OBJECT-ORIENTED PROGRAMMING II
2024-2025 SPRING SEMESTERS
PROJECT

Project Name:
ADVANCED IMAGE PROCESSING INTERFACE

Interface Developer:
Doğukan AVCI

NO: 151220202051

INSTRUCTOR:
Dr. Burak KALECİ

SUBMISSION DATE:
May 14, 2025

1. Introduction

This project, titled *Advanced Image Processing Interface*, was developed as part of the final laboratory assignment for the Object-Oriented Programming II course in the Spring 2024–2025 semester. The main purpose of this project is to design a modular, interactive, and object-oriented graphical user interface (GUI) for performing common image processing operations. It allows users to perform real-time grayscale and HSV color conversion, apply advanced segmentation techniques, and detect image edges using various algorithms — all through a responsive and intuitive GUI built with PyQt5.

The application architecture heavily utilizes the principles of object-oriented design, including **abstraction**, **inheritance**, **polymorphism**, **encapsulation**, and **data hiding**. These principles were applied to ensure code modularity, maintainability, and future extensibility. Every image processing functionality (e.g., RGB to Grayscale conversion or Sobel edge detection) is encapsulated in its own class and invoked through polymorphic handler interfaces.

The backend of the system is built upon widely used scientific Python libraries including **scikit-image** for image transformations, **NumPy** for numerical operations, and **PyQt5** for GUI development. The interface was designed using Qt Designer and then integrated into Python code using the **pyuic5** conversion tool. Furthermore, all operations are handled asynchronously using multithreading (**QThread**) to ensure a smooth user experience and prevent the GUI from freezing during intensive image processing tasks.

In this report, the architectural decisions, implemented object-oriented structures, and functional capabilities of the system will be detailed, along with an evaluation of how the application fulfills the assignment criteria.

2. System Design and Architecture

The *Advanced Image Processing Interface* was designed as a modular, maintainable, and extensible application using strong object-oriented design principles. The architecture follows a **layered modular structure**, in which each functionality (conversion, segmentation, edge detection, I/O, state control) is implemented as a separate module and interacts with the GUI controller (main.py) through abstract interfaces.

The diagram below illustrates the **class hierarchy and module relationships** in the form of a UML Class Diagram.

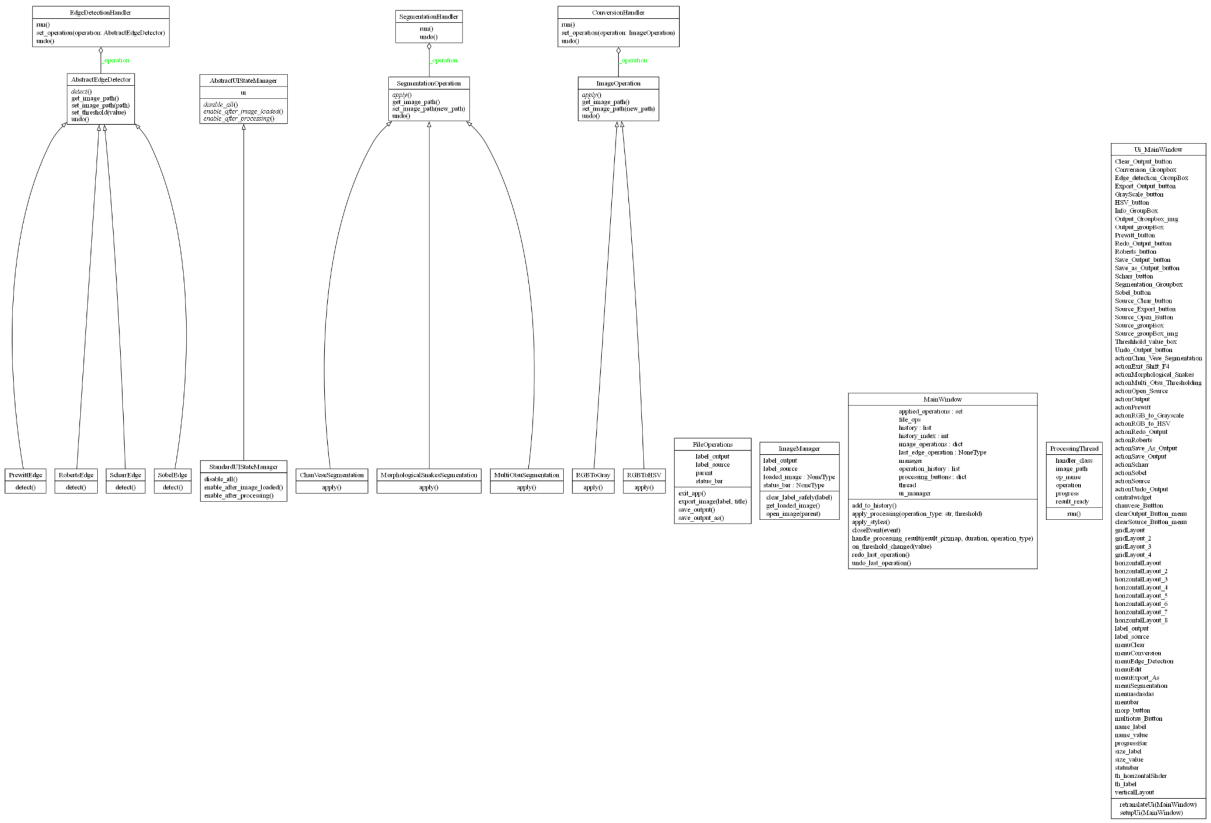

Figure 1. *UML Class Diagram of the Overall System Architecture*

Architectural Layers & Key Modules

UI Layer: Designed with Qt Designer (**LabFinal.ui**) and converted via **pyuic5**, then managed by **main.py**.

Conversion Module (conversion.py):

Contains **ImageOperation** (abstract class) and concrete implementations **RGBToGray** and **RGBToHSV**. These use **skimage.color** for pixel transformation and **numpy** for array manipulation.

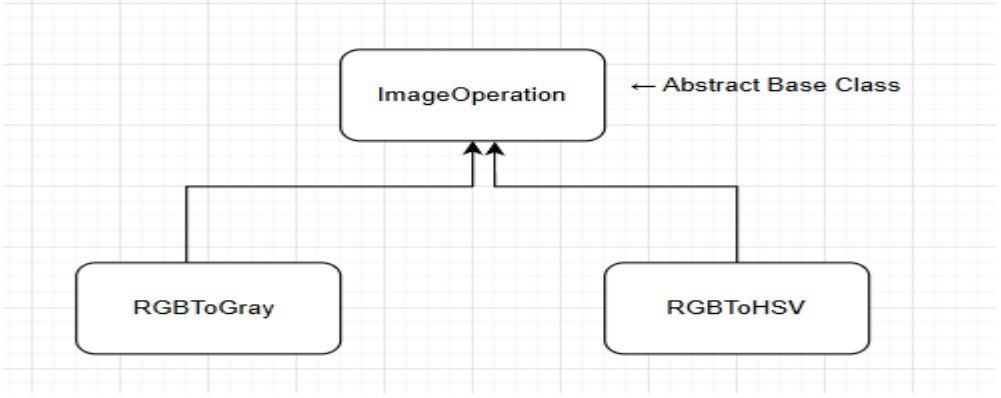

Figure 2. *Inheritance structure of the conversion.py module*

This inheritance structure demonstrates a clear application of **abstraction**, **inheritance**, and **polymorphism**. The abstract class ***ImageOperation*** defines a generic interface through the **apply()** method, which must be implemented by all derived classes. This ensures that every image conversion operation adheres to a unified contract.

Both ***RGBToGray*** and ***RGBToHSV*** override the **apply()** method to execute their specific color transformations. Despite their differing internal logic, they can be used interchangeably through the same interface, thanks to **polymorphism**. This is particularly useful in the main controller (main.py), where different operation classes are passed dynamically to a **ConversionHandler** object, allowing for flexible and extensible processing logic without hardcoded dependencies.

Segmentation Module (segmentation.py):

Defines an abstract class **SegmentationOperation** and concrete implementations like **MultiOtsuSegmentation**, **ChanVeseSegmentation**, and **MorphologicalSnakesSegmentation**. Internally uses **NumPy** operations and **skimage.segmentation**.

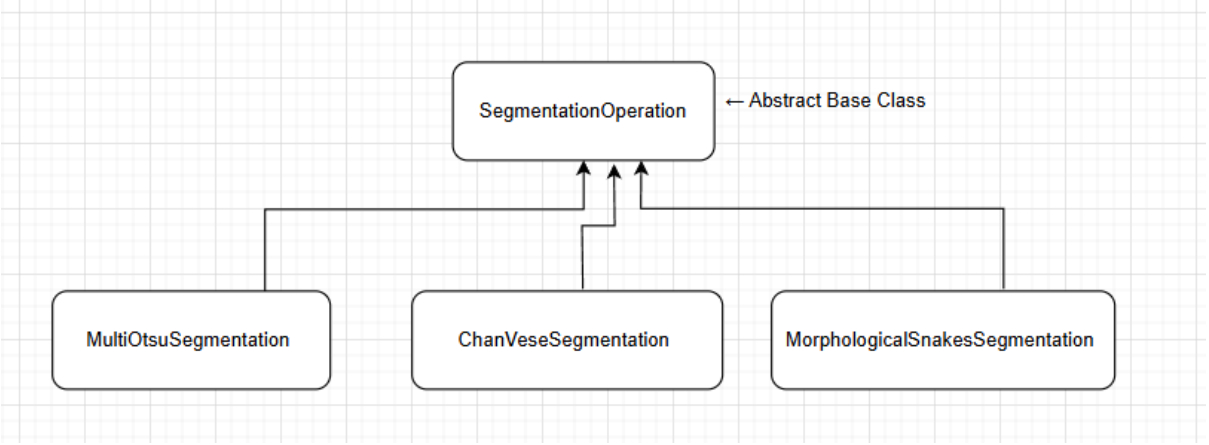

Figure 3. *Inheritance and polymorphic segmentation structure in segmentation.py*

The segmentation module is a clear and effective implementation of both **inheritance** and **polymorphism**. The abstract base class ***SegmentationOperation*** defines a generic **apply()** method that is **overridden** in each subclass to execute the respective segmentation algorithm. The **subclasses** — ***MultiOtsuSegmentation***, ***ChanVeseSegmentation***, and ***MorphologicalSnakesSegmentation*** — all provide unique implementations while sharing a consistent interface.

This design enables **runtime polymorphism**: in the main controller (main.py), a segmentation operation is instantiated dynamically (e.g.,

ChanVeseSegmentation(image_path)) and passed to the SegmentationHandler, which is unaware of the specific subclass type. The handler simply calls operation.apply(), relying on the correct overridden version to execute. This allows the application to easily support new segmentation algorithms without changing the main control logic — a core advantage of **polymorphic** design.

Additionally, shared functionalities such as undo support and QPixmap conversion are encapsulated in the base class, promoting **code reuse** and **consistency**. Internally, each class processes the image using **NumPy** for array manipulation and segmentation functions from **skimage.segmentation** such as chan_vese and morphological_chan_vese. The segmented result is returned as a QPixmap and rendered on the GUI in real time.

Edge Detection Module (edge_detection.py):

Inherits from ***AbstractEdgeDetector*** and supports ***SobelEdge***, ***RobertsEdge***, ***ScharrEdge***, and ***PrewittEdge***, with **thresholding** logic applied via **NumPy** array filtering.

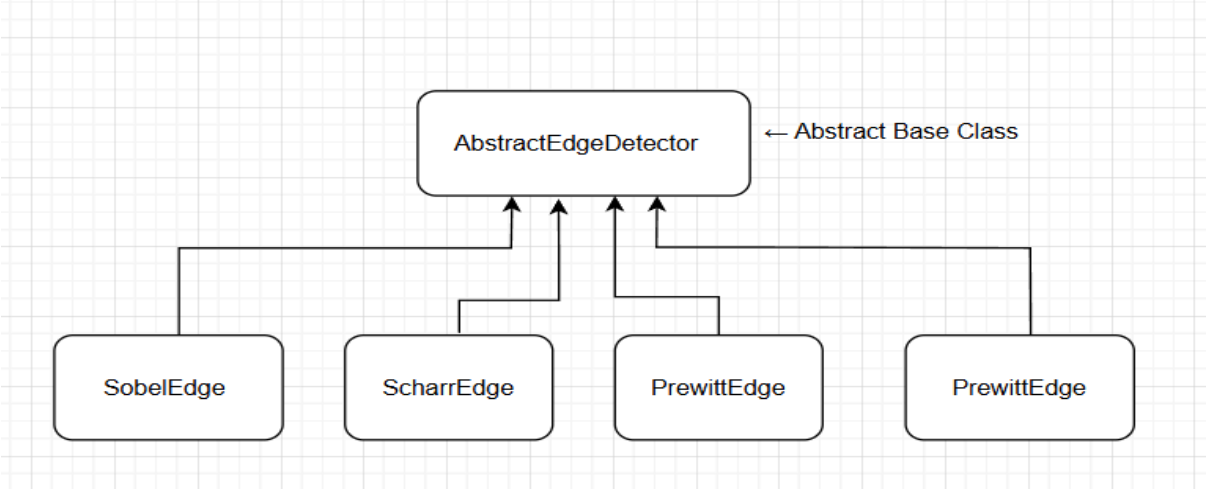

Figure 4. *Object-oriented edge detection hierarchy with threshold support*

The edge detection module follows a well-structured object-oriented hierarchy where ***AbstractEdgeDetector*** serves as the **abstract base class**. This base class defines a common interface — typically the detect() method — that must be overridden by all subclasses. The concrete implementations (***SobelEdge***, ***ScharrEdge***, ***PrewittEdge***, and ***RobertsEdge***) each apply a different edge detection algorithm from the **scikit-image.filters** module.

This design allows the main application to instantiate and invoke edge detectors **polymorphically**. In the main.py controller, the user's selected edge detection method (e.g., "sobel" or "roberts") is mapped to the corresponding class and passed

into a common ***EdgeDetectionHandler***, which simply calls detect() on the object — without needing to know its specific type. This is a textbook example of runtime polymorphism, ensuring flexibility and code modularity.

Each class uses NumPy arrays internally for image thresholding and filtering. Additionally, a threshold parameter is dynamically injected from the GUI's slider widget and used in real time. The final output is a QPixmap object that is displayed in the GUI via QLabel, allowing the user to see the edge-detected result with adjustable thresholding.

Overall, this structure showcases a high level of object-oriented design, combining abstraction, inheritance, and polymorphism to support multiple edge detection methods with minimal code duplication.

Threading (processing_worker.py):

Defines **ProcessingThread**, a subclass of **QThread**, which allows time-consuming operations to run in parallel without blocking the UI.

History & Undo/Redo:

Uses a list as a **stack** (operation_history[]) to track operation history with **tuples** of (operation_type, image_path, threshold).

Applied operations per image are tracked in a dict (image_operations[filename] = [...]), and duplicated operations are prevented with a **set** (**applied_operations**).

UI State Management (ui_state_manager.py):

Uses **inheritance**: ***StandardUIStateManager*** extends ***AbstractUIStateManager***, enabling or disabling interface components based on application state.

Exception Handling:

- *ProcessingThread.run()* catches and reports errors using **try-except** blocks with ***QMessageBox***.
- I/O errors (image save/load) in **file_operations.py** are wrapped in error-handling constructs to inform the user of issues.

Built-in Data Structures & Libraries Used

Type	Usage
list	Operation history for undo/redo (operation_history[])
dict	Tracks applied operations per image (image_operations[filename])
set	Prevents redundant processing (applied_operations)
tuple	Stores operation details as immutable entries in history list: (operation_type, image_path, threshold)
numpy.ndarray	All pixel-level image data is processed using NumPy arrays (e.g., thresholding, segmentation masks
skimage	Core image transformations (rgb2gray, chan_vese, sobel, etc.) are from scikit-image
QPixmap,QImage	Image display objects for PyQt GUI integration

3. Implemented Functionalities

The application provides a set of image processing functionalities organized under five main menu groups: **File**, **Edit**, **Conversion**, **Segmentation**, and **Edge Detection**. Each functionality is also accessible through dedicated toolbar buttons with appropriate icons, shortcuts, and status tips.

- **File Menu**:
  Allows the user to open source images (.jpg, .png), save processed output (with overwrite or “Save As” dialog), and export images in different formats (e.g., .jpg to .png). Buttons and menu items are dynamically enabled based on application state.

- **Edit Menu**:
  Includes *Clear Source*, *Clear Output*, and *Undo/Redo Output*. The undo/redo system is built on a history stack and supports polymorphic operation reversal using operation_history and handler reruns.

- **Conversion Menu**:
  Offers *RGB to Grayscale* and *RGB to HSV* operations using scikit-image.color. These are handled by polymorphic classes and displayed instantly in the output pane.

- **Segmentation Menu**:
  Implements *Multi-Otsu*, *Chan-Vese*, and *Morphological Snakes* segmentation techniques from scikit-image.segmentation, applied via NumPy arrays and visualized using QPixmap.

- **Edge Detection Menu**:
  Supports four methods: *Sobel*, *Scharr*, *Prewitt*, and *Roberts*. Each method takes a dynamic threshold value from the GUI slider and applies real-time edge filtering.

All operations are executed in a background thread (ProcessingThread) for GUI responsiveness. Results are displayed in the output section with smooth fade-in animations, and status messages guide the user throughout.

Output Images:

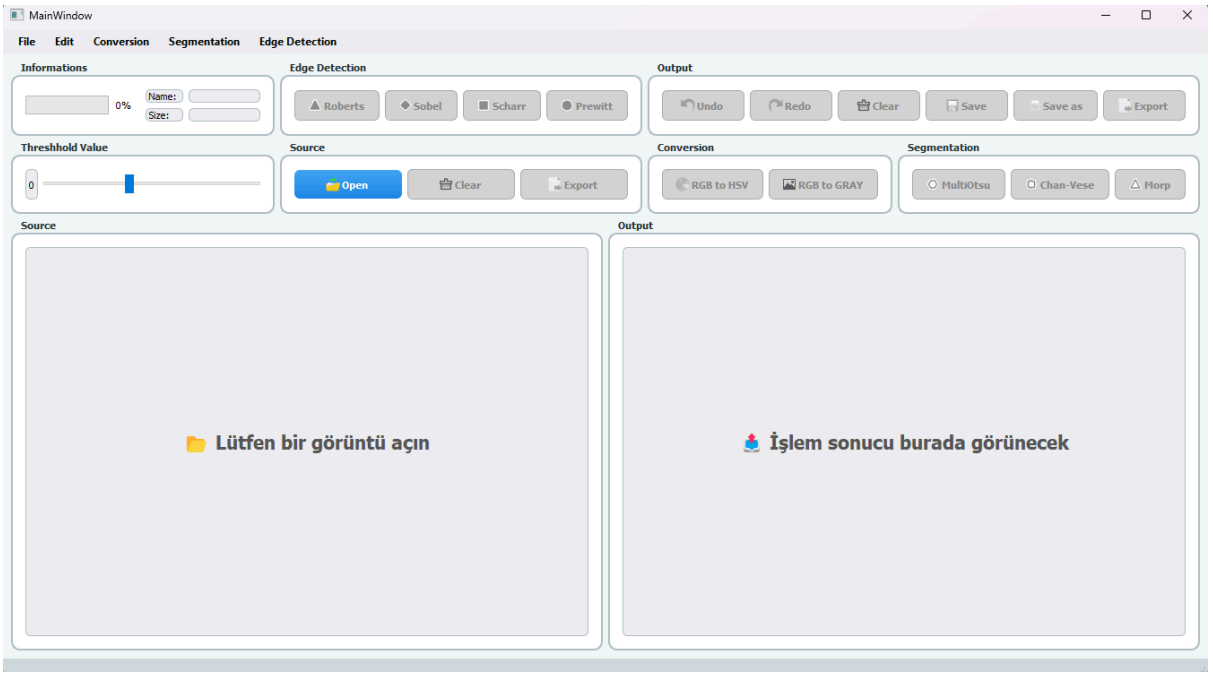

Figure 1. Initial GUI before loading any image

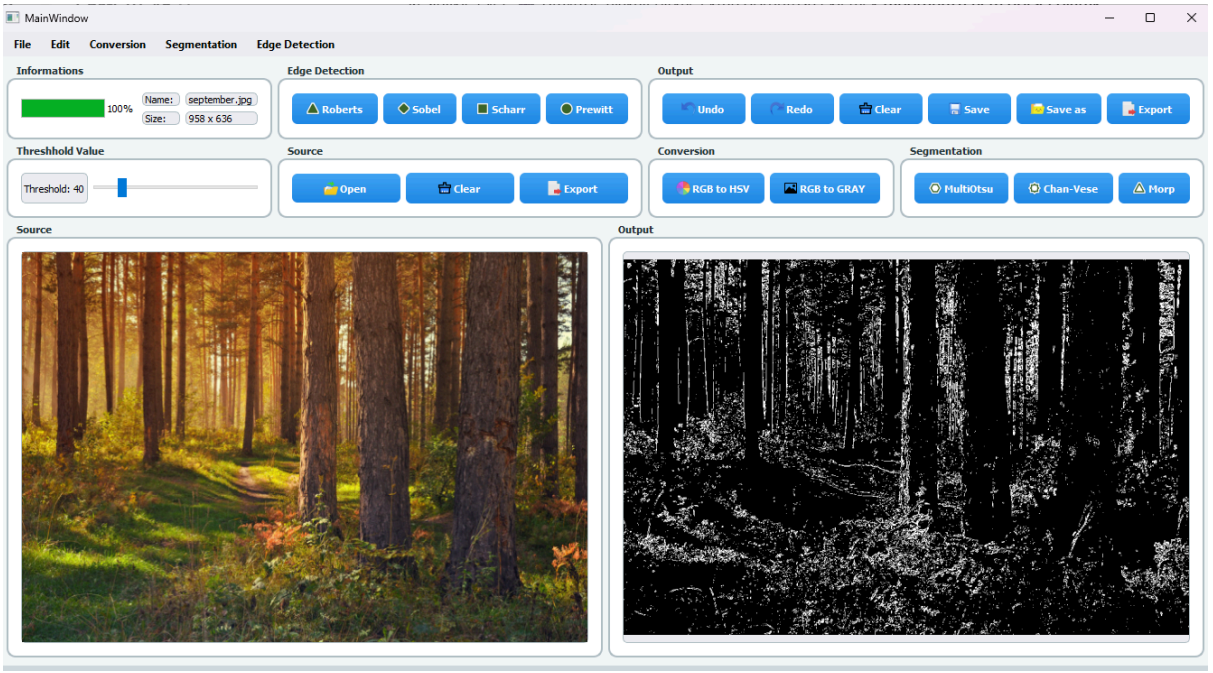

Figure 2. Output after Chan-Vese segmentation applied

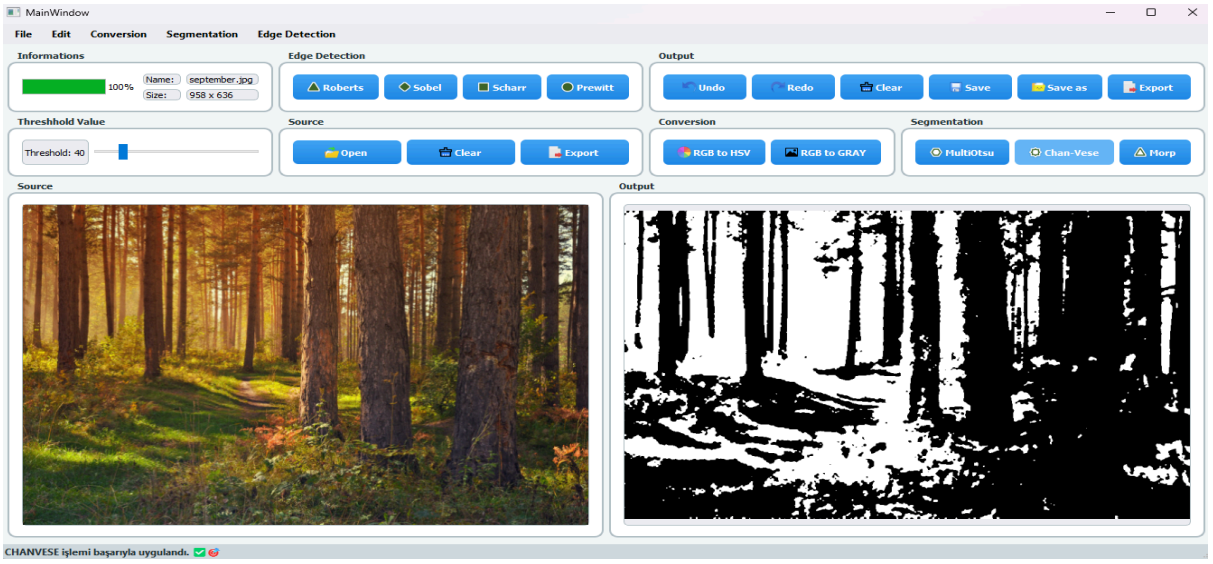

Figure 3. Edge detection with dynamic threshold

4. Conclusion

This project demonstrated a clean and modular application of object-oriented programming principles in building a functional image processing interface. Core concepts such as inheritance, polymorphism, and abstraction were effectively used alongside Python libraries like scikit-image, NumPy, and PyQt5. The system is responsive, extendable, and meets all functional requirements including segmentation, conversion, and edge detection with real-time interaction.