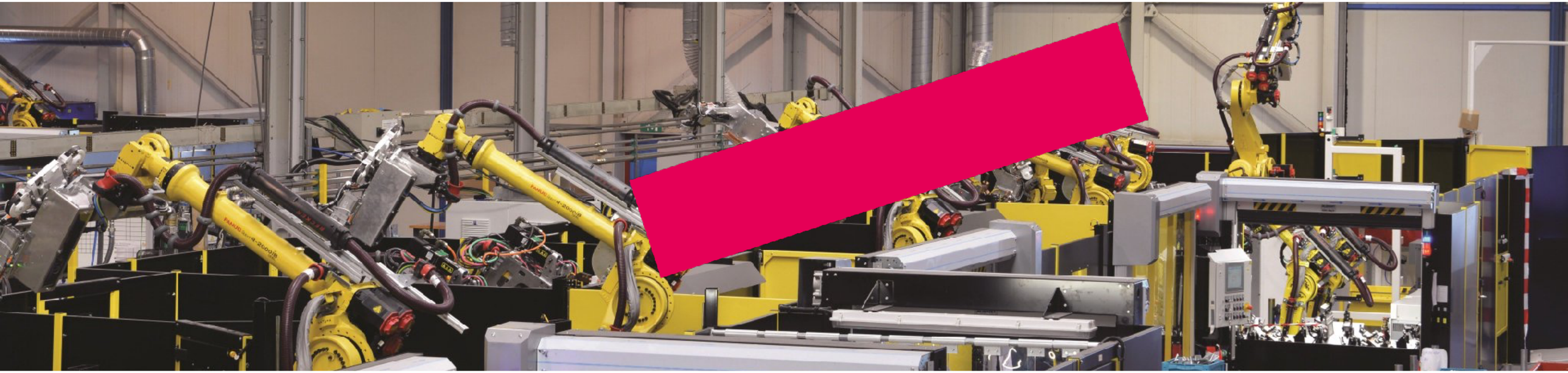
Programming 6



Workshop on CMake

johan.korten@han.nl

V1.0 Nov 2025

HAN_UNIVERSITY
OF APPLIED SCIENCES

Workshop Topics overview

Clean code (beyond SOLID)

CMake

Unit testing

Commentaar

Patterns (Clean architecture)

Workshop Topics for Today

- **Testing in General**
- **V-Model vs Testing Pyramid**

General Warning on Testing

“Program testing can be used to show the presence of bugs,
but never to show their absence!”

— Edsger W. Dijkstra

General Warning on Testing

“Program testing can be used to show the presence of bugs,
but never to show their absence!”

— Edsger W. Dijkstra

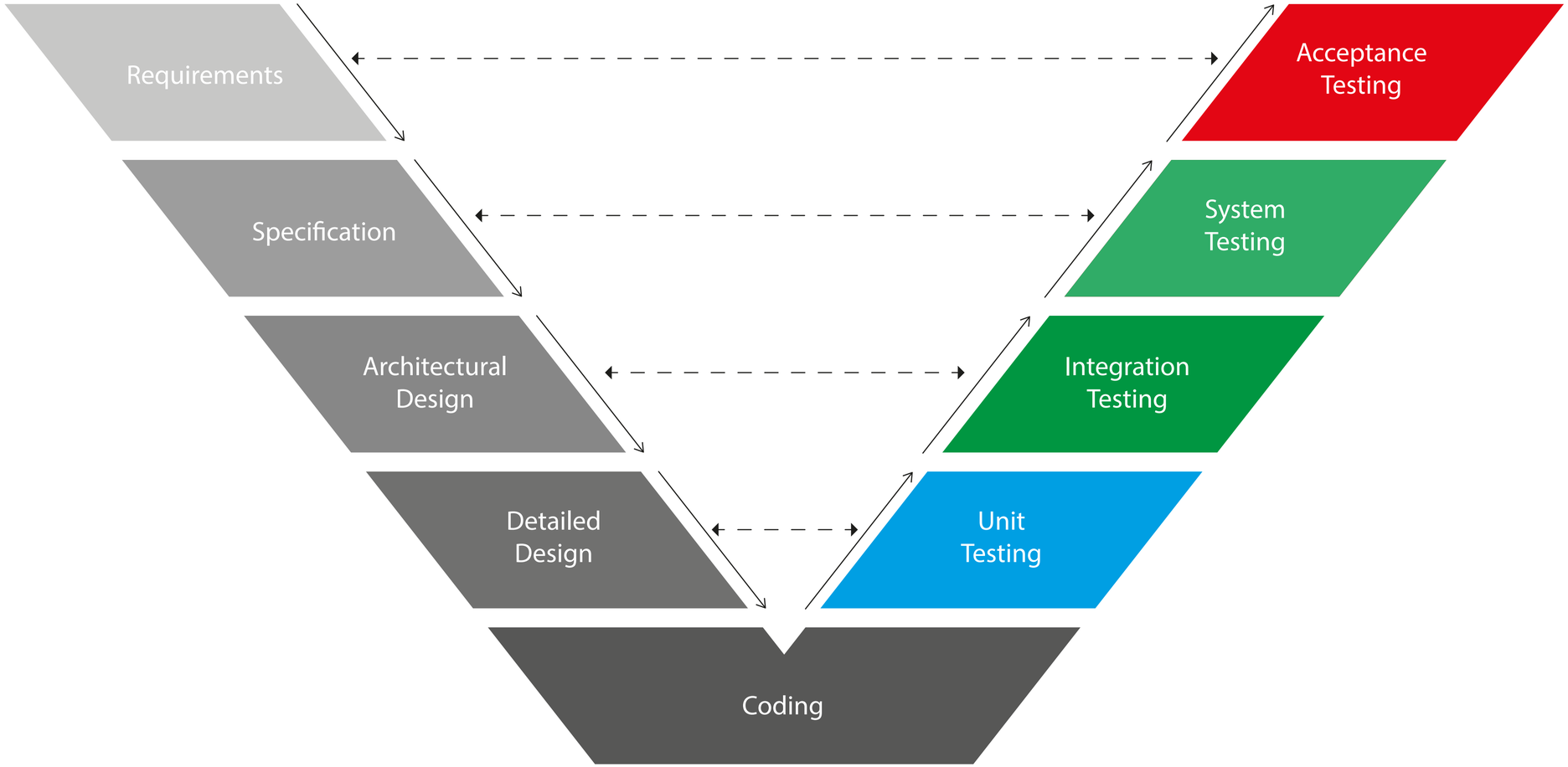
- Testing is inherently incomplete.
- You can run a million tests and still miss the one input that breaks everything.
- Tests prove "this scenario works," not "all scenarios work."

Testing Focus

- **V-model** = *what* and *when* (traceability to requirements, phase gates)
- **Pyramid** = *how many* (economics of test distribution)
- **Both agree**: unit tests are the foundation

Essential for e.g. medical device development (IEC 62304) or automotive (ISO 26262)

V-Model


Requirements
Specification
Architectural Design
Detailed Design
Coding
Unit Testing
Integration Testing
System Testing
Acceptance Testing

V-Model: Acceptance Test

- Validation
- With real users/environment

V-Model: System Test

- Hardware + Software
- Real target, real peripherals

V-Model: Integration Testing

Software Integration:

- Simulated Hardware

Hardware Integration:

- Board Bring-up

Unit Testing

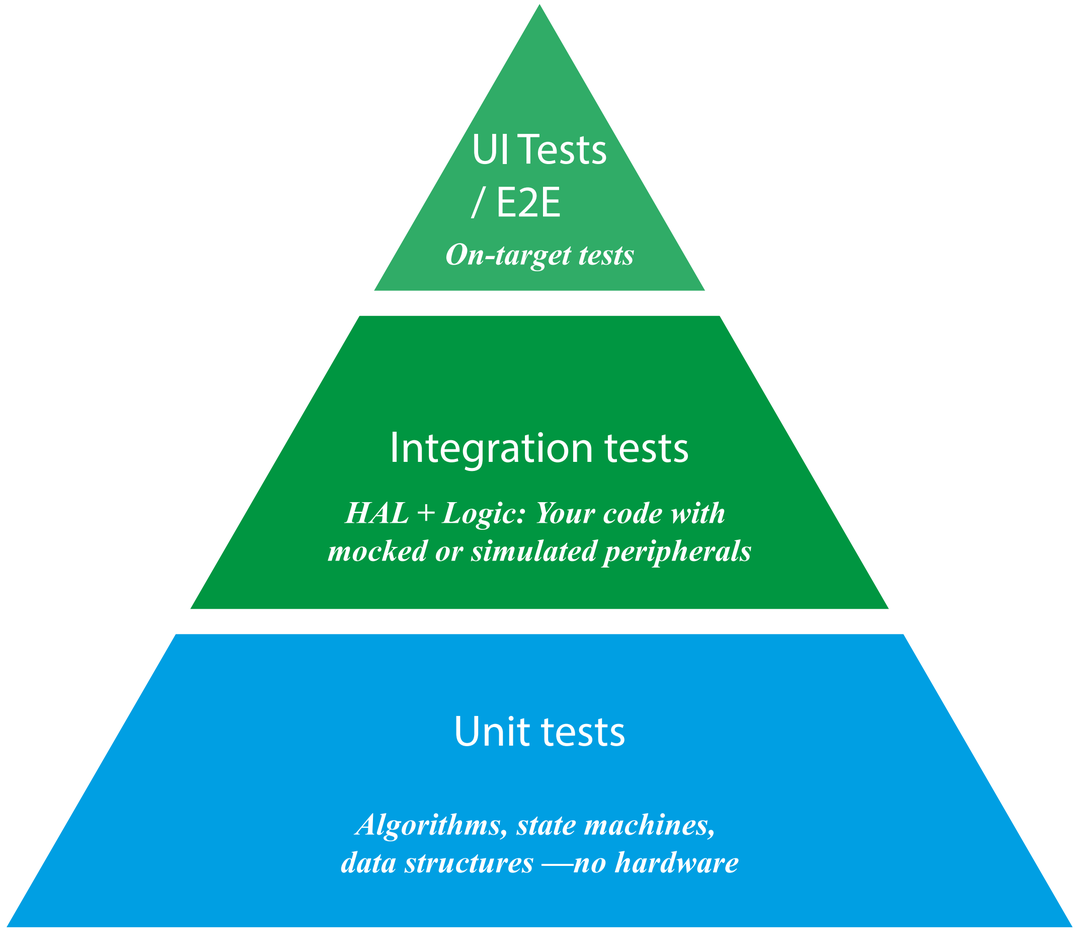
Software Unit Testing:

- Today’s Main Topic

Hardware Unit Testing:

- Component Test

Testing Pyramid vs. V-Model



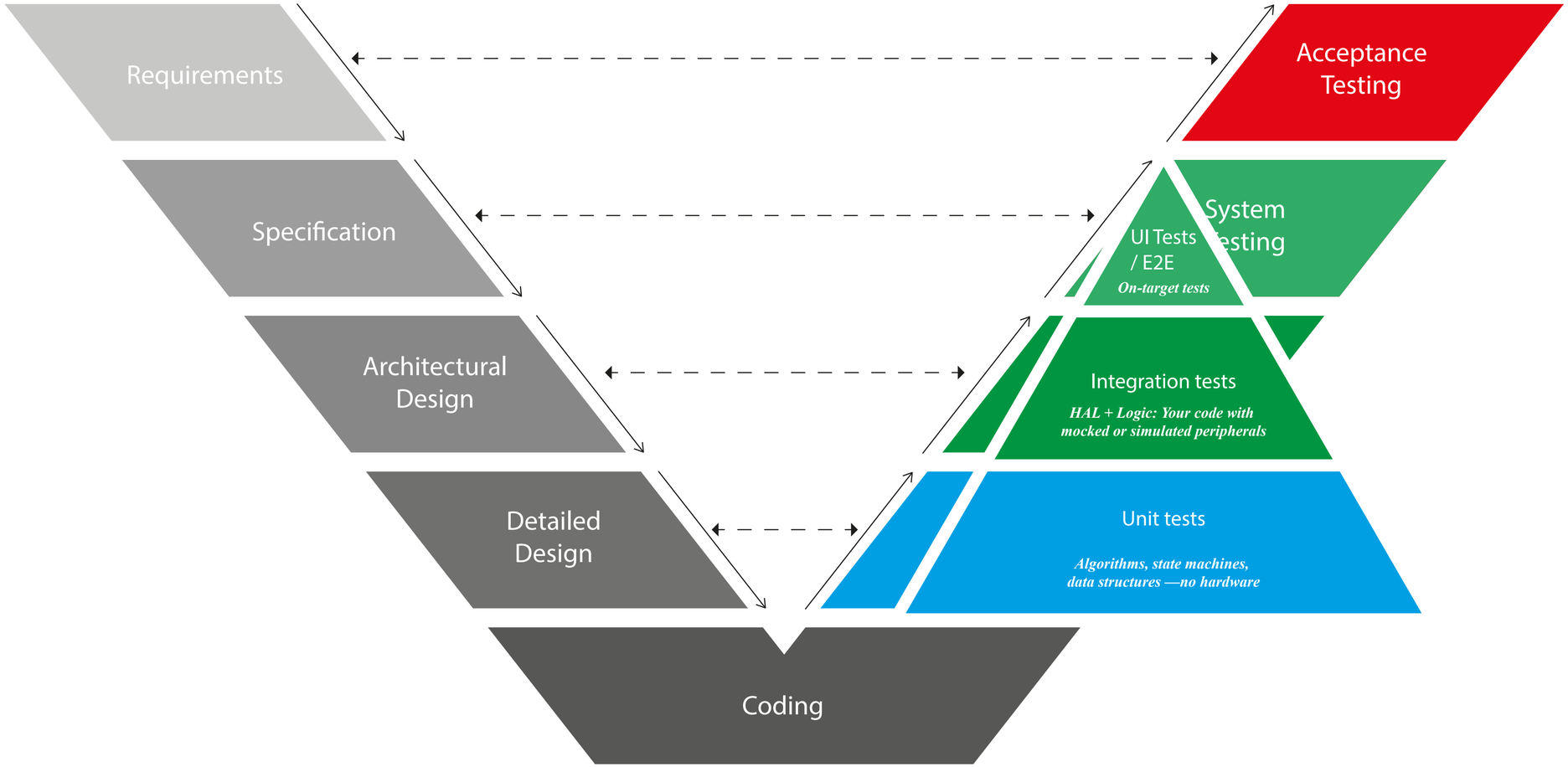
Testing versus Verification

Computer Science Theorists (Dijkstra formalized this) prefer Verification over Testing, but:

- Formal verification is expensive and limited in scope
- Testing catches many bugs cheaply

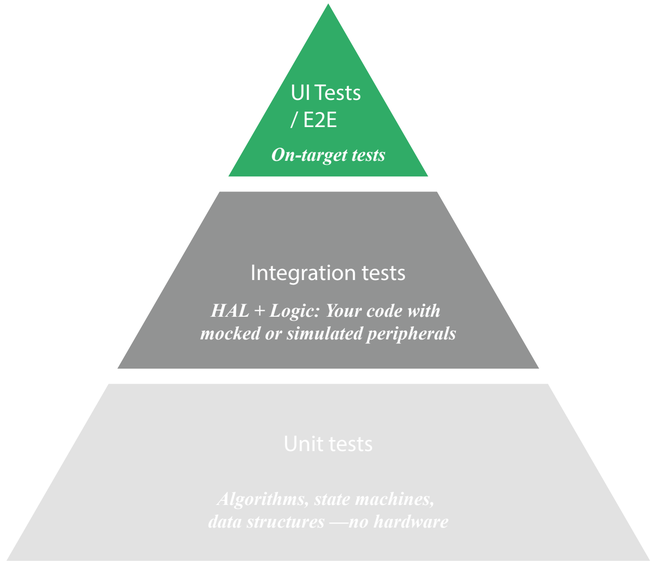
Approach	Proves	Cost
Formal verification	Correctness (for specific properties)	Very high
Unit tests	Stops execution entirely—timing-sensitive code breaks	Low
Integration/system tests	Changes memory layout, stack usage, cache behavior	Medium
No testing	Nothing	Zero (until production fails)

Testing Pyramid vs. V-Model



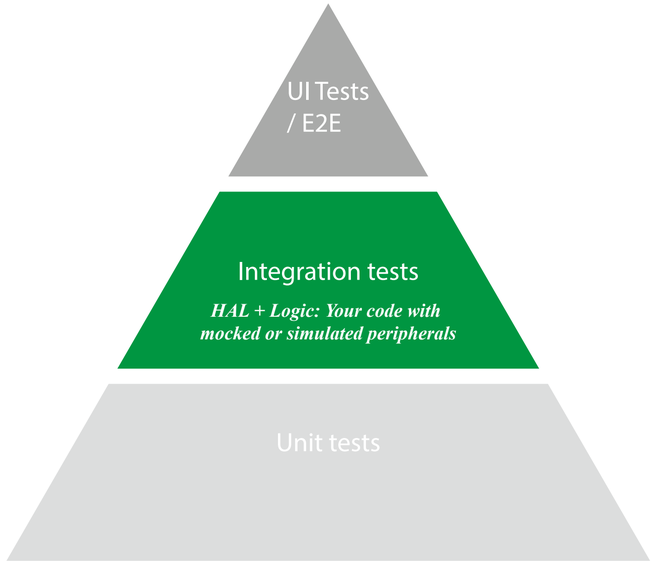
UI / End-to-end / Hardware tests (top, fewest tests)

- Real target, real peripherals
- Slow, expensive, sometimes flaky
- Essential but not where you want to catch bugs



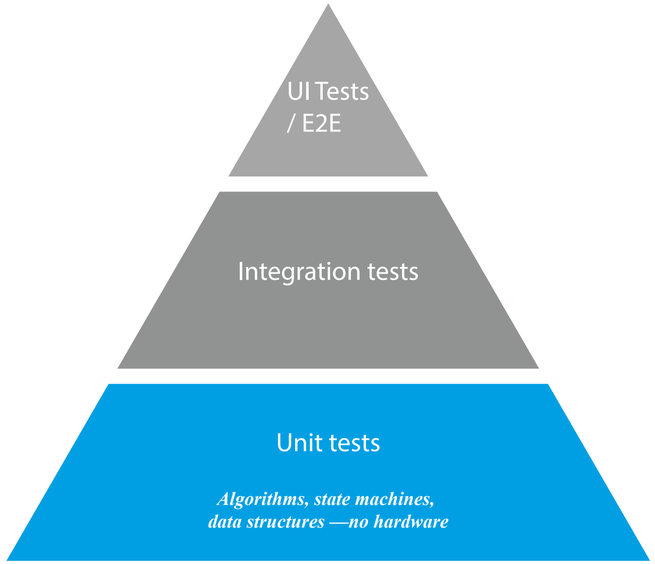
Integration tests (middle)

- Test interactions between components
- May include HAL with simulated peripherals
- Slower, but still on PC or emulator

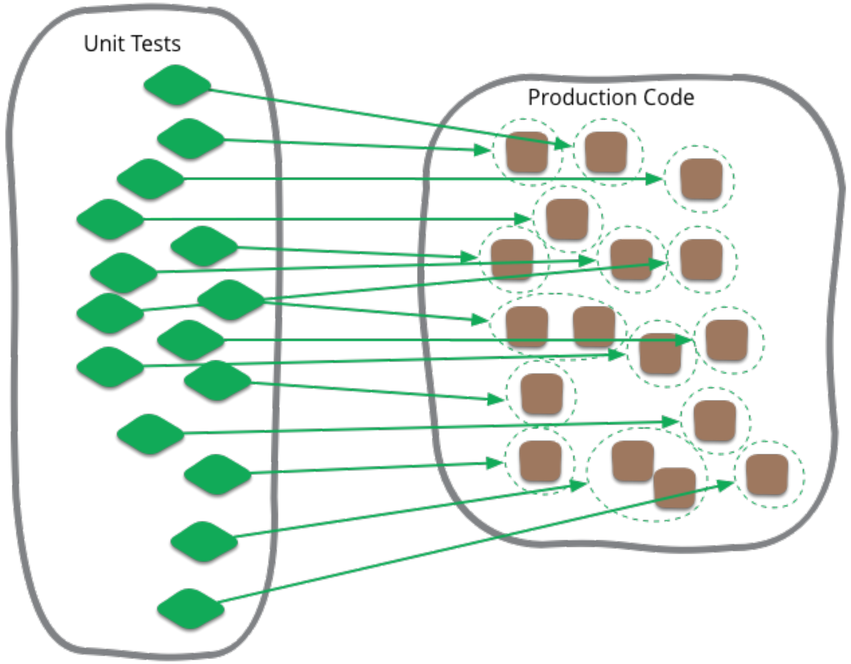


Unit tests (base, most tests)

- Fast: milliseconds per test
- Cheap: no hardware needed
- Precise: failures pinpoint the problem
- Stable: no flaky hardware timing issues



Unit Testing



Basic Unit Test mechanics

```cpp
#include <iostream>

void check_equal(int expected, int actual, const char* message) {
    if (expected != actual) {
        std::cerr << "FAIL: " << message
                  << " (expected " << expected
                  << ", got " << actual << ")\n";
    } else {
        std::cout << "PASS: " << message << "\n";
    }
}

int main() {
    check_equal(4, 2 + 2, "addition works");
    check_equal(0, 1 - 1, "subtraction works");
    check_equal(5, 2 + 2, "this should fail");
    return 0;
}
```

Tests that can't fail…

No Asserts

```
1 TEST(TemperatureController, HeaterTurnsOnWhenCold) {
2     // Student forgets to add assertion
3     TemperatureController controller{20.0f};
4     controller.update(15.0f);
5     // No CHECK or LONGS_EQUAL — test passes but proves nothing
6 }
```

Tests that can't fail…

Tautology

```
1 TEST(CircularBuffer, IsEmptyAfterCreation) {
2     CircularBuffer buffer{10};
3     CHECK_FALSE(buffer.isEmpty());  // Inverted logic –
4                                     // passes if bug exists
5 }
```

Tests that can't fail…

Wrong assertion

```
1 TEST(Range, ContainsValue) {
2     Range r{0, 100};
3     int value = 50;
4     CHECK_TRUE(value == value);  // Oops — not testing Range at all
5 }
```

C++ Unit Testing Frameworks

Framework	Style	Embedded Suitability	Notes
Catch2	Modern, header-heavy	Good	Nice syntax, no macros for test registration, slow compile
GoogleTest	Full-featured	Good	Industry standard, mature, more setup
doctest	Lightweight Catch2	Excellent	Fastest compile, minimal footprint, Catch2-like syntax
Unity	Pure C	Excellent	ThrowTheSwitch, tiny, runs anywhere
CppUTest	Embedded-focused	Excellent	Made for embedded, James Grenning's book uses it
µTest	Minimal	Excellent	Header-only, tiny

CppUTest

- Purpose-built for embedded
- James Grenning's *Test-Driven Development for Embedded C* uses it
- Better memory leak detection out of the box
- More C-friendly if they mix C and C++
- Has CppUMock built in

Please pull the repo

https://github.com/AEAEmbedded/ESE_PROG/tree/main/Workshops/UnitTesting

https://github.com/AEAEmbedded/ESE_PROG.git

MinimalTest using CppUTest

Case 0

```
MinimalTest
    ├── CMakeLists.txt
    ├── main.cpp
    └── test_first.cpp
```

MinimalTest using CppUTest

Case 0

- main.cpp

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int argc, char** argv) {
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

MinimalTest using CppUTest

Case 0

- CMake to fetch CppUTest

```
# Fetch CppUTest
include(FetchContent)
FetchContent_Declare(
    CppUTest
    GIT_REPOSITORY https://github.com/cpputest/cpputest.git
    GIT_TAG master
)
```

MinimalTest using CppUTest

Case 0

- test_first.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest) {
};

TEST(FirstTest, TrueIsTrue) {
    CHECK_TRUE(true);
}

TEST(FirstTest, OnePlusOneIsTwo) {
    LONGS_EQUAL(2, 1 + 1);
}
```

TemperatureTest using CppUTest

Case 1

```
TemperatureTest/
    ├── CMakeLists.txt
    ├── temperature_sensor.hpp
    ├── temperature_sensor.cpp
    ├── test_temperature_sensor.cpp
    └── main.cpp
```

TemperatureTestComplex using CppUTest

Case 3

- Test stateful classes using CppUTest's `TEST_GROUP` with `setup()` and `teardown()`.

```
TemperatureTestComplex/
        ├── CMakeLists.txt
        ├── temperature_filter.hpp
        ├── test_temperature_filter.cpp
        └── main.cpp
```

TemperatureTestMocking using CppUTest

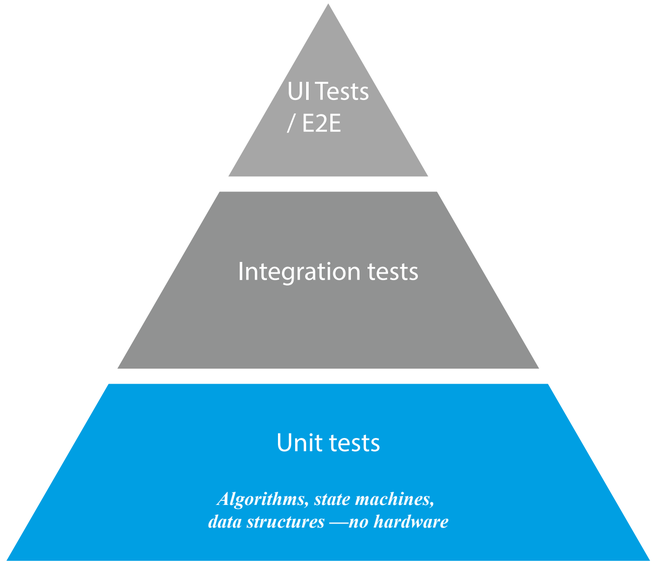
Case 4

- Mocking / Dependency injection (more on Patterns week after next week)

```
TemperatureTestMocking/
    ├── CMakeLists.txt
    ├── i_temperature_sensor.hpp
    ├── i_heater.hpp
    ├── temperature_controller.hpp
    ├── temperature_controller.cpp
    ├── mock_temperature_sensor.hpp
    ├── mock_heater.hpp
    ├── test_temperature_controller.cpp
    └── main.cpp
```

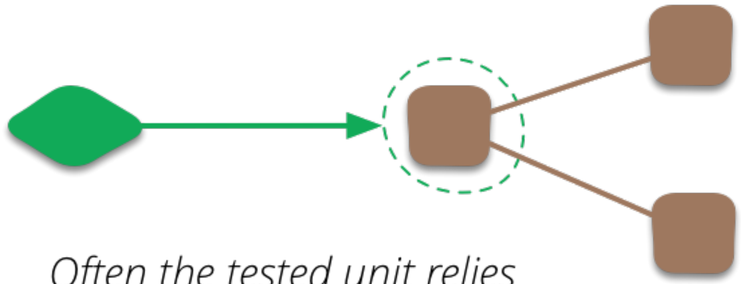
Unit tests (base, most tests)

- Fast: milliseconds per test
- Cheap: no hardware needed
- Precise: failures pinpoint the problem
- Stable: no flaky hardware timing issues



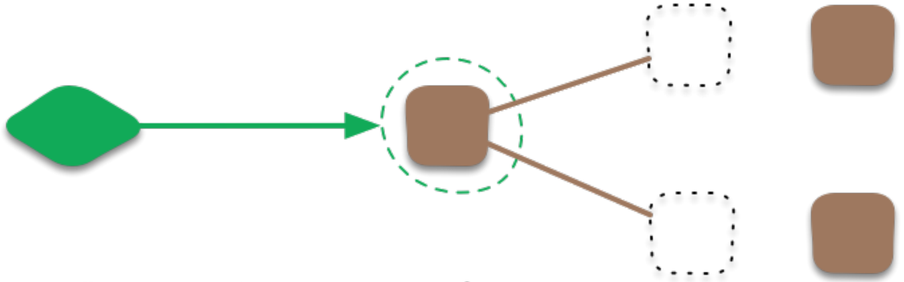
Isolation versus Integration

Sociable Tests



Often the tested unit relies on other units to fulfill its behavior

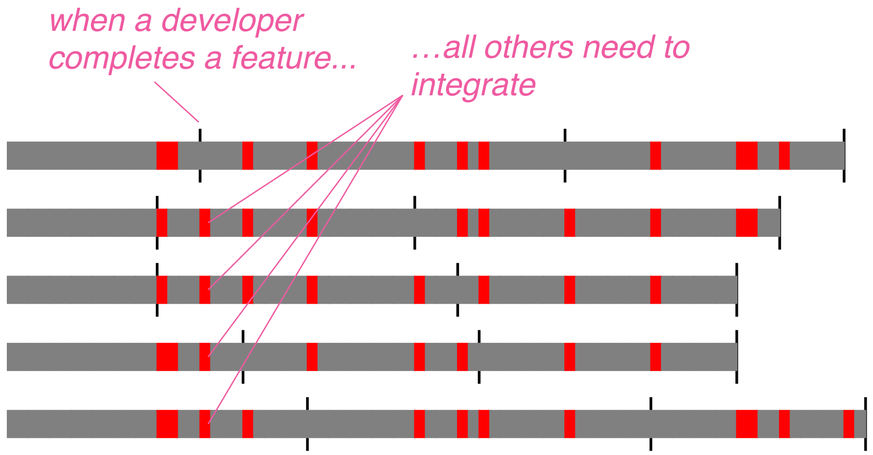
Solitary Tests



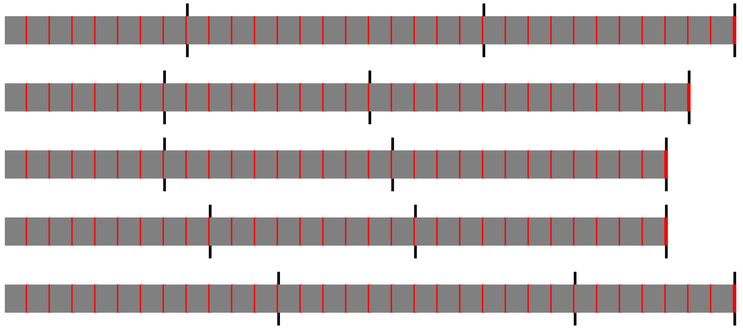
Some unit testers prefer to isolate the tested unit

Continuous Integration

Continuous Integration



Continuous Integration

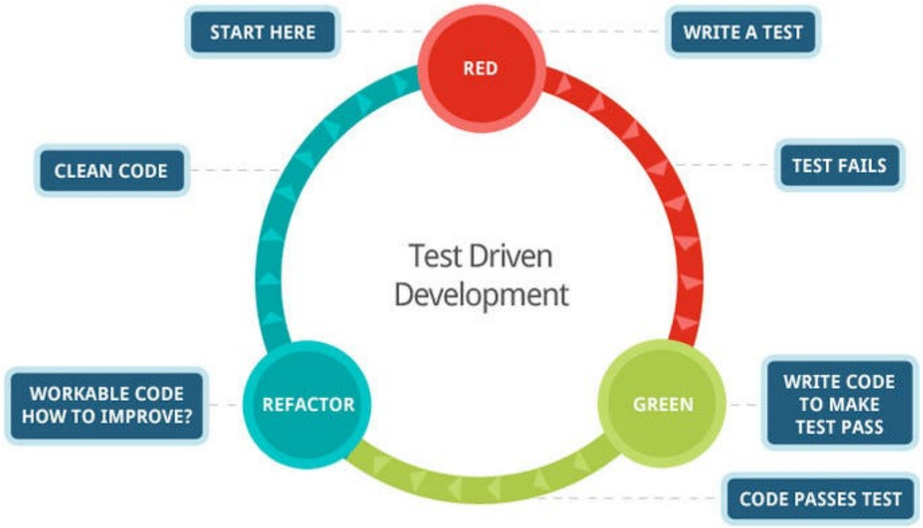


Test Driven Development / XP

Kent Beck: *Test-Driven Development: By Example, 2002* (Together with Ward Cunningham)

Extreme Programming emphasizes short feedback loops, incremental development, and code that's always working.

Integrating Unit Testing: Test Driven Development (TDD)



Test Driven Development (TDD)

For embedded systems specifically:

TDD lets you develop and verify logic on your PC, fast, without waiting for hardware.
The discipline of writing testable code forces you to separate your algorithms from your hardware abstractions — which is exactly the architecture you want anyway.

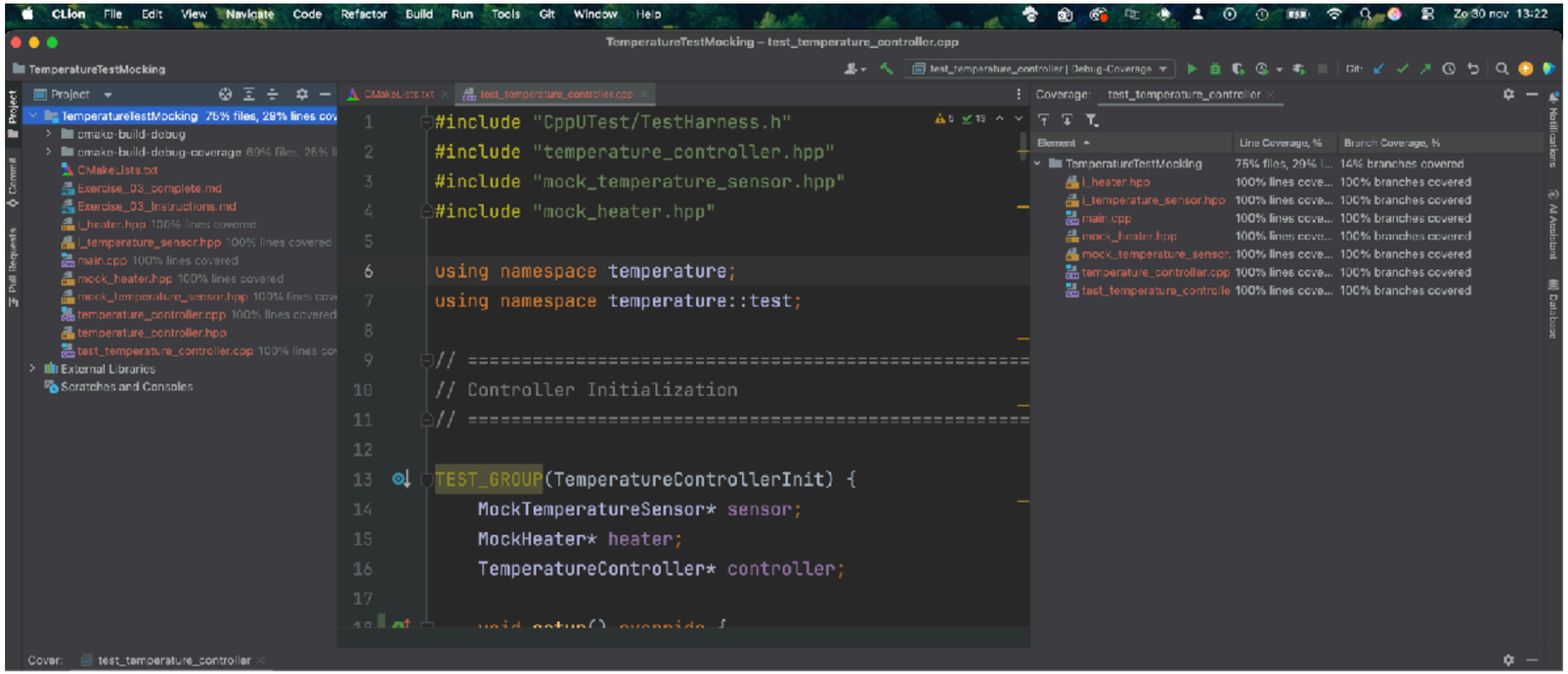
What TDD is not:

- a guarantee of correctness,
- a replacement for integration testing,
- or a rule that every line must be test-driven.

It's a tool for design feedback. Use it where it helps.

IDE influence on testing

CLion has native test coverage support



VSC use a “Coverage Gutters” extension

One more thing…

Back to Dijkstra.

- Testing can't prove correctness—only find bugs

The Heisenberg problem: Observing can hide bugs—instrumentation changes behavior

Heisenberg's Uncertainty Principle (Quantum Mechanics)

You can't precisely measure both position and momentum of a particle simultaneously.

Observing it disturbs it.

Heisenberg problem in debugging

When you add instrumentation to observe your program, you change its behavior:

The Classic Symptoms

- Bug disappears when you add printf to find it
- Bug only happens in release build, not debug
- Race condition vanishes when you set a breakpoint
- Interrupt timing bug that only occurs without the debugger attached

Heisenberg problem of debugging

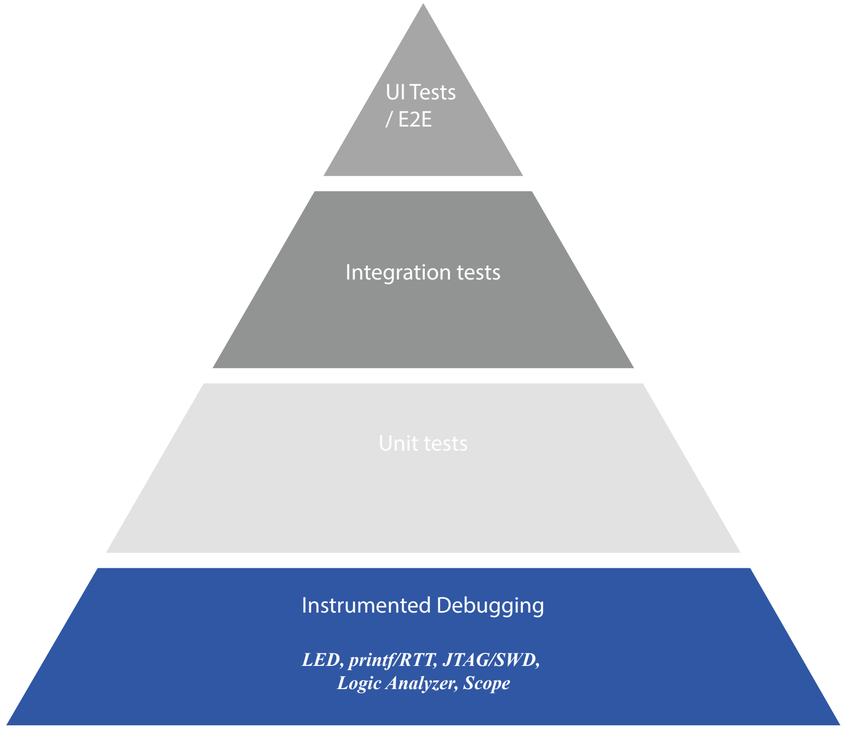
Instrumentation	How It Disturbs
printf	Takes CPU cycles, blocks on UART, changes timing by milliseconds
Breakpoint (JTAG)	Stops execution entirely—timing-sensitive code breaks
Extra variables for inspection	Changes memory layout, stack usage, cache behavior
Debug build (-O0)	Different code paths than release (-O2), different timing

Hardware Unit Testing: Instrumented Debugging

- JTAG, SWD, JLink / ST-Link
- Ozone Debugger
- Segger SystemView

Expensive tools like:

- Lauterbach TRACE32 Debug- and Trace Solutions



Instrumentation Debugging

Tool	What It Shows	Strengths	Limitations
printf / RTT	Program flow, variable values	Easy, familiar	Slow, intrusive, manual interpretation
JTAG / SWD	Breakpoints, memory, registers, call stack	Non-intrusive stop, realtime inspection	Stops execution, changes timing
Logic Analyzer	Digital signal timing, protocols (SPI, I2C, UART)	Exact timing, protocol decode	External view only, no internal state
Oscilloscope	Analog signals, rise times, noise, glitches	Ground truth for electrical behavior	No software insight
Segger SystemView / Tracealyzer	RTOS task timing, events	System-level behavior over time	Requires instrumentation

Trace Debugging

Trace Type	ARM Core	What It Captures
ETM (Embedded Trace Macrocell)	Cortex-M3/M4/M7, Cortex-A	Full instruction trace, branch history
ITM (Instrumentation Trace Macrocell)	Cortex-M3/M4/M7	Software-generated trace (like fast printf)
MTB (Micro Trace Buffer)	Cortex-M0+	Lightweight trace to on-chip RAM
ETB (Embedded Trace Buffer)	Various	On-chip circular buffer for trace
Segger SystemView / Tracealyzer	RTOS task timing, events	System-level behavior over time

Final Message: Dijkstra + Heisenberg problem

Testing is doubly limited: it can't prove absence of bugs, and the act of testing might hide bugs that exist

State space is enormous — timing, interrupts, hardware states, race conditions

Observation is intrusive — printf/breakpoints change the system

Real environment is unpredictable — EMI, voltage sag, temperature

Final Message: Dijkstra + Heisenberg problem

Dijkstra said testing can only show the presence of bugs, never their absence. He's right. But showing the presence of bugs is exactly what we want—find them before the customer does.

Unit tests won't prove your code is correct. They'll catch the mistakes you make at 4pm on a Friday. That's valuable!

“

That’s all…

Any questions?