Team Flame Code Overview (Jan. 24, 2026)

Table of Contents

Code Structure Summary

Architecture Overview

Team Flame's robot code is built on an **object-oriented architecture** that separates concerns into logical components. The design emphasizes **modularity**, **reusability**, and **maintainability** through inheritance-based patterns and clear separation of hardware abstraction from mission logic.

Core Design Principles

- **Inheritance-Based Design:** Base classes (`Attachment` and `Run`) define common behavior, while subclasses implement mission-specific logic
- **Single Robot Instance:** One `Robot` object manages all hardware interactions (motors, sensors, drive base)
- **Color-Based Run Selection:** Color sensor reads cartridges to automatically select and execute the appropriate mission run
- **Motor Configuration System:** Attachments use declarative motor dictionaries for easy configuration and automatic initialization
- **Stall Detection:** Built-in motor stall detection prevents damage and enables robust operation
- **Hardware Abstraction:** Complex Pybricks API calls are wrapped in simple, descriptive methods
- **Arc Movement:** Curved movement capabilities (`arc_left` , `arc_right`) enable smooth navigation around obstacles

Component Overview

Component	Purpose	Key Responsibilities
Robot	Hardware abstraction layer	Movement (straight, turning, arcs), sensor reading, system initialization, battery/IMU checks
Attachment	Base class for robot attachments	Motor management, stall detection, common attachment operations
Run	Base class for mission runs	Mission execution, debugging, logging, flag management
Runner	Run orchestration	Color detection, run selection, execution management, error handling

Class Hierarchy Summary

- **Robot:** Inherits from PrimeHub (Pybricks), manages DriveBase, ColorSensor, and Motors
- **Attachments (5 types):**
  - `BrushAndReveal` - Surface brushing and map reveal missions
  - `ShipDropper` - Flag deployment for salvage operation
  - `ForgeAndSilo` - Hammer and arm mechanisms for forge/silo missions
  - `MineshaftAttachment` - Grabber and arm for mineshaft retrieval
  - `TesterAttachment` - Development and testing attachment
- **Runs (9 types):**
  - `SalvageOperation` - Run 1: Mission 12 (ship flag deployment)
  - `SurfaceBrushing` - Run 2: Missions 1, 2 (brush collection and map reveal)
  - `MineshaftRetrieval_Old` - Run 3 (old implementation): Missions 8, 9, 10 (mineshaft, tip the scales, what's on sale) - currently registered in main()
  - `MineshaftRetrieval` - Run 3 (new implementation): Missions 8, 9, 10 (mineshaft, tip the scales, what's on sale) - alternative implementation
  - `Silo` - Run 4: Missions 5, 6, 7 (silo, forge, heavy lifting)
  - `CrossTable` - Run 5: Cross-table navigation
  - `Forum` - Run 6: Mission 14 (forum delivery)
  - `Test` - Development and debugging
  - `Demo` - Demonstration and testing

Code Organization Strategy

Key Techniques and Patterns

Object-Oriented Design Patterns

- **Template Method Pattern:** Base classes (`Attachment` , `Run`) define the structure, subclasses provide specific implementations
- **Strategy Pattern:** Different run strategies are encapsulated in separate classes, selected at runtime based on color sensor input

- **Facade Pattern:** `Robot` class provides simplified interface to complex hardware operations
- **Composition:** Robot contains DriveBase, Motors, and ColorSensor rather than inheriting from them
- **Factory Pattern:** Motor initialization from configuration dictionaries in Attachment subclasses

Code Organization Techniques

- **Declarative Motor Configuration:** Each attachment defines motors in a class-level dictionary with port, gear ratios, and direction settings
- **Automatic Motor Initialization:** `Attachment._init_motors()` automatically converts configuration dictionaries to Motor instances
- **Stall Detection:** All motor movements monitor for stall conditions and gracefully stop to prevent damage
- **Hardware Abstraction:** All Pybricks API calls are wrapped in `Robot` methods with descriptive names (`move_forward` , `turn_left` , `arc_left` , `arc_right` , etc.)
- **Color Calibration:** Custom HSV color definitions ensure accurate color detection in competition lighting conditions
- **Safety Checks:** Battery voltage validation and IMU readiness checks prevent execution with insufficient power or uncalibrated sensors
- **Debug Logging:** Toggle-able logging in `Run` base class for development and troubleshooting
- **Error Handling:** Comprehensive error handling for missing runs, invalid motor configurations, and sensor failures

Mission Run Architecture

Each mission run follows a consistent pattern:

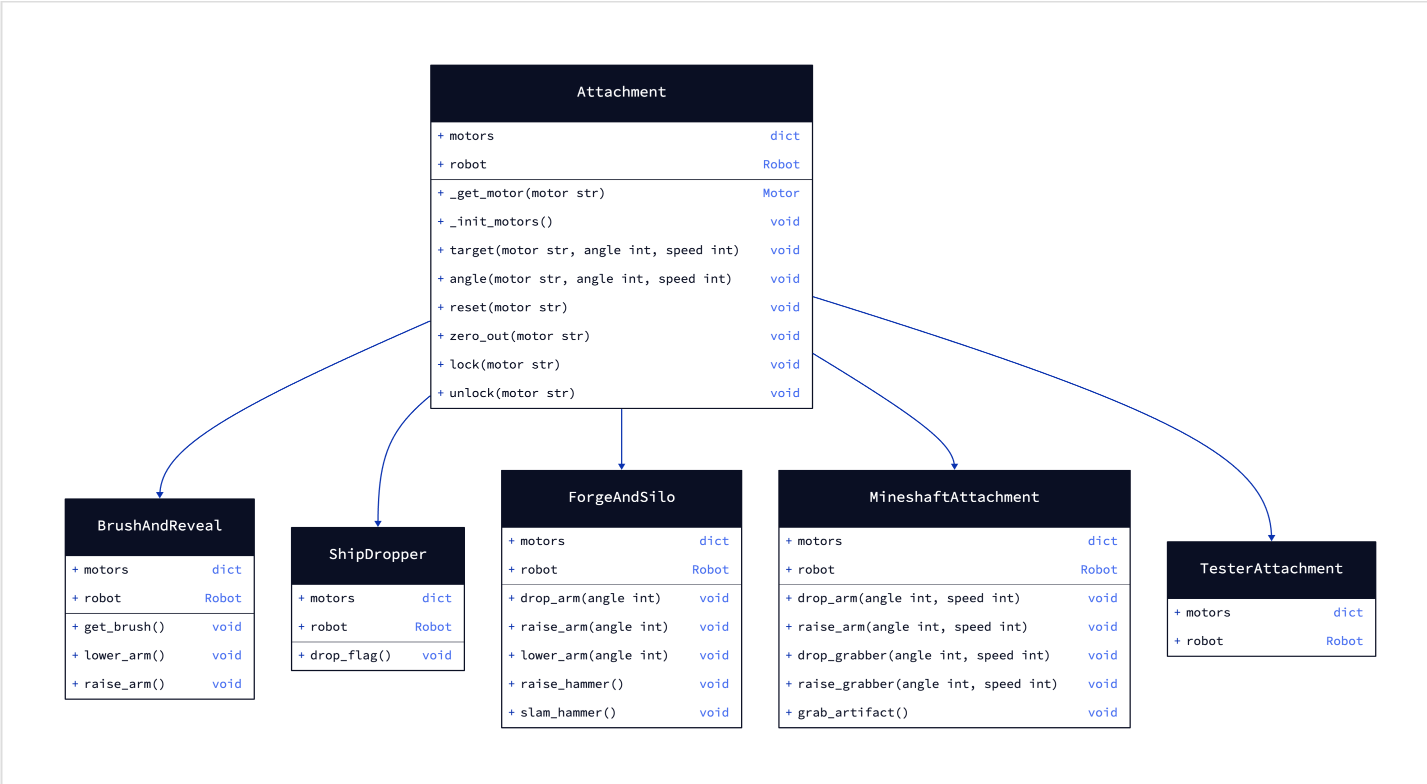
1. Color cartridge is placed on sensor
2. `Runner.begin()` reads color and matches to run class
3. Selected run's `execute()` method is called
4. Run instantiates required attachments
5. Run performs movement and attachment operations
6. Completion is signaled and robot returns to ready state

UML Diagrams

Core Classes

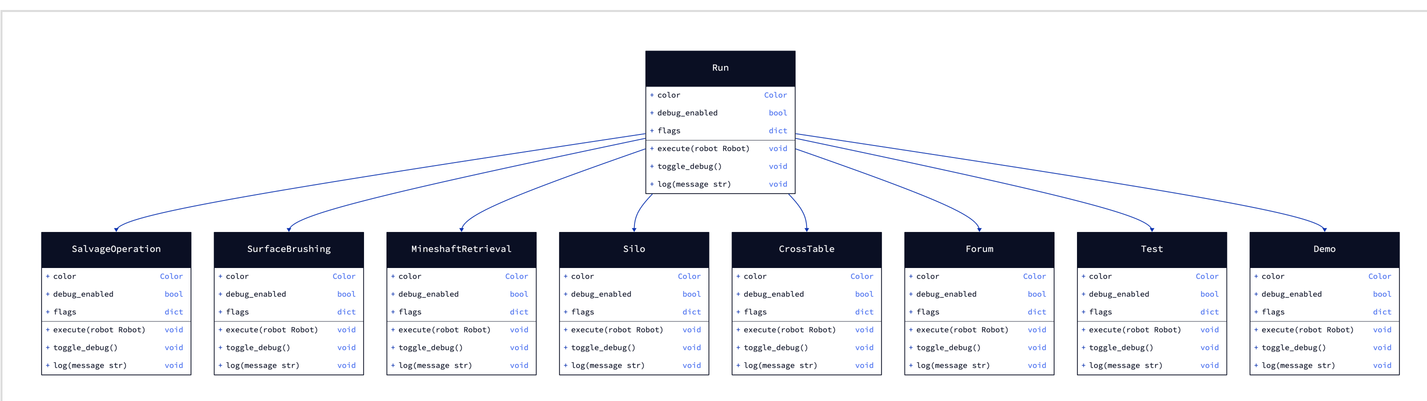
This diagram shows the foundational classes of our robot system. The `Robot` class acts as the central hub, containing the drive base, color sensor, and managing motor operations. The Robot class provides comprehensive movement capabilities including straight movement, turning, and arc-based navigation. Both `Attachment` and `Run` classes depend on the robot instance to perform their operations. The `Runner` class orchestrates the execution of runs based on color sensor input, implementing the strategy pattern for mission selection.

Attachments Overview



All attachments inherit from the base `Attachment` class, which provides motor management, stall detection, and common operations. Each specialized attachment adds mission-specific methods while leveraging the base class's motor configuration and control capabilities. The system supports five distinct attachment types for different mission requirements.

Runs Overview



Each mission run inherits from the base `Run` class and implements the `execute()` method with mission-specific logic. Color attributes enable automatic run selection by the `Runner` class. The system currently supports nine different run types: SalvageOperation (Mission 12), SurfaceBrushing (Missions 1-2), MineshaftRetrieval_Old (Missions 8-10, currently registered), MineshaftRetrieval (Missions 8-10, alternative implementation), Silo (Missions 5-7), CrossTable (cross-table navigation), Forum (Mission 14), Test (development), and Demo (demonstration). Runs are executed in priority order based on color detection.