, the Main Program sends the marked frame to the High-Level GUI by calling imshow(frame), which renders the final visual output to the user. This sequence of reading, processing, detecting, marking, and displaying repeats continuously to maintain the real-time nature of the application.

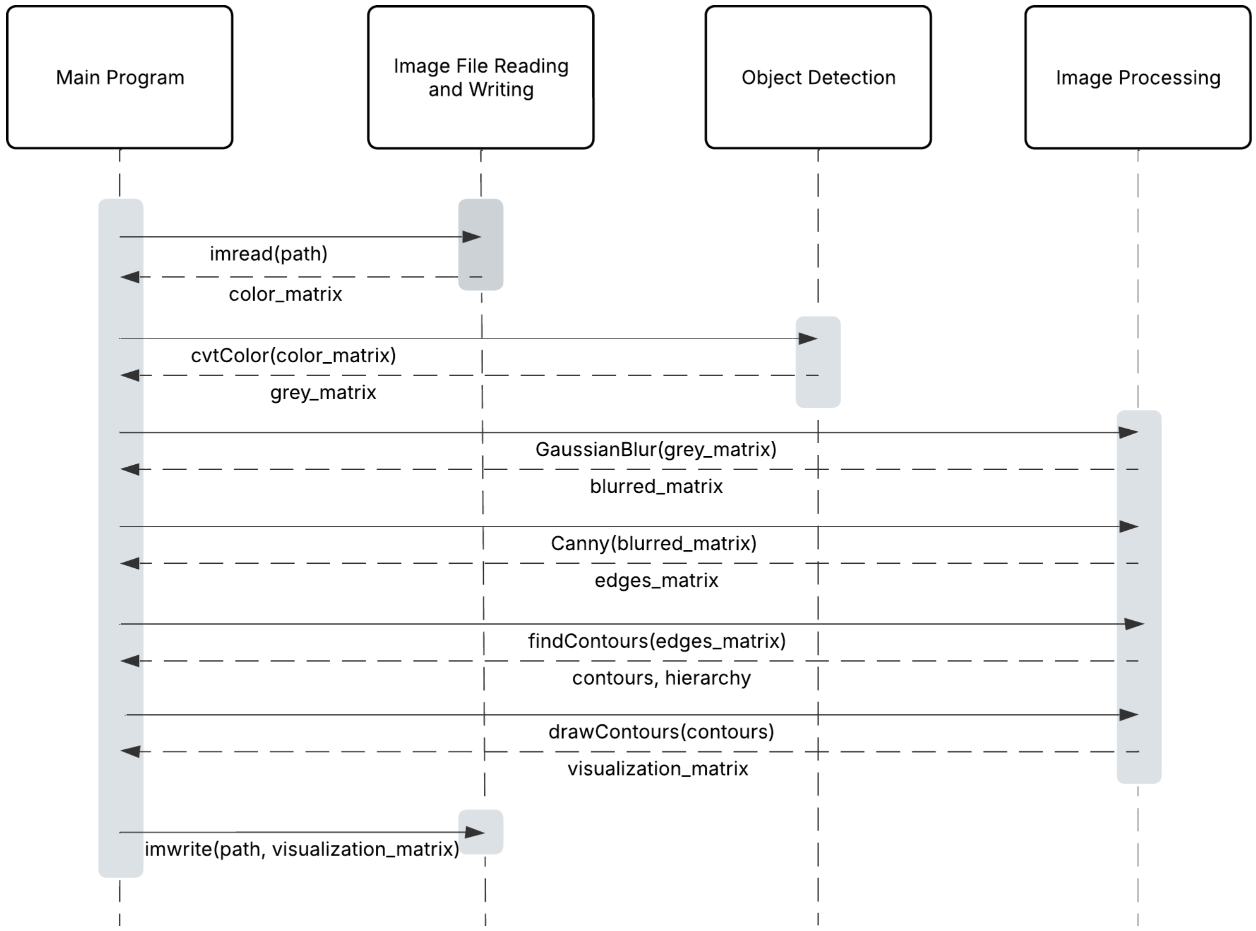

Figure 6 - Automated Architectural Feature Extraction via Contours

The use case shown in Figure outlines the automated process of transforming a raw architectural image, such as a scanned blueprint or a drone photograph of a building, into a structured data format suitable for Computer-Aided Design (CAD) software. The core of the solution relies on the sequential execution of various functions within the OpenCV library, demonstrating a clear inter-module collaboration.

The process is initiated by the Client Application which first uses the Image File Reading and Writing module, calling the imread() function to load the image file from the disk into memory, utilizing the central data structure: the Image Matrix (cv::Mat).

Once loaded, the control flow immediately shifts to the Image Processing module to begin the image processing pipeline. This pipeline involves several sequential function calls:

1. Preprocessing: The image matrix is converted to grayscale (cvtColor) and then subjected to a smoothing filter (GaussianBlur). This step is crucial as it reduces noise (smoothed_matrix) while preserving key structural information, making subsequent edge detection more reliable.
2. Feature Detection: The smoothed matrix is passed to the core detection function, Canny(). The Canny algorithm identifies sharp intensity changes, transforming the image into a binary edges_matrix, where only the outlines of the architectural elements remain.
3. Contour Extraction: The final processing step within the Image Processing module involves calling findContours(). This function analyzes the edges_matrix to mathematically group the edge pixels into meaningful geometric shapes, returning a list of coordinate points (contours) and their spatial relationships (hierarchy).

Finally, the Client Application gains back control to process the extracted data. It typically iterates over the contours list to filter, simplify, or approximate the shapes (e.g., converting pixel boundaries into lines and arcs). For visualization, the drawContours() function can be used. The processed or visualized data is then written back to the disk using the Image File Reading and Writing module's imwrite() function, often exporting the contours into a format like DXF for CAD import. This entire sequence clearly illustrates how data flows between specialized OpenCV functions, where the output of one method serves as the essential input for the next, forming a powerful data transformation chain.

Data Dictionary

There are no terms needed to be defined that haven’t been defined in the report. This section is N/A.

Naming Conventions

There are no abbreviations or naming conventions used in this report that require further clarification. This section is N/A.

Conclusions

After analyzing OpenCV’s architecture on a deeper level, our team felt confident with our choices of architecture style and conceptual mockups originally created. Using SciTools’ Understand helped us get a stronger grasp of OpenCV’s code and organization. Our research has confirmed that our selections of the repository, layered, and pipe-and-filter architecture styles are well suited for this system. Each style plays a role in maintaining performance, reusability, and modularity.

The Core Manager acted as the central repository containing data structures, algorithms, functions, and more that nearly every other subsystem depends on. Layering was also evident, as higher level modules depend on lower level ones for different operations and hardware abstraction. This ties it to the pipe-and-filter pattern, as Processing, DNN, and Feature and Object managers form pipelines to analyze and transform data. Tools such as LSEdit, used for architectural visualization, helped provide clearer understandings of these different relations.

To conclude, the concrete architecture of OpenCV accurately reflects our discoveries made in the conceptual architecture analysis. There is a wide range of functionality supported through well defined modular interfaces, a strong emphasis on performance optimization, and an overall design that allows for scalable transformations that encourage its evolution. Using different architecture analysis tools and techniques that provide deep, code level insight proved to be crucial for revealing these details, providing a stronger understanding of OpenCV's structure.

Lessons Learned

During this assignment, our team was faced with many challenges throughout the analysis process. One early example was creating the containment file, where defining the subsystems and mapping files required lots of accuracy and a solid understanding of OpenCV's structure. Similarly, using LSEdit was difficult at first, as trying to achieve the desired layout required multiple trials and errors to be fixed. Eventually, we learned to use the tool effectively to produce the diagrams we wanted that reflected our understanding of the conceptual architecture. This experience helped us understand how technical of a process it can be when analyzing architecture, even when tools such as Understand or LSEdit do lots of work for you. We realized there is always a significant amount of manual effort involved that requires a high attention to detail, and persisting throughout is essential for maintaining accuracy and producing strong results.

References

[1] opencv, "opencv/opencv," *GitHub*, Dec. 12, 2019. https://github.com/opencv/opencv
[2] "EECS 4314 Lab Notes Part I: Source Code Download and Environment Configuration." Accessed: Oct. 31, 2025. [Online]. Available: http://www.cse.yorku.ca/~zmjiang/teaching/eecs4314/slides/EECS4314ArchRecoveryLabNotes.pdf
[3] "A Guide to Generating LSEdit Landscapes." Accessed: Oct. 31, 2025. [Online]. Available: https://www.swag.uwaterloo.ca/lsedit/landscapes.html