OBJECT-ORIENTED PROGRAMMING II
2024-2025 SPRING SEMESTERS
PROJECT

Project Name:
AGRIEDGE AGRICULTURAL ROBOT INTERFACE

Interface Developers:

Doğukan AVCI
NO:151220202051

Sinan İLBEY
NO:151220202096

1-Project Summary

AGRIEDGE agricultural robot is an autonomous robot supported by artificial intelligence, developed to assist farmers in modern agricultural processes. The main function of this robot is to mow weeds with its mechanical hoe. It also provides both visual and numerical information to the farmer by detecting environmental data (such as humidity, temperature, light intensity) and plant locations in the field. It detects plants, people, animals and stones with YOLO technology. Equipped with advanced sensors, the robot can precisely locate plants, map and analyze plant health.

There is a special interface that provides management of this system and interaction with the user. This easy-to-use interface consists of three main pages (pages):

Autonomous Control Panel: On this page, the user can start and stop the robot's mapping, hoeing and data recording processes and monitor the system status in real time. In the image taken from the robot camera, tomatoes and diseased plants identified with the YOLOv8 algorithm are classified and presented to the user with distance information. At the same time, humidity, temperature and light intensity analyses are displayed graphically at the top right. Finally, interaction is increased by giving and receiving commands with the control panel. In addition, the interface is started with voice support.

Manual Driving Panel: Allows the user to manually control the robot forward, backward, right and left with the directional keys. In addition, the user can switch between control modes with a single button.

Agricultural Assistant Screen: This is the section where the farmer can get detailed analysis about the field. Production conditions on the field are evaluated regionally with the help of maps created according to humidity, temperature and light amount. The analysis results are given to the farmer and it is ensured that he improves the field conditions. Automatic weather forecast and production suggestions are provided along with special suggestions for the farmer. It can set reminders for the system and the farmer using the calendar.

Thanks to this interface, all functions of the robot can be managed intuitively, field analyses can be followed effectively and all operations are carried out with minimum user intervention. The images show sample screens of these three pages.

2-Overview Interface and UML

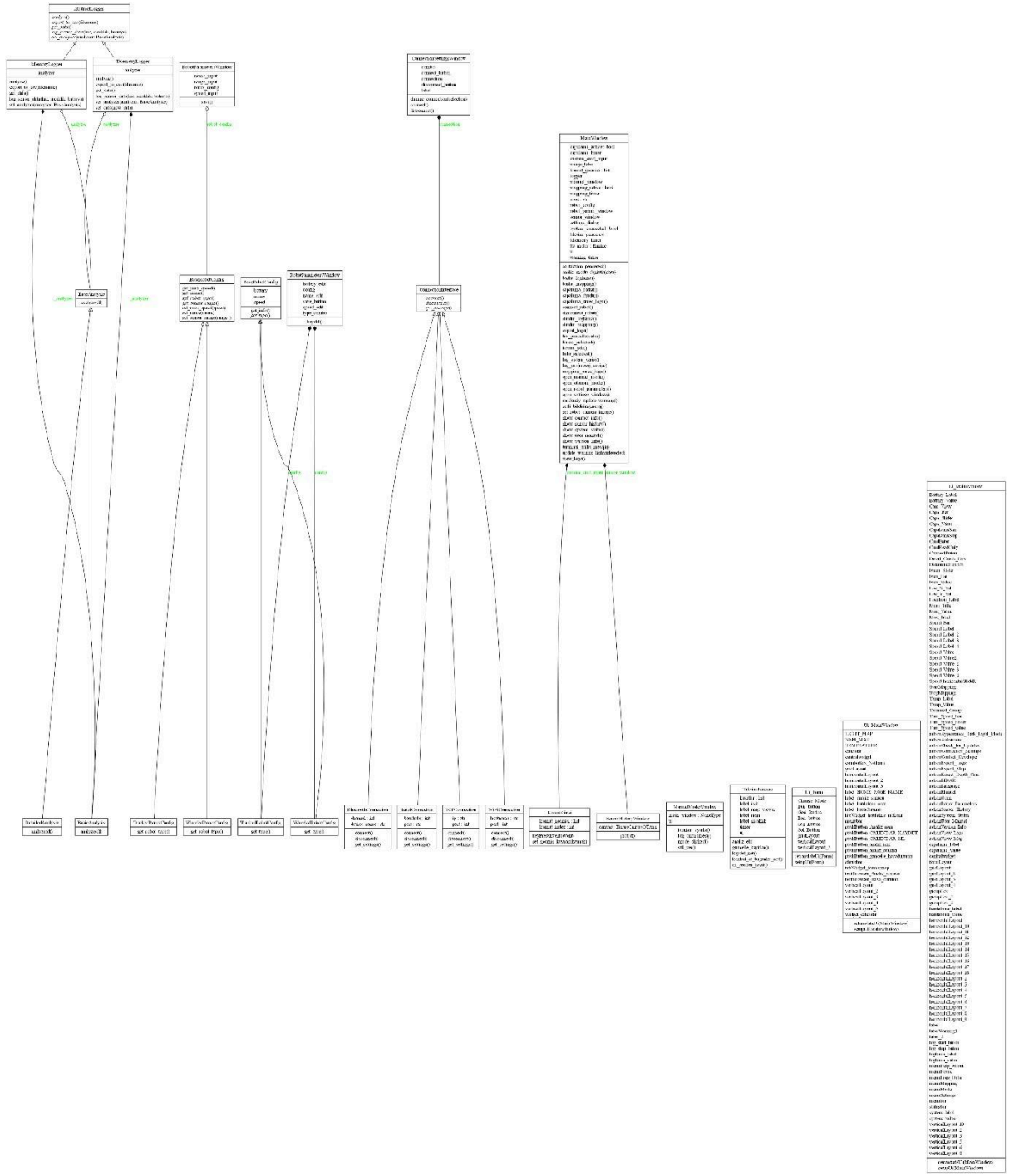

Figure 1. *Overall UML class and module diagram of the AGRIEDGE project*

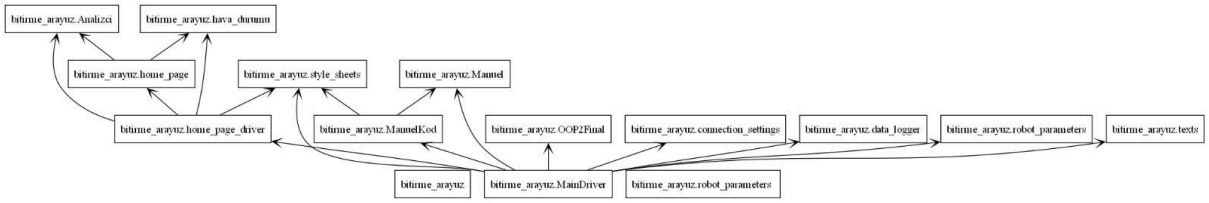

Figure 2. *Package-level dependency diagram of the AGRIEDGE system*

2.1 MainDriver.py – Central Control Architecture

The MainDriver.py module serves as the core orchestration layer of the AGRIEDGE smart agriculture interface, implementing the main user interface and system controller. The principal class defined in this module is MainWindow, which inherits from PyQt5's built-in QMainWindow class, thus gaining access to the full event-driven GUI architecture. This class embodies the architectural root of the system, coordinating robot connection, logging, manual/autonomous mode switching, data visualization, command interpretation, and real-time voice feedback via the **pyttsx3** text-to-speech engine. The code follows essential object-oriented programming (OOP) paradigms.

Inheritance is applied through both external and internal class relationships: the MainWindow class derives from QMainWindow, while the internally defined KomutGirisi class inherits from QLineEdit to specialize the command input field. KomutGirisi is seamlessly integrated into the GUI layout, overriding behavior while maintaining the base class's rendering and interaction capabilities, showcasing **polymorphism**.

Several **encapsulated components** such as the TelemetryLogger and MemoryLogger (imported from other modules) are instantiated and managed privately within MainWindow. Although Python does not enforce access control strictly, variables such as self.__telemetry_data in logger classes exemplify **data hiding**, and all operational logic is modularized to avoid direct cross-component interference.

The design leverages **built-in data structures**, including Python lists (for command history), dictionaries (for dynamically assigning icons and resources), and datetime objects (for timestamped terminal logs). GUI elements are organized using PyQt5's QWidget, QLabel, QPushButton, and layout managers (QVBoxLayout, QGridLayout). Meanwhile, **external libraries** such as matplotlib enable real-time data plotting, platform provides environment introspection, and pyttsx3 adds auditory alerts for better human-robot interaction.

Periodic tasks like mapping or hoeing are controlled using QTimer, a Qt class inheriting from **QObject**, which emits signals on timeouts to trigger related operations. This demonstrates both **composition** and **event-driven design** patterns. Overall, the MainWindow class encapsulates logic, maintains modular responsibilities, and interfaces with both abstract and concrete logger/analysis implementations, forming a polymorphic and extensible foundation for the entire AGRIEDGE platform.

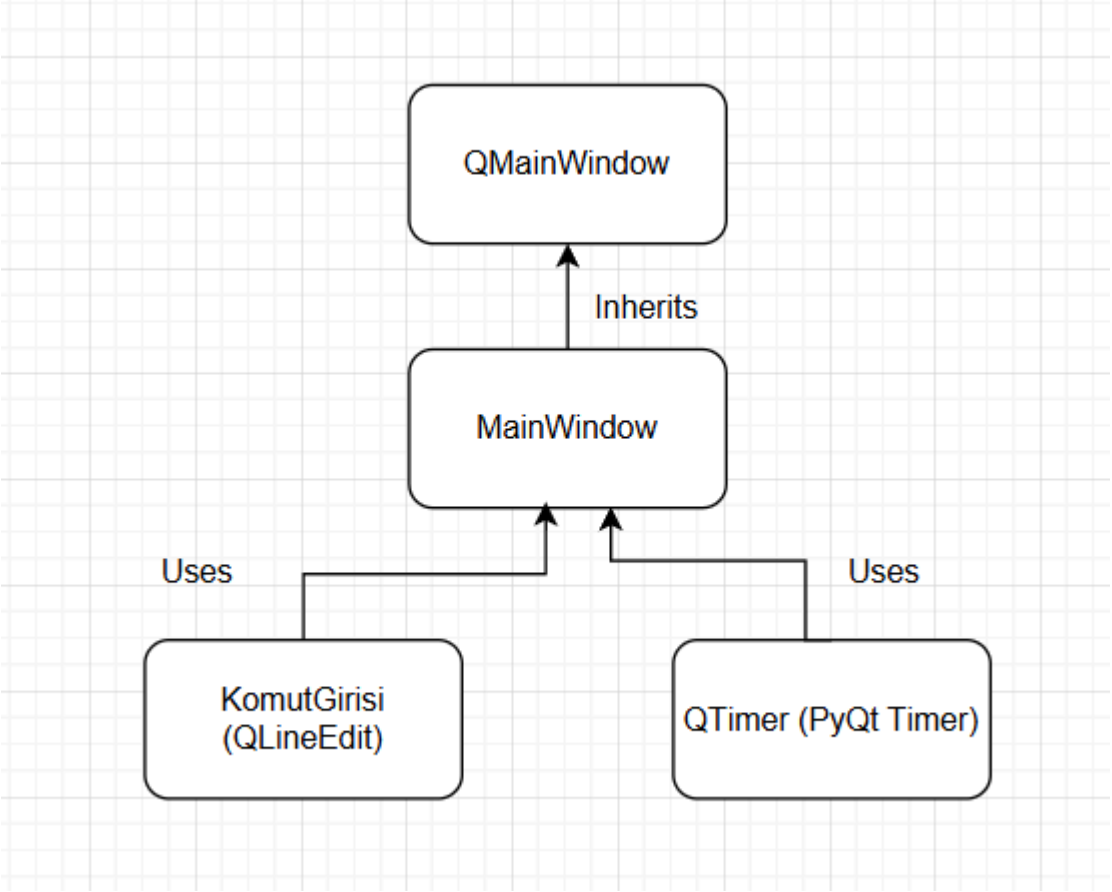

Figure 3. *UML class diagram showing the* ***MainWindow*** *class inheriting from* ***QMainWindow*** *and its associated GUI components.*

2.2 ManuelKod.py – Manual Driving Interface Module

The ManuelKod.py module defines the GUI logic for the manual driving mode of the AGRIEDGE agricultural robot system. Its central class, ManuelModeWindow, inherits directly from PyQt5's QWidget, making it a self-contained interactive window. This class acts as a bridge between the user and the robot, offering directional control (forward, backward, left, right, stop) through a simplified, icon-driven panel. It leverages **inheritance** to extend QWidget with customized control logic, and applies **composition** by embedding a UI class instance (Ui_Form) generated via Qt Designer from the Manuel.ui layout.

The manual control logic follows clear **encapsulation** principles. All UI components are managed internally through the self.ui namespace, preventing external manipulation. Button click events are connected using PyQt's signal-slot mechanism to lambda functions that invoke a log_bildir() method. This method optionally communicates user actions to the main interface (MainWindow) if it is passed during

construction — a design decision that ensures loose coupling between modules and reinforces **data hiding** by preventing direct external UI access.

The module also demonstrates **modularity and reusability**. It imports style constants (BUTON_STIL) from the global style_sheets.py module, ensuring consistent theming across the application. Furthermore, it loads icons dynamically using QIcon, pointing to structured asset folders (Assets/Icons/). This avoids hardcoding and allows UI elements to be updated independently of core logic.

Although ManuelModeWindow itself does not implement polymorphic behavior directly, it acts polymorphically from the perspective of the main application, where multiple control windows (manual, calendar, parameter settings) are launched through unified mechanisms. Built-in Python constructs such as lambda expressions, dictionaries (for icon paths), and optional function arguments (like main_window=None) are used to maximize flexibility without introducing complexity.

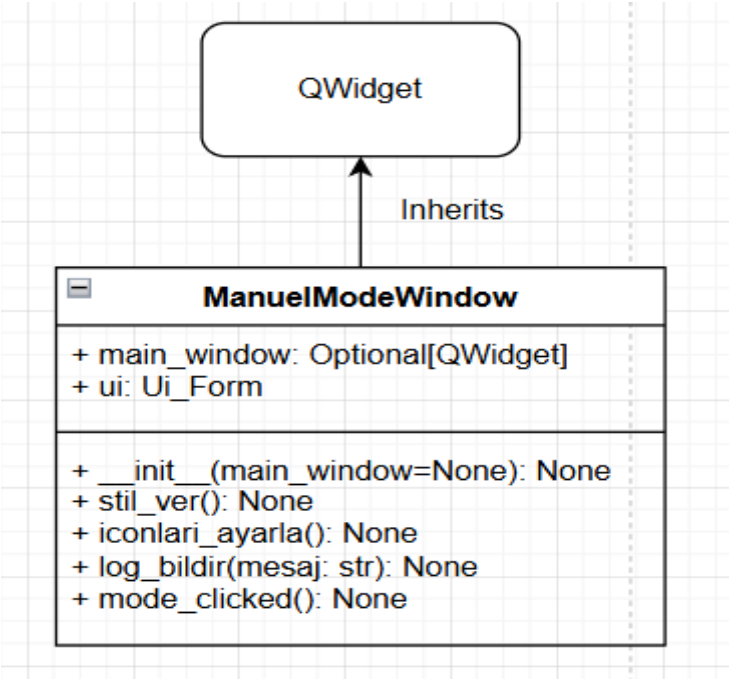

Figure 4. *Internal structure of the* ***ManuelModeWindow*** *class, including* ***QPushButton*** *controls and style integration.*

2.3 robot_parameters.py – Robot Configuration System

The robot_parameters.py module implements a complete object-oriented configuration system for robot parameter management. At its core is the abstract base class BaseRobotConfig, which defines common attributes such as name, speed, and battery, and declares an abstract method get_type(). This design

introduces **abstraction** and **polymorphism**, allowing concrete subclasses to implement behavior specific to different robot types.

Two subclasses, WheeledRobotConfig and TrackedRobotConfig, inherit from BaseRobotConfig. These subclasses override the abstract get_type() method to return robot-specific descriptors ("Tekerlekli" and "Paletli", respectively), making runtime polymorphism possible wherever a BaseRobotConfig object is referenced.

Encapsulation is maintained through the use of instance attributes, while **data hiding** is applied semantically (e.g., attributes are not exposed globally and only accessed via controlled methods). Though Python lacks strict private access, the structure enforces logical separation of concerns.

The module also contains a PyQt5 GUI class named RobotParametersWindow, which inherits from QWidget. This class provides an interactive dialog for entering a robot's name, speed, battery level, and selecting its type from a QComboBox. Once "Save" is clicked, a new object is instantiated from the appropriate configuration subclass, demonstrating **factory pattern behavior** via runtime class selection. Internally, the form uses built-in data types like str and float and performs validation before object creation.

Altogether, this module exemplifies clean OOP design in Python, emphasizing flexibility, modularity, and strong adherence to software design patterns.

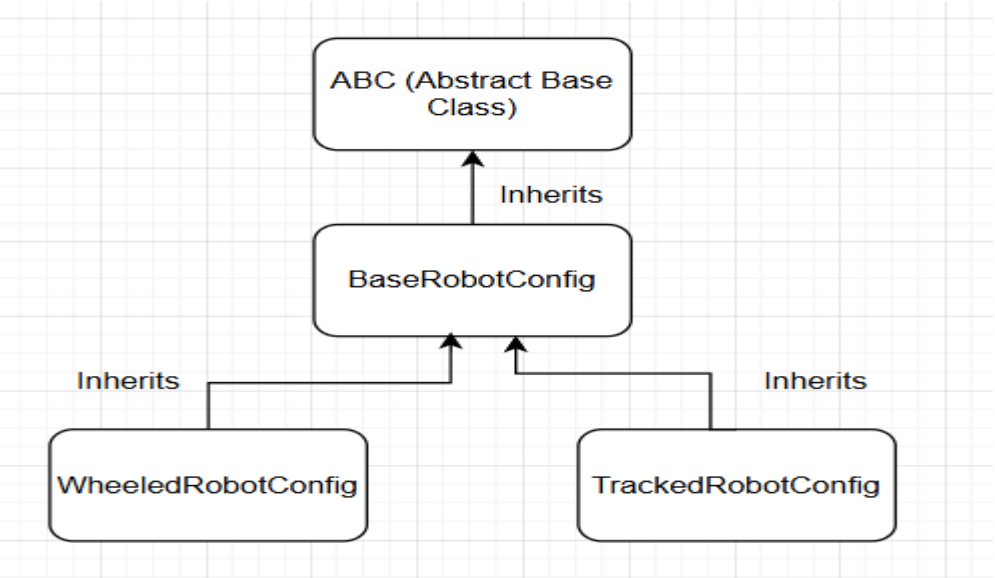

Figure 3. *Inheritance hierarchy of* ***BaseRobotConfig*** *with* ***WheeledRobotConfig*** *and* ***TrackedRobotConfig*** *as concrete subclasses.*

2.4 data_logger.py – Telemetry Logging & Strategy System

The data_logger.py module defines a polymorphic, modular telemetry logging system using abstract base classes and interchangeable analysis strategies. The

abstract class AbstractLogger specifies the contract for all loggers, enforcing method signatures like log_sensor_data(), analyze(), and export_to_csv(). Two concrete classes — TelemetryLogger and MemoryLogger — inherit from AbstractLogger and implement these methods using different internal data structures (Pandas DataFrame vs. Python list). This allows for flexible swapping of logging backends.

Similarly, the BaseAnalysis abstract class defines a standard interface for analysis strategies. BasicAnalysis and DetailedAnalysis both inherit from BaseAnalysis and override the analyze() method. These analysis objects are injected into logger instances at runtime, demonstrating the **strategy pattern** and **runtime polymorphism**.

Encapsulation and data hiding are enforced using internal attributes like __telemetry_data in TelemetryLogger, which hides raw logs from external access. The overall design enables future extensions by allowing new logger or analysis types to plug into the system without modifying existing code — following the **Open/Closed Principle** in OOP.

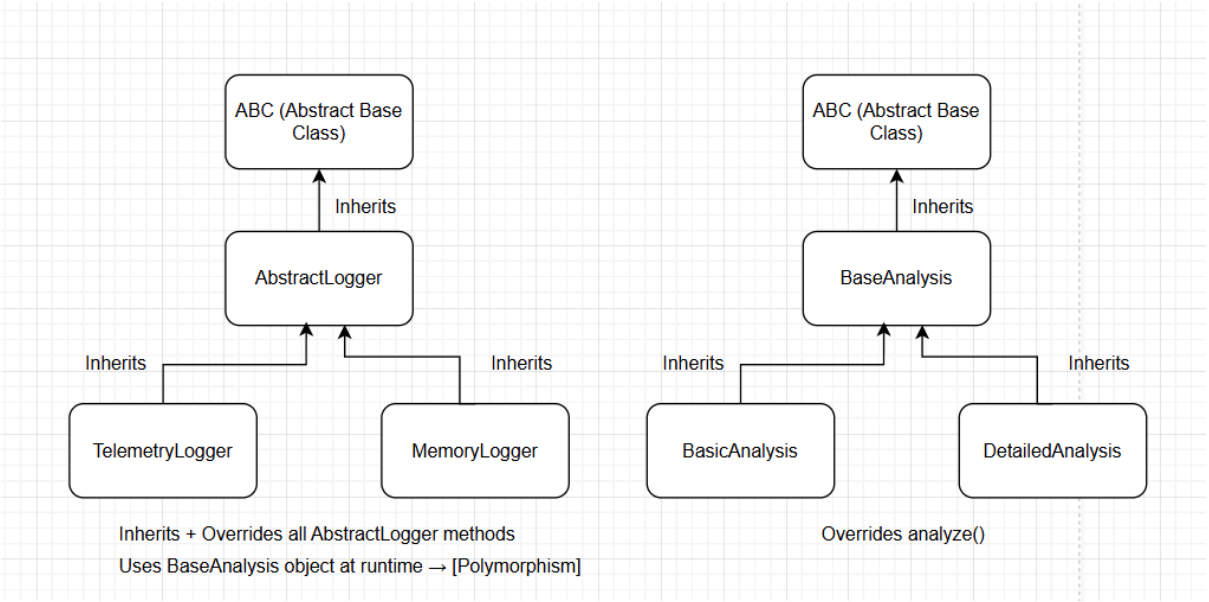

Figure 4. *Polymorphic relationship between* ***AbstractLogger*** *and its subclasses* ***TelemetryLogger*** *and* ***MemoryLogger****, using runtime analysis strategies.*

2.5 hava_durumu.py – Weather Forecast Integration System

The hava_durumu.py module provides a dedicated weather information integration service for the AGRIEDGE interface. It defines a single functional component, guncelle_hava_durumu(text_browser), which fetches 7-day weather forecast data from the **Open-Meteo API** using the requests library and dynamically renders it in HTML format into a PyQt5 QTextBrowser widget. This module demonstrates

effective use of **external libraries** (requests, datetime) and **HTML generation** for user-facing output.

The function extracts daily maximum and minimum temperatures and precipitation levels from the API response, then classifies each day's condition with an emoji-based icon system (e.g., ☀️ for sunny, 🌧️ for rain). This simple heuristic forms a lightweight rule-based decision layer, enabling clear visual communication without requiring a full weather engine.

Although the module does not define any classes or inheritance, it demonstrates **modular design** and **separation of concerns**, encapsulating weather logic in a single callable unit. It also utilizes built-in Python data structures such as lists and strings to format and present forecast data in a compact HTML table. In the context of the AGRIEDGE system, this module acts as a **plug-and-play utility** that enhances the usability of the calendar and home screen interfaces.

2.6 Analizci.py – RGB-Based Agricultural Map Analysis

The Analizci.py module implements an RGB-based visual analysis system for interpreting agricultural maps representing environmental factors such as moisture, light intensity, and temperature. The core function analiz_yap(resim_yolu, harita_tipi) processes a provided image by dividing it into four directional regions (North, South, East, West) and then computes average intensities of red, green, and blue pixels in each region.

This logic is encapsulated in a single, pure function that utilizes **external libraries** such as Pillow for image processing, NumPy for matrix operations, pandas for tabular data formatting, and matplotlib for generating bar charts. The design reflects a **functional programming paradigm** with clear input-output isolation and no side effects beyond returning an HTML string for GUI rendering.

The function exhibits modular design by separating analysis logic (renk_oranlari) from visualization and reporting. It builds an HTML-formatted string that includes a statistics table and an embedded base64-encoded bar chart image, which can be rendered in a PyQt5 QTextBrowser. This makes it suitable for direct integration into UI workflows without requiring class inheritance or GUI subclassing.

Although the module contains no classes, it demonstrates **encapsulation** through internal helper functions and **data abstraction** via structured tabular and graphical outputs. The system's output is both human-readable and visualization-rich, offering actionable insights to users in a compact format.

2.7 home_page_driver.py – Calendar & Map Analysis Controller

The home_page_driver.py module implements the event-driven logic for the AGRIEDGE calendar and analysis interface through the TakvimPencere class. This class inherits from QMainWindow and serves as a GUI controller that binds front-end elements (defined in home_page.py) to backend functionalities, including RGB map analysis, calendar-based reminders, and real-time weather updates.

The class structure follows classic **Model-View-Controller (MVC)** separation, where UI layout is loaded from an external module (Ui_MainWindow), and all user interaction logic (e.g., button clicks, image rendering, and reminder management) resides within the TakvimPencere controller. The class applies **encapsulation** by managing all calendar notes through internal lists and dynamically populating image placeholders (QLabel) based on predefined image paths.

Built-in Qt widgets like QCalendarWidget, QLabel, and QPushButton are used, and QTimer is employed to periodically refresh reminder lists — showing practical use of PyQt's signal-slot mechanism. External utility functions such as analiz_yap() (for image-based diagnostics) and guncelle_hava_durumu() (for weather updates) are called directly, showcasing **modular composition** rather than inheritance.

Although the module does not apply polymorphism explicitly, it features **runtime behavior switching** based on user interaction (e.g., switching between different maps and triggering appropriate analysis). The system design allows for scalable GUI logic while maintaining a clean connection to analysis and weather services.

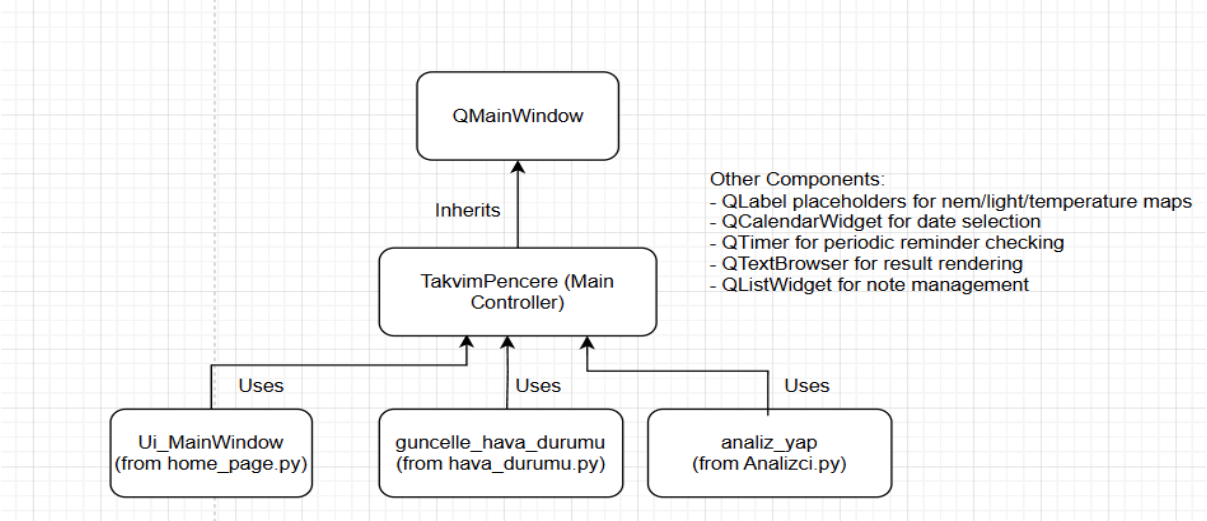

Figure 5. *UML diagram of the* ***TakvimPencere*** *class*

2.8 texts.py – Static Text Resource Management

The texts.py module serves as a centralized repository for all static text resources used across the AGRIEDGE interface. It defines global string constants such as VERSION_TEXT, USER_MANUAL_TEXT, and CONTACT_TEXT, which are referenced in various parts of the GUI, including version dialogs, user help menus,

and contact information panels. This approach demonstrates strong **separation of content and logic**, as it prevents hardcoded strings from being scattered throughout the codebase.

Although the module contains no classes, inheritance, or polymorphism, it exemplifies **modularity** and adheres to the **Single Responsibility Principle** by isolating all user-facing text content. Each constant is formatted with placeholders (e.g., {os_name}, {python_version}), making them easily adaptable at runtime using Python's str.format() method. This enables dynamic generation of version reports and flexible integration with internationalization systems (i18n), should multilingual support be added in the future.

The use of well-named constants also enhances code readability and maintainability. Since all user-visible information is stored in one place, future updates to terminology, developer contacts, or documentation are simplified and risk-free with regard to business logic.

2.9 style_sheets.py – Centralized GUI Styling Definitions

The style_sheets.py module defines a centralized set of CSS-like style templates that control the visual appearance of the AGRIEDGE interface. These style definitions are stored as Python string constants, including GREEN_BUTTON_STYLE, RED_BUTTON_STYLE, TERMINAL_STYLE, and MAIN_TITLE_STYLE, among others. Each constant is associated with a specific type of widget (e.g., QPushButton, QLabel, QGroupBox) and is applied at runtime using the setStyleSheet() method.

This modular design reflects the **separation of concerns** principle: visual styling is completely decoupled from application logic. As a result, UI elements can be restyled globally without touching business logic or controller code. The module also supports **reusability**, allowing the same visual theme to be shared across multiple windows and widgets, improving interface consistency and reducing redundancy.

Although no inheritance or class-based polymorphism is present, the design enables **functional polymorphism** by allowing the same style to be applied to different widget instances dynamically. For example, any number of QPushButton widgets can adopt the same GREEN_BUTTON_STYLE, thus enforcing a consistent design system across the entire application.

The use of named constants also supports **maintainability**. Instead of manually repeating styles or relying on Qt Designer’s hardcoded formats, developers can update the visual appearance of the application from a single module — an approach aligned with scalable, long-term software engineering practices.

2.10 connection_settings.py – Polymorphic Connection Management Interface

The connection_settings.py module introduces a clean, interface-based architecture to handle multiple robot communication protocols, including Serial, TCP, Bluetooth, and Wi-Fi. At the core of this design lies the abstract base class ConnectionInterface, which defines a common contract (connect(), disconnect(), get_settings()) that all connection types must implement. This is a textbook application of **abstraction**, ensuring consistency across all communication implementations.

Concrete classes such as SerialConnection, TCPConnection, BluetoothConnection, and WiFiConnection all **inherit from** ConnectionInterface and override its methods with protocol-specific logic. This enables **runtime polymorphism**, as the GUI window (ConnectionSettingsWindow) stores a single ConnectionInterface reference (self.connection) and interacts with it via virtual methods — regardless of its underlying type. Inside the ConnectionSettingsWindow class, the user can dynamically select the desired communication type from a dropdown menu. The system reacts by instantiating the corresponding class at runtime, demonstrating both **polymorphic instantiation** and the **factory pattern**. The GUI class itself inherits from QDialog and encapsulates its UI logic, keeping backend communication logic abstracted through the interface.

In addition to strong OOP principles, this module demonstrates **encapsulation** by keeping protocol-specific attributes (e.g., port, IP, baudrate) within each connection object. These values are never directly exposed — instead, get_settings() provides controlled, human-readable access, implementing **data hiding**.

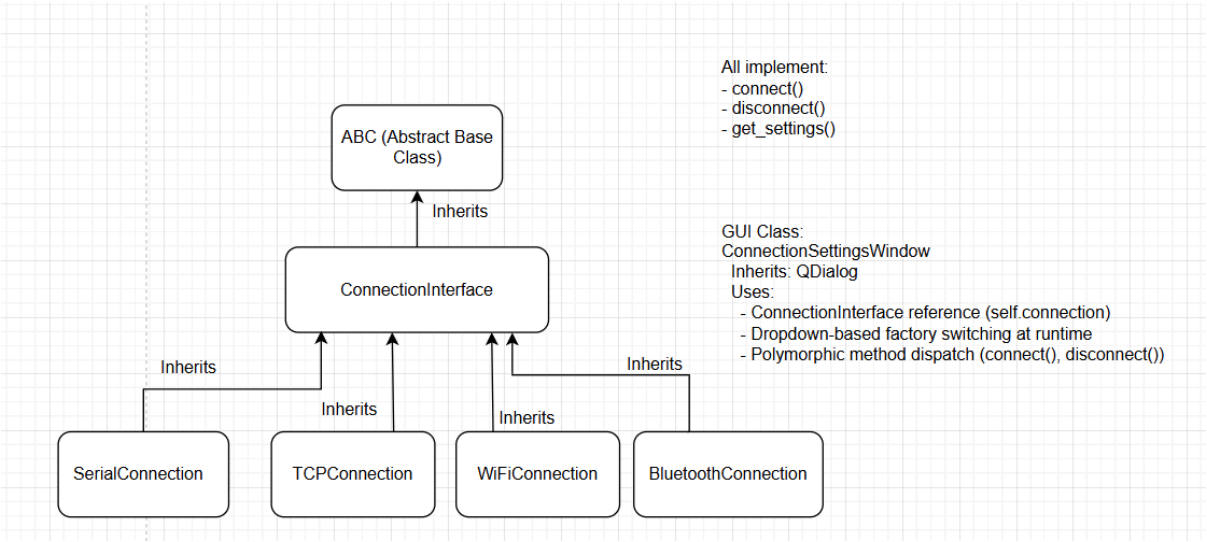

Figure 8. *Interface-based inheritance structure for connection types, demonstrating* ***runtime polymorphism*** *using* ***ConnectionInterface****.*

Built-in Data Structures Usage in AGRIEDGE

Table 1. *Summary of built-in Python data structures used in the AGRIEDGE system*

Data Structure	Used In Module(s)	Purpose
list	data_logger.py, Analizci.py, home_page_driver.py, texts.py	Storing time-series sensor data, region-wise RGB values, calendar notes, menu items
dict	MainDriver.py, connection_settings.py	Mapping UI elements to icons or actions, runtime class selection (factory pattern)
tuple	Analizci.py , robot_parameters.py	RGB statistics per region, grouped parameter returns
str	all modules	Logging, UI label content, file paths, HTML construction
set		

3 Main Page

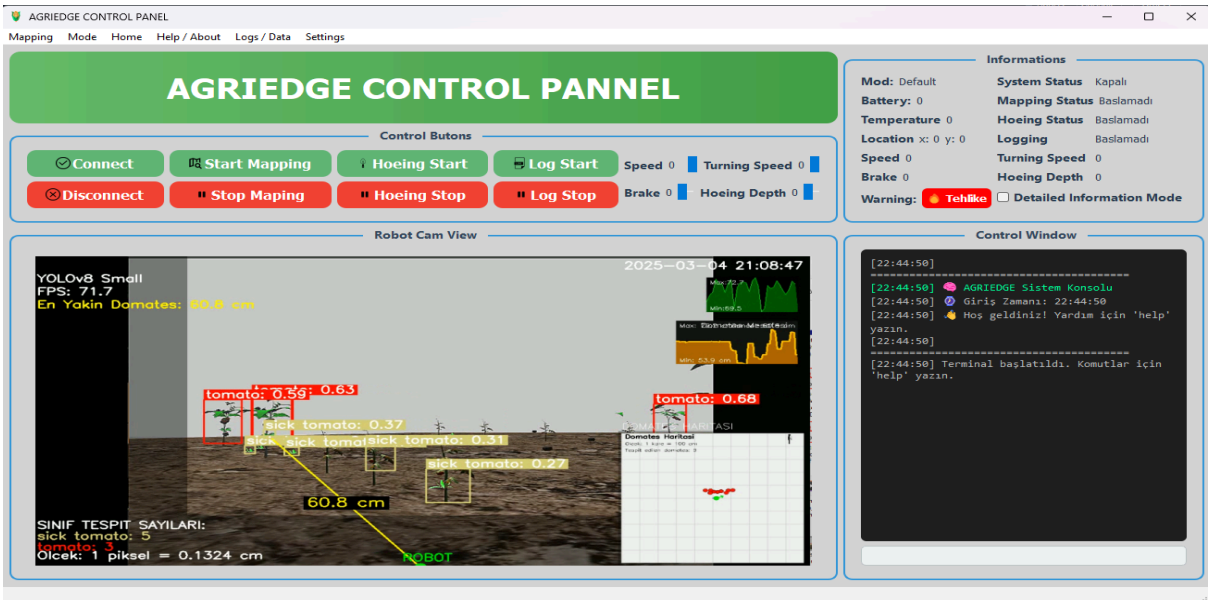

Figure 9. *Interface Main Page Image*

3.1 Control Buttons

This section contains buttons used to operate and manage the robot:

Table 2. *Description of core control panel buttons and their corresponding functions in the AGRIEDGE interface*

Button	Function
Connect	Establishes a connection with the robot
Disconnect	Terminates the existing robot connection
Start Mapping	Begins autonomous field mapping
Stop Mapping	Stops the mapping process
Hoeing Start	Starts the mechanical hoeing (weed removal) process
Hoeing Stop	Stops the hoeing process
Log Start	Starts data logging
Log Stop	Stops data logging

3.2 Slider Controls

On the right-hand side, sliders allow manual tuning of movement and tool parameters:

- **Speed (Hız):** Controls forward/backward speed
- **Rotation Speed (Dönüş Hızı):** Controls turn rate
- **Brake (Fren):** Adjusts braking force
- **Hoe Depth (Çapa Derinliği):** Sets depth of the hoeing tool

3.3 Informations Panel

Displays real-time status information from the robot:

Table 3. Overview of status and telemetry indicators displayed in the feedback panel

Info	Description
Mode	Selected mode (e.g., Default, Manual, Autonomous)
System Status	Indicates whether the robot is ON or OFF
Battery	Battery level
Temperature	Ambient temperature from sensors
Position	Current X, Y coordinates of the robot
Speed / Rotation / Brake / Depth	Real-time control values
Warning	Displays alert (e.g., ⚠️ Danger)
Detailed Info Mode	Toggles expanded feedback panel

3.4 Robot Cam View

This section shows the live camera feed processed with object detection:

- **YOLOv8 Small**: AI model used for object classification
- **FPS**: Current frame rate (e.g., 71.7)
- **Nearest Tomato**: Shows distance to closest tomato (60.8 cm)
- **Object Classification**: Detects tomato, sick tomato, etc.
- **Mini Field Map**: A bottom-right mini-map showing detected plant positions

3.5 Robot Cam View

This section shows the live camera feed processed with object detection:

- **YOLOv8 Small**: AI model used for object classification

- **FPS**: Current frame rate (e.g., 71.7)
- **Nearest Tomato**: Shows distance to closest tomato (60.8 cm)
- **Object Classification**: Detects tomato, sick tomato, etc.
- **Mini Field Map**: A bottom-right mini-map showing detected plant positions

Figure 10. *User interface of the Manuel Sürüş Paneli (ManuelModeWindow),*

This panel allows the user to directly control the robot's movement using directional buttons. It's designed for manual navigation and testing purposes.

Control Buttons & Functions

Table 4. *Description of manual driving control buttons and their corresponding actions*

Button	Function Description
↑ **Forward (İleri)**	Moves the robot forward
← **Left (Sol)**	Rotates or steers the robot to the left
→ **Right (Sağ)**	Rotates or steers the robot to the right
↓ **Backward (Geri)**	Moves the robot backward

× **Stop (Dur)**	Immediately stops all movement
⟳ **Change Mode**	Toggles between **manual mode** and **autonomous mode**

This interface is typically used during field testing, debugging, or when autonomy is temporarily disabled.

The “Change Mode” button is essential for toggling control authority between the operator and the onboard autonomous system.

4 Agricultural Assistant Page

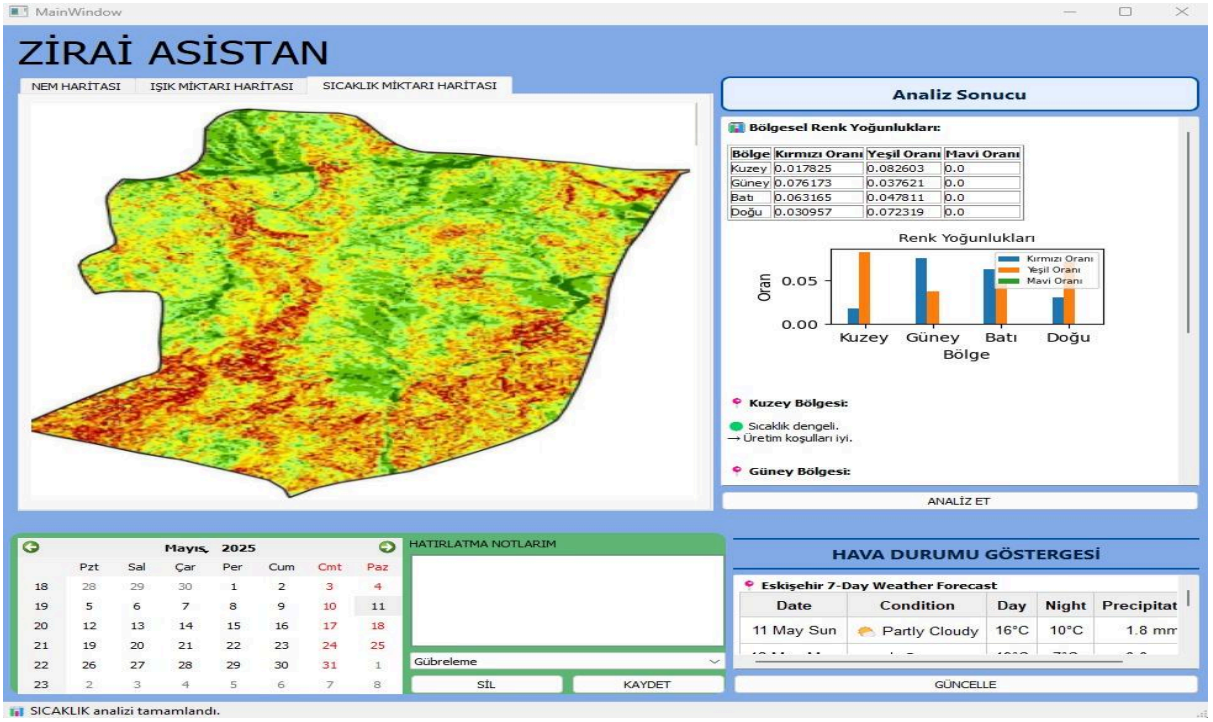

Figure 11. *Interface Agricultural Asisant Page Image*

This page is designed to assist farmers by offering **region-based agricultural analysis** based on RGB image data, weather forecasting, and daily task planning. It serves as a **decision support tool** that interprets agricultural map images and helps farmers manage field operations efficiently.

4.1 Tabs – Map Views

At the top of the interface, three tabs are visible:

1. **NEM HARİTASI (Moisture Map)**
2. **IŞIK MİKTARI HARİTASI (Light Intensity Map)**
3. **SICAKLIK MİKTARI HARİTASI (Temperature Map)**

Each tab shows a different heatmap (in RGB format), indicating environmental distribution across the field. These maps are derived from the sensor data and image analysis performed by the robot.

4.2 RGB-Based Regional Analysis

On the right side of the map:

- The **"Analiz Et" (Analyze)** button triggers a real-time analysis based on **color density** (RGB values) in different regions of the field.
- A table shows the **Red, Green, and Blue pixel density ratios** in **North, South, East, and West** sections of the field.
- A corresponding **bar chart** visualizes these ratios for comparison.
- Below the chart, qualitative comments are generated (e.g., “Sıcaklık dengeli, üretim koşulları iyi.”) based on predefined thresholds of color ratios.

4.3 Calendar & Task Notes Panel

- The calendar located on the lower left corner allows users to **schedule daily agricultural operations**.
- For example, users can plan irrigation, fertilization, or hoeing tasks by **clicking a date** and entering a note in the **“Gübreleme” (Fertilization)** field.
- The **“Kaydet” (Save)** button stores the task, and the **“Sil” (Delete)** button removes it.
- This module helps in **managing the robot’s routine tasks** over a planting season.

4.4 Live Weather Panel

- On the bottom right corner, the **“HAVA DURUMU GÖSTERGESİ” (Weather Status Display)** shows a **7-day weather forecast**.

- This data is automatically fetched from an external API and updated when the **“GÜNCELLE” (Update)** button is clicked.

5. OOP Approach:

The software architecture follows strict object-oriented programming principles for modularity and scalability:

- **Abstraction**: Core functionalities like file operations, image processing, and sensor communication are abstracted into separate modules.

- **Encapsulation**: GUI components and processing logic are isolated, avoiding tight coupling.

- **Inheritance**: Common operations like image handling, edge detection, and segmentation inherit from abstract base classes.

- **Polymorphism**: Control handlers interact with operation classes through shared interfaces, allowing flexibility in algorithm application.

6. Technologies & Libraries Used:

- **GUI Framework**: PyQt5

- **Image Processing**: skimage

- **Data Processing**:

    - **numpy** for numerical operations,

    - **pandas** for tabular data management,

    - **matplotlib** for plotting graphs and statistical analysis.

- **Threading**: Python's QThread was used to handle long-running image operations without freezing the interface.

7. Conclusion – Final Architectural Evaluation

The AGRIEDGE smart agriculture interface demonstrates a robust, scalable, and maintainable software architecture grounded in solid object-oriented design principles. Throughout the system, key OOP concepts such as **inheritance**, **abstraction**, **polymorphism**, **encapsulation**, and **data hiding** are consistently applied across both backend logic and GUI components.

Inheritance is leveraged in several core modules to implement extensible class hierarchies, such as the connection system (ConnectionInterface) and robot configurations (BaseRobotConfig). Polymorphism enables runtime flexibility and interchangeability, particularly in logging (AbstractLogger) and connection management. This allows the system to dynamically adapt to user selections and operate generically on abstract references — ensuring code generality and adherence to the **Open/Closed Principle**.

Encapsulation is effectively enforced via class-scoped attributes and clear method interfaces, while data hiding is demonstrated by shielding internal configuration data and sensor records from direct access. UI logic is separated from content and styling, thanks to supporting modules like texts.py and style_sheets.py, which improves code readability and eases future modifications or localization (i18n).

Built-in Python data structures (such as list, dict, tuple, and str) are used efficiently throughout the codebase, providing lightweight, expressive means for storing dynamic UI states, sensor data, and configuration mappings. Modular design is achieved by splitting functionality into dedicated files, each responsible for a single aspect of the system — analysis, logging, communication, and visualization — enabling clean code reuse and testability.

In summary, AGRIEDGE showcases a well-structured and forward-compatible design. The system can be easily extended with new robot types, sensors, or data analysis tools with minimal changes to the existing architecture. It stands as a strong example of how domain-specific applications — in this case, smart farming — can benefit from general-purpose OOP strategies and design patterns.