2025-11-01

Repository: https://github.com/theSadeQ/dip-smc-pso
Documentation: academic/paper/presentations/podcasts/episodes/

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, Seven Brains: All controllers (Classical SMC, STA, Adaptive, Hybrid, Swing-Up, Conditional, MPC) implement the SAME interface.
Result: Change one line in config.yaml and swap algorithms without touching code!

Why This Matters

Common Pitfall

Without Interface:

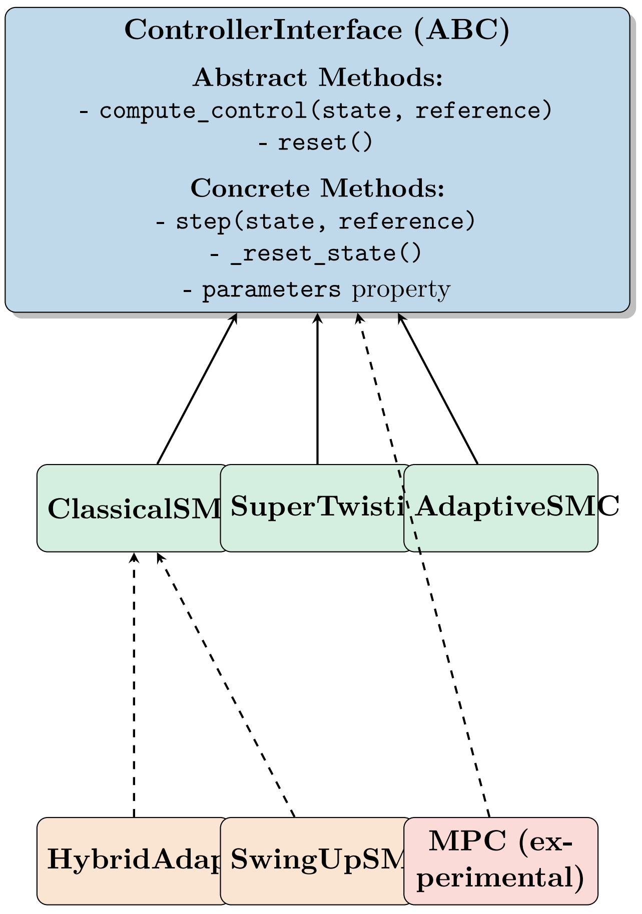
- Each controller has different method names
- Simulation code full of if/else chains
- Adding new controller = rewrite everything
- Testing = nightmare (7 different APIs)

Pro Tip

With Interface:

- One method: `compute_control()`
- Simulation code agnostic to controller type
- New controller = implement interface
- Testing = same harness for all 7

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)

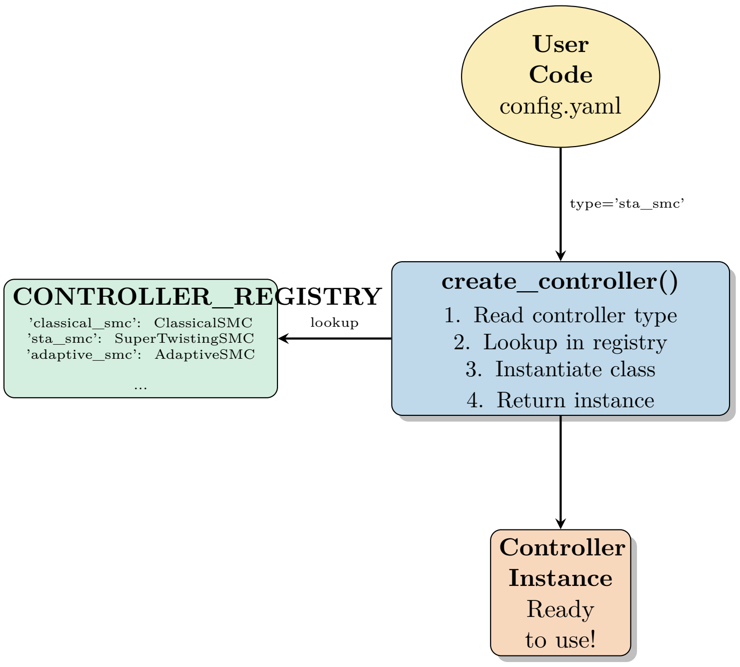
```python
from abc import ABC, abstractmethod
from typing import Optional, Tuple, Any
import numpy as np

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, theta1dot, theta2, theta2dot]
            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 = {'saturated': bool(abs(control) >= self.max_force),
                'control_raw': control}
        return control, info

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

Factory Pattern: One Function to Rule Them All

Key Concept

Problem: How do you create 7 different 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
CONTROLLER_REGISTRY = {
    'classical_smc': ClassicalSMC,
    'sta_smc': SuperTwistingSMC,
    'adaptive_smc': AdaptiveSMC,
    'hybrid_adaptive_sta_smc': HybridAdaptiveSTASMC,
    'swingup_smc': SwingUpSMC,
    'mpc': MPCController,
}

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_smc')
        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 Controller-

Interface

- **Discoverable**: List available controllers programmatically

Alias Support

```
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 = load_config("config.yaml")
lstnumberCreate controller (type from config, NOT hardcoded!)  controller = create_controller(ctrl_type = 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):  state =
    get_current_state() control = controller.compute_control(state) 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):

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

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

Good (Weak Reference):

```python
import weakref

class ClassicalSMC:
    def __init__(self, dynamics_model):
        # Weak reference
        self._dynamics_ref = weakref.ref(dynamics_model)

    @property
    def dyn(self):
        return self._dynamics_ref()
```

Weak reference doesn't prevent garbage collection!

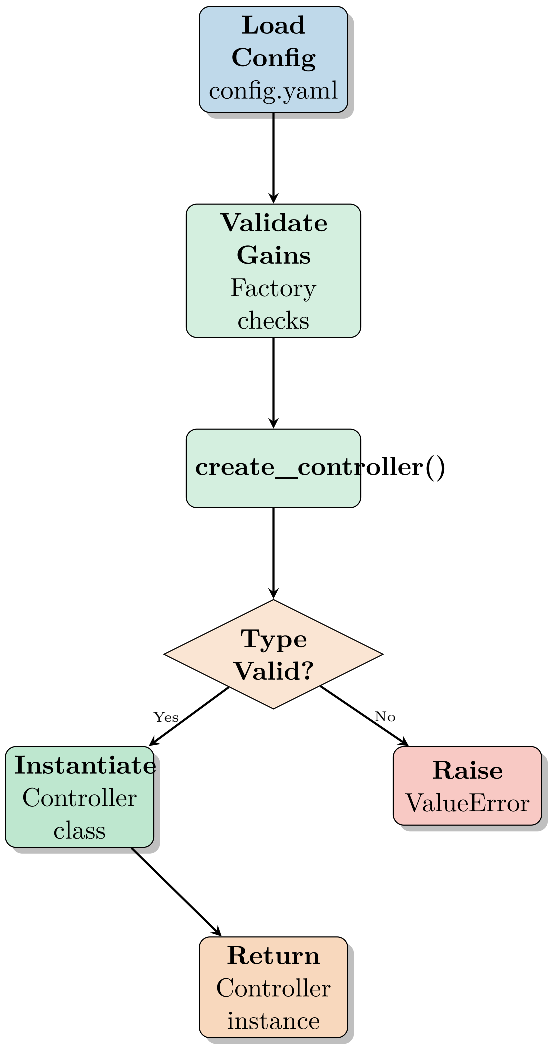
Cleanup Pattern

```python
class ClassicalSMC(ControllerInterface):
    def cleanup(self) -> None:
        """Explicit memory cleanup to prevent leaks."""
        # Nullify dynamics reference
        if hasattr(self, '_dynamics_ref'):
            self._dynamics_ref = lambda: None

        # Clear large NumPy arrays
        if hasattr(self, 'L'):
            self.L = None
        if hasattr(self, 'B'):
            self.B = None

    def __del__(self) -> None:
        """Destructor for automatic cleanup."""
        try:
            self.cleanup()
        except Exception:
            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

enumi**Abstract Base Class (ABC)**: Enforces contract via Python's `@abstractmethod`

0. enumi**Factory Pattern**: Registry-based instantiation decouples creation from usage

0. enumi**Weak Reference Pattern**: Prevents memory leaks from circular references

0. enumi**Strategy Pattern**: Controllers are interchangeable strategies for same problem

Practical Examples: Using the Factory

Example 1: Command-Line Simulation

```python
# File: simulate.py
import argparse
from src.controllers.factory import create_controller
from src.config import load_config

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(’--ctrl’, default=’classical_smc’,
                        help=’Controller␣type’)
    args = parser.parse_args()

    # Load config
    config = load_config("config.yaml")

    # Create controller from command-line argument
    controller = create_controller(
        ctrl_type=args.ctrl,  # User-specified type!
        config=config[’controllers’][args.ctrl],
        gains=config[’gains’][args.ctrl]
    )

    # Run simulation (same code for ALL controllers)
    results = simulate(controller, initial_state, dt=0.01, duration=10.0)
    plot_results(results)

# Usage:
# python simulate.py --ctrl classical_smc
# python simulate.py --ctrl sta_smc
# python simulate.py --ctrl adaptive_smc
```

Example 2: Batch Comparison

```python
from src.controllers.factory import list_available_controllers, create_controller

# Discover all available controllers programmatically
all_controllers = list_available_controllers()
# Returns: [’classical_smc’, ’sta_smc’, ’adaptive_smc’, ...]

results = {}
for ctrl_type in all_controllers:
    # Create controller
    controller = create_controller(ctrl_type, config, default_gains[ctrl_type])

    # Run simulation
    metrics = run_simulation(controller)
    results[ctrl_type] = metrics

    # Clean up memory (IMPORTANT for batch!)
    controller.cleanup()

# Compare all controllers
plot_comparison(results)
```

Example 3: PSO Optimization

```python
from src.controllers.factory import create_smc_for_pso, get_gain_bounds_for_pso

def objective_function(gains):
    """PSO evaluates this function 1000s of times."""
    # Create controller with candidate gains
    controller = create_smc_for_pso(’sta_smc’, gains, max_force=20.0, dt=0.01)

    # Simulate
    cost = simulate_and_evaluate(controller)

    # Clean up (prevents memory leak over 1000 iterations!)
    controller.cleanup()

    return cost

# Get valid gain bounds for chosen controller type
bounds = get_gain_bounds_for_pso(’sta_smc’)  # Returns: [(K1_min, K1_max), (K2_min, K2_max), ...]

# Run PSO
best_gains = pso_optimize(objective_function, bounds, n_particles=30, n_iterations=50)
```

Quick Reference: Factory API

Factory Functions

lstnumberDiscovery functions list$_a vailable_c ontrollers() -> list[str] list_a ll_c ontrollers() -> dict Withmetadata$
lstnumberValidation validate$_c ontroller_g ains(ctrl_t ype, gains) -> ValidationResult get_d efault_g ains(ctrl_t ype) ->$
$list[float] get_g ain_b ounds(ctrl_t ype) -> list[tuple]$
lstnumberPSO integration
create$_s mc_f or_p so(ctrl_t ype, gains, **params) -> ControllerInterface get_g ain_b ounds_f or_p so(ctrl_t ype) -> list[tuple]$
lstnumberType utilities canonicalize$_c ontroller_t ype(name) -> str Resolvesaliases$

Configuration Example (config.yaml)

```
lstnumber# Controller selection (CHANGE THIS ONE LINE to swap algorithms!)
lstnumbercontroller_type: 'sta_smc'  # Try: classical_smc, adaptive_smc, hybrid_adaptive_sta_smc
lstnumber
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: [10.0, 5.0, 8.0, 3.0, 15.0, 2.0]  # [k1, k2, lam1, lam2, K, kd]
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

1. **Interface Unity**: `ControllerInterface` enforces contract - all 7 controllers implement `compute_control()`
2. **Factory Power**: One function creates any controller via registry pattern - no if/else chains!
3. **Memory Safety**: Weakref pattern prevents circular reference leaks (critical for batch simulations)
4. **Configuration-Driven**: Change controller algorithm by editing ONE line in config.yaml
5. **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

- `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/smc/classic_smc.py:187-190` - Weakref example