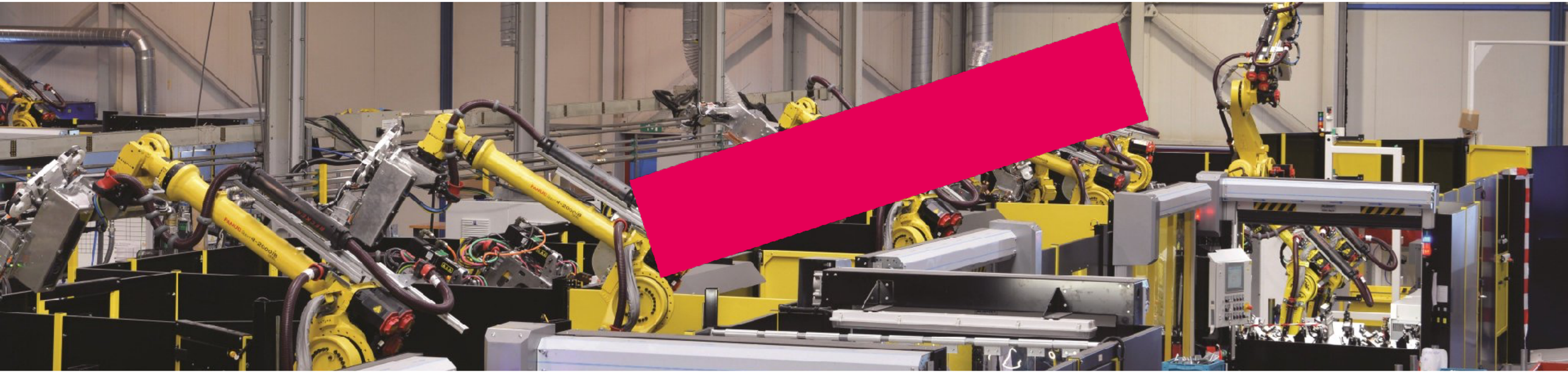
Programming 6



Workshop on Commenting

johan.korten@han.nl

V1.0 Nov 2025

HAN_UNIVERSITY
OF APPLIED SCIENCES

Workshop Topics overview

Clean code (beyond SOLID)

CMake

Unit testing

Commenting

Patterns (Clean architecture)

Overview of Today's Topics

- The self-documenting code debate
- The "why" of commenting
- Different levels of documentation
- Practical exercises
- The role of AI in commenting and documenting

Reasons why we don't comment

- Good code is self-documenting

- Commenting takes up too much of our time
- Comments become out of date and become misleading
- Most comments I have seen are all worthless so why bother

John Ousterhout (2022)

Remember: Reading code

"Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code.”

Robert C. Martin (2008)

What the heck is this code…?!

```
7   if (isnan(h) || isnan(t) || isnan(f)) {
8     Serial.println("Failed to read from sensor!");
9     return;
10  }
11
12  float hif = sensor.computeHeatIndex(f, h);
13  float hic = sensor.computeHeatIndex(t, h, false);
```

- What are h, t, f, hif and hic?

What the heck is this code…?!

```
7   if (isnan(h) || isnan(t) || isnan(f)) {
8     Serial.println("Failed to read from sensor!");
9     return;
10  }
11
12  float hif = sensor.computeHeatIndex(f, h);
13  float hic = sensor.computeHeatIndex(t, h, false);
```

- What are h, t, f, hif and hic?
- How do you know?

What the heck is this code…?!

```
7   if (isnan(h) || isnan(t) || isnan(f)) {
8     Serial.println("Failed to read from sensor!");
9     return;
10  }
11
12  float hif = sensor.computeHeatIndex(f, h);
13  float hic = sensor.computeHeatIndex(t, h, false);
```

- What are h, t, f, hif and hic?

- How do you know?

- Now imagine you're debugging this at 2 AM, six months from now, in a project with three different sensors. Still confident about what *h* means?

- What if this is running on a medical device and someone needs to verify what's being measured? Do they have to guess that *f* is Fahrenheit, or should the code tell them?

What the heck is this code…?!

```
1  void loop() {
2    delay(2000);
3    float h = dht.readHumidity();
4    float t = dht.readTemperature();
5    float f = dht.readTemperature(true);
6
7    if (isnan(h) || isnan(t) || isnan(f)) {
8      Serial.println("Failed to read from DHT sensor!");
9      return;
10   }
11
12   float hif = dht.computeHeatIndex(f, h);
13   float hic = dht.computeHeatIndex(t, h, false);
14 }
```

Two confessions:

- I tricked you a little bit, this is the actual code
- I did remove the comments and the final output

What the heck is this code…?!

Let’s have a look at this code:

https://github.com/adafruit/DHT-sensor-library/blob/master/examples/DHTtester/DHTtester.ino

Ok we are the ‘lucky’ few who can fairly easily read this code because we happen to know the sensor etc.

(See also our Repo / DHTtester.ino)

What the heck is this code…?!

This is *official library example code*, written by experienced developers at Adafruit, and it's completely dependent on comments to be comprehensible.

Without comments, ‘professional-grade’ code from MIT-educated engineers becomes opaque.

If Lady Ada – one of the most respected figures in the Arduino/maker community – writes code that entirely relies on comments to understand, what does that say about the "good code is self-documenting" argument?

What the heck is this code…?!

```
/*
 We just proved: Without comments, even this trivial code becomes ambiguous.
 This sets up beautifully for your Ousterhout quote later: ”The process of writing comments,
 if done correctly, will actually improve a system's design.”

 Because if Adafruit had been forced to make this code self-documenting, they might have
 written:
*/
float humidity_percent = ambientSensor.readHumidity();
float temperature_celsius = ambientSensor.readTemperature();
float temperature_fahrenheit = ambientSensor.readTemperature(true);

// No comments needed. The act of trying to remove the comment dependency would have improved
the design.
```

The Commenting Dilemma

Ok so now we have found ourselves a big dilemma:

- Comment everything
- Comment nothing

The Commenting Dilemma: comment everything

Comment everything:

- “Every line must have a comment"
- Results in noise:

```
i++; // increment i
```

- Comments become wallpaper – ignored
- Doubles the maintenance burden
- Comments drift out of sync with code

The Commenting Dilemma: comment everything

Comment nothing:

- “Good code is self-documenting"
- Works fine... until it doesn't
- Assumes perfect naming (rare)
- Assumes reader knows the domain (dangerous)
- Assumes code can express intent and why (it can't)

What/How versus Why

Code tells you how. Comments tell you why.

What the code does → should be obvious from the code itself

Why the code does it → that's what comments are for

What/How versus Why

Code tells you how. Comments tell you why.

What the code does → should be obvious from the code itself

Why the code does it → that's what comments are for

```
// Using 2-second delay because SHT31 datasheet specifies
// minimum 1.5s between measurements for accurate readings
delay(2000);
```

Documentation Challenges: three fragile knowledge sources

Nawrocki et. al (2002): Extreme Programming and its development practices.

There are three sources of knowledge about the software that are required but are hard to maintain in the long run:

- the code,
- test cases; and
- the memory of the developers.

Documentation Challenges: Why documentation fails

- Informal documentation is hard to understand.
- Documentation is considered waste.
- Productivity is measured by the amount of working software only.
- Documentation is out-of-sync with the software.
- Short-term focus.

In memory of our late colleague Theo Teunissen (2025).

Theo's tribute: Reasons why we don't comment

The TL;DR culture problem – Diminished attention spans and "digital natives" are processing information differently.

Documentation ≠ waste – Lean interpretation that documentation is waste, don't document too much, make "just enough" documentation.

Theunissen, 2023

Theo's tribute: Knowledge Vaporization

Knowledge vaporization – This is his central concept: the reduction in documentation quantity and quality due to Agile/Lean/DevOps leads to knowledge evaporating over time. The three knowledge sources from Nawrocki (code, test cases, developer memory) are particularly vulnerable.

Theunissen, 2023

Why do we still need commenting

Software is complex;

Embedded systems are arguably even more complex;

Diagrams and comments or essential to document these systems.

Goals:

- Describing our code and structures in Natural Language
- Improving understandability
- Hiding complexity

The Commenting Paradox

“The process of writing comments, if done correctly, will actually improve a system’s design.”

- John Ousterhout (2022), Stanford University

The Commenting Paradox

“The overall idea behind comments is to capture information that was in the mind of the designer but couldn’t be represented in the code.”

- John Ousterhout (2022), Stanford University

The Commenting Paradox

Robert C. Martin: “Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.

Comments are not like Schindler's List. They are not "pure good." Indeed, comments are, at best, a necessary evil. If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much -- perhaps not at all.

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

So when you find yourself in a position where you need to write a comment, think it through and see whether there isn't some way to turn the tables and express yourself in code. Every time you express yourself in code, you should pat yourself on the back. Every time you write a comment, you should grimace and feel the failure of your ability of expression.”

The Commenting Paradox

Martin	Ousterhout
Comments = failure to express in code	Comments = different *type* of expression
Code is the only truth	Code says *what*, comments say *why*
Minimise comments	Write *good* comments
Comments lie	*Unmaintained* comments lie

Martin sees comments as a *fallback* for *inadequate code*.

Ousterhout sees comments as a *complement to code* – both need to be well-written, and good comments actually improve design by forcing you to think clearly about intent.

The Commenting Paradox

```
const uint8_t WATCHDOG_TIMEOUT_MS = 120;
const uint8_t SAFETY_MARGIN_MS = 25;
const uint8_t MAX_PROCESSING_DELAY_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS;

void processData() {
    readSensors();
    calculateOutput();
    delay(MAX_PROCESSING_DELAY_MS);
    transmitResult();
}
```

- Is this code well-written?

The Commenting Paradox

```
const uint8_t WATCHDOG_TIMEOUT_MS = 120;
const uint8_t SAFETY_MARGIN_MS = 25;
const uint8_t MAX_PROCESSING_DELAY_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS;

void processData() {
    readSensors();
    calculateOutput();
    delay(MAX_PROCESSING_DELAY_MS);
    transmitResult();
}
```

- Is this code well-written?
- Why was it written the way it was?

The Commenting Paradox

```
const uint8_t WATCHDOG_TIMEOUT_MS = 120;
const uint8_t SAFETY_MARGIN_MS = 25;
const uint8_t MAX_PROCESSING_DELAY_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS;

void processData() {
    readSensors();
    calculateOutput();

    // See safety certification document SEC-2024-0042.
    delay(MAX_PROCESSING_DELAY_MS);

    transmitResult();
}
```

- Is this better?
- Could we do without the comment?

The Commenting Paradox

```
const uint8_t WATCHDOG_TIMEOUT_MS = 120;
const uint8_t SAFETY_MARGIN_MS = 25;
const uint8_t MAX_PROCESSING_DELAY_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS;

void processData() {
    readSensors();
    calculateOutput();

    // Rate limit: CAN bus protocol requires minimum interval between frames.
    // See safety certification document SEC-2024-0042.
    delay(MAX_PROCESSING_DELAY_MS);

    transmitResult();
}
```

- Is this better?
- Could we really do without the comment?

The Commenting Paradox

```
// main.cpp

void processData() {
    readSensors();
    calculateOutput();
    waitForNextTransmissionWindow();
    transmitResult();
}

// goryDetails.hpp

const uint8_t WATCHDOG_TIMEOUT_MS = 120;
const uint8_t SAFETY_MARGIN_MS = 25;
const uint8_t TRANSMISSION_INTERVAL_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS;

/**
 * Rate limit: CAN bus protocol requires minimum interval between frames.
 *
 * WARNING: Do not exceed 100ms - watchdog resets MCU at 120ms.
 * See safety certification document SEC-2024-0042.
 */
void waitForNextTransmissionWindow() {
    delay(TRANSMISSION_INTERVAL_MS);
}
```

The Commenting Paradox: Keeping Martin and Ousterhout happy…

Level	What you see	What you need
High (caller)	waitForNextTransmissionWindow()	Just the name
Low (implementation)	The delay, the constants, the comment	The full story

- processData() reads like a story – no comments needed at that level
- The what is in the function name: waitForNextTransmissionWindow()
- The why is documented once, where the implementation lives
- A curious developer can drill down; a casual reader doesn't need to

"Hiding complexity" done right.

The Commenting Paradox

The code *can* say:

- waitForNextTransmissionWindow() – what we're doing
- WATCHDOG_TIMEOUT_MS = 120 – what the value is
- TRANSMISSION_INTERVAL_MS = WATCHDOG_TIMEOUT_MS - SAFETY_MARGIN_MS – how it's calculated

The code *cannot* say:

- Why the watchdog is set to 120ms (hardware/safety decision)
- Where that requirement comes from (SEC-2024-0042)
- What happens if you violate it (MCU reset)
- Why this matters (safety certification)

The Commenting Paradox

Before you write a comment, ask: “Could I express this in a better variable name, function name, or code structure?”

If *yes* → *refactor the code*, *delete the comment*.

If *no* → *write the comment*. No guilt. No grimace.

The Commenting Paradox

Things that are never obvious from code:

- Why this value and not another (datasheet, protocol, clinical validation)
- Why this sequence matters (hardware errata, race conditions)
- Why this code exists at all (business rule, safety requirement, bug workaround)
- Where to find more information (document references, ticket numbers)
- What happens if you change it (consequences, warnings)
- Who approved it (regulatory traceability)

The Commenting Paradox

Code tells you *what* and *how*.

Comments tell you *why* and *where* to find more.

Exercise 1: Strip and suffer

"What does this code do? What are these values?"

Reasons why we don't comment

Levels of documentation: Inline → function → module → architectural

Clean code, clear comments

Levels of documentation: **Inline** → function → module → architectural

```
delay(15);
```

Clean code, clear comments

Levels of documentation: **Inline** → function → module → architectural

```
const uint8_t SHT31_HIGH_REPEATABILITY_DELAY_MS = 15;  // Datasheet Table 4

delay(SHT31_HIGH_REPEATABILITY_DELAY_MS);
```

Clean code, clear comments

Levels of documentation: **Inline** → function → module → architectural

```
const uint8_t SHT31_DATASHEET_TABLE4_HIGH_REPEATABILITY_MAX_MS = 15;

delay(SHT31_DATASHEET_TABLE4_HIGH_REPEATABILITY_MAX_MS);
```

Clean code, clear comments

Levels of documentation: **Inline** → function → module → architectural

```
// SHT31 Datasheet Table 4: Measurement duration (max)
const uint8_t SHT31_MEASUREMENT_DURATION_MS = 15;

delay(SHT31_MEASUREMENT_DURATION_MS);
```

Clean code, clear comments

Levels of documentation: Inline → **function** → module → architectural

Function documentation

Audience: Developer using the function – not reading the implementation

Purpose: The contract – what it *does*, what it *needs*, what it *returns*, what *can go wrong*

Clean code, clear comments

Levels of documentation: Inline → **function** → module → architectural

```
/**
 * @brief Validates chest compression depth for neonatal CPResuscitation training.
 *
 * @param depth_mm  Measured compression depth in millimetres
 * @return true if depth within protocol limits
 *
 * @pre Sensor must be calibrated (call calibrateSensor() first)
 * @note Thresholds per ERC protocol (2025), Section 4.2
 * @see Validation report HCL-VAL-2024-017
 */
bool isCompressionDepthValid(float depth_mm);
```

Clean code, clear comments

The key insight: you document the declaration, not the implementation.

This goes in the .h/hpp) file, not the .c/cpp file.

Rationale:

The user of your function:

- Sees the header
- Doesn’t (and shouldn't need to) read the implementation

Needs to know *what it does*, **not** *how it does it.*

Clean code, clear comments

The key insight: you document the declaration, not the implementation.

This goes in the .h/hpp) file, not the .c/cpp file.

Rationale:

The user of your function:

- Sees the header
- Doesn’t (and shouldn't need to) read the implementation

Needs to know *what it does*, **not** *how it does it.*

Clean code, clear comments

Levels of documentation: Inline → **function** → module → architectural

```
/**
 * This function checks the compression depth.
 */
bool isCompressionDepthValid(float depth_mm);
```

- What about this comment?

Clean code, clear comments

Levels of documentation: Inline → **function** → module → architectural

Tag	Purpose
@brief	One line – what does it do?
@param	What goes in? (units! constraints!)
@return	What comes out? (units! ranges!)
@pre	*What must be true before calling?*
@post	What will be true after calling?
@note	Important things to know
@warning	Things that can go wrong
@see	Related functions, documents, references

Clean code, clear comments

Levels of documentation: Inline → function → **module** → architectural

Module documentation

Audience: Developer working on, integrating, or maintaining this module

Purpose: What this module *is*, what it's responsible for, what it's *not* responsible for, and how it fits into the system

Exercise 2: Comment this

Where did 12.5 come from?

Where did 0.0625 come from?

Why 100?

Exercise 3: Good vs bad (5 min)

Which comment is useful? Why?

Clean code, clear comments - Modules

```
/**
 * @file compression_sensor.hpp
 * @brief Chest compression depth measurement for neonatal CPR trainer
 * @version 2.1.0
 * @date 2024-03
 * @author J. Korten
 *
 * This module handles all interaction with the VL6180X time-of-flight
 * sensor used to measure chest compression depth during training.
 *
 * Responsibilities:
 * - Sensor initialization and calibration
 * - Raw measurement acquisition
 * - Conversion to millimetres
 * - Validation against clinical thresholds
 *
 * NOT responsible for:
 * - Feedback to user (see feedback_controller.h)
 * - Data logging (see session_logger.h)
 * - Communication with host (see can_interface.h)
 *
 * Hardware: VL6180X on I2C1 (address 0x29)
 *
 * @see ERC Neonatal Resuscitation Protocol (2025)
 * @see Hardware schematic SCH-2024-017
 * @see Calibration procedure HCL-CAL-2024-003
 *
 * Version history:
 * - 2.1.0: Added ex-phantoma calibration support
 * - 2.0.0: Changed to VL6180X (was GP2Y0A21YK)
 * - 1.1.0: Added ERC protocol validation
 * - 1.0.0: Initial release
 */
```

Clean code, clear comments - Modules

Levels of documentation: Inline → function → **module** → architectural

Tag	Purpose
@file	File name
@brief	One line – what is this module?
@version	Version info
@author/date	Who to ask
@author/date	When it was written
@description	What problem does it solve?
Responsibilities	What it *does*
NOT responsible	What it *doesn't do* (boundaries)
Hardware	Physical dependencies
Dependencies	Dependencies / Other modules it relies on
@see	References: Datasheets, protocols, design docs
Version history	Who to ask, when it was written

Note: For medical devices / safety-critical systems, @version isn't just nice to have – it's often **required for traceability**.

Clean code, clear comments

Levels of documentation: Inline → function → module → **architectural**

Architectural documentation

Audience: New team member, future maintainer, external auditor, your future self (e.g. in 2 years)

Purpose: The big picture – why this system exists, how the parts relate, what decisions were made and why

Clean code, clear comments

Levels of documentation: Inline → function → module → **architectural**

Architectural documentation

Audience: New team member, future maintainer, external auditor, your future self (e.g. in 2 years)

Purpose: The big picture – why this system exists, how the parts relate, what decisions were made and why

Clean code, clear comments

Levels of documentation: Inline → function → module → **architectural**

This lives outside the code:

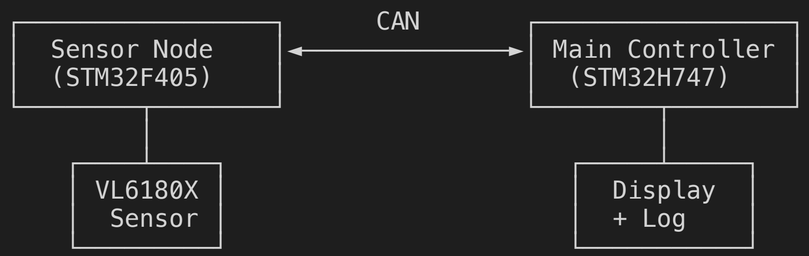
- README.md
- ARCHITECTURE.md
- Architecture Decision Records (ADRs)
- System diagrams (block diagrams, class diagrams, sequence diagrams, etc)
- Design rationale documents

Clean code, clear comments - Modules

```
# Neonatal CPR Training Manikin – Firmware

Embedded firmware for the HCL infant resuscitation trainer,
developed in collaboration with Wilhelmina Children's Hospital.

## What is this?

A training device that provides real-time feedback on chest
compression depth, rate, and hand positioning for neonatal CPR.

## Hardware

- MCU: STM32F103C8T6
- Compression sensor: VL6180X (I2C)
- Communication: MCP2515 CAN bus
- Feedback: WS2812B LED ring + piezo buzzer

## Building

    mkdir build && cd build
    cmake ..
    make

## Architecture

See [ARCHITECTURE.md](docs/ARCHITECTURE.md)

## Documentation

- [Hardware schematic](docs/SCH-2024-017.pdf)
- [CAN Protocol](docs/HCL-CAN-Protocol-v2.1.pdf)
- [Calibration procedure](docs/HCL-CAL-2024-003.pdf)
- [Validation report](docs/HCL-VAL-2024-017.pdf)
```

Clean code, clear comments - Modules

```
# System Architecture

## Overview
```




```
## Design Decisions

### Why CAN bus instead of I2C?

- Distance: sensor node is 40cm from controller
- Noise immunity: operating near motors
- Medical standard: common in medical devices

See [ADR-001](docs/adr/ADR-001-can-bus.md)

### Why VL6180X instead of ultrasonic?

- Precision: ±1mm vs ±3mm
- Speed: 10ms vs 50ms response time
- Size: fits inside manikin chest cavity

See [ADR-002](docs/adr/ADR-002-tof-sensor.md)
```

Clean code, clear comments - Modules

```
# ADR-001: Use CAN bus for sensor communication

## Status
Accepted

## Context
We need communication between the compression sensor node
and the main controller. Distance is ~40cm, environment
includes stepper motors for lung simulation.

## Options Considered
1. I2C – simple, but limited distance, noise sensitive
2. SPI – fast, but requires many wires, limited distance
3. CAN – robust, standard in medical, designed for noise
4. RS-485 – robust, but requires protocol design

## Decision
CAN bus using MCP2515 controller.

## Consequences
– Additional component cost (~€3 per node)
– Team needs CAN protocol knowledge
– Easy to add more sensor nodes later
– Compliant with medical device expectations

## References
– MCP2515 Datasheet
– ISO 11898 (CAN standard)
```

Clean code, clear comments

Levels of documentation: Inline → function → module → **architectural**

The test: *Can a new team member understand why the system is built this way – not just how it works?*

Can they understand why you chose *[CAN]* over *[I2C]* without having to ask you?

If *no* → your architectural documentation is *incomplete*.

Clean code, clear comments

Levels of documentation: Inline → function → module → architectural

Level	Where	Changes	Audience
Inline	In the code	Often	Developer reading this line
Function	Header files	Sometimes	Developer using this function
Module	Top of file	Rarely	Developer working on this component
Architectural	Separate docs	Very rarely	Everyone, including future team

Clean code, clear comments, clear documentation

Theo Theunissen's *"knowledge vaporization"* – when the original developers leave, the *why* disappears. The code shows *what*. Only documentation preserves *why*.

Exercise 4: Write the function header

Write the Doxygen documentation.

What questions would a user need answered?

Exercise 5: Bonus

Volunteer needed: Is anyone willing to show some code from Prog 1 - 4?

Final topic: Vibe-documenting

"Hey ChatGPT, add comments to this code"

"Hey Copilot, document this function"

```
/**
 * @brief Reads temperature from sensor
 * @param sensor Pointer to sensor instance
 * @return Temperature value in degrees Celsius
 */
float readTemperature(Sensor* sensor) {
    // Read the raw value from the sensor
    uint16_t raw = sensor->read();
    // Convert to temperature using formula
    return raw * 0.0625f - 40.0f;
}
```

Vibe-documenting

The problem:

- Where did 0.0625f come from? Where did 40.0f come from?

The AI doesn't know. It pattern-matched from training data. Maybe it's right. Maybe it's hallucinated.

You/we don’t know either – we might just accepted it.

This is vibe documenting:

- Comments that look correct
- But aren't grounded in actual knowledge
- Written by someone (AI) who never read the datasheet
- Accepted by someone (developer) who also never read the datasheet

Vibe-documenting

Knowledge vaporization at the point of creation

Theo Theunissen described knowledge evaporating over time.

- With AI-generated comments, there was *never* any knowledge to begin with.

The comment says *what*. But the *why* was never known by anyone:

- Not by the AI (it doesn't "know" anything)
- Not by the developer (they didn't research it)
- Not by the code reviewer (they see professional-looking docs and approve)

Vibe-documenting

Knowledge vaporization at the point of creation

Theo Theunissen described knowledge evaporating over time.
- With AI-generated comments, there was *never* any knowledge to begin with.

The comment says *what*. But the *why* was never known by anyone:
- Not by the AI (it doesn't "know" anything)
- Not by the developer (they didn't research it)
- Not by the code reviewer (they see professional-looking docs and approve)

The real danger:
- Six months later, someone asks: "Why 40.0f? Can we change it?"
- Nobody knows. The "documentation" is fiction.
- The original developer can't answer because they never knew.
- The knowledge didn't vaporize – it never condensed in the first place.

LLM's usefulness in Commenting and Documenting

When AI *can* help

AI is useful for:

- Boilerplate: *"Generate Doxygen template for this function"*
- Grammar/clarity: *"Improve the wording of this comment"*
- Format conversion: *"Convert these comments to Doxygen format"*

But you must supply:

- The *why*
- The datasheet references
- The design decisions
- The validation evidence

LLM's usefulness in Commenting and Documenting

The rule

AI can help you write comments.

AI cannot help you know what to write.

Final topic: Vibe-documenting

Is this documentation complete?

```
/**
 * @brief Waits for sensor to stabilise
 * @param ms Milliseconds to wait
 */
void stabilisationDelay(uint16_t ms) {
    delay(ms);
}
```

LLM/AI as documentation assistant

AI as documentation assistant, not documentation author

Bad prompt:

“Add comments to this code”

LLM/AI as documentation assistant

AI as documentation assistant, not documentation author

Bad prompt:

“Add comments to this code”

Better prompt:

“Here is the SHT31 datasheet. Document this code with references to the relevant datasheet sections.”

LLM/AI as documentation assistant

AI as documentation assistant, not documentation author

Bad prompt:

“Add comments to this code”

Better prompt:

“Here is the SHT31 datasheet. Document this code with references to the relevant datasheet sections.”

Best prompt:

“Here is the SHT31 datasheet. Here is our calibration report HCL-CAL-2024-003. Document this code explaining: Which datasheet tables the timing values come from, Why we chose high repeatability mode, How the calibration offset was determined”

LLM/AI documentation assistant workflow

The workflow

- You read the datasheet
- You make the design decisions
- You understand *why*

AI LLMs help you write it clearly, consistently, in proper Doxygen format

AI LLMs are writing tools, not a knowledge tools.

Take away message

Comments explain what the code cannot say.

AI only knows what the code already says.

That's why AI LLMs can't write the comments that matter.

AI + datasheet = useful documentation

AI + nothing = professional-looking fiction

“

That’s all…

Any questions?