2025-11-01

E007: Testing and Quality Assurance

4,563 Tests in 45 Seconds

Part 2 · Duration: 15-20 minutes

Beginner-Friendly Visual Study Guide

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

Learning Objective: Understand test pyramid, coverage standards (85/95/100%), property-based testing, and quality gates for research vs production

The Testing Challenge

Key Concept

Question: How many tests to validate 105,000 lines of code?
Answer: 4,563 tests - BUT quality > quantity!
Real question: What deserves a test? When do you have enough?

Three Hard Questions

Common Pitfall

- **1. What to test?** One input or all possible inputs? (All is impossible ⇒ sample strategically)
- **2. When to stop?** Can always write more tests - what's "good enough"?
- **3. What to validate?** Implementation details (break on refactor) vs behavior (stable contract)?

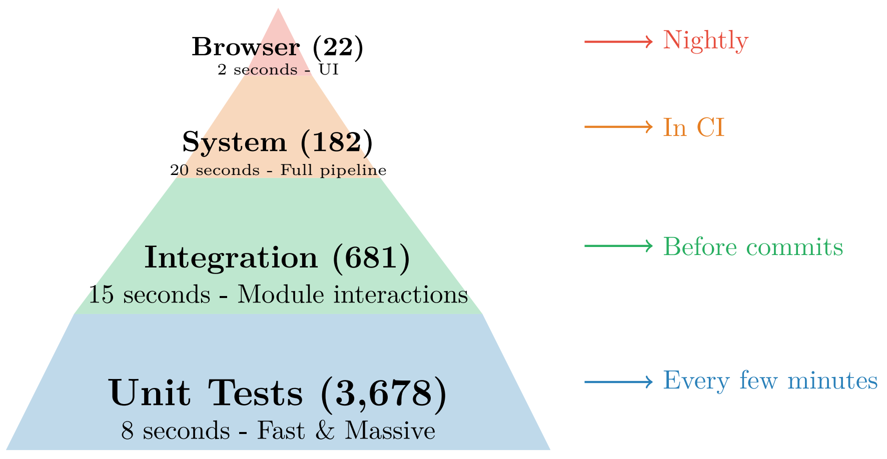
Testing is strategic choice-making under time constraints

Test Suite Breakdown: 4,563 Tests

Test Suite Distribution

Level	Count	Percent	Time/Test
Unit Tests	3,678	81%	200 μs
Integration Tests	681	15%	50 ms
System Tests	182	4%	2 s
Browser Tests	22	0.5%	5 s
TOTAL	**4,563**	**100%**	**45 seconds**

The Test Pyramid



Pro Tip

Pyramid Philosophy:

- **Wide base:** Lots of fast tests (run constantly during development)
- **Narrow top:** Few slow tests (run before commits, in CI)
- **Speed enables feedback:** 45 seconds total ⇒ run every few minutes!

Unit Tests: The Foundation

Example

3,678 unit tests, 200 microseconds each (8 seconds total)
Example: Classical SMC has 51 unit tests covering:

- Zero state (equilibrium)
- Maximum gains (saturation)
- Boundary layer transitions ($s \to 0$)
- Edge cases (NaN, infinity, negative values)

Pattern: Pass known state → Get control signal → Verify matches expected

Integration Tests: Module Interactions

681 integration tests, 50 ms each (15 seconds total)
What they test:

- Factory + Config: Parse YAML, create controller
- Controller + Dynamics: Interface compatibility
- PSO + Simulator: Batch evaluation

Example:

```
lstnumberdef test_factory_config():
lstnumber    config =
        load_config("test.yaml")
lstnumber    ctrl = create_controller(
lstnumber        config.controller.type,
lstnumber        config.controller.params
lstnumber    )
lstnumber    assert ctrl.gains ==
        config.gains
```

Coverage Standards: 85 / 95 / 100

Key Concept

Three-tier coverage requirements:

- **1. Overall project:** 85% minimum (aggregate across all files)
- **2. Critical modules:** 95% minimum (controllers, dynamics, PSO)
- **3. Safety-critical:** 100% required (saturation, validation, monitoring)

Why Different Standards?

Risk-based prioritization:
Utility function (formats logs):

- Failure ⇒ Garbled log entry
- **Cost:** Annoying
- **Coverage:** 85% OK

Saturation function (limits force):

- Failure ⇒ Command 10,000 N to 150 N actuator
- **Cost:** Broken hardware!
- **Coverage:** 100% MANDATORY

The 100% Coverage List (10 Modules)

Safety-Critical Modules

Safety Modules:

- Saturation - Prevents actuator damage (10,000 N → 150 N max)
- Validation - Stops physically impossible configs (negative mass!)
- Deadband - Prevents actuator oscillation near setpoint

Correctness Modules:

- Reproducibility - Deterministic random seeds (peer review requirement!)
- State Manager - Prevents simulation corruption
- Config Validator - Catches errors before simulation

Core Interfaces:

- Base Controller - Inherited by all 7 controllers
- Dynamics Interface - Swappable plant models
- PSO Bounds - Keeps optimization within valid ranges

Monitoring:

- Latency Tracker - Detects missed control deadlines

Common Pitfall

Why reproducibility is critical:
If reviewer can't reproduce results (bad random seeds) ⇒ **Paper invalid!**
Consequences: Rejected paper → Broken $50,000 robot
Reproducibility is not optional in research software.

CI Enforcement

Example

Pull Request Rules:

- **1.** Add 100 lines to critical module
- **2.** Must add tests to maintain 95%+ coverage
- **3.** If coverage drops below 95% ⇒ **Build FAILS**
- **4.** Cannot merge until tests added

This prevents "I'll add tests later" syndrome!

Property-Based Testing with Hypothesis

Key Concept

Traditional testing: Write test with ONE specific input
Property-based testing: Write property that holds for ALL inputs
Hypothesis framework: Generates hundreds of random inputs, checks property for each

Example: Saturation Function

Traditional Test:

```
lstnumberdef test_saturation():
lstnumber    result = saturate(200, max=150)
lstnumber    assert result == 150
lstnumber# Tests ONE case (200)
```

What about: 151? 10,000? 1 million?

Property-Based Test:

```
lstnumber@given(value=st.floats(
lstnumber    min_value=151, max_value=1e6))
lstnumberdef test_saturation_property(value):
lstnumber    result = saturate(value,
        max=150)
lstnumber    assert result == 150
lstnumber# Tests 100 random cases!
```

Pro Tip

It's like having a robot stress-test your code while you sleep!
Hypothesis generates edge cases you never thought of: `NaN`, `inf`, negative zero, etc.

Properties We Test

System Properties

Controller Properties:

- Control signal must be bounded ($|u| \leq u_{\max}$)
- Control must not contain NaN or infinity
- Control must be deterministic (same state $\Rightarrow$ same output)

Dynamics Properties:

- State derivatives must be finite (no explosions!)
- Energy conserved in absence of friction
- Linearization matches finite-difference approximation

PSO Properties:

- Best cost must never increase (monotonic improvement)
- Final best particle within search bounds
- Optimization reproducible with same seed

Coverage Campaign: Week 3 Bug Hunt

Example

December 20-21, 2025: The 16.5-Hour Sprint
Mission: Bring 10 critical modules to 100% coverage before holidays
Results:

- 668 tests created
- 11 modules validated (beat goal by one!)
- 2 silent killers found and fixed same-day

Felt like defusing bombs while clock ticked down

Bug 1: Factory API Mismatch

Common Pitfall

Problem: Factory expected gains as `list`, config provided `numpy.ndarray`
Symptom: Worked in most cases, failed when serializing to JSON
Fix: Explicitly convert to list in factory
Found via: Integration test for controller state serialization

Bug 2: Memory Leak in Adaptive Controller

Common Pitfall

The Silent Killer:
Adaptive controller stored reference to EVERY simulation's full history (for debugging).
Never released memory (hoarding!).
Impact:

- After 1,000 simulations (typical PSO run): 500 MB RAM
- Overnight optimizations crashed at hour 9 of 10-hour run

Fix: Use `weakref` - "Remember where object is, but don't hold it hostage"
Found via: Property-based test running 10,000 consecutive simulations, asserting memory growth = 0

Test Execution: 45 Seconds for 4,563 Tests

Execution Breakdown

Test Type	Time	Why So Fast?
Unit (3,678)	8 seconds	No I/O, pure functions
Integration (681)	15 seconds	Load configs, few timesteps
System (182)	20 seconds	Simplified dynamics + Numba JIT
Browser (22)	2 seconds	Parallel with pytest-xdist
TOTAL	**45 seconds**	**10 ms/test average**

Pro Tip

Why speed matters:
10-minute tests ⇒ Developers don't run during development ⇒ Commit broken code ⇒ Wait for CI failure ⇒ Slow iteration
45-second tests ⇒ Run every few minutes locally ⇒ Catch failures before commit ⇒ Fast feedback loop

Quality Gates: Research vs Production

8 Quality Gates (5/8 Pass)

Gates We PASS (Research-Ready):

- **1.** Zero critical bugs (all P0 issues resolved)
- **2.** 100% test pass rate (4,563/4,563 tests)
- **3.** Memory validated (10,000 sims, zero growth)
- **4.** Thread-safe (11/11 parallel PSO tests pass)
- **5.** Zero high-priority issues

Gates We FAIL (Production Blockers):

- **1.** Coverage measurement broken (reports 2.86%, real is 89%)
- **2.** No production CI/CD (dev pipelines only)
- **3.** No hardware validation (never run on actual robot/PLC)

Key Concept

Bottom Line:
Research-Ready (5/8): Can publish papers, run experiments, validate theories
Production-Ready (8/8) ⚠: Can deploy to industrial plant (need gates 6-8)
Verdict: Science is sound. Engineering needs hardening for production.

Quick Reference: Testing Commands

Run Full Test Suite

```
lstnumberWith coverage report pytest tests/ -cov=src -cov-report=html
lstnumberParallel execution (faster on multi-core) pytest tests/ -n auto
```

Run Specific Test Levels

```
lstnumberIntegration tests (15 seconds) pytest tests/test_integration/
lstnumberSystem tests (20 seconds) pytest tests/test_system/
lstnumberBrowser tests (2 seconds) pytest tests/test_browser/
```

Property-Based Testing

```
lstnumber@given(value=st.floats(min_value = 151, max_value = 1e6))deftest_saturation_property(value) : result =
        saturate(value, max = 150)assertresult ==
        150assertresult <= 150Property!
```

Key Takeaways

Quick Summary

Test Pyramid: 81% unit (fast), 15% integration, 4% system, 0.5% browser
Coverage Tiers: 85% overall, 95% critical, 100% safety-critical
Property-Based Testing: Hypothesis generates 100 random cases, finds edge cases you never thought of
Quality Gates: 5/8 pass (research-ready), need 8/8 for production
Speed Enables Feedback: 45 seconds for 4,563 tests ⇒ run every few minutes
Bugs Found: Factory API mismatch + Memory leak (500 MB after 1,000 sims)

What's Next?

Key Concept

E008: Research Outputs & Publications - 11 research tasks, submission-ready paper (v2.1), 14 figures
Remember: Testing is strategic choice-making under constraints. Quality > quantity!