

2025-11-01

E015: Architectural Standards & Patterns

Intentional Patterns, Design Decisions,
Quality Gates, File Naming Conventions

Part 3 · Duration: 15-20 minutes

Beginner-Friendly Visual Study Guide

🎯 **Learning Objective:** Understand architectural invariants (never "fix" these!), design patterns (factory/strategy/DI), code conventions (naming/structure), and quality gates enforcement

Why Architectural Standards Matter

💡 Key Concept

Problem: Without standards, these look like BUGS:

- `simulation_context.py` in 3 locations (duplication?)
- 8 different dynamics files (redundant?)
- `src/optimizer/` + `src/optimization/` (typo?)

Reality: All INTENTIONAL architectural patterns!

Solution: Document standards so patterns become FEATURES, not mistakes

Codebase Without Standards = City Without Zoning Laws

⚠️ Common Pitfall

Chaos Example:

- `controller.py`
- `Controller_v2.py`
- `CONTROLLER_NEW.py`
- `my_controller_final_FINAL.py`
- `temp_controller_backup_old_2.py`

Which is production? Which is safe to delete? **Nobody knows!**

With Standards: Clean Organization

📁 Zoning Laws for Code

Controllers District: `src/controllers/`

- `classical_smc.py`
- `sta_smc.py`
- `adaptive_smc.py`
- No version suffixes, no "final" labels

Standards Answer 4 Questions:

- 1. **WHERE** to put code (directory structure)
- 2. **HOW** to name things (files, classes, functions)
- 3. **WHAT** patterns to use (factory, not if-else chains)
- 4. **WHEN** to deviate (documented exceptions only)

Intentional Architectural Patterns (DO NOT "FIX"!) ---

💡 Key Concept

Critical: These patterns look like mistakes to newcomers but are DELIBERATE design decisions

Pattern 1: Compatibility Layers

🔗 Example

Observed: Both `src/optimizer/` AND `src/optimization/` exist

Looks Like: Typo or inconsistency

Reality: Backward compatibility layer!

Reason: Legacy code uses `from src.optimizer import PSOTuner`

New code uses `from src.optimization import PSOTuner`

`src/optimizer/__init__.py` re-exports from `src/optimization/`

DO NOT DELETE `src/optimizer/` - breaks backward compatibility!

Pattern 2: Re-Export Chains

🔗 Example

Observed: `simulation_context.py` in 3 locations:

- `src/core/simulation_context.py` (implementation)
- `src/simulation_context.py` (re-export)
- `src/interfaces/simulation_context.py` (alias)

Reason: Import path flexibility - all these work:

```
lstnumberfrom src.core import SimulationContext
lstnumberfrom src import SimulationContext
lstnumberfrom src.interfaces import SimulationContext
```

Benefit: Users can import from convenient location without knowing internal structure

Pattern 3: Model Variants (8 Dynamics Files)

Accuracy/Performance Tradeoffs

NOT Redundant - Each Serves Different Use Case:

File	Compute Time	Accuracy
simplified_dynamics.py	5 μ s	Linear approx
full_nonlinear_dynamics.py	50 μ s	Full physics
lowrank_dynamics.py	15 μ s	Reduced model
coriolis_dynamics.py	60 μ s	With Coriolis
+ 4 more variants	varies	Specialized

Strategy: Develop with simplified (fast), validate with full (accurate)

PSO Example: 5,000 evaluations with simplified = 3 minutes, full = 30 minutes

Pattern 4: Framework Files (NOT Test Files!)

Common Pitfall

Common Mistake: Moving `src/interfaces/hil/test_automation.py` to `tests/`
Why It's WRONG:

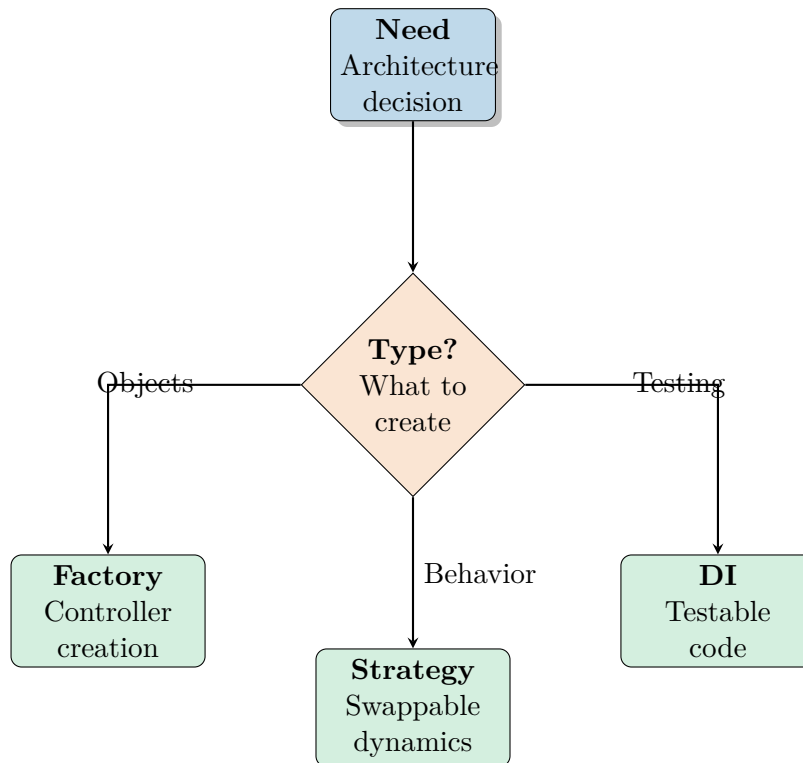
- Has `pytest` imports BUT is PRODUCTION code
- Provides test automation FRAMEWORK for HIL validation
- Used by actual deployed systems, not just CI

Rule: File location determined by PURPOSE, not imports!

Classification Checklist:

- 1. Exported in `__init__.py`? → Production (`src/`)
- 2. Imported by production code? → Production (`src/`)
- 3. Framework/infrastructure? → Production (`src/`)
- 4. Has `test_*` name AND only for `pytest`? → Test (`tests/`)

Design Patterns Used



⚙️ Core Design Patterns

1. Factory Pattern (Controller Creation)

```

lstnumber# BAD: Giant if-else chain
lstnumberif ctrl_type == "classical":
lstnumber    return ClassicalSMC(gains)
lstnumberelif ctrl_type == "sta":
lstnumber    return STASMC(gains)
lstnumber# ... 5 more elif branches
lstnumber
lstnumber# GOOD: Factory pattern
lstnumbercontroller = create_controller(ctrl_type, config, gains)
  
```

2. Strategy Pattern (Swappable Dynamics)

```

lstnumber# Use different dynamics without changing controller
lstnumberdynamics = SimplifiedDynamics() # Fast
lstnumber# OR
lstnumberdynamics = FullNonlinearDynamics() # Accurate
lstnumber
lstnumbercontroller.simulate(dynamics) # Same interface
  
```

3. Dependency Injection (Testable Code)

```

lstnumber# Inject dependencies, not hardcode
lstnumberdef run_simulation(controller, dynamics, config):
lstnumber    # Easy to test with mock objects
lstnumber    pass
  
```

File Naming Conventions

Naming Standards

Production Code (**src/**):

- lowercase_with_underscores.py
- ClassNameInPascalCase
- function_name_in_snake_case()

Test Files (**tests/**):

- test_*.py (prefix with "test_")
- Mirror structure: `src/controllers/classical_smc.py` → `tests/test_controllers/test_classical_smc.py`

Scripts (**scripts/**):

- Executable tools (not imported)
- generate_figures.py, analyze_results.py

NEVER Create:

- Directories with braces: {dir}/
- Spaces in names: my folder/
- Windows device names: nul, con, prn

Directory Placement Rules

src/ - Production Code

- Controllers, models, utils
- Frameworks (even with pytest imports!)
- Anything imported by production

scripts/ - Dev Tools

- Executed, not imported
- One-off analysis scripts
- Figure generators

tests/ - Pytest Tests

- test_*.py files
- Unit, integration, system tests
- Mirror **src/** structure

Root - Essentials Only

- Target: 19 visible items
- Currently: 14 items [OK]
- No temporary files!

Quality Gates Enforcement

✓ Mandatory Quality Gates

Gate 1: Critical Issues = 0 [MANDATORY]

Any P0 bug blocks merge

Gate 2: High-Priority Issues 3 [REQUIRED]

P1 issues tracked, must have timeline

Gate 3: Test Pass Rate = 100% [MANDATORY]

4,563/4,563 tests must pass

Gate 4: Root Items 19 [REQUIRED]

Currently 14 [OK] - prevents root clutter

Gate 5: Malformed Names = 0 [MANDATORY]

No braces, spaces, device names in paths

Key Takeaways

☰ Quick Summary

Architectural Invariants: Patterns that LOOK like bugs but are INTENTIONAL (document these!)

Compatibility Layers: `src/optimizer/` → `src/optimization/` (backward compat, don't delete!)

Re-Export Chains: `simulation_context.py` in 3 locations (import flexibility)

Model Variants: 8 dynamics files = different accuracy/speed tradeoffs (NOT redundant)

Framework Files: `test_automation.py` has pytest imports BUT is production code (`src/`, not `tests/`)

Design Patterns: Factory (controller creation), Strategy (swappable dynamics), Dependency Injection (testable)

File Naming: `lowercase_underscore.py`, `test_*.py` prefix, mirror structure

Directory Rules: `src/` (production), `scripts/` (dev tools), `tests/` (pytest), Root (19 items)

Quality Gates: 5 mandatory checks (critical bugs=0, test pass=100%, malformed names=0, root items 19, high-priority 3)

Classification Checklist: Exported? Imported by production? Framework? → Determines location

Quick Reference: Architecture Commands

📄 Validate Architecture

```
lstdirectoryList files without corresponding tests python scripts/architecture/find_untested.py
lstdirectoryCheck naming conventions python scripts/architecture/check_naming.py
```

What's Next?

💡 Key Concept

E016: Attribution & Citations

Dependency licenses, academic citations, contributor acknowledgments, compliance requirements

Remember: Document intentional patterns - what looks like a bug might be a feature!