

2025-11-01

E030: Controller Base Classes & Factory

**Understanding the Foundation: How
5 Controllers Share One Interface**

Part 5A · Duration: 25-30 minutes

Beginner-Friendly Visual Study Guide

⌚ **Learning Objective:** Understand the abstract base class that all controllers inherit from, the factory pattern for creating controllers, and how this enables seamless controller swapping

The Design Challenge

💡 Key Concept

One Interface, Five Controllers: All controllers (Classical SMC, STA, Adaptive, Hybrid Adaptive STA, Conditional Hybrid) implement the SAME interface. The factory pattern makes adding new controllers (e.g., Swing-Up, MPC) trivial - just register them!

Result: Change one line in config.yaml and swap algorithms without touching code!

Why This Matters

⚠ Common Pitfall

Without Interface:

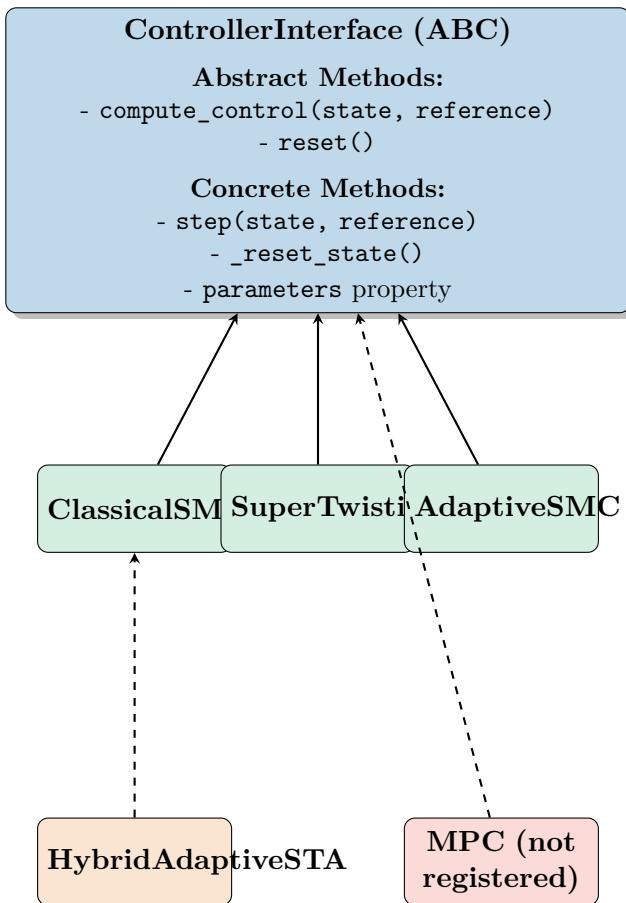
- Each controller has different method names
- Simulation code full of if/else chains
- Adding new controller = rewrite everything
- Testing = nightmare (5 different APIs)

💡 Pro Tip

With Interface:

- One method: `compute_control()`
- Simulation code agnostic to controller type
- New controller = implement interface
- Testing = same harness for all 5

The ControllerInterface Abstract Base Class



Example

Python ABC Pattern: Abstract Base Class (ABC) defines a contract. Subclasses MUST implement abstract methods or Python raises TypeError at instantiation.

Core Interface Code (src/controllers/base/controller_interface.py)

```

from abc import ABC, abstractmethod
from typing import Optional, Tuple, Any
import numpy as np

class ControllerInterface(ABC):
    """Abstract base class for all controllers in the DIP system."""
    def __init__(self, max_force: float = 20.0, dt: float = 0.01):
        """Initialize base controller with common parameters."""
        self.max_force = max_force # Actuator saturation limit (N)
        self.dt = dt # Sampling timestep (s)
        self._reset_state()

    @abstractmethod
    def compute_control(self, state: np.ndarray,
                        reference: Optional[np.ndarray] = None) -> float:
        """THE KEY METHOD - Compute control force for given state.
        Args:
            state: [x, xdot, theta1, thetadot1, theta2, thetadot2]
            reference: Target state (default: upright equilibrium)
        Returns:
            float: Control force to apply to cart (N)
        """
        pass # Subclasses MUST implement

    @abstractmethod
    def reset(self) -> None:
        """Reset controller internal state (for multi-simulation)."""
        pass

    def step(self, state: np.ndarray,
             reference: Optional[np.ndarray] = None) -> Tuple[float, Any]:
        """Perform one control step with saturation."""
        control = self.compute_control(state, reference)

        # Apply actuator limits (CRITICAL for real hardware!)
        control = np.clip(control, -self.max_force, self.max_force)

        # Return control + diagnostics
        info = f'saturated': bool(abs(control) >= self.max_force),
        return control, info
  
```

```
lstnū0er         'control_raw': control}
lstnū0er     return control, info
lstnū0er
lstnū0er
lstnū0er     @property
lstnū0er     def parameters(self) -> dict:
lstnū0er         """Get controller parameters for logging/analysis."""
lstnū0er         return {'max_force': self.max_force, 'dt': self.dt}
```

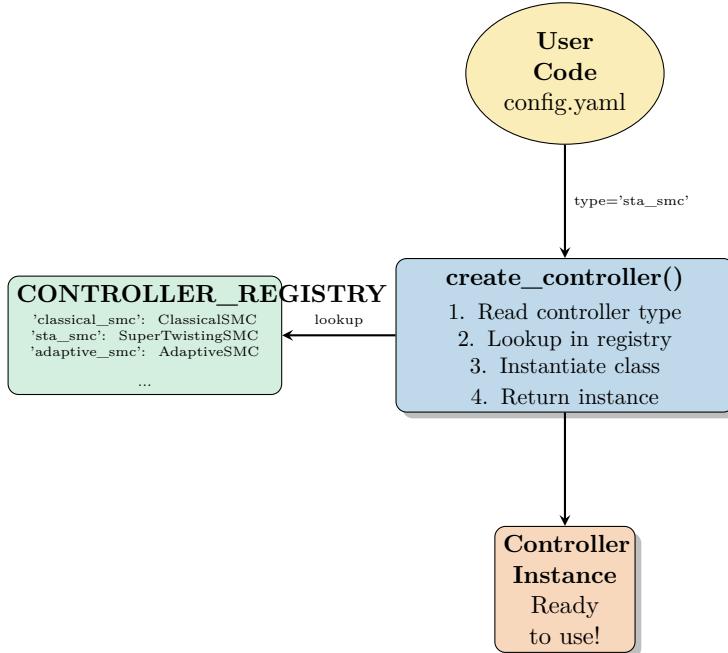
Factory Pattern: One Function to Rule Them All

Key Concept

Problem: How do you create 5 controller types from configuration without massive if/else chains?

Solution: Factory pattern - one function `create_controller()` that uses a registry to instantiate the right class.

Factory Pattern Flow



Factory Code (src/controllers/factory/base.py)

```

# Simplified version for clarity (verified 2026-02-04)
CONTROLLER_REGISTRY = {
    'classical_sm': ClassicalSMC,
    'sta_sm': SuperTwistingSMC,
    'adaptive_sm': AdaptiveSMC,
    'hybrid_adaptive_stasmc': HybridAdaptiveSTASMC,
    'conditional_hybrid': ConditionalHybrid,
    # Note: SwingUp and MPC not yet registered (can be added by extending registry)
}

def create_controller(ctrl_type: str, config: dict, gains: list) -> ControllerInterface:
    """Factory function to create any controller type.

    Args:
        ctrl_type: Controller identifier (e.g., 'sta_sm')
        config: Configuration dictionary
        gains: Controller gains (validated before instantiation)

    Returns:
        Controller instance implementing ControllerInterface

    Raises:
        ValueError: Unknown controller type
    """
    # Canonicalize type (handle aliases)
    ctrl_type = canonicalize_controller_type(ctrl_type)

    # Lookup controller class in registry
    if ctrl_type not in CONTROLLER_REGISTRY:
        raise ValueError(f"Unknown controller: {ctrl_type}")

    controller_class = CONTROLLER_REGISTRY[ctrl_type]

    # Instantiate controller with validated parameters
    return controller_class(gains=gains, **config)
  
```

Registry Benefits

- **No if/else chains:** Dictionary lookup = O(1)
- **Easy to extend:** Add new controller = register it
- **Type-safe:** All values implement ControllerInterface

- **Discoverable:** List available controllers programmatically

```
lstnumberCONTROLLER_ALIASES = {
lstnumber    'classical': 'classical_smc',
lstnumber    'sta': 'sta_smc',
lstnumber    'super_twisting': 'sta_smc',
lstnumber    'adaptive': 'adaptive_smc',
lstnumber    # User-friendly names
lstnumber}
```

Usage Example: Swapping Controllers

Python Usage

```
lstnumberLoad configuration config = loadconfig("config.yaml")
lstnumberCreate controller (type from config, NOT hardcoded!) controller = create_controller(ctrltype = config[
lstnumber'controller_type'], 'sta_smc', config = config['controller_params'], gains = config['controller_gains'])
lstnumberSimulation loop - controller type doesn't matter here! for t in np.arange(0, 10, dt):
lstnumber    state = get_current_state()
lstnumber    control = controller.compute_control(state)
lstnumber    apply_control(control)
lstnumberWant to test different algorithm? Change ONE line in config.yaml!
```

Memory Management: Breaking Circular References

⚠ Common Pitfall

The Circular Reference Problem:

Controller → holds reference to → Dynamics Model

Dynamics Model → sometimes holds reference to → Controller

Result: Python garbage collector can't free memory (memory leak!)

Solution: Weakref Pattern

Bad (Strong Reference):

```
lstnumberclass ClassicalSMC:
lstnumber    def __init__(self, dynamics_model):
lstnumber        # Strong reference
lstnumber        self.dyn = dynamics_model
```

Good (Weak Reference):

```
lstnumberimport weakref
lstnumber
lstnumberclass ClassicalSMC:
lstnumber    def __init__(self, dynamics_model):
lstnumber        # Weak reference
lstnumber        self._dynamics_ref = weakref.ref(dynamics_model)
lstnumber
lstnumber    @property
lstnumber    def dyn(self):
lstnumber        return self._dynamics_ref()
```

Problem: If `dynamics_model` holds controller, both objects never freed!

Weak reference doesn't prevent garbage collection!

Cleanup Pattern

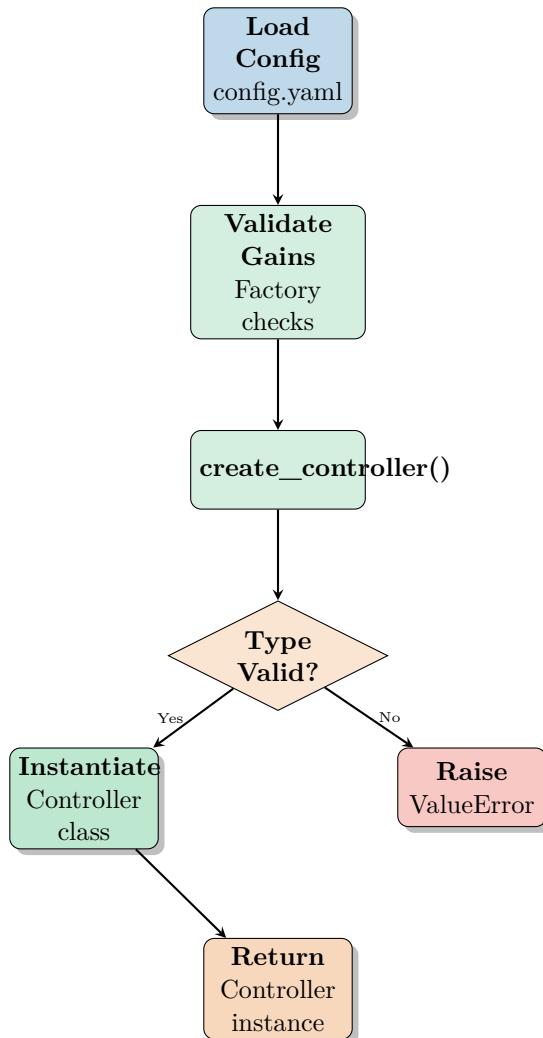
```
lstn0erclass ClassicalSMC(ControllerInterface):
lstn0er    def cleanup(self) -> None:
lstn0er        """Explicit memory cleanup to prevent leaks."""
lstn0er        # Nullify dynamics reference
lstn0er        if hasattr(self, '_dynamics_ref'):
lstn0er            self._dynamics_ref = lambda: None
lstn0er
lstn0er        # Clear large NumPy arrays
lstn0er        if hasattr(self, 'L'):
lstn0er            self.L = None
lstn0er        if hasattr(self, 'B'):
lstn0er            self.B = None
lstn0er
lstn0er    def __del__(self) -> None:
lstn0er        """Destructor for automatic cleanup."""
lstn0er        try:
lstn0er            self.cleanup()
lstn0er        except Exception:
lstn0er            pass # Prevent exceptions during finalization
```

💡 Pro Tip

Memory Management Guideline: Call `cleanup()` when done with controller, especially in batch simulations or PSO optimization (1000s of instantiations).

The `__del__()` destructor provides automatic cleanup, but explicit `cleanup()` is more reliable.

Controller Initialization Flow



Design Patterns Identified

Four Key Patterns

enumiAbstract Base Class (ABC): Enforces contract via Python's `@abstractmethod`

- 0. **enumiFactory Pattern:** Registry-based instantiation decouples creation from usage
- 0. **enumiWeak Reference Pattern:** Prevents memory leaks from circular references
- 0. **enumiStrategy Pattern:** Controllers are interchangeable strategies for same problem

Practical Examples: Using the Factory

Example 1: Command-Line Simulation

```

lstn0er# File: simulate.py
lstn0erimport argparse
lstn0erfrom src.controllers.factory import create_controller
lstn0erfrom src.config import load_config
lstn0er
lstn0erdef main():
lstn0er    parser = argparse.ArgumentParser()
lstn0er    parser.add_argument('--ctrl', default='classical_smc',
lstn0er                help='Controller type')
lstn0er    args = parser.parse_args()
lstn0er
lstn0er    # Load config
lstn0er    config = load_config("config.yaml")
lstn0er
lstn0er    # Create controller from command-line argument
lstn0er    controller = create_controller(
lstn0er        ctrl_type=args.ctrl, # User-specified type!
lstn0er        config=config['controllers'][args.ctrl],
lstn0er        gains=config['gains'][args.ctrl]
lstn0er    )
lstn0er
lstn0er    # Run simulation (same code for ALL controllers)
lstn0er    results = simulate(controller, initial_state, dt=0.01, duration=10.0)
lstn0er    plot_results(results)
lstn0er
lstn0er# Usage:
lstn0er# python simulate.py --ctrl classical_smc
lstn0er# python simulate.py --ctrl sta_smc
lstn0er# python simulate.py --ctrl adaptive_smc

```

Example 2: Batch Comparison

```

lstn0erfrom src.controllers.factory import list_available_controllers, create_controller
lstn0er
lstn0er# Discover all available controllers programmatically
lstn0erall_controllers = list_available_controllers()
lstn0er# Returns: ['classical_smc', 'sta_smc', 'adaptive_smc', ...]
lstn0er
lstn0erresults = {}
lstn0erfor ctrl_type in all_controllers:
lstn0er    # Create controller
lstn0er    controller = create_controller(ctrl_type, config, default_gains[ctrl_type])
lstn0er
lstn0er    # Run simulation
lstn0er    metrics = run_simulation(controller)
lstn0er    results[ctrl_type] = metrics
lstn0er
lstn0er    # Clean up memory (IMPORTANT for batch!)
lstn0er    controller.cleanup()
lstn0er
lstn0er# Compare all controllers
lstn0erplot_comparison(results)

```

Example 3: PSO Optimization

```

lstn0erfrom src.controllers.factory import create_smc_for_pso, get_gain_bounds_for_pso
lstn0er
lstn0erdef objective_function(gains):
lstn0er    """PSO evaluates this function 1000s of times."""
lstn0er    # Create controller with candidate gains
lstn0er    controller = create_smc_for_pso('sta_smc', gains, max_force=20.0, dt=0.01)
lstn0er
lstn0er    # Simulate
lstn0er    cost = simulate_and_evaluate(controller)
lstn0er
lstn0er    # Clean up (prevents memory leak over 1000 iterations!)
lstn0er    controller.cleanup()
lstn0er
lstn0er    return cost
lstn0er
lstn0er# Get valid gain bounds for chosen controller type
lstn0erbounds = get_gain_bounds_for_pso('sta_smc') # Returns: [(K1_min, K1_max), (K2_min, K2_max), ...]
lstn0er
lstn0er# Run PSO
lstn0erbest_gains = pso_optimize(objective_function, bounds, n_particles=30, n_iterations=50)

```

Quick Reference: Factory API

Factory Functions

```

lstnumberDiscovery functions listAvailableControllers() -> list[str]
lstnumberValidation validateControllerGains(ctrl_type, gains) -> ValidationResult
lstnumber getFaultGains(ctrl_type) -> list[float]
lstnumber getGainBounds(ctrl_type) -> list[tuple]
lstnumberPSO integration
    createSmcForPSO(ctrl_type, gains, **params) -> ControllerInterface
    getGainBoundsForPSO(ctrl_type) -> list[tuple]
lstnumberType utilities canonicalizeControllerType(name) -> str
    Resolves aliases

```

Configuration Example (config.yaml)

```

lstnumber# Controller selection (actual config.yaml structure verified 2026-02-04)
lstnumber# No single controller_type field - specify via simulate.py --ctrl flag
lstnumber# Controller-specific parameters
lstnumbercontrollers:
lstnumber    classical_smc:
lstnumber        max_force: 20.0
lstnumber        boundary_layer: 0.1
lstnumber        switch_method: 'tanh'
lstnumber
lstnumber    sta_smc:
lstnumber        max_force: 20.0
lstnumber        dt: 0.01
lstnumber        boundary_layer: 0.01
lstnumber        damping_gain: 0.5
lstnumber
lstnumber    adaptive_smc:
lstnumber        max_force: 20.0
lstnumber        leak_rate: 0.1
lstnumber        K_min: 1.0
lstnumber        K_max: 50.0
lstnumber        dead_zone: 0.05
lstnumber
lstnumber# Controller gains (tuned via PSO or manually)
lstnumbergains:
lstnumber    classical_smc: [23.068, 12.854, 5.515, 3.487, 2.233, 0.148] # MT-8 robust optimized gains
lstnumber    sta_smc: [15.0, 10.0, 5.0, 3.0, 2.0, 1.0] # [K1, K2, k1, k2, lam1, lam2]
lstnumber    adaptive_smc: [8.0, 4.0, 6.0, 2.5, 0.8] # [k1, k2, lam1, lam2, gamma]

```

Key Takeaways

☰ Quick Summary

- Interface Unity:** ControllerInterface enforces contract - all 5 controllers implement compute_control()
- Factory Power:** One function creates any controller via registry pattern - no if/else chains!
- Memory Safety:** Weakref pattern prevents circular reference leaks (critical for batch simulations)
- Configuration-Driven:** Change controller algorithm by editing ONE line in config.yaml
- Discoverable:** Programmatically list/validate controllers for testing and optimization

What's Next?

💡 Key Concept

E031: Classical SMC Implementation - Deep-dive into the baseline algorithm: sliding surface, switching control, equivalent control, and the chattering phenomenon

E032: Super-Twisting Algorithm (STA) - 2nd-order sliding mode for smooth control without chattering

E033-E036: Adaptive controllers, Swing-Up, MPC, and testing strategies

Code References

- 0. `src/controllers/base/controller_interface.py:12-101` - Base class definition
- `src/controllers/factory/base.py:25-90` - Factory function
- `src/controllers/factory/registry.py:10-60` - Controller registry
- `src/controllers/smooth/smooth.py:187-190` - Weakref example