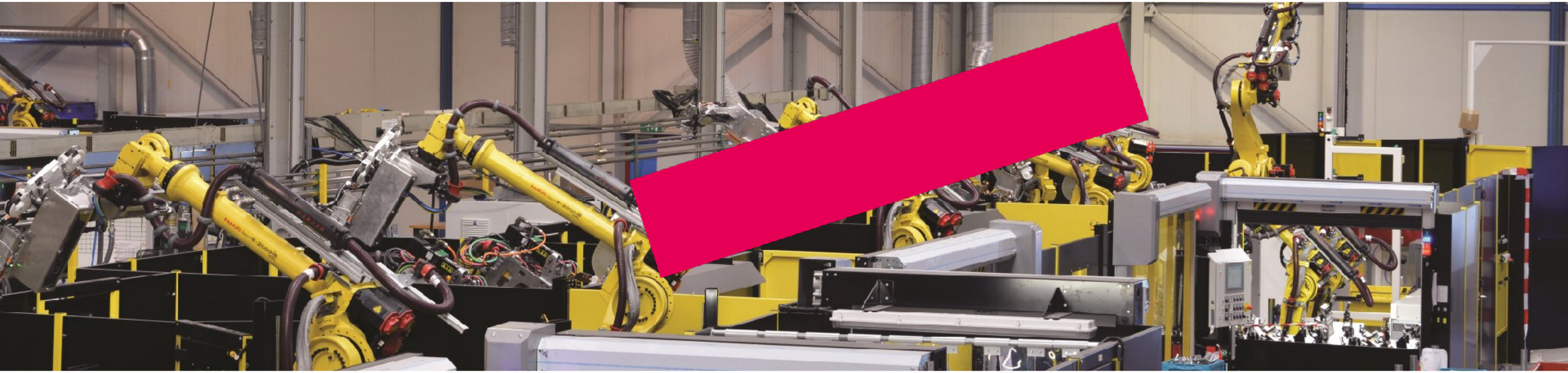
Programming 6



Workshop on Design Patterns and Clean Architecture for Embedded Systems

johan.korten@han.nl

V1.0 Dec 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

- Why Clean Architecture is vital

Overview of Today's Topics

- Refer to RGTData.h from the CleanCode workshop
- How would you test the fixed-point conversion *konverteerSpanning*?
- How would you test sending a command?

Overview of Today's Topics

- Refer to RGTData.cpp
- Why is there a doeZelftest test case in the code?

Pro's and Con's of the example code

Uses templates for data packets

Has self-test capability (doeZelftest())

Platform abstraction via #ifdef for different STM32 boards

Type aliases (using Spanning = float)

Pro's and **Con's** of the example code

Tight Coupling to Hardware

Low-level byte manipulation everywhere

Mixed responsibilities

Not testable

App-titude Test

"Getting an app to work is what I call the *App-titude test* for a programmer. Programmers, embedded or not, who just concern themselves with getting their app to work are doing their products and employers a disservice.”

Bob Martin - Clean Architecture
(Chapter 29: Clean Embedded Architecture, chapter by James Grenning)

Kent Beck

Kent Beck's three activities:

1. "First make it work." — You are out of business if it doesn't work.
2. "Then make it right." — Refactor so you and others can understand it.
3. "Then make it fast." — Refactor for "needed" performance.

The Target-Hardware Bottleneck

When embedded code is structured without applying clean architecture principles and practices, you will often face the scenario in which you can test your code only on the target.

If the target is the only place where testing is possible, the target-hardware bottleneck will slow you down.

Design Patterns

A design pattern is like a software recipe: you often face problems that can be solved in a certain way, you can of course code the solution entirely by yourself, but you could also try to apply a design pattern.

Design Patterns

A design pattern is like a software recipe: you often face problems that can be solved in a certain way, you can of course code the solution entirely by yourself, but you could also try to apply a design pattern.

Fair ***Warning***: The recipe metaphor has a useful extension: a recipe assumes you can already chop an onion.

Design Patterns - GoF

Gang of Four (GoF): “Design Patterns: Elements of Reusable Object-Oriented Software" in 1994 — Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

The book catalogued 23 patterns they'd observed recurring across well-designed object-oriented systems.

Design Patterns: Types

Creational

Structural

Behavioral

Design Patterns: Types (GoF)

Creational

- Abstract Factory, Builder, Factory Method, Prototype, Singleton

Structural

Behavioral

Design Patterns: Types (GoF)

Creational

Structural

- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

Behavioral

Design Patterns: Types (GoF)

Creational

Structural

Behavioral

- Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Design Patterns: Types (GoF)

The GoF book set that the following template:

Each pattern entry includes:

Intent — the one-sentence goal

Motivation — a concrete scenario showing the problem

Applicability — when to use it (and implicitly, when not to)

Consequences — the trade-offs, both benefits and liabilities

Related Patterns — what's similar, what competes, what combines well

Humble Object Pattern

Intent

Separate hard-to-test code from easy-to-test logic by pushing all difficult-to-test concerns (hardware, UI, external systems) into objects so simple they hardly need testing.

Motivation

A motor controller needs complex ramp-up logic, acceleration curves, and stall detection. Testing this requires the physical motor, power supply, and oscilloscope to verify behavior — slow, non-deterministic, and impossible in CI. But the logic itself is just math and state transitions. By extracting all register writes into a trivial driver class, the interesting logic becomes testable on any machine in milliseconds.

Humble Object Pattern

Applicability

Use Humble Object when:

Code directly manipulates hardware registers or peripherals

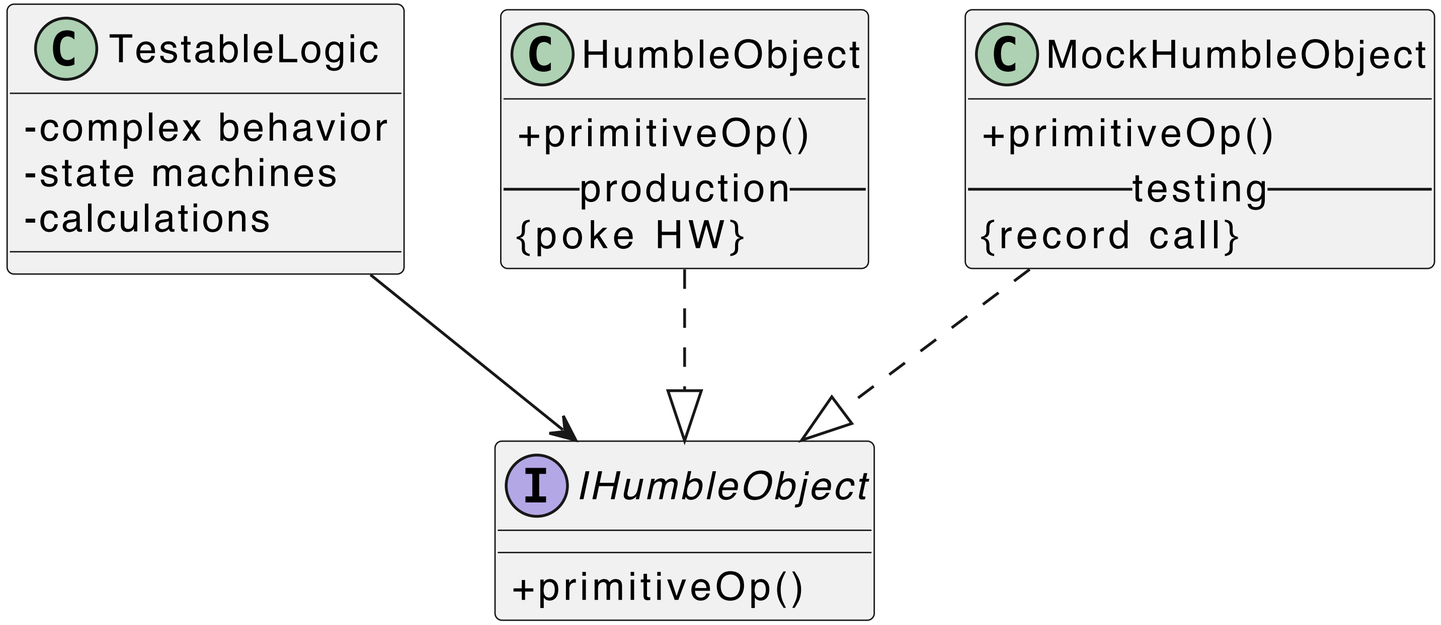
Code depends on UI frameworks with render loops or event systems

Code interacts with databases, file systems, or network

Timing constraints or asynchronous behavior make tests non-deterministic

You need to test on a different platform than the deployment target

Humble Object Pattern



Humble Object Pattern: Pro's

Benefits:

Logic testable without target hardware or external systems

Tests execute at CPU speed, enabling fast feedback cycles

Tests become deterministic — no flaky failures from hardware variance

Clear architectural boundary between policy and mechanism

Logic layer becomes portable across platforms

Humble Object Pattern: Con's

The humble object itself remains untested or undertested — bugs in register configuration, timing, or hardware interaction only surface during integration

Proliferation of interfaces, one per hardware boundary

Abstraction design requires anticipating what the logic layer needs; mistakes cause leaky abstractions

High unit test coverage can create false confidence — real failures often occur at the boundary you deliberately stopped testing

Overhead may not justify itself for genuinely simple systems

Humble Object Pattern: Implementation Notes

Make the humble object *genuinely humble* — if you're tempted to test it, you haven't pushed enough logic out

The interface should express intent, not hardware mechanism (setMotorSpeed(rpm) not writeRegister(0x4F, value))

Consider whether the humble object needs any configuration or initialization logic — this often belongs in the testable layer

Humble Object Pattern: Related Patterns

Dependency Injection: the mechanism for supplying humble objects to the logic layer

Adapter Pattern: useful when the humble object must translate between your interface and an existing API

Gateway Pattern: similar intent applied to databases and external services (Martin Fowler)

Strategy Pattern: the testable logic may itself use strategies; don't confuse the two separations

Facade Pattern: a humble object sometimes simplifies a complex hardware subsystem

Refactoring CSKompaktData using Humble Object Pattern

Original:

```
// CSKompaktData – conversion logic is PRIVATE and STATIC
class CSKompaktData {
    public:
        explicit CSKompaktData(const SampleMoment, const Spanning mv,
                               const Spanning rv, const Spanning cv);
        Spanning geefMeting() const;
        FoutCode doeZelftest();  // <- Author knew testing was needed!

    private:
        // Can't test these – they're private!
        static UInt16 konverteerSpanning(const Spanning &);
        static Spanning konverteerFixedPoint(const UInt16 &);

        static constexpr UInt16 fraktieBereik = 0xfff;  // Q4.12 format
        // ...
};
```

Refactoring CSKompaktData using Humble Object Pattern

1. Extract the converter (new file: FixedPointQ412.hpp)

```cpp
#ifndef FIXED_POINT_Q412_HPP
#define FIXED_POINT_Q412_HPP

#include <cstdint>

namespace fixedpoint {


class FixedPointQ412 {
public:
    using FloatType = float;
    using FixedType = uint16_t;

    static constexpr uint16_t FRACTION_BITS = 12U;
    static constexpr uint16_t FRACTION_MASK = 0x0FFFU;
    static constexpr uint8_t  INTEGER_MASK  = 0x0FU;

    /**
     * @brief Convert floating-point to Q4.12 fixed-point
     * @param value Floating-point value in range [0.0, 15.999...]
     * @return Q4.12 fixed-point representation
     */
    static FixedType toFixed(FloatType value) {
        const auto integer = static_cast<uint8_t>(value) & INTEGER_MASK;
        const auto fraction = static_cast<uint16_t>((value - integer) * FRACTION_MASK);
        return static_cast<FixedType>((integer << FRACTION_BITS) + fraction);
    }

    /**
     * @brief Convert Q4.12 fixed-point to floating-point
     * @param fixed Q4.12 fixed-point value
     * @return Floating-point representation
     */
    static FloatType toFloat(FixedType fixed) {
        const uint16_t integer = fixed >> FRACTION_BITS;
        const uint16_t fraction = fixed & FRACTION_MASK;
        return static_cast<FloatType>(integer) +
               static_cast<FloatType>(fraction) / FRACTION_MASK;
    }

    /**
     * @brief Maximum quantization error for this format
     * @return Approximately 1/4095 = 0.000244
     */
    static constexpr FloatType maxError() {
        return 1.0F / FRACTION_MASK;
    }
};
```

Refactoring CSKompaktData using Humble Object Pattern

1. Extract the converter (new file: FixedPointQ412.hpp)

```cpp
#ifndef FIXED_POINT_Q412_HPP
#define FIXED_POINT_Q412_HPP

#include <cstdint>

namespace fixedpoint {


class FixedPointQ412 {
public:
    using FloatType = float;
    using FixedType = uint16_t;

    static constexpr uint16_t FRACTION_BITS = 12U;
    static constexpr uint16_t FRACTION_MASK = 0x0FFFU;
    static constexpr uint8_t  INTEGER_MASK  = 0x0FU;

    /**
     * @brief Convert floating-point to Q4.12 fixed-point
     * @param value Floating-point value in range [0.0, 15.999...]
     * @return Q4.12 fixed-point representation
     */
    static FixedType toFixed(FloatType value) {
        const auto integer = static_cast<uint8_t>(value) & INTEGER_MASK;
        const auto fraction = static_cast<uint16_t>((value - integer) * FRACTION_MASK);
        return static_cast<FixedType>((integer << FRACTION_BITS) + fraction);
    }

    /**
     * @brief Convert Q4.12 fixed-point to floating-point
     * @param fixed Q4.12 fixed-point value
     * @return Floating-point representation
     */
    static FloatType toFloat(FixedType fixed) {
        const uint16_t integer = fixed >> FRACTION_BITS;
        const uint16_t fraction = fixed & FRACTION_MASK;
        return static_cast<FloatType>(integer) +
               static_cast<FloatType>(fraction) / FRACTION_MASK;
    }

    /**
     * @brief Maximum quantization error for this format
     * @return Approximately 1/4095 = 0.000244
     */
    static constexpr FloatType maxError() {
        return 1.0F / FRACTION_MASK;
    }
};
```

Refactoring CSKompaktData using Humble Object Pattern

3. Unit tests

```cpp
#include "CppUTest/TestHarness.h"
#include "FixedPointQ412.hpp"

using namespace fixedpoint;

// ============================================================================
// Round-trip Conversion Tests
// ============================================================================

TEST_GROUP(RoundTrip) {
};

TEST(RoundTrip, ValuePreservedWithinQuantizationError) {
    constexpr float INPUT = 3.45678F;

    auto fixed = FixedPointQ412::toFixed(INPUT);
    auto result = FixedPointQ412::toFloat(fixed);

    DOUBLES_EQUAL(INPUT, result, FixedPointQ412::maxError());
}

TEST(RoundTrip, ZeroConvertsExactly) {
    constexpr float INPUT = 0.0F;

    auto fixed = FixedPointQ412::toFixed(INPUT);
    auto result = FixedPointQ412::toFloat(fixed);

    DOUBLES_EQUAL(INPUT, result, 0.0001);
}

TEST(RoundTrip, IntegerValueConvertsExactly) {
    constexpr float INPUT = 5.0F;

    auto fixed = FixedPointQ412::toFixed(INPUT);
    auto result = FixedPointQ412::toFloat(fixed);

    DOUBLES_EQUAL(INPUT, result, 0.0001);
}
```

Refactoring CSKompaktData using Humble Object Pattern

3. Unit tests

```
// ============================================================================
// toFixed() Tests
// ============================================================================

TEST_GROUP(ToFixed) {
};

TEST(ToFixed, ZeroProducesZeroFixed) {
    LONGS_EQUAL(0x0000, FixedPointQ412::toFixed(0.0F));
}

TEST(ToFixed, MaxIntegerProducesCorrectFixed) {
    // Q4.12: integer 15 = 0xF, shifted left 12 bits = 0xF000
    LONGS_EQUAL(0xF000, FixedPointQ412::toFixed(15.0F));
}

TEST(ToFixed, OneProducesCorrectFixed) {
    // Q4.12: integer 1 = 0x1, shifted left 12 bits = 0x1000
    LONGS_EQUAL(0x1000, FixedPointQ412::toFixed(1.0F));
}

// TODO: Students implement
// TEST(ToFixed, HalfProducesCorrectFixed)
// Hint: 0.5 in Q4.12 should be approximately 0x0800 (2048/4095)

// TODO: Students implement
// TEST(ToFixed, QuarterProducesCorrectFixed)
// Hint: 0.25 in Q4.12 should be approximately 0x0400 (1024/4095)

// ============================================================================
// toFloat() Tests
// ============================================================================
```

Refactoring CSKompaktData using Humble Object Pattern

3. Unit tests

```
TEST_GROUP(ToFloat) {
};

TEST(ToFloat, ZeroFixedProducesZero) {
    DOUBLES_EQUAL(0.0F, FixedPointQ412::toFloat(0x0000), 0.0001);
}

TEST(ToFloat, MaxIntegerFixedProducesMaxInteger) {
    DOUBLES_EQUAL(15.0F, FixedPointQ412::toFloat(0xF000), 0.0001);
}

TEST(ToFloat, IntegerPlusFractionConvertsCorrectly) {
    // 5.25 -> integer=5 (0x5000), fraction=0.25 (~0x0400)
    // Expected fixed: 0x5400 (approx)
    constexpr float EXPECTED = 5.25F;
    constexpr uint16_t FIXED_5_25 = 0x53FF;  // 5 + 1023/4095

    auto result = FixedPointQ412::toFloat(FIXED_5_25);

    DOUBLES_EQUAL(EXPECTED, result, FixedPointQ412::maxError());
}

// TODO: Students implement
// TEST(ToFloat, MidpointValueConvertsCorrectly)
// Hint: 0x8000 should produce approximately 8.0
```

Refactoring CSKompaktData using Humble Object Pattern

3. Unit tests

```
// ============================================================================
// Edge Cases
// ============================================================================

TEST_GROUP(EdgeCases) {
};

TEST(EdgeCases, MaxValueConvertsCorrectly) {
    // 0xFFFF = integer 15, fraction 4095/4095 = 15.999...
    constexpr uint16_t MAX_FIXED = 0xFFFF;
    constexpr float EXPECTED_MAX = 15.0F + (4095.0F / 4095.0F);

    DOUBLES_EQUAL(EXPECTED_MAX, FixedPointQ412::toFloat(MAX_FIXED), 0.001);
}

TEST(EdgeCases, VoltageTypicalAdcRange) {
    // Typical STM32 ADC: 0-3.3V mapped to 12-bit (0-4095)
    // If we store 3.3V in Q4.12 format:
    constexpr float VOLTAGE = 3.3F;

    auto fixed = FixedPointQ412::toFixed(VOLTAGE);
    auto result = FixedPointQ412::toFloat(fixed);

    DOUBLES_EQUAL(VOLTAGE, result, FixedPointQ412::maxError());
}

// TODO: Students implement
// TEST(EdgeCases, NegativeValueBehavior)
// Question: What happens with negative input? Is this a valid use case?
// Hint: The current implementation uses unsigned types...
```

Hardware Proxy Pattern (aka HAL)

Problem

Code that directly accesses hardware registers cannot be tested off-target.

Example: GPIO_PORTA->ODR |= (1 << 5); // Can't run this on your laptop!

Solution

Create an abstract interface for hardware access. Provide two implementations

1. Real implementation (talks to actual hardware)
2. Mock/Fake implementation (for testing)

Clean Architecture (Ch.29):

"The hardware is a detail. A clean embedded architecture's software is testable OFF the target hardware.”

Douglass (Ch.3):

"The Hardware Proxy pattern encapsulates the interface to hardware devices behind a software interface."

Architecture - abstraction layers

Layer	Purpose
HAL (Hardware Abstraction Layer)	Hides GPIO, registers, peripherals
PAL (Processor Abstraction Layer)	Hides processor-specific C extensions
OSAL (OS Abstraction Layer)	Hides RTOS specifics

Cyclic Executive Pattern

Intent

Execute a fixed sequence of tasks repeatedly in a deterministic main loop, dividing time into predictable slots.

Motivation

An embedded system must read sensors every 10 ms, update a display every 100 ms, and check communication every 50 ms.

Rather than using an RTOS with its overhead and complexity, a cyclic executive divides time into a major frame (e.g., 100 ms) containing minor frames (e.g., 10 ms each).

Each minor frame runs a predetermined subset of tasks, guaranteeing all deadlines are met through static scheduling.

Cyclic Executive Pattern

Applicability

Use when timing requirements are well-understood at design time, tasks have harmonic periods (multiples of each other), worst-case execution times are known, and determinism is more important than flexibility.

Avoid when task sets change dynamically or when periods are not harmonic.

Consequences

Fully deterministic with zero scheduling overhead at runtime, easy to analyse for timing correctness, and requires no RTOS (reducing memory footprint and complexity).

However, the schedule is brittle, adding or changing tasks requires redesigning the entire frame structure.

Long-running tasks must be manually split across frames.

Unused time in slots is wasted.

Cyclic Executive Pattern

Related Patterns

Competes with RTOS-based preemptive scheduling when flexibility matters. Often combined with Super Loop for the outer structure.

State Machine pattern helps decompose long tasks into frame-sized chunks.

Rate Monotonic Scheduling provides the theoretical basis for assigning slots when moving to a preemptive model.

Debouncing Patterns

Delay-Based Debouncing

Intent

Eliminate switch bounce by waiting a fixed time after the first edge before accepting a new state.

Motivation

A button press triggers multiple interrupts within milliseconds.

By ignoring all input for a set period after the first edge, subsequent bounces are filtered out.

Applicability

Use when simplicity and predictable timing matter more than response latency.

Requires a timer or counter.

Consequences

Simple to understand and implement with predictable behaviour, but adds fixed latency (10–50 ms) to every button response.

Related Patterns

Can be combined with Integrator for noisy environments.

Hardware RC filter achieves similar results passively.

Debouncing Patterns

Shift Register Debouncing

Intent

Track recent input history as a bit pattern and accept state change only when all samples agree.

Motivation

Polling a button every 5 ms and storing results in an 8-bit register means a stable press shows 0xFF and a stable release shows 0x00.

Intermediate values indicate ongoing bounce.

Applicability

Use when memory and CPU cycles are constrained.

Requires consistent polling rate.

Configurable depth via data type (uint8_t for 8 samples, uint16_t for 16).

Consequences

Very efficient (bit shifts only) with minimal memory (one byte), but behaviour depends on polling consistency.

Related Patterns

Conceptually similar to Integrator but uses binary history rather than a count.

Competes with Delay-Based for simple applications.

Debouncing Patterns

Integrator Debouncing

Intent

Increment or decrement a counter based on input state and change output only when the counter saturates.

Motivation

A noisy mechanical switch produces erratic readings.

The integrator absorbs this noise by requiring sustained input before triggering, with separate saturation thresholds possible for press and release.

Applicability

Use when signal quality is poor or when hysteresis (different press/release thresholds) is needed.

Accepts inconsistent polling.

Consequences

Handles noisy signals gracefully with smooth behaviour, but requires more complex logic and additional memory for the counter.

Related Patterns

Combines well with hardware Schmitt trigger for extra noise immunity.

More robust alternative to Shift Register in harsh environments.

State Pattern

Intent

Allow an object to alter its behavior when its internal state changes, appearing to change its class.

Motivation

A traffic light controller must behave differently depending on whether it is in the Red, Green, or Yellow state.

Using conditionals (switch/if-else) throughout the code leads to scattered logic that is difficult to maintain and extend.

The State pattern encapsulates each state as a separate object (or function pointer in C), with each state defining its own behavior for events like timer expiry or pedestrian button press.

The controller delegates to the current state object, which handles the event and determines the next state.

State Pattern

Applicability

Use when an object's behavior depends on its state and it must change behavior at runtime based on that state.

Particularly valuable when state-specific logic is duplicated across multiple conditionals, when the number of states may grow, or when state transitions follow complex rules.

Avoid for trivial state machines with only two or three states and simple transitions.

Consequences

Localizes state-specific behavior into separate classes or functions, making each state explicit and self-contained.

Adding new states requires no changes to existing state code (Open/Closed Principle).

State transitions become explicit and traceable.

However, increases the number of classes or function pointers, which can feel heavyweight for simple machines.

State objects may need access to the context's internals, potentially breaking encapsulation.

State Pattern

Related Patterns

Often used with State Machine pattern as the underlying model.

Strategy pattern has identical structure but different intent (interchangeable algorithms vs. state-dependent behavior).

Flyweight can reduce memory when many contexts share the same state instances.
Singleton often applies to state objects when they carry no instance-specific data.

Observer Pattern

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Motivation

A temperature sensor in a medical device must inform multiple subsystems when a reading changes: the display needs to update, the alarm module must check thresholds, and the data logger must record the value.

Hardcoding these dependencies creates tight coupling—the sensor module would need to know about every consumer.

The Observer pattern inverts this: interested parties register themselves with the sensor (the subject), which maintains a list of observers and notifies them all when data changes, without knowing what they do with the information.

Observer Pattern

Applicability

Use when a change in one object requires changing others and you do not know how many objects need to change.

Valuable when an abstraction has two aspects where one depends on the other, and you want to vary or reuse them independently.

Common in event-driven architectures, UI frameworks, and publish-subscribe systems.

Avoid when the update order matters and cannot be controlled, or when notification cascades could cause performance issues.

Consequences

Achieves loose coupling between subject and observers—the subject only knows the observer interface, not concrete implementations.

Supports broadcast communication where any number of observers can subscribe.

Enables dynamic relationships at runtime.

However, observers are notified in undefined order, which can cause subtle bugs if dependencies exist between them.

Cascading updates can be difficult to trace and debug.

Memory management requires care: observers must unsubscribe before destruction to avoid dangling pointers.

Observer Pattern

Related Patterns

Mediator can centralize complex update logic when observers have interdependencies.
Singleton often applies to subjects that represent global resources like hardware peripherals.

Command pattern can encapsulate notifications for queuing or logging.

Publish-Subscribe extends Observer with message channels for greater decoupling.

State Pattern

Intent

Allow an object to alter its behavior when its internal state changes, appearing to change its class.

Motivation

A traffic light controller must behave differently depending on whether it is in the Red, Green, or Yellow state.

Using conditionals (switch/if-else) throughout the code leads to scattered logic that is difficult to maintain and extend.

The State pattern encapsulates each state as a separate object (or function pointer in C), with each state defining its own behavior for events like timer expiry or pedestrian button press.

The controller delegates to the current state object, which handles the event and determines the next state.

State Pattern

Applicability

Use when an object's behavior depends on its state and it must change behavior at runtime based on that state.

Particularly valuable when state-specific logic is duplicated across multiple conditionals, when the number of states may grow, or when state transitions follow complex rules. Avoid for trivial state machines with only two or three states and simple transitions.

Consequences

Localizes state-specific behavior into separate classes or functions, making each state explicit and self-contained.

Adding new states requires no changes to existing state code (Open/Closed Principle).

State transitions become explicit and traceable.

However, increases the number of classes or function pointers, which can feel heavyweight for simple machines.

State objects may need access to the context's internals, potentially breaking encapsulation.

State Pattern

Related Patterns

Often used with *State Machine* pattern as the underlying model.

Strategy pattern has identical structure but different intent (interchangeable algorithms vs. state-dependent behavior).

Flyweight pattern can reduce memory when many contexts share the same state instances.

Singleton pattern often applies to state objects when they carry no instance-specific data.

Different State related strategies

Aspect	Switch/Case	State Pattern	State Table	Hierarchical State Machine
Complexity	Low	Medium	Medium	High
States < 5	Ideal	Overkill	Overkill	Overkill
States 5–15	Manageable	Good fit	Good fit	If duplication
States > 15	Unmaintainable	Verbose	Compact	Best
Complex per-state logic	Messy	Excellent	Poor	Excellent
Uniform transitions	Fine	Verbose	Excellent	Good
Duplicated transitions	Copy/paste	Copy/paste	Copy/paste	Inheritance
Runtime configurable	No	Difficult	Easy	Possible
Testability	Whole machine	Per state	Whole machine	Per level
Memory (RAM)	Minimal	Pointers	Table size	Hierarchy + stack
Typical use	Trivial FSMs	Business logic	Protocols, parsers	Modes, UI, control

Singleton Pattern

Intent

Ensure a class has exactly one instance and provide a global point of access to it.

Singleton Pattern

Example

Maybe we should prevent the following?!

```
// In sensor_task.cpp
UartDriver uart;  // Configures baud rate to 115200
uart.init();

// In debug_task.cpp
UartDriver uart;  // Reconfigures to 9600, corrupting sensor comms
uart.init();
```

Singleton Pattern

Example

We could solve it using a UartDriver singleton:

```
// Both tasks now share the same, consistently configured driver
UartDriver::instance().send(data);
```

Singleton Pattern

Applicability

Use Singleton when:

Hardware abstraction — Peripherals (UART, SPI, I2C/I3C, ADC) exist exactly once per channel. The singleton models physical reality.

System-wide services — A central logger, error handler, or watchdog manager that everything needs to access.

Configuration stores — A single source of truth for calibration data or runtime settings.

Resource managers — Memory pools, task schedulers, or mutex managers where centralized control prevents fragmentation or deadlock.

Singleton Pattern

Avoid

Avoid Singleton when:

- You’re using it merely to avoid passing parameters — that's global state in disguise
- Multiple instances might legitimately exist (e.g., multiple SPI buses → not a singleton, parameterize by bus ID instead)
- Testing requires substituting implementations — singletons make mocking painful
- The "single instance" requirement is accidental, not fundamental to the domain
- Also: do not overuse (make every class a singleton…)

Singleton Pattern

Benefits	Liabilities
Controlled access — The class itself enforces the single-instance constraint, not programmer discipline	Hidden dependencies — Code that calls Foo::instance() has an invisible coupling; you can't see it in the function signature
Lazy initialization — Instance created on first use, avoiding startup-order issues	Testing friction — Hard to substitute mocks; state persists between tests unless you add explicit reset() (which undermines the pattern's guarantee)
No heap required — Meyer's singleton lives in static storage, suitable for embedded	Concurrency hazards — If instance() is called from ISR before main() initializes it, you get undefined behavior
Namespace tidiness — Avoids polluting global scope with extern declarations	Destruction order — Static destruction happens in reverse construction order across translation units; accessing a destroyed singleton is undefined behavior (UB)
Single point of configuration — One place to set up the resource correctly	Lifetime ambiguity — "Lives forever" is fine for embedded, but problematic in systems with dynamic module loading

Singleton Pattern

Pattern	Relationship
Monostate	Alternative approach: all instances share the same static state. Looks like a normal class from outside, but every object sees identical data. Easier to mock (you can subclass), but less explicit about intent.
Factory Method	Can be combined — `instance()` is essentially a factory that always returns the same object. Useful when the singleton's concrete type varies by platform.
Façade	Singletons often serve as façades, providing a simplified interface to a complex subsystem (e.g., `HardwareAbstractionLayer::instance()` hiding register-level details).
Service Locator	A generalized singleton that returns different services by key. More flexible, more indirection, more complexity. Sometimes appropriate when you need runtime-switchable implementations.
Dependency Injection	The philosophical opposite — instead of code reaching out to get dependencies via `instance()`, dependencies are passed in explicitly. Better for testability, but requires infrastructure that may be too heavy for deeply embedded systems.

Singleton Pattern

Final remark

The singleton's bad reputation comes from overuse as a "global variable with extra steps." The pattern is legitimate when the domain genuinely has a single instance: hardware peripherals being the canonical embedded example.

The question to ask: "Is there exactly one of these in reality, or am I just being lazy about parameter passing?" If the latter, reconsider.

Singleton Pattern

```cpp
class Logger {
public:
    static Logger& instance() {
        static Logger instance;  // Created on first call, lives until program ends
        return instance;
    }

    void log(const char* msg) { /* ... */ }

    // Delete copy/move to prevent accidents
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

private:
    Logger() = default;  // Private constructor
};

// Usage
Logger::instance().log("Boot complete");
```

See also: [Implementing a Singleton in C++]

Dependency Injection for UART - Interface

```cpp
// Abstract interface – defines what we need, not how it's implemented
class IUart {
public:
    virtual ~IUart() = default;
    virtual void send(uint8_t byte) = 0;
    virtual bool receive(uint8_t& byte) = 0;
};
```

Dependency Injection for UART - Implementation of IUart

```
// Concrete implementation for real hardware
class Uart0 : public IUart {
public:
    explicit Uart0(uint32_t baud) {
        // Configure actual UART0 peripheral registers
    }

    void send(uint8_t byte) override {
        while (!(UART0->SR & TX_READY)) {}
        UART0->DR = byte;
    }

    bool receive(uint8_t& byte) override {
        if (UART0->SR & RX_READY) {
            byte = UART0->DR;
            return true;
        }
        return false;
    }
};
```

Dependency Injection for UART

```
// Consumer receives its dependency — doesn't fetch it
class SensorTask {
public:
    explicit SensorTask(IUart& uart) : uart_{uart} {}

    void run() {
        uart_.send(reading_);  // Uses whatever was injected
    }

private:
    IUart& uart_;
    uint8_t reading_{};
};
```

Dependency Injection for UART - main

```
int main() {
    // Create the single hardware instance – but as a local, not a singleton
    Uart0 uart{115200};

    // Inject into components that need it
    SensorTask sensors{uart};
    DebugConsole console{uart};

    while (true) {
        sensors.run();
        console.run();
    }
}
```

Dependency Injection for UART - Mock testing

```cpp
// Mock for unit tests
class MockUart : public IUart {
public:
    void send(uint8_t byte) override {
        sent_bytes.push_back(byte);
    }

    bool receive(uint8_t& byte) override {
        if (rx_queue.empty()) return false;
        byte = rx_queue.front();
        rx_queue.pop();
        return true;
    }

    std::vector<uint8_t> sent_bytes;
    std::queue<uint8_t> rx_queue;
};
```

```cpp
// Test without touching real hardware
TEST(SensorTask, SendsReadingOverUart) {
    MockUart mock;
    SensorTask task{mock};

    task.run();

    EXPECT_EQ(mock.sent_bytes.size(), 1);
}
```

Singleton Pattern vs Dependency Injection

Aspect	Singleton	Dependency Injection
Dependencies	Hidden inside implementation	Explicit in constructor
Testability	Requires real instance or invasive hooks	Inject mocks directly
Flexibility	One implementation, forever	Swap implementations at composition time
Boilerplate	Minimal	Requires interface + wiring
Runtime cost	None (direct call)	Virtual dispatch (one indirection)
Code navigation	"Where does this come from?"	Follow constructor parameter

Dependency Injection code on an embedded system

On Cortex-M, a virtual call costs roughly 3–5 extra cycles
(load vtable pointer, load function pointer, branch).

For a UART that's already waiting on hardware, this is noise.

But if you're in a tight ISR, consider using e.g. a template version.

Note: vtable-visualization.html shows the overhead concerns

Template-based injection (Singleton + DI alternative)

```cpp
// Template-based injection — zero overhead, no virtuals
template<typename Uart>
class SensorTask {
public:
    explicit SensorTask(Uart& uart) : uart_{uart} {}
    void run() { uart_.send(reading_); }
private:
    Uart& uart_;
};

// Usage — type is resolved at compile time
Uart0 uart{115200};
SensorTask<Uart0> sensors{uart};
```

Singleton Pattern vs Dependency Injection

Situation	Recommendation
Small firmware, single developer, no tests	Singleton is pragmatic
Team project, needs unit tests	Dependency Injection pays off quickly
Multiple UART channels	DI naturally — `Uart0`, `Uart1` are separate instances
Performance-critical ISR code	Template-based DI (no vtable)
Teaching software design principles	DI demonstrates decoupling and testability

Embedded Software Engineering

Why we call it *Engineering*

Virtually everything we do is a trade-off.

Even when we use patterns there is no one size fits all!

“

That’s all…

Any questions?