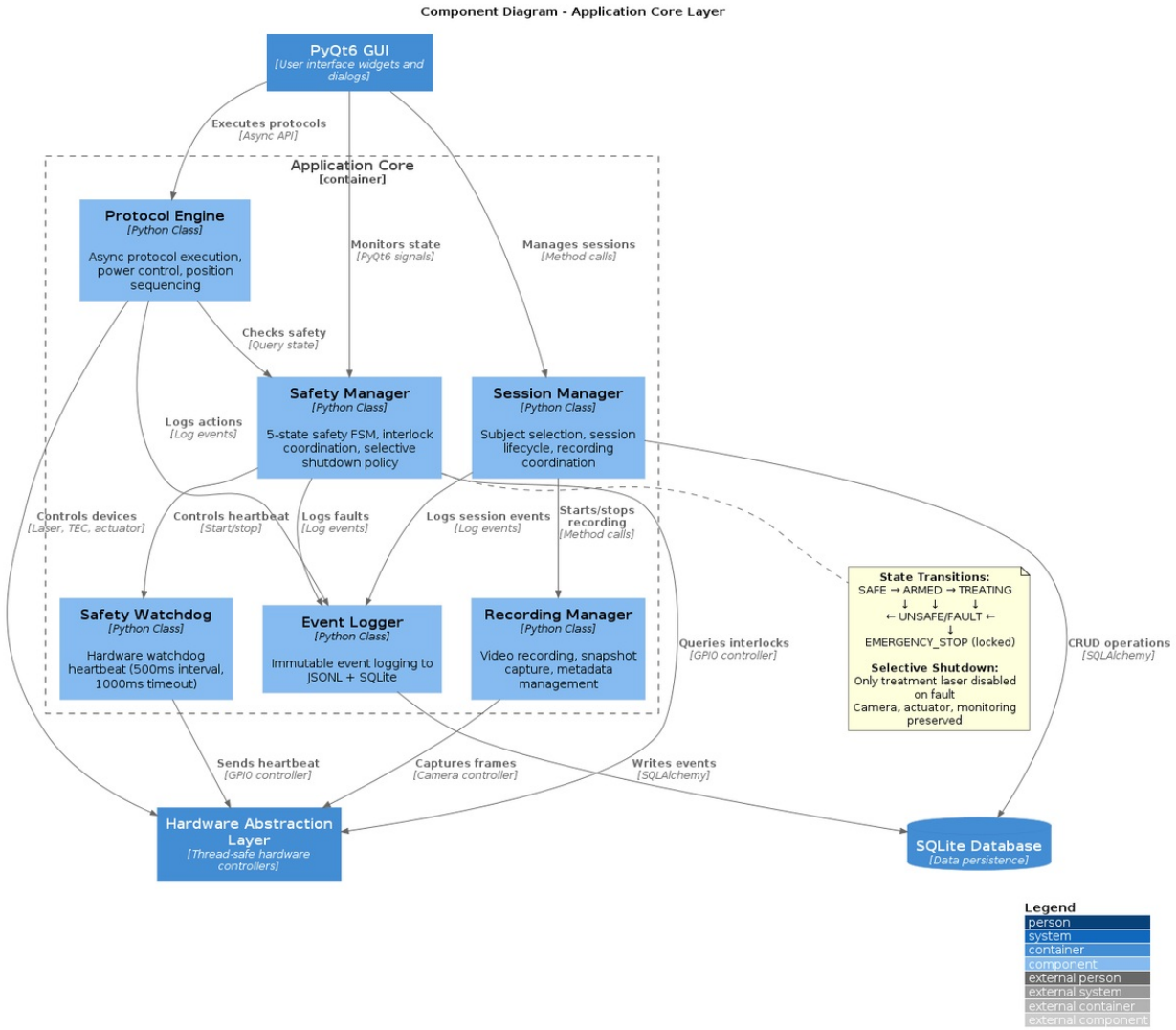
Test Architecture

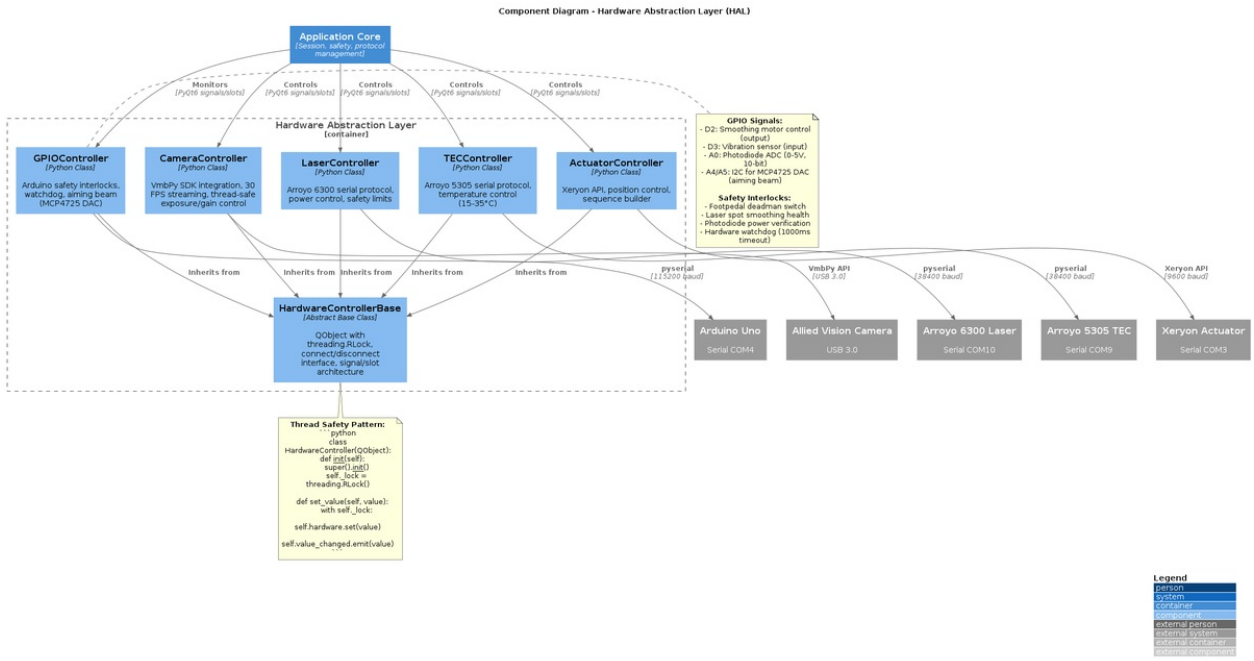
Architecture Diagrams

Figure 1: TOSCA Component Diagram - Application Core



TOSCA Component Diagram - Application Core

Figure 2: TOSCA Component Diagram - Hardware Abstraction Layer



TOSCA Component Diagram - Hardware Abstraction Layer

Document Version: 1.0 **Last Updated:** 2025-10-26 **Status:** Implemented - Phase 5 Testing Complete **Priority:** CRITICAL - Required for FDA validation

Table of Contents

Overview

Purpose

This document describes the test architecture for the TOSCA Medical Laser Control System, including the MockHardwareBase pattern, test organization, coverage strategy, and FDA validation requirements.

Regulatory Context: - **IEC 62304 Class C:** Enhanced Documentation Level - **FDA 21 CFR Part 820:** Design Controls and Verification/Validation - **ISO 14971:** Risk Management Requirements

Testing Objectives

1. **Verification:** Prove software meets specifications (design verification)
2. **Validation:** Prove software meets user needs (design validation)
3. **Regression Prevention:** Detect breaking changes early
4. **FDA Compliance:** Demonstrate comprehensive testing for 510(k) submission
5. **Safety Assurance:** Validate safety-critical code paths

Test Metrics

Metric	Target	Current Status
Code Coverage	≥90% (safety-critical: 100%)	Week 1: ~85%
Test Count	≥500 tests	30 files, expanding
Safety Tests	100% path coverage	[DONE] Complete
Hardware Mock Coverage	All 4 controllers	[DONE] 4/4 implemented
CI/CD	All tests pass before merge	[PENDING] Phase 6

Testing Philosophy

Test-Driven Development (TDD)

Workflow: 1. **Write test first** (defines expected behavior) 2. **Implement minimum code** (make test pass) 3. **Refactor** (improve code while tests still pass) 4. **Repeat**

Benefits: - [DONE] Better test coverage (written before code) - [DONE] Clearer specifications (tests document expected behavior) - [DONE] Fewer bugs (caught early in development) - [DONE] Safer refactoring (tests catch regressions)

Testing Pyramid

```
1. **Manual        5% (exploratory testing, usability)**
2. **Validation**
3. **Integration       25% (system-level, hardware integration)**
4. **Tests**
5. **Unit Tests           70% (fast, isolated, comprehensive)**
```

Unit Tests: - Test individual functions/classes in isolation - Fast execution (<1ms per test) - Mock external dependencies - High coverage (90%+ target)

Integration Tests: - Test interaction between components - Moderate execution time (~100ms per test) - Use real components where possible, mock hardware - Focus on interfaces and data flow

Manual Validation: - Exploratory testing with real hardware - Usability testing with clinicians - Safety validation (before clinical trials) - Regulatory submission testing

---

Mock Hardware Pattern

Motivation

Problem: TOSCA depends on 4 expensive hardware devices: 1. **Arroyo Laser** ($20,000+) - Safety-critical treatment laser 2. **Xeryon Actuator** ($15,000+) - Precision linear stage 3. **Allied Vision Camera** ($2,000+) - High-speed image capture 4. **Arduino Nano GPIO** ($25) - Safety interlocks

Traditional Approach:

```python
# BAD: Tightly coupled to real hardware
class SafetyManager:
    def __init__(self):
        self.laser = ArroyoLaserController()  # ← Requires real laser
        self.laser.connect(port="COM3")       # ← Fails if not connected
```

Problems with Traditional Approach: - [FAILED] Cannot run tests without hardware - [FAILED] Hardware failures break all tests - [FAILED] Slow (real hardware has delays) - [FAILED] Cannot simulate edge cases (connection failures, timeouts)

MockHardwareBase Solution

Pattern: Dependency injection + abstract base class

```
# GOOD: Decoupled from real hardware
class SafetyManager:
    def __init__(self, laser: HardwareControllerBase):
        self.laser = laser  # ← Can be real OR mock

# Production code uses real hardware
real_laser = ArroyoLaserController()
safety = SafetyManager(laser=real_laser)

# Test code uses mock
mock_laser = MockLaserController()
safety = SafetyManager(laser=mock_laser)
```

Benefits: - [DONE] Tests run without hardware (faster, more reliable) - [DONE] Can simulate failures (connection errors, timeouts, edge cases) - [DONE] Full control over behavior (deterministic testing) - [DONE] Supports PyQt6 signals (verify UI responses)

MockHardwareBase Implementation

Location: `tests/mocks/mock_hardware_base.py`

Key Features:

1. **Configurable Behaviors:**

```
mock = MockHardwareBase()

# Simulate connection failure
mock.simulate_connection_failure = True
result = mock.connect()  # Returns False

# Simulate slow response (timeout testing)
mock.response_delay_s = 5.0
result = mock.get_status()  # Takes 5 seconds

# Simulate error during operation
mock.simulate_status_error = True
status = mock.get_status()  # Emits error_occurred signal
```

2. **Call Tracking (Verification):**

```
mock = MockHardwareBase()
mock.connect(port="COM4")
mock.disconnect()

# Verify calls were made with correct arguments
assert ("connect", {"port": "COM4"}) in mock.call_log
assert mock.disconnect_call_count == 1
```

3. **PyQt6 Signal Support:**

```
# Mock emits real PyQt6 signals
mock = MockLaserController()

# Connect signal to callback
received_signals = []
mock.power_changed.connect(lambda p: received_signals.append(p))

# Trigger signal
mock.set_power(50.0)

# Verify signal was emitted
assert 50.0 in received_signals
```

Specialized Hardware Mocks

MockLaserController (`tests/mocks/mock_laser_controller.py`) - Simulates Arroyo laser control - Supports power setting, enable/disable, interlock simulation - Emits `power_changed`, `state_changed`, `error_occurred` signals

MockActuatorController (`tests/mocks/mock_actuator_controller.py`) - Simulates Xeryon actuator motion - Supports position control, homing, limit switch detection - Configurable: `simulate_homing_failure`, `simulate_limit_hit`

MockCameraController (`tests/mocks/mock_camera_controller.py`) - Simulates Allied Vision camera - Supports frame capture, exposure control, trigger modes - Returns synthetic test frames (black, pattern, or custom)

MockGPIOController (`tests/mocks/mock_gpio_controller.py`) - Simulates Arduino Nano GPIO - Supports interlock state simulation (footpedal, smoothing device) - Configurable pin states (HIGH/LOW)

MockQObjectBase (`tests/mocks/mock_qobject_base.py`) - Base class for PyQt6 signal support - Enables signal/slot testing without QApplication - Supports `pyqtSignal` emitting in tests

---

Test Organization

Directory Structure

```
**High-Level Structure:**

- **tests/**

**Key Files:**
- __init__.py               # Exports all mocks
- mock_hardware_base.py     # Base mock (foundation)
- mock_laser_controller.py  # Laser mock
- mock_actuator_controller.py # Actuator mock
- mock_camera_controller.py  # Camera mock
- mock_gpio_controller.py    # GPIO mock
- mock_qobject_base.py       # PyQt6 signal support
- example_basic_usage.py
```

See full project structure in source repository

Naming Conventions

Test Files: - `test_*.py` - Unit/integration tests - `test_mock_*.py` - Mock validation tests - `debug_*.py` - Debug scripts (not run by pytest)

Test Functions: - `test_<feature>_<scenario>()` - Clear, descriptive names - Examples: - `test_laser_enable_success()` - `test_laser_enable_when_interlock_fails()` - `test_emergency_stop_disables_laser()`

Test Classes (Optional):

```
class TestSafetyManager:
    """Group related tests for SafetyManager."""

    def test_emergency_stop_disables_laser(self):
        ...

    def test_emergency_stop_maintains_camera(self):
        ...
```

Test Coverage Strategy

Coverage Targets

Component	Coverage Target	Rationale
Safety System	100%	Safety-critical (FDA requirement)
Treatment Protocol	100%	Safety-critical (FDA requirement)
Laser Control	100%	Safety-critical (Class 4 laser)
Interlock Logic	100%	Safety-critical (patient safety)
Database Layer	95%+	Audit trail integrity (21 CFR Part 11)
GUI Layer	70-80%	User interaction (manual validation)
Hardware Abstraction	90%+	Reliability critical
Image Processing	85%+	Treatment effectiveness

Safety-Critical Code Paths

Definition: Code that could cause patient harm if it fails

Examples: 1. Laser enable/disable logic 2. Emergency stop handling 3. Interlock validation 4. Power limit enforcement 5. Session timeout handling 6. Treatment protocol execution

Testing Requirements: - [DONE] 100% branch coverage (every if/else tested) - [DONE] All edge cases (boundary values, error conditions) - [DONE] Failure modes (connection loss, timeout, hardware error) - [DONE] Concurrent access (threading, race conditions) - [DONE] Signal/slot correctness (PyQt6 event handling)

Coverage Measurement

Tool: `pytest-cov` (built on `coverage.py`)

Commands:

```
# Run tests with coverage
pytest --cov=src tests/

# Generate HTML report
pytest --cov=src --cov-report=html tests/

# View report
open htmlcov/index.html
```

Coverage Report Example:

```
Name                                Stmts   Miss  Cover
-----------------------------------------------------
src/core/safety.py                    245      3    99%
src/core/protocol_engine.py           189      8    96%
src/hardware/laser_controller.py      156      0   100%
src/hardware/actuator_controller.py   203     15    93%
src/database/manager.py               178      5    97%
-----------------------------------------------------
TOTAL                                2134    127    94%
```

Missing Coverage Identification

Workflow: 1. Run coverage report 2. Identify uncovered lines (red in HTML report) 3. Write tests for uncovered code 4. Re-run coverage 5. Repeat until target met

Priority for Coverage: - **P0:** Safety-critical code (must be 100%) - **P1:** Core business logic (90%+ target) - **P2:** UI code (70%+ target) - **P3:** Debug/logging code (best effort)

Testing Guidelines

Writing Good Unit Tests

Characteristics of Good Unit Tests: 1. **Fast** - <1ms execution (use mocks, not real hardware) 2. **Isolated** - Test one thing, no dependencies between tests 3. **Repeatable** - Same result every time (no randomness) 4. **Self-checking** - Assert pass/fail automatically 5. **Timely** - Written before or with implementation code

Example: Good Unit Test

```
def test_laser_enable_requires_all_interlocks() -> None:
    """
    Verify laser enable fails if any interlock is not satisfied.
```

```
    Safety requirement: All 7 interlocks must pass for laser operation.
    """
    # Arrange
    mock_laser = MockLaserController()
    mock_gpio = MockGPIOController()
    safety = SafetyManager(laser=mock_laser, gpio=mock_gpio)

    # Act: Simulate footpedal not pressed
    mock_gpio.set_footpedal_state(False)
    result = safety.enable_laser()

    # Assert
    assert result is False
    assert safety.laser_enable_permitted is False
    assert mock_laser.is_enabled is False
```

Why This Test is Good: - [DONE] Fast (uses mocks, no hardware) - [DONE] Isolated (one specific scenario) - [DONE] Clear purpose (docstring explains why) - [DONE] Arrange-Act-Assert structure (readable) - [DONE] Explicit assertions (documents expected behavior)

Writing Integration Tests

Integration Test Example:

```
def test_treatment_session_end_to_end() -> None:
    """
    Test complete treatment session workflow.

    Validates:
    1. Session creation
    2. Camera initialization
    3. Ring detection
    4. Treatment protocol execution
    5. Video recording
    6. Database logging
    """
    # Arrange: Set up all components with mocks
    mock_laser = MockLaserController()
    mock_camera = MockCameraController()
    mock_actuator = MockActuatorController()
    mock_gpio = MockGPIOController()

    session_manager = SessionManager(
        laser=mock_laser,
        camera=mock_camera,
        actuator=mock_actuator,
        gpio=mock_gpio
    )

    # Act: Execute full session workflow
    session = session_manager.create_session(patient_id="TEST-001")
    session_manager.start_treatment(protocol="standard_5x5")

    # Simulate operator pressing footpedal
    mock_gpio.set_footpedal_state(True)

    # Simulate treatment completion
    session_manager.wait_for_completion(timeout_s=10.0)

    # Assert: Verify complete workflow
    assert session.status == SessionStatus.COMPLETED
    assert len(session.treatment_events) > 0
    assert session.video_path.exists()
    assert session.total_pulses_delivered == 25  # 5x5 grid
```

Fixture Usage (pytest)

Shared Setup with Fixtures:

```
# conftest.py
import pytest
from tests.mocks import MockLaserController, MockGPIOController

@pytest.fixture
def mock_laser():
    """Provide fresh MockLaserController for each test."""
    return MockLaserController()

@pytest.fixture
def mock_gpio():
    """Provide fresh MockGPIOController for each test."""
    return MockGPIOController()

@pytest.fixture
def safety_manager(mock_laser, mock_gpio):
    """Provide configured SafetyManager for testing."""
    return SafetyManager(laser=mock_laser, gpio=mock_gpio)


# test_safety.py
def test_laser_enable_success(safety_manager, mock_gpio) -> None:
    """Use fixtures to reduce boilerplate."""
    mock_gpio.set_all_interlocks_satisfied(True)

    result = safety_manager.enable_laser()

    assert result is True
```

Benefits: - [DONE] Reduces duplicate setup code - [DONE] Ensures clean state (fresh fixture per test) - [DONE] Improves readability (clear dependencies)

Parameterized Tests

Test Multiple Scenarios Efficiently:

```
import pytest
```

```python
@pytest.mark.parametrize("power,expected_valid", [
    (0.0, False),      # Zero power invalid
    (1.0, True),       # Minimum valid power
    (50.0, True),      # Mid-range power
    (100.0, True),     # Maximum valid power
    (101.0, False),    # Over maximum invalid
    (-5.0, False),     # Negative power invalid
])
def test_laser_power_validation(power, expected_valid) -> None:
    """Test laser power validation for various inputs."""
    mock = MockLaserController()

    result = mock.set_power(power)

    assert (result is not None) == expected_valid
```

Benefits: - [DONE] Tests multiple cases with single function - [DONE] Clear boundary value testing - [DONE] Easy to add new test cases

---

Continuous Integration

GitHub Actions (Future - Phase 6)

Planned Workflow (.github/workflows/tests.yml):

```yaml
name: Test Suite

on: [push, pull_request]

jobs:
  test:
    runs-on: windows-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python 3.12
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install pytest pytest-cov pytest-qt

      - name: Run tests with coverage
        run: |
          pytest --cov=src --cov-report=xml tests/

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml

      - name: Enforce coverage threshold
        run: |
          pytest --cov=src --cov-fail-under=90 tests/
```

Benefits: - [DONE] Automatic testing on every commit - [DONE] Prevents merging broken code - [DONE] Coverage enforcement (fail if <90%) - [DONE] Historical coverage tracking

Pre-Commit Hooks (Local)

Planned `.pre-commit-config.yaml`:

```yaml
repos:
  - repo: local
    hooks:
      - id: pytest
        name: pytest
        entry: pytest
        args: [--cov=src, --cov-fail-under=90]
        language: system
        pass_filenames: false
        always_run: true
```

Benefits: - [DONE] Catches failures before push - [DONE] Faster feedback loop - [DONE] Reduces CI/CD failures

---

FDA Validation Requirements

Design Verification Testing

IEC 62304 Requirements:

1. **Test Plan** (this document + test files)
2. **Test Cases** (pytest test functions)
3. **Test Results** (pytest output + coverage report)
4. **Traceability Matrix** (requirements ↔ tests)

Traceability Matrix Example

Requirement ID	Requirement Description	Test File	Test Function
SAFE-001	All interlocks must pass for laser enable	`tests/test_safety.py`	`test_laser_enable_requires_all_interlocks()`
SAFE-002	Emergency stop disables laser only	`tests/test_safety.py`	`test_emergency_stop_selective_shutdown()`
SAFE-003	Laser disabled on interlock loss	`tests/test_safety.py`	`test_laser_disable_on_interlock_loss()`

PROT-001	Protocol validates power limits	`tests/test_protocol.py`	`test_protocol_power_limit_validation()`
AUDIT-001	All treatment events logged	`tests/test_database.py`	`test_treatment_event_logging()`

Test Documentation for FDA Submission

Required Artifacts: 1. [DONE] Test Architecture (this document) 2. [DONE] Test Plan (test organization + fixtures) 3. [DONE] Test Cases (all test functions with docstrings) 4. [DONE] Test Results (pytest HTML report) 5. [DONE] Coverage Report (pytest-cov HTML report) 6. [DONE] Traceability Matrix (requirements ↔ tests) 7. [PENDING] Design Verification Report (summary of testing)

References

Testing Frameworks

- **pytest:** https://docs.pytest.org/
- **pytest-cov:** https://pytest-cov.readthedocs.io/
- **pytest-qt:** https://pytest-qt.readthedocs.io/

FDA Guidance

- **FDA Software Validation Guidance:** General Principles of Software Validation
- **IEC 62304:** Medical device software - Software life cycle processes
- **FDA 21 CFR Part 820:** Quality System Regulation (Design Controls)

Best Practices

- **Test-Driven Development:** Kent Beck, “Test-Driven Development By Example”
- **Unit Testing Best Practices:** Martin Fowler, “Mocks Aren’t Stubs”
- **Medical Device Testing:** AAMI TIR45, Guidance on the use of AGILE practices

---

Document Owner: Test Architect **Last Updated:** 2025-10-26 **Next Review:** Before Phase 6 implementation **Status:** Implemented - 30 test files, MockHardwareBase pattern operational