

OpenPilot

CONCRETE ARCHITECTURE



Table of contents

01

Introduction

02

**Top level
Architecture**

03

**Subsystems and
their interactions**

04

Design Patterns

05

Diagrams

06

Reflexion analysis

07

Conclusion



01

Introduction



Introduction

- **Objective:** Provide a comprehensive understanding of the Camerad subsystem within the OpenPilot project, focusing on camera management and operation.
- **Scope:** Examination of the subsystem architecture, including module organization, operational states, data flow, and interaction with external systems.
- **Key Components:**
 - Camerad Subsystem: Central hub for camera-related features.
 - Cameras Subsystem: Manages hardware and software for camera operation.
 - Camera Utility Module, Camera QCOM2 Module, and Camera Common Module: Specific roles in the subsystem for efficiency and scalability.
- **Design Patterns:** Highlight the use of the Singleton pattern in managing system resources efficiently and ensuring global access to the MemoryManager class.
- **Interaction with Sensors:** Overview of the SensorInfo class and its role in encapsulating sensor information, promoting code modularity and flexibility.



02

Top level Architecture



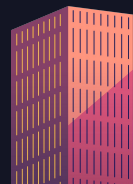
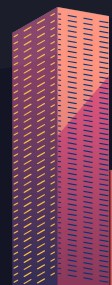
Conceptual Architecture:

At a high level, OpenPilot's conceptual architecture is built around several key architectural styles:

1. Event-Driven Architecture: Components communicate primarily through events or messages, promoting loose coupling and responsiveness to real-time data and situations.

2. Process Control Architecture: OpenPilot is designed to continuously monitor inputs (like camera and sensor data) and adjust outputs (vehicle controls) in a feedback loop, critical for real-time decision-making in autonomous driving.

3. Layered Architecture: The software is structured in layers, from low-level hardware interaction to high-level decision-making and user interface, facilitating abstraction and separation of concerns.



Concrete Architecture:

The concrete architecture is implemented through various daemons and components, each responsible for specific tasks within the OpenPilot ecosystem:

1. camerad: Manages camera inputs, processing images for further analysis by the system.

2. boardd: Handles communication with the vehicle's hardware, particularly over the CAN bus, ensuring OpenPilot's commands are executed by the car.

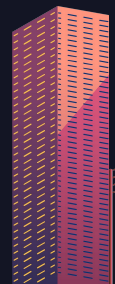
3. sensord: responsible for managing data from various sensors (excluding the camera), integrating this data into the system's decision-making processes.

4. modeld: Used for processing data using machine learning models, crucial for tasks like interpreting the vehicle's surroundings and making driving decisions.

5. ubloxd and qcomgpsd: These daemons handle GPS data, with "ubloxd" likely focusing on u-blox devices and "qcomgpsd" on Qualcomm devices, ensuring the system has accurate positioning information.

6. proclogd and logcatd: Responsible for logging, these daemons collect, manage, and store logs from various system components, essential for debugging and monitoring.

7. manager: This component oversees the operation of various daemons, ensuring they're running correctly and coordinating their activities



Data Flow

1. Data Collection: o Camera Data: The camera(s) capture video frames, which are crucial for understanding the vehicle's surroundings. The camerad daemon processes this data, which involves tasks like image correction, cropping, and encoding before passing it on for further analysis.

o Sensor Data: Various sensors, including GPS (handled by ublox or qcomgpsd), IMU (Inertial Measurement Unit), wheel speed sensors, and others, provide additional data about the vehicle's state and environment. The sensord daemon is responsible for aggregating and processing this sensor data.

2. Data Processing: o Model Processing: The processed camera and sensor data are fed into machine learning models (via modeld or a similar component). These models interpret the data to understand the vehicle's environment, detect objects and determine the optimal path and actions for the vehicle.

o Localization: GPS data, combined with other sensor inputs, helps the system understand the vehicle's precise location and orientation, which is essential for navigation and path planning

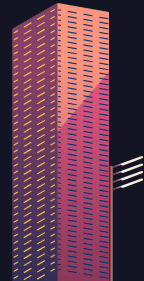
3. Decision Making:

o The results from the model processing stage are used to make driving decisions. This involves determining the appropriate steering, acceleration, and braking commands to safely and efficiently navigate the vehicle along its intended path.

4. Command Execution:

o Vehicle Control: The determined commands are sent to the vehicle's control systems via the board daemon, which communicates with the vehicle's CAN bus. This daemon translates the high-level commands into specific messages that the vehicle's electronic control units (ECUs) can understand and act upon, adjusting the steering, throttle, and brake as needed.

o Feedback Loop: The system continuously monitors the vehicle's response to the issued commands and the evolving driving environment. This feedback loop allows OpenPilot to adjust its decisions in real-time, ensuring that the vehicle responds appropriately to dynamic road conditions.

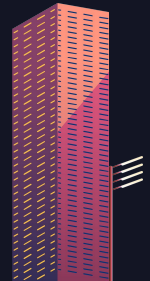


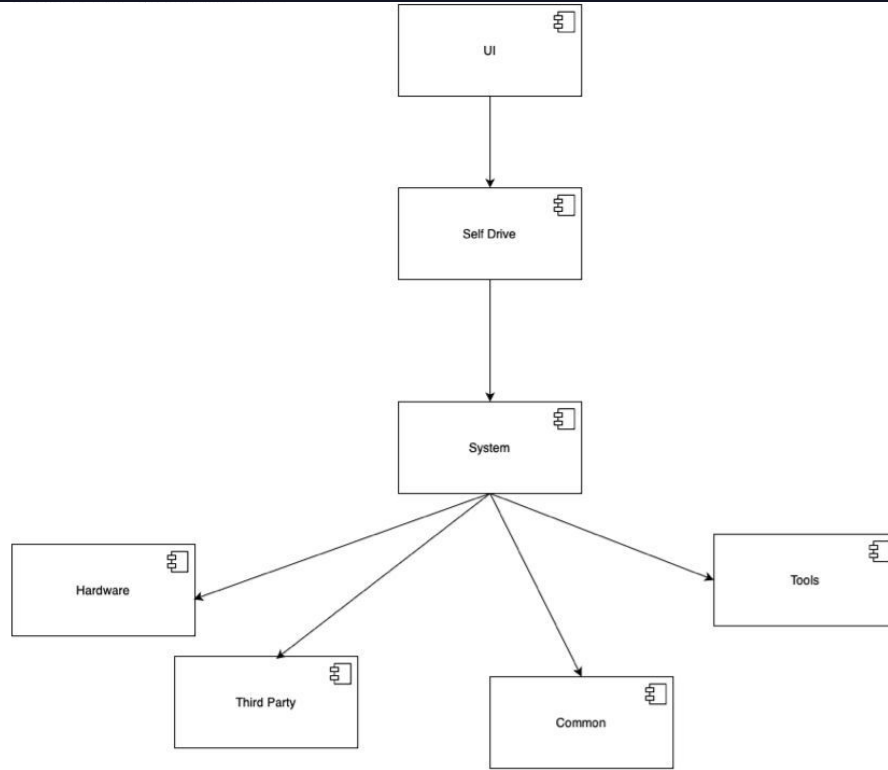
5. Logging and Monitoring:

o Daemons like `proclogd` and `logcatd` are responsible for logging system activity and monitoring the health of various components. These logs are crucial for debugging, performance monitoring, and ensuring the system operates within its safety parameters.

6. User Interface:

o While not directly a part of the core data flow for driving, OpenPilot also includes a user interface component that provides feedback to the driver, displays system status, and allows for user inputs and interactions





03

Subsystems and their Interactions



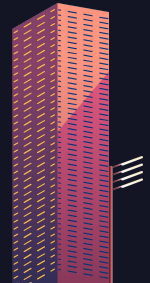
SUBSYSTEM BREAKDOWN

- **Camerad Subsystem:**
 - **Cameras Subsystem:**
 - **Camera Utility Module:** Handles utility functions and interfaces with third-party libraries.
 - **Camera QCOM2 Module:** Adapts or enhances camera functionalities for Qualcomm-specific hardware.
 - **Camera Common Module:** Central hub for camera operations, interfacing with selfdrive, tools, and other camera modules.



cameraD Subsystem

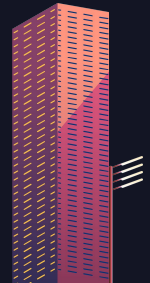
The general subsystem that houses all camera-related features is called the Camerad Subsystem. It includes the camera module and all of its features.



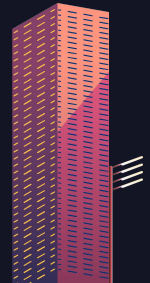
cameras Subsystem

This subsystem of camerad is in charge of the hardware and software for the cameras. It is made up of various parts, each playing a different role:

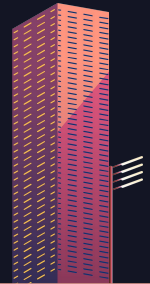
Camera Utility Module :The camera_util.h and camera_util.cc files are part of the camera utility module. Utility functions and communication with external libraries or modules are handled by these files. This might be viewed as an interfacing layer or utility within the Cameras Subsystem due to their substantial interaction with external components (24 inputs from.h and.cc combined).



Camera QCOM2 Module: This module, which is exclusive to Qualcomm hardware (probably, QCOM2 refers to a Qualcomm chipset or camera specification), is composed of `camera_qcom2.cc` and `camera_qcom2.h`. It interacts with the common camera functionalities and has a sizable number of third-party inputs (181 from.cc), suggesting that it plays a part in customizing or improving the common functionality for hardware that is special to Qualcomm.



Camera Common Module: which includes `camera_common.h` and `camera_common.cc`, is the central component of the Cameras Subsystem. It receives 282 + 1 (from selfdrive) + 8 (from tools) + inputs from `camera_util.cc`, `camera_qcom2.cc`, and `camera_qcom2.h`. It interfaces directly with other components of the subsystem. It's likely that the basic camera functions and utilities provided by this module are improved upon or used by other modules.

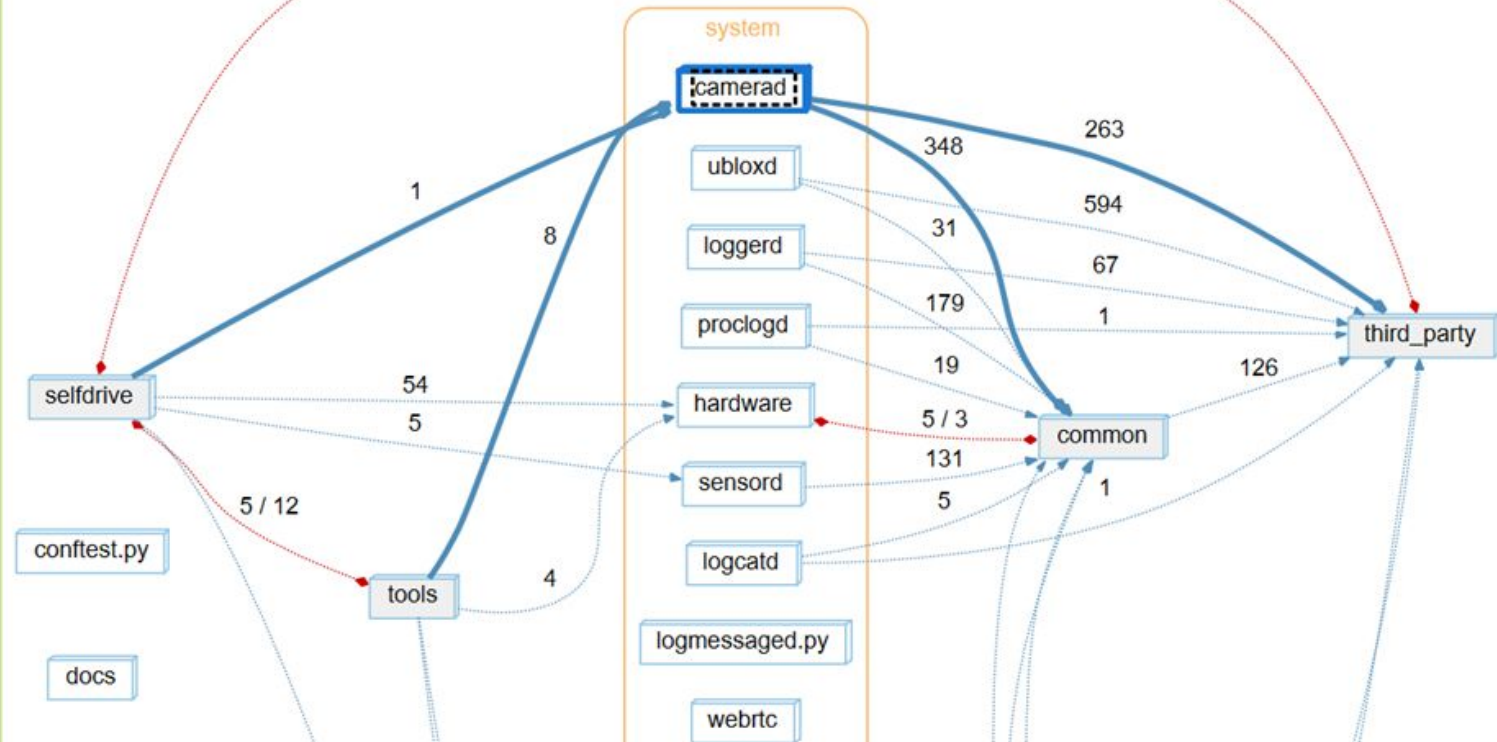


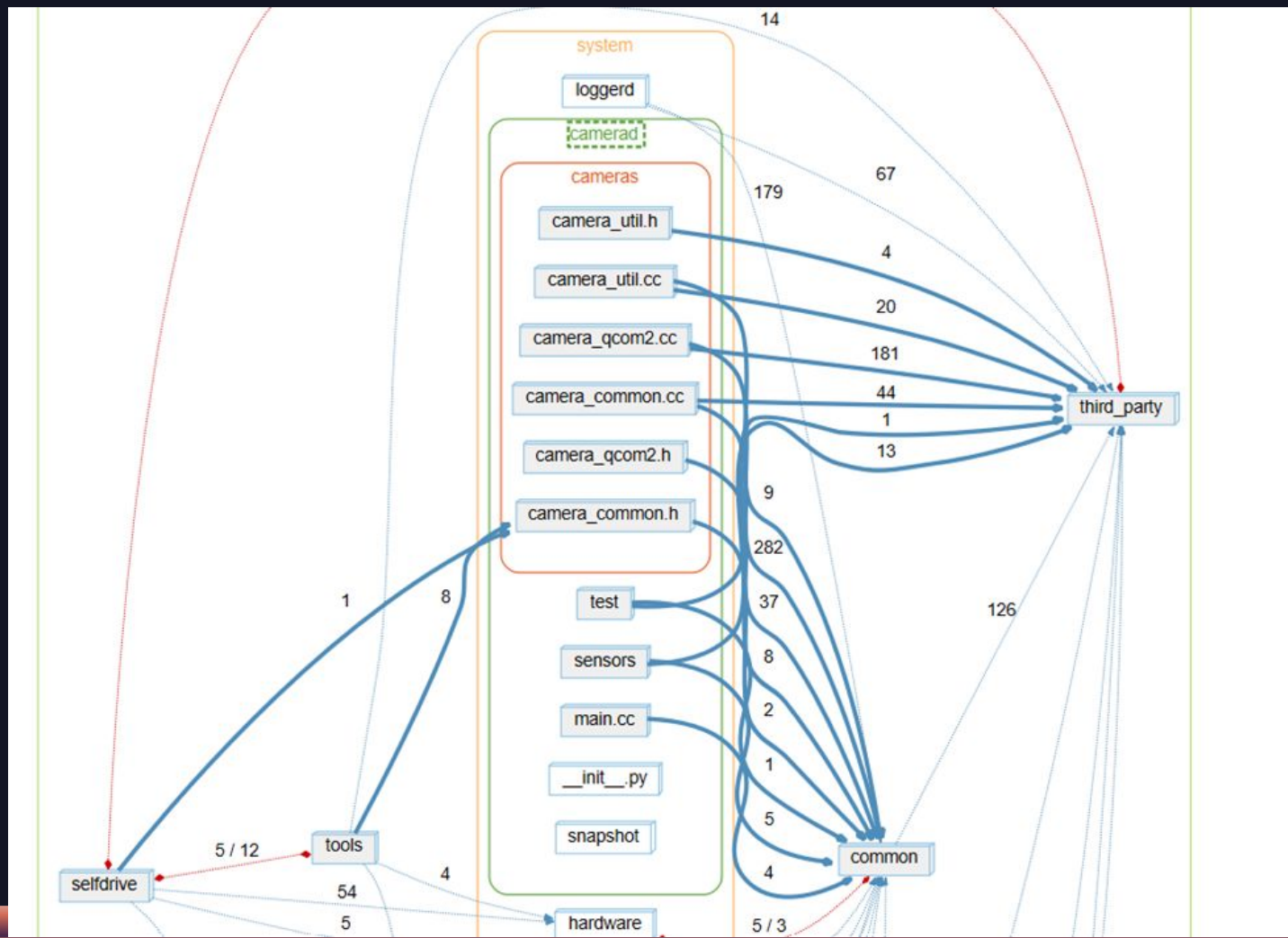
External Interactions

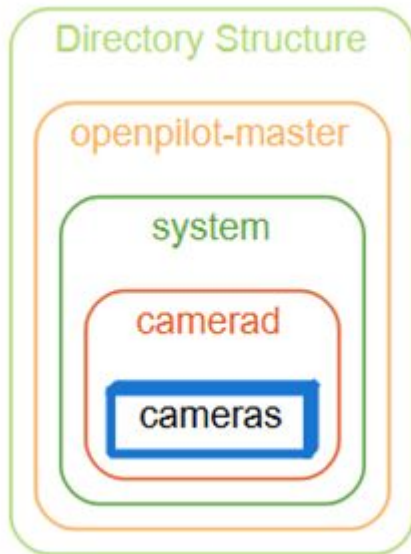
- a. **Tools and selfdrive inputs:** According to these inputs into `camera_common.h`, the Camera Common Module interacts with both driving features and development tools to function as a hub for camera-related operations in the larger OpenPilot system.
- b. **Interactions with third parties:** A number of modules have interactions with libraries or modules from third parties, which suggests external dependencies or integrations with the OpenPilot project. Codecs, drivers, and other libraries for multimedia processing may be examples of this.

openpilot-master

403 / 19



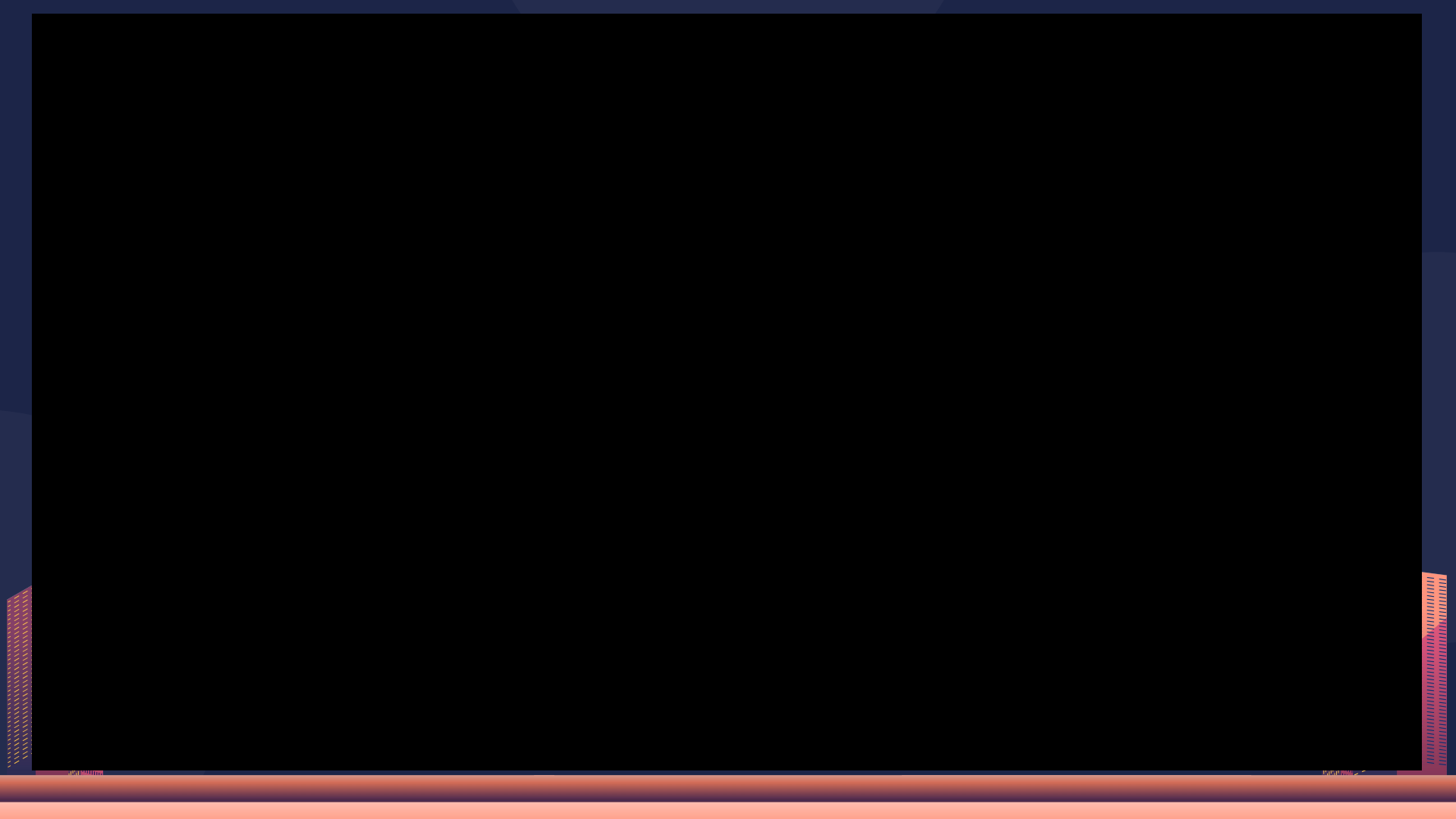




04

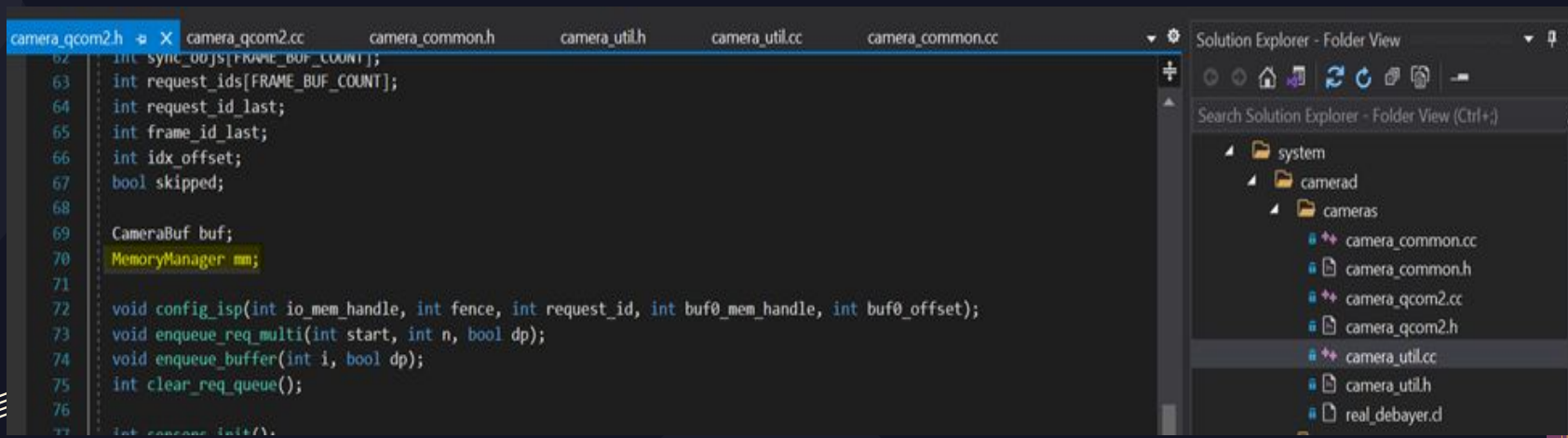
Design Patterns





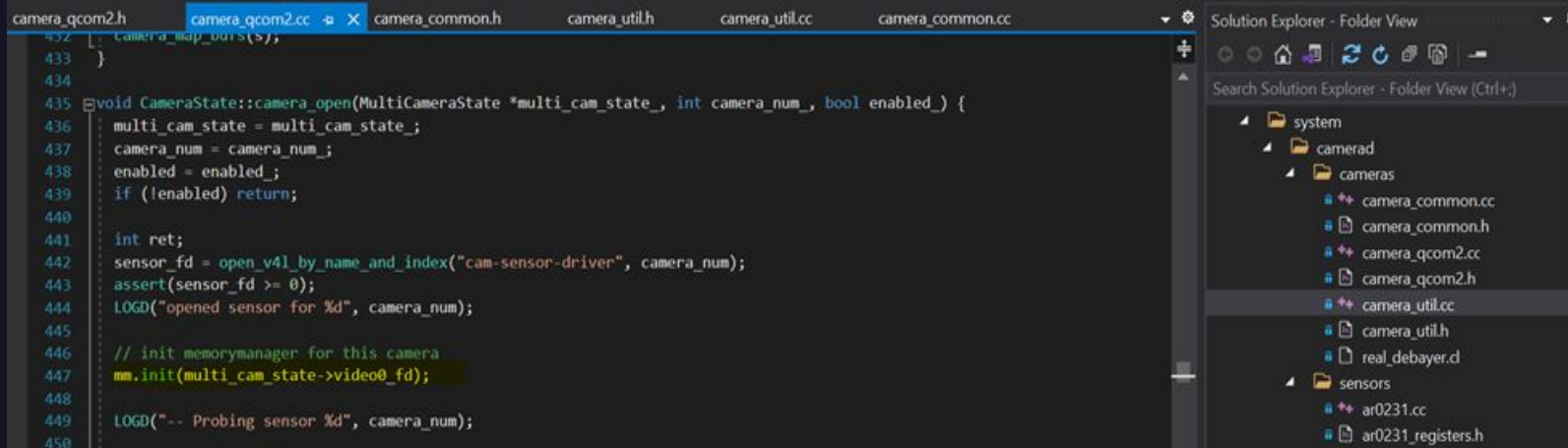
Singleton pattern

- The camera system utilizes the Singleton pattern, ensuring that only one instance of the MemoryManager class exists throughout the application lifecycle
- This approach guarantees global access to a single, shared instance, maintaining consistency and preventing unnecessary resource duplication



Singleton pattern

- The MemoryManager class (mm) serves as a central component of the system, managing memory-related operations
- The constructor of the MemoryManager class in camera_util.h is not public, indicating that instantiation is restricted to within the class itself
- By restricting instantiation to a single instance, the application promotes efficient resource utilization and simplifies memory management tasks
- This encapsulation reinforces the Singleton patterns intent, emphasizing that only one instance should be created and accessed globally

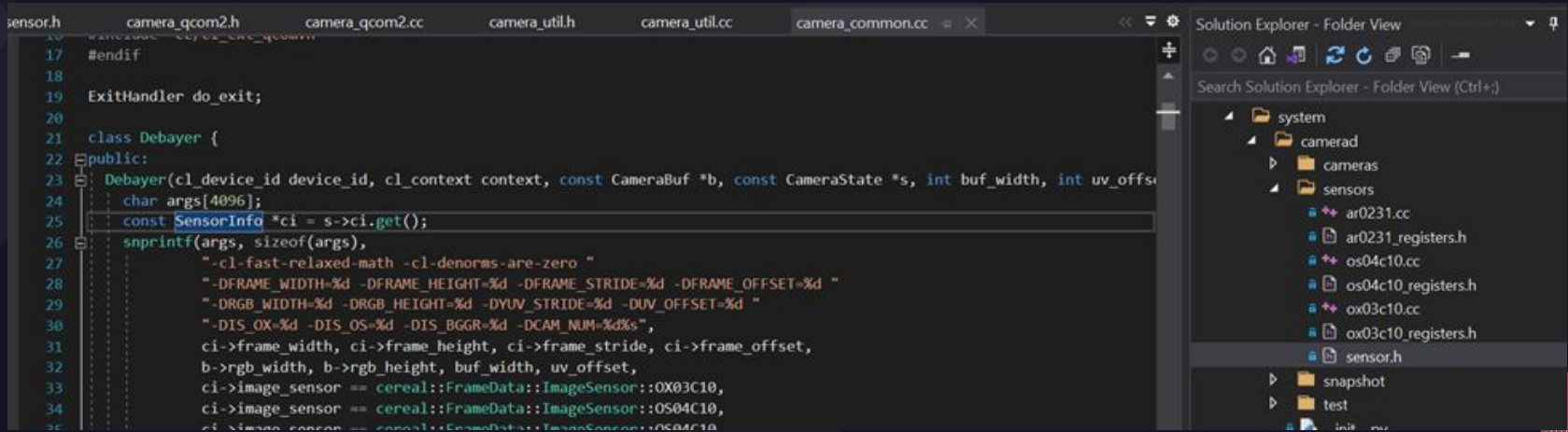


Singleton pattern

- Other parts of the code interact with the MemoryManager instance through a global variable (mm)
- This centralized access point facilitates seamless communication and coordination between different components of the system, promoting cohesion and reducing coupling
- By enforcing a single instance of the MemoryManager class, the application maintains consistency in memory management operations and ensures that changes to memory-related functionality are applied universally
- This centralized control simplifies debugging, optimization, and maintenance efforts, enhancing the overall robustness and reliability of the system

Factory Pattern

- The SensorInfo class encapsulates common functionality for various sensor information objects, providing a clear and concise interface
- Its abstract nature allows for polymorphic behavior, enabling the creation of concrete implementations tailored to specific sensor models



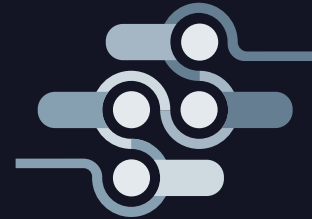
Factory Pattern

- Concrete subclasses such as AR0231, OX03C10, and OS04C10 offer specific implementations of sensor information, each catering to the unique characteristics of different sensor models. This approach ensures modularity and reusability of code components

```
63 class AR0231 : public SensorInfo {
64 public:
65     AR0231();
66     std::vector<i2c_random_wr_payload> getExposureRegisters(int exposure_time, int new_exp_g, bool dc_gain_enabled) const override;
67     float getExposureScore(float desired_ev, int exp_t, int exp_g_idx, float exp_gain, int gain_idx) const override;
68     int getSlaveAddress(int port) const override;
69     void processRegisters(CameraState *c, cereal::FrameData::Builder &framed) const override;
70
71 private:
72     mutable std::map<uint16_t, std::pair<int, int>> ar0231_register_lut;
73 };
74
75 class OX03C10 : public SensorInfo {
76 public:
77     OX03C10();
78     std::vector<i2c_random_wr_payload> getExposureRegisters(int exposure_time, int new_exp_g, bool dc_gain_enabled) const override;
79     float getExposureScore(float desired_ev, int exp_t, int exp_g_idx, float exp_gain, int gain_idx) const override;
80     int getSlaveAddress(int port) const override;
81 };
82
83 class OS04C10 : public SensorInfo {
84 public:
85     OS04C10();
86     std::vector<i2c_random_wr_payload> getExposureRegisters(int exposure_time, int new_exp_g, bool dc_gain_enabled) const override;
87     float getExposureScore(float desired_ev, int exp_t, int exp_g_idx, float exp_gain, int gain_idx) const override;
88     int getSlaveAddress(int port) const override;
89 };
```

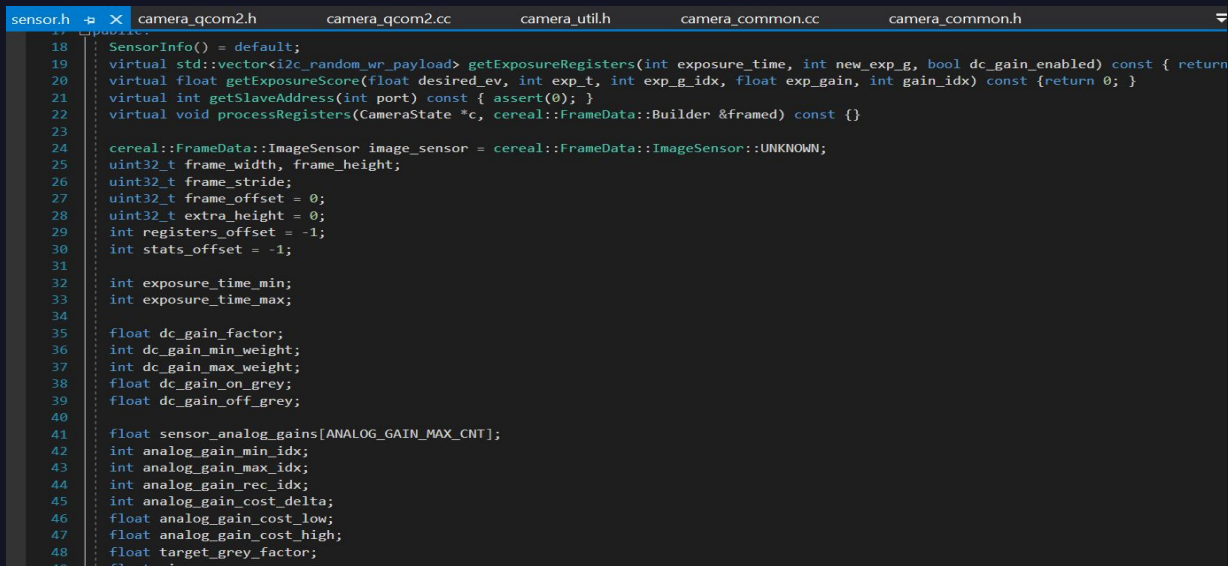
Factory Pattern

- The factory aspect of the design pattern comes into play when instances of SensorInfo or its subclasses are created based on runtime conditions or configurations. By abstracting the object creation process, the system achieves flexibility and scalability, facilitating the addition of new sensor types in the future without requiring modifications to existing client code
- Client code interacts with the SensorInfo interface without needing to be aware of the specific subclass being used. This decoupling enhances code maintainability and simplifies future enhancements or modifications, as changes to concrete implementations can be isolated without impacting other parts of the system.



Factory Pattern

- The architecture allows for seamless extension and modification of the system to accommodate new sensor types. Adding support for additional sensor models simply involves implementing new subclasses of SensorInfo and updating the factory logic, ensuring minimal disruption to existing functionality and promoting code longevity.



```
18 SensorInfo() = default;
19 virtual std::vector<i2c_random_wr_payload> getExposureRegisters(int exposure_time, int new_exp_g, bool dc_gain_enabled) const { return
20 virtual float getExposureScore(float desired_ev, int exp_t, int exp_g_idx, float exp_gain, int gain_idx) const {return 0; }
21 virtual int getSlaveAddress(int port) const { assert(0); }
22 virtual void processRegisters(CameraState *c, cereal::FrameData::Builder &framed) const {}
23
24 cereal::FrameData::ImageSensor image_sensor = cereal::FrameData::ImageSensor::UNKNOWN;
25 uint32_t frame_width, frame_height;
26 uint32_t frame_stride;
27 uint32_t frame_offset = 0;
28 uint32_t extra_height = 0;
29 int registers_offset = -1;
30 int stats_offset = -1;
31
32 int exposure_time_min;
33 int exposure_time_max;
34
35 float dc_gain_factor;
36 int dc_gain_min_weight;
37 int dc_gain_max_weight;
38 float dc_gain_on_grey;
39 float dc_gain_off_grey;
40
41 float sensor_analog_gains[ANALOG_GAIN_MAX_CNT];
42 int analog_gain_min_idx;
43 int analog_gain_max_idx;
44 int analog_gain_rec_idx;
45 int analog_gain_cost_delta;
46 float analog_gain_cost_low;
47 float analog_gain_cost_high;
48 float target_grey_factor;
```

05

Diagrams



Use-Case Diagram

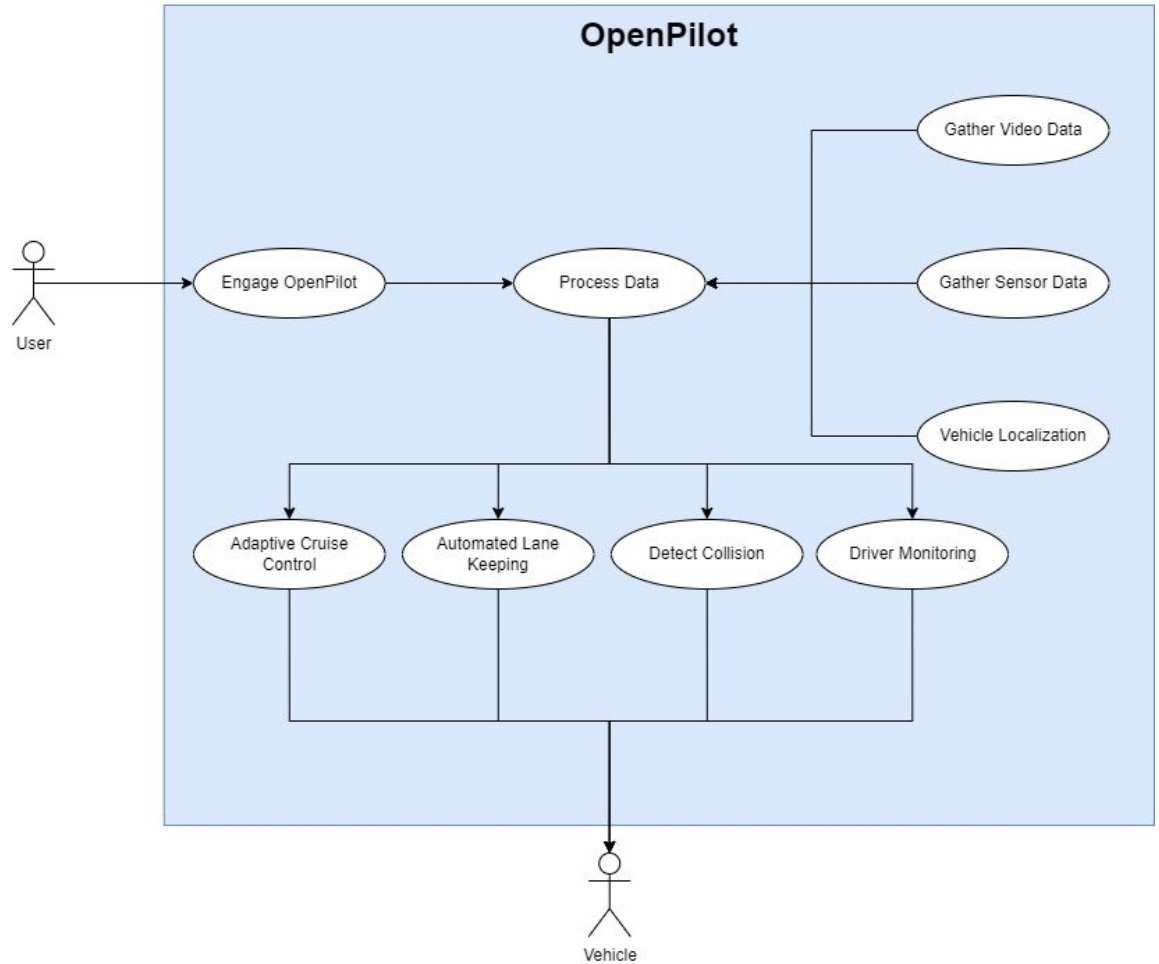
LEGEND



Actor



Use Case



Sequence Diagram

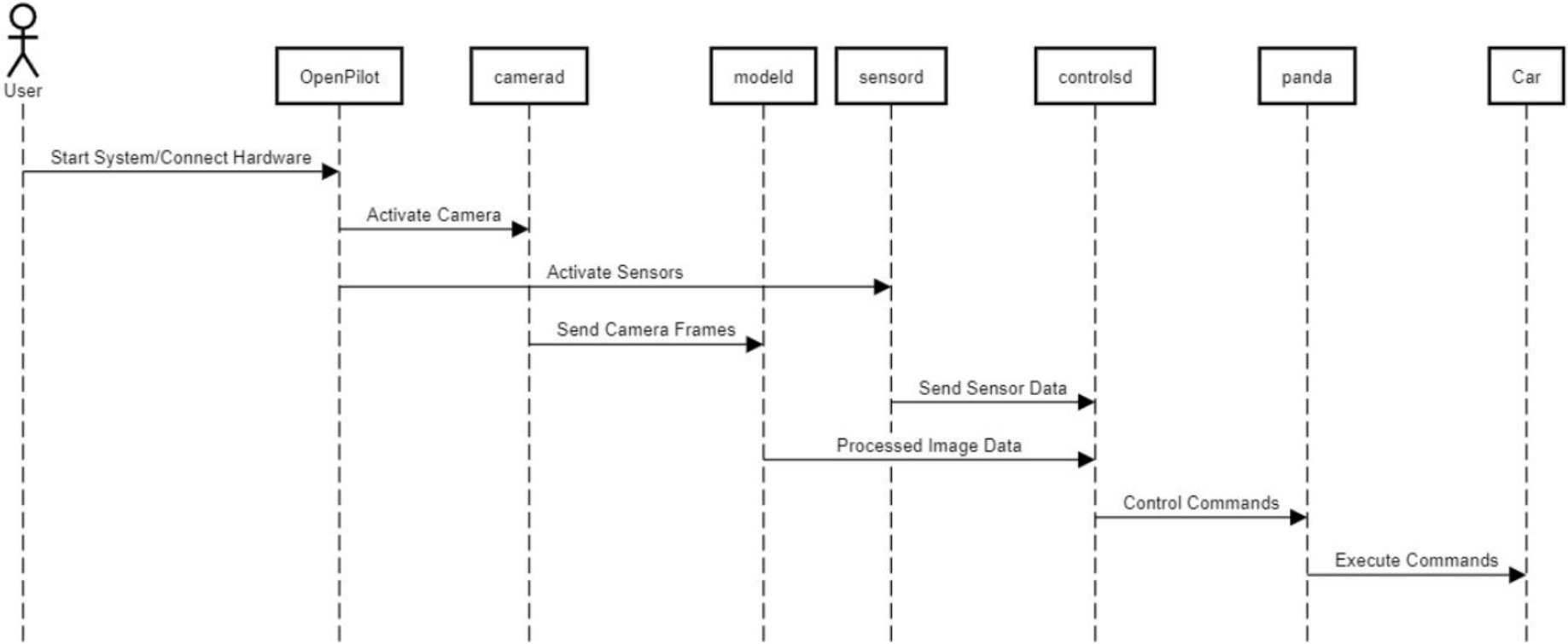
LEGEND



Actor



Object
Lifeline



State Diagram: Camerad

The camerad subsystem operates in a cycle to manage the vehicle's camera inputs.

LEGEND



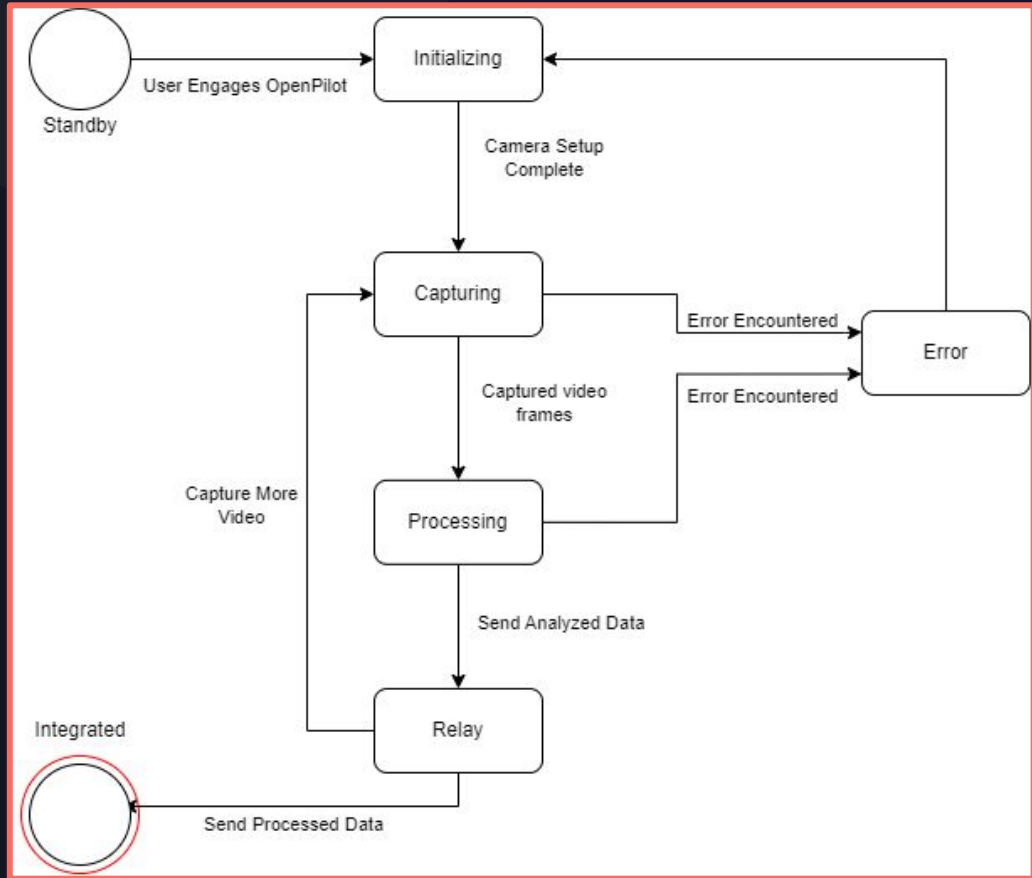
Start State



Final State






Simple State

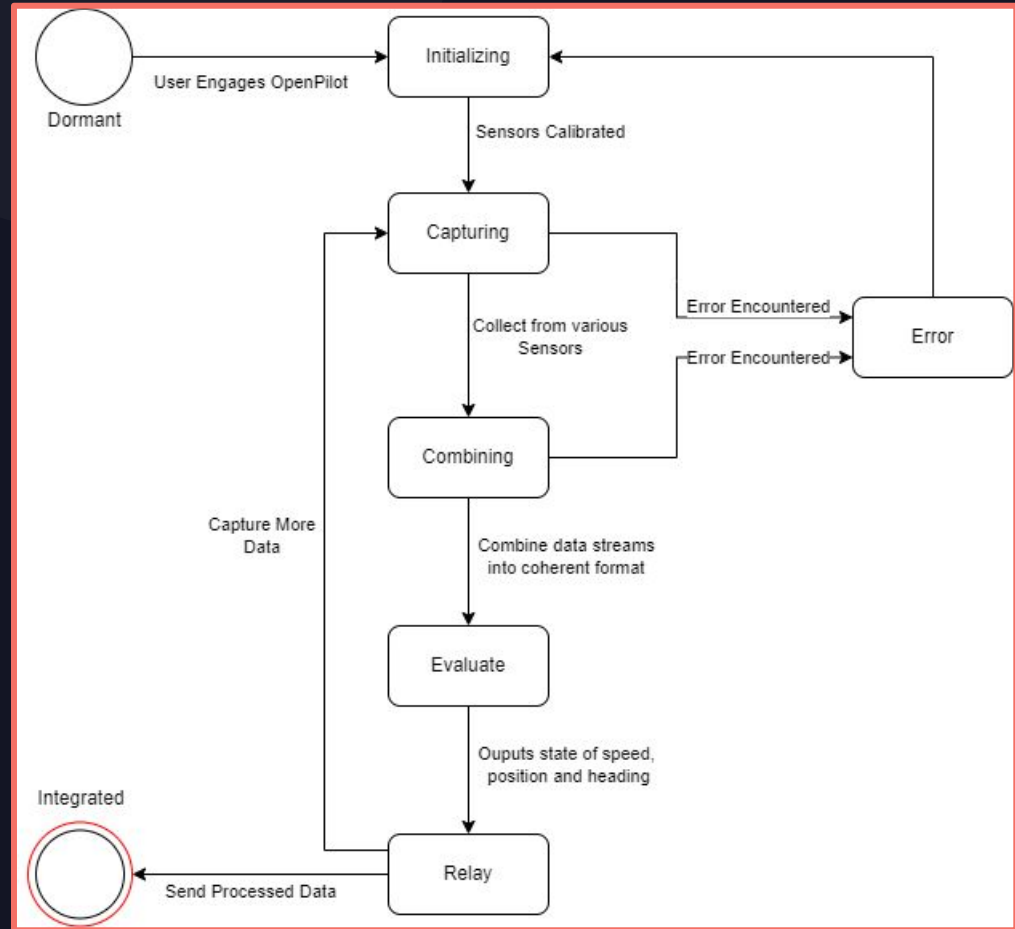


State Diagram: Sensord

The sensord subsystem actively collects data from various sensors. This data is combined into a coherent format.

LEGEND

-  Start State
-  Final State
-  Simple State



06

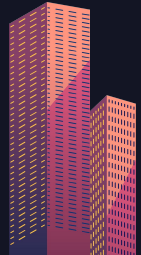
Reflexion analysis



1. Event-Driven Architecture (Conceptual) vs. Message Passing and InterModule Communication (Concrete):

Conceptually, OpenPilot uses an event-driven architecture, where different components communicate and coordinate actions through events. This allows the system to be responsive and modular.

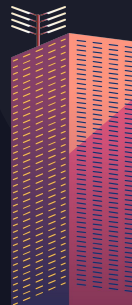
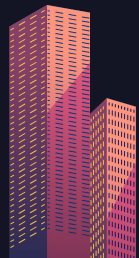
In the concrete implementation, this is visible in how different modules in OpenPilot communicate through event signaling/message passing.



2. Process Control Architecture (Conceptual) vs. Real-Time Monitoring and Control Loops (Concrete):

At the conceptual level, OpenPilot follows a process control architecture, continuously monitoring inputs and adjusting outputs (vehicle controls) in a feedback loop.

Looking at the repository, the concrete implementations of this in the form of real-time data processing, control algorithms, and feedback mechanisms.



11

Conclusion



- **Achievements:** Successful organization of the Camerad and Cameras subsystems into a cohesive architectural framework that enhances OpenPilot's camera management capabilities.
- **Design Efficiency:** Implementation of the Singleton pattern for the MemoryManager class has streamlined memory management across the subsystem, demonstrating effective resource utilization.
- **Flexibility and Scalability:** The SensorInfo class and its subclasses illustrate the system's adaptability to accommodate various sensor models, ensuring the subsystem's longevity and ease of integration.
- **Future Directions:** The modular and scalable nature of the subsystem architecture lays a solid foundation for future enhancements. The system is poised for seamless integration of new camera technologies and sensor types, facilitating continuous improvement of the OpenPilot project.
- **Closing Remark:** Through strategic subsystem organization and adherence to proven design principles, the Camerad subsystem stands as a testament to the robust, flexible, and scalable architecture driving OpenPilot's success in autonomous vehicle technology.