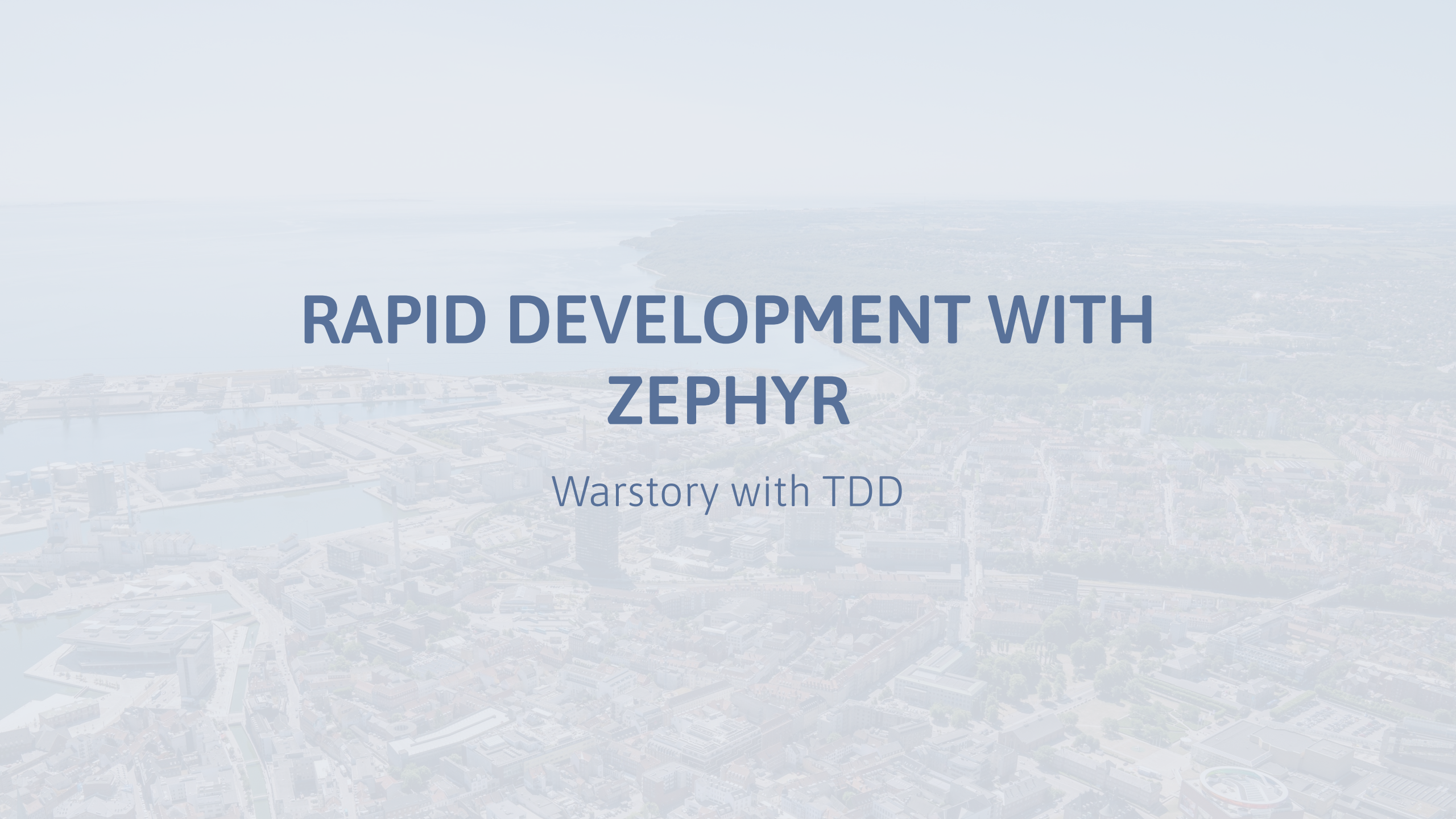


RAPID DEVELOPMENT WITH ZEPHYR

Warstory with TDD

TABLE OF CONTENTS

INTRO

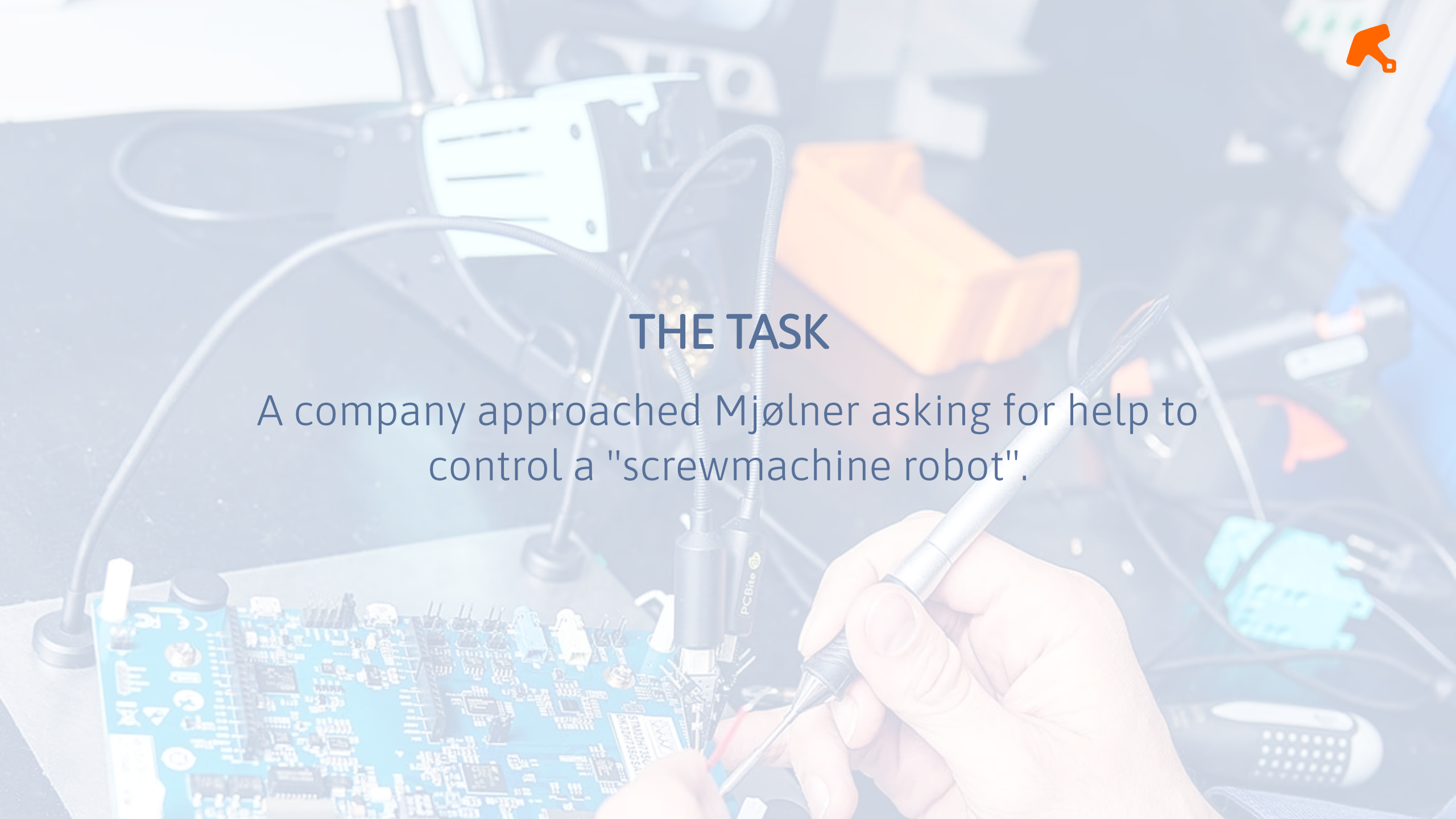
A real life example of rapid product development using Zephyr, with TDD as the driving force with unclear and changing requirements.

WHO AM I

Solution Architect in Embedded @ Mjølner.

Computer Science background: *Top down approach to Zephyr*

- [X] Metal
- [X] Abstractions & type safety -> C++
- [X] Fast feedback -> TDD & CI/CD
- [ ] Boolean parameters

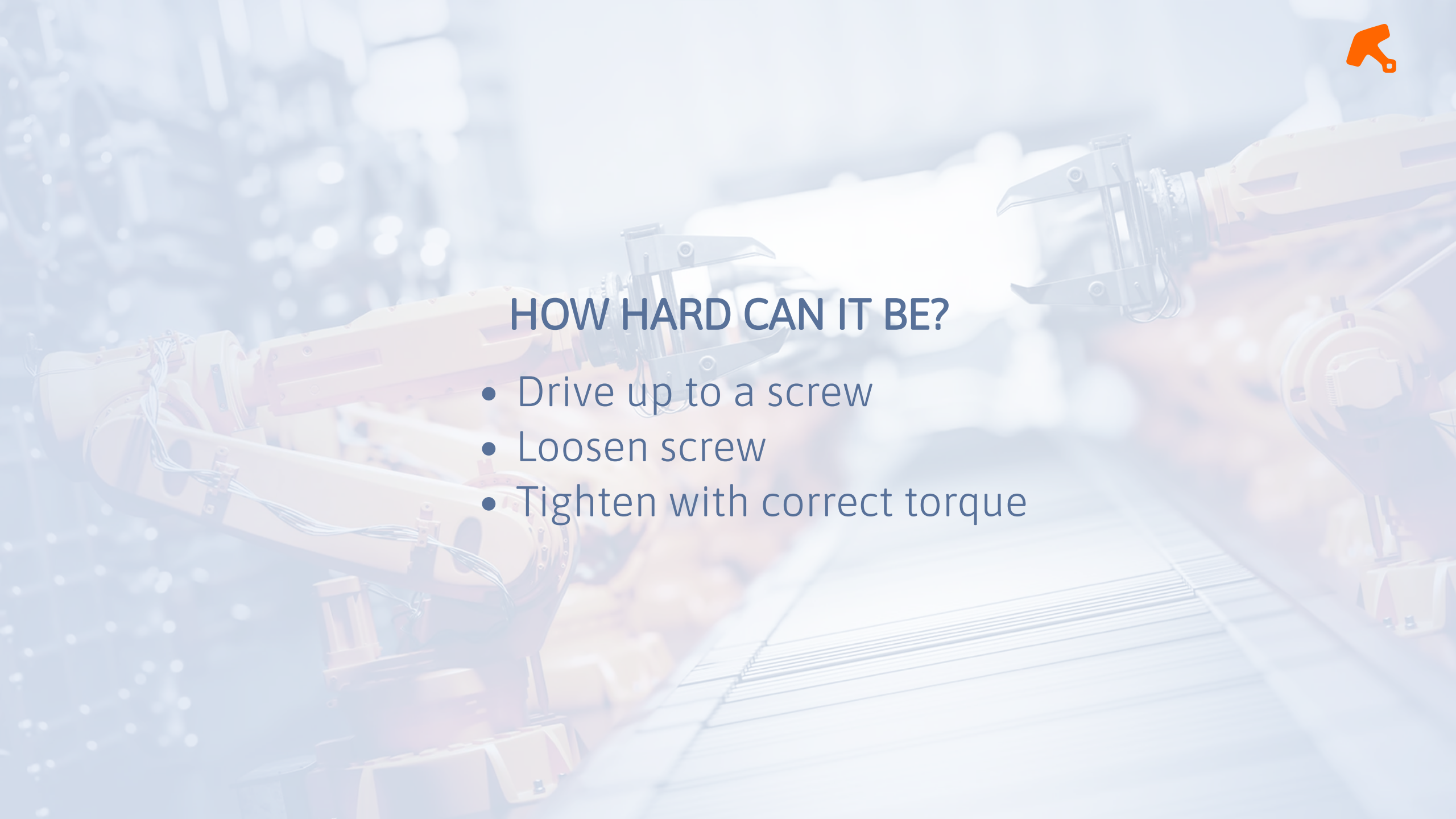


THE TASK

A company approached Mjølner asking for help to control a "screwmachine robot".

BY THE WAY

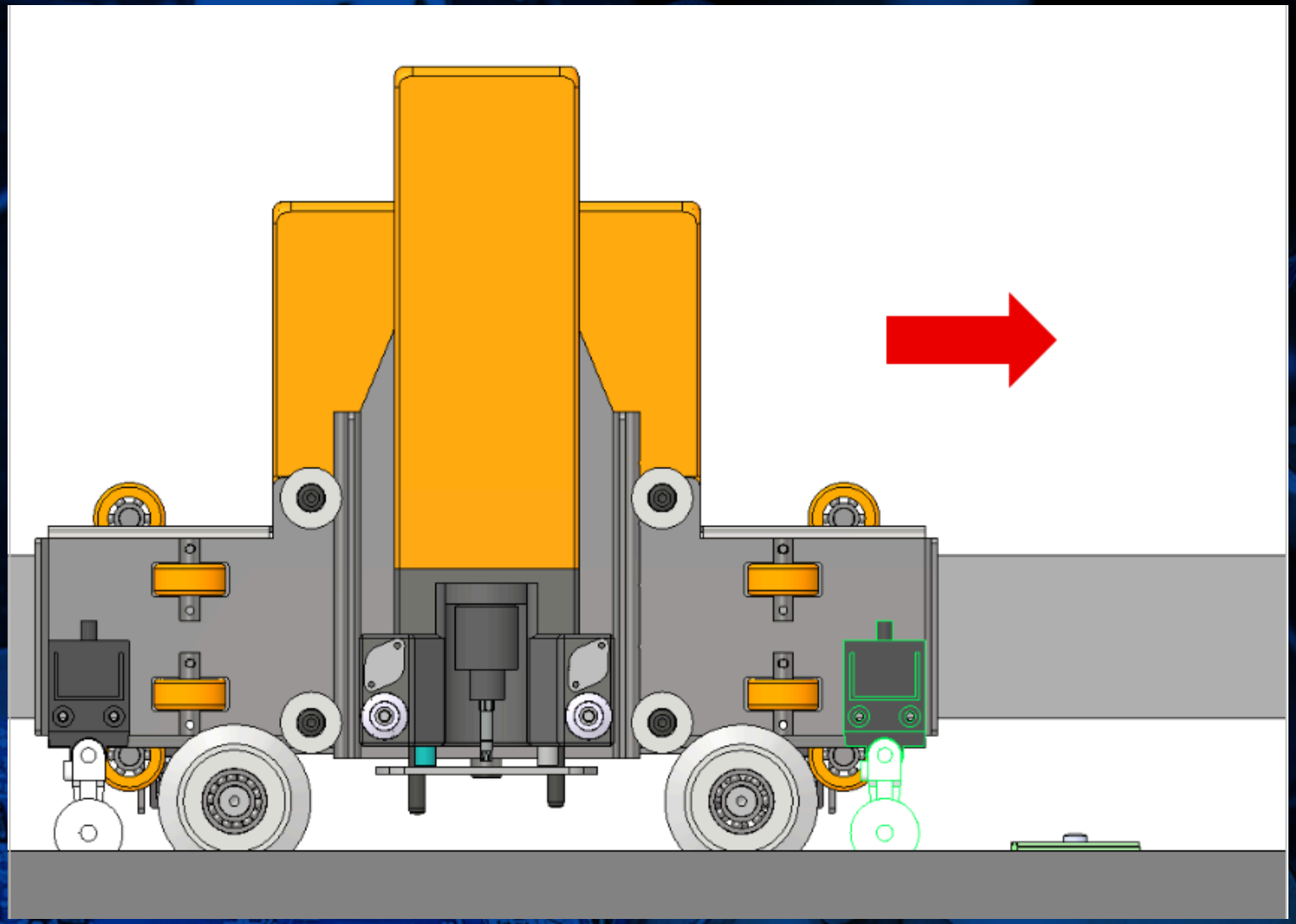
- Deliver next week.
- It must be cheap.
- It must be fast & reliant.



HOW HARD CAN IT BE?

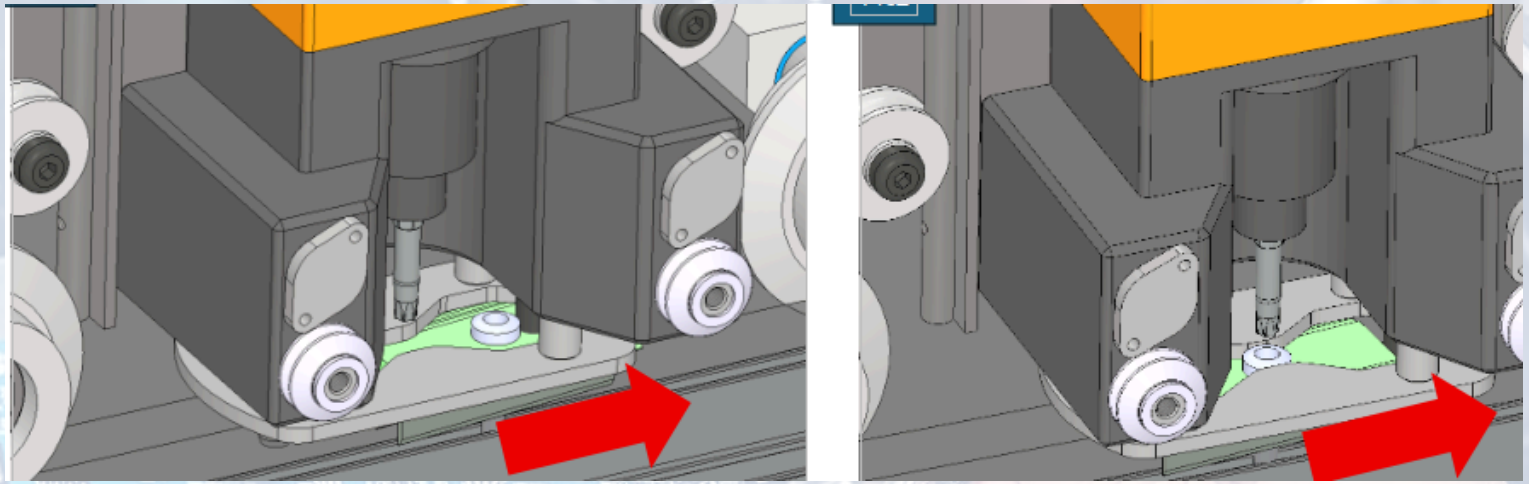
- Drive up to a screw
- Loosen screw
- Tighten with correct torque

THE MACHINE

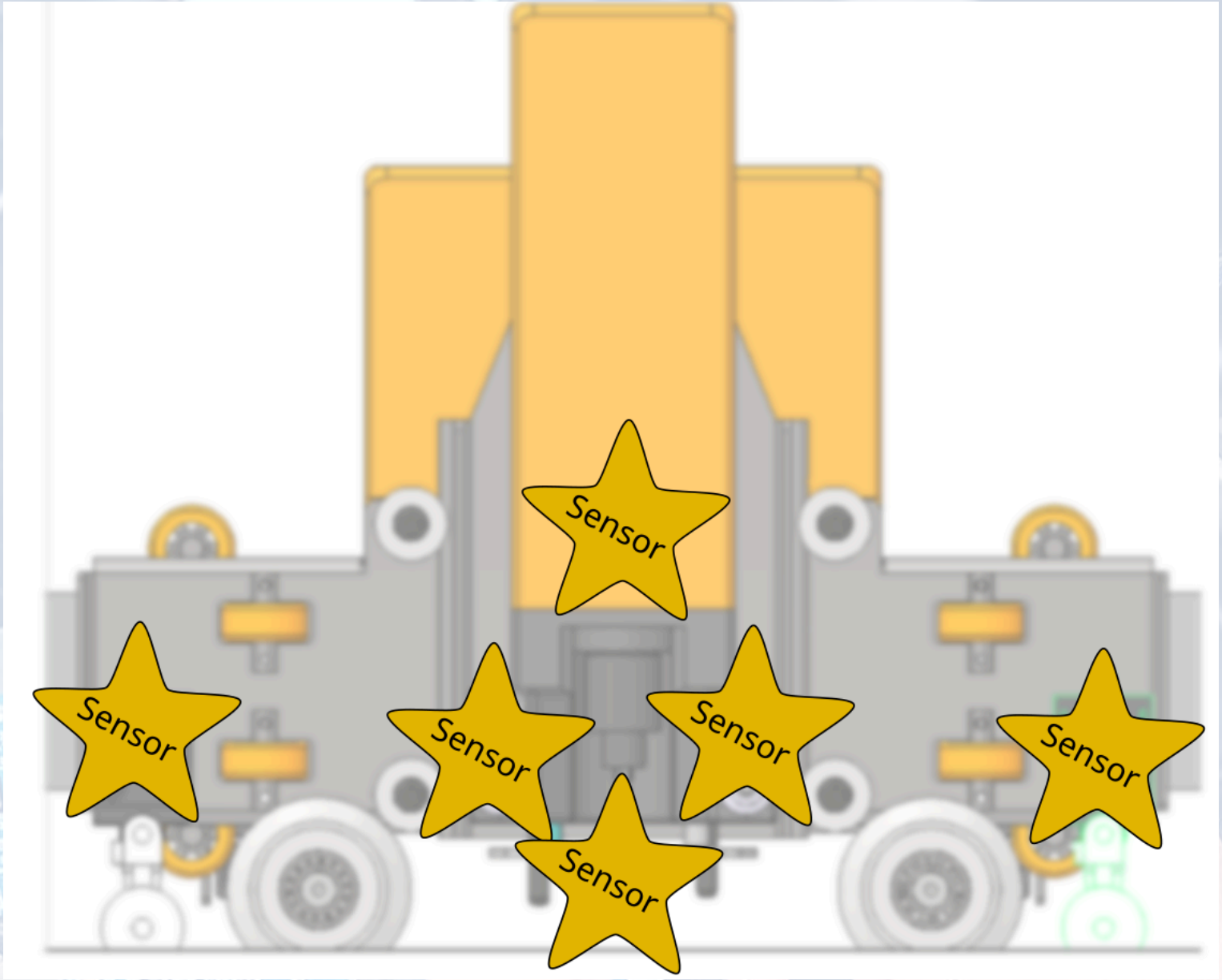


REDUCE FIRMWARE SCOPE

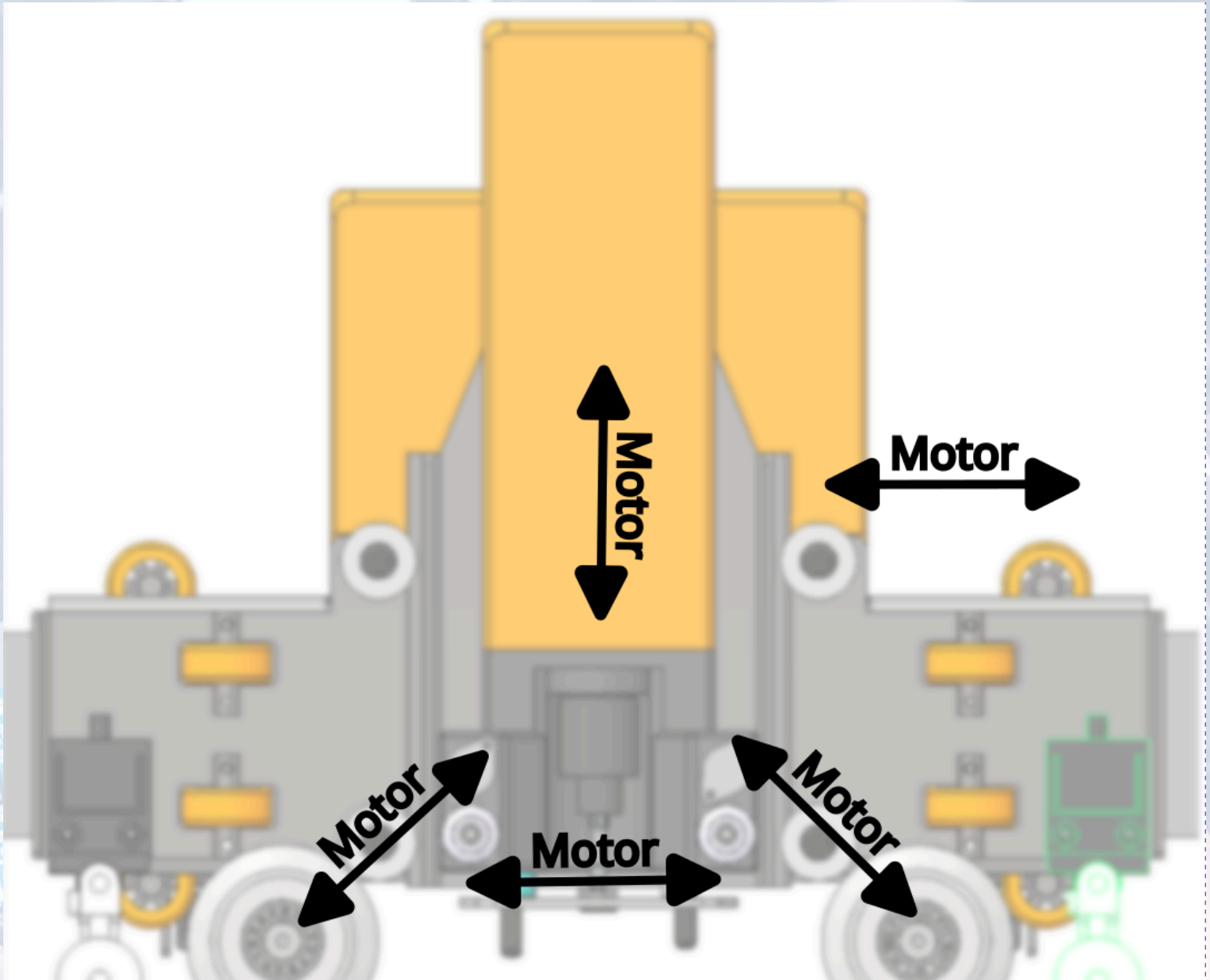
- No camera
- Mechanical alignment of position
- Mechanical adjustable sensors to tweak settings
- Retry logic for torx engagement



SENSORS



MOTORS



CHANGES:

- Stepper -> DC, with and without encoder
- Encoder generating too many ticks
- RoboClaw too late, but good solution.
- Created own PCB / hat <-— DTS for the win
- Yet another motor
- Yet another sensor for alignment

CHALLENGES:

- Hardware very late in development
- **one machine**: one motor, one battery, one ….
- First customer demo day, was first day attaching the real encoder.

TIMELINE

- May: Initial project rumors
- June: Development starts
- July: Mechanical prototyping
- August: Integration / Development / PCB
- September: Field testing
- October: *done*...

THE CHOSEN SOLUTION

We had a say in choice of components.

- Zephyr: We are familiar with it.
- MCU: nRF52840DK
- Modern C++: for the application

STARTING FROM AN IDEA

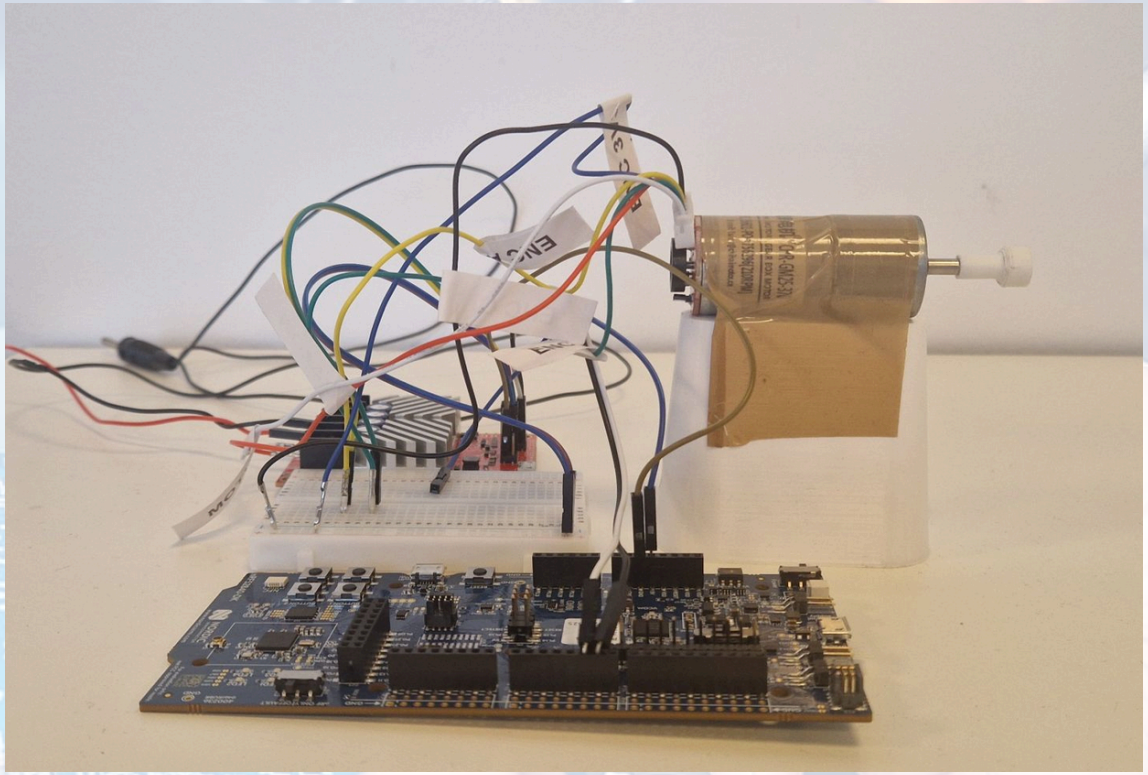
No hardware or test device when starting development!

We started in two places:

- Hardware bring up with functionally equivalent components
- Firmware bring up with functionally equivalent interfaces

HARDWARE BRING UP

- Device Tree Specification (DTS)
- Drivers
- Instrument with specific utilities for test:
  `ShellCommands`



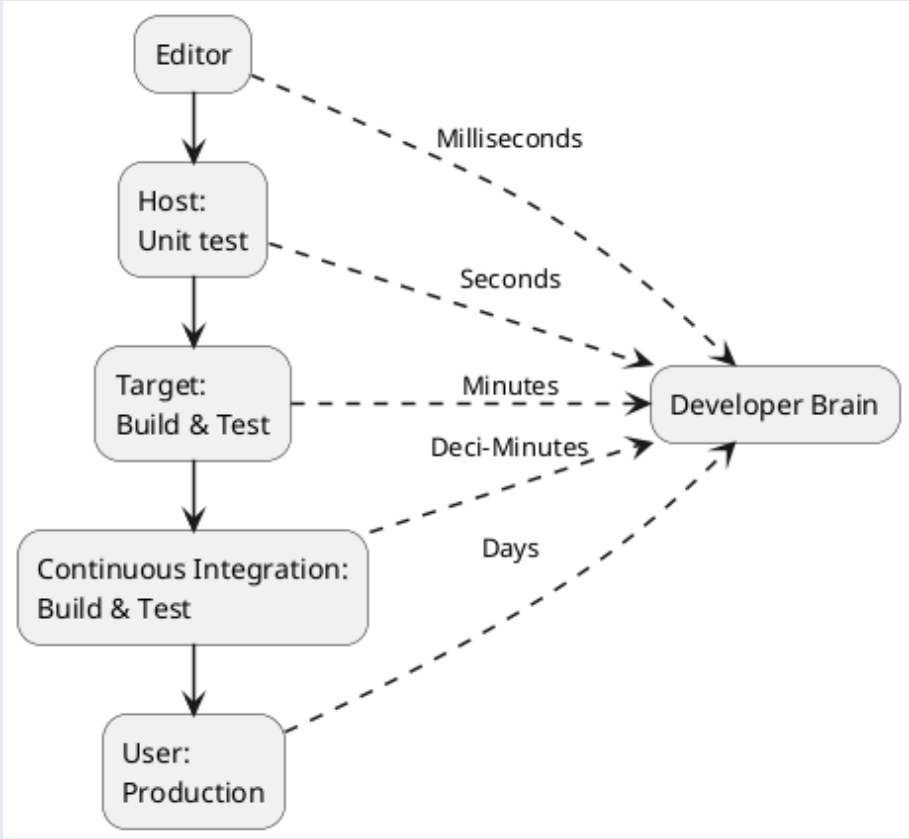
FIRMWARE BRING UP

- Event driven architecture
- Interfaces for components
- Unit tests

TDD

- Goal is fast feedback
- Stay *in the zone*
- Work in small increments
- Avoid rework

FAST FEEDBACK



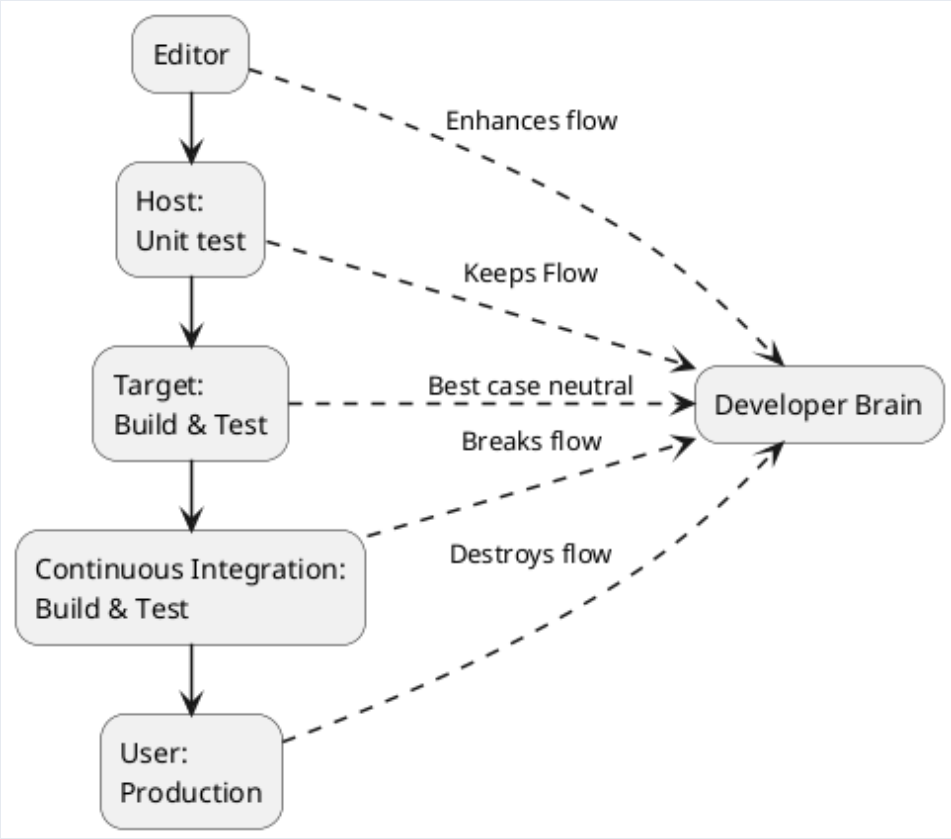
THE ZONE

... people who experience flow tend to describe it similarly. There's a feeling of timelessness. The task seems easy and things just "come together."

There's this focus that, once it becomes intense, leads to a sense of ecstasy, a sense of clarity: you know exactly what you want to do from one moment to the other; you get ***immediate feedback***

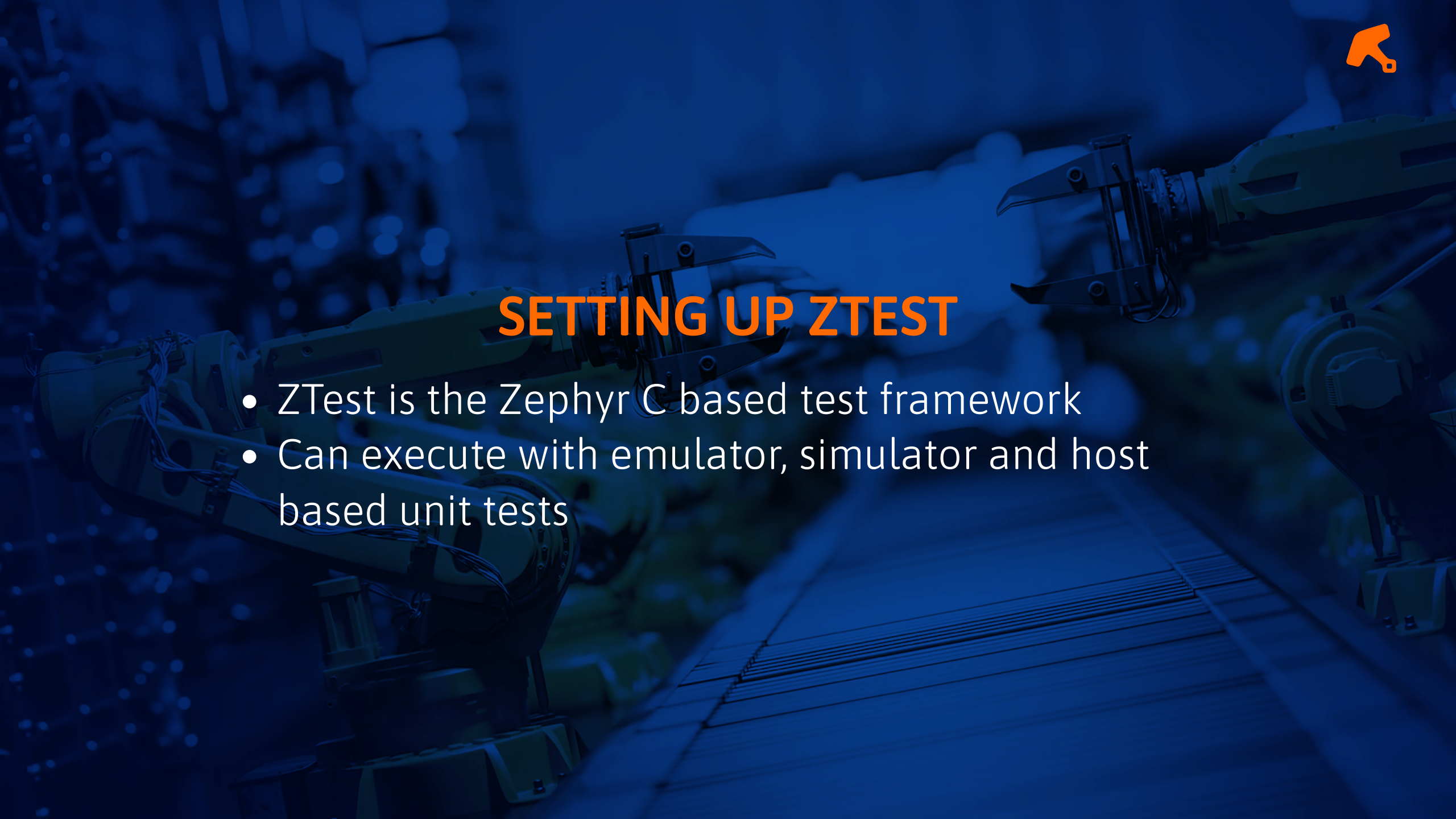
Mihaly Csikszentmihalyi (Me-high Cheek-sent-me-high)

FEEDBACK TO ENHANCE FLOW



THE INNER LOOP

1. Small change
2. Observe failure
3. Fix failure
4. Enjoy
5. Refactor

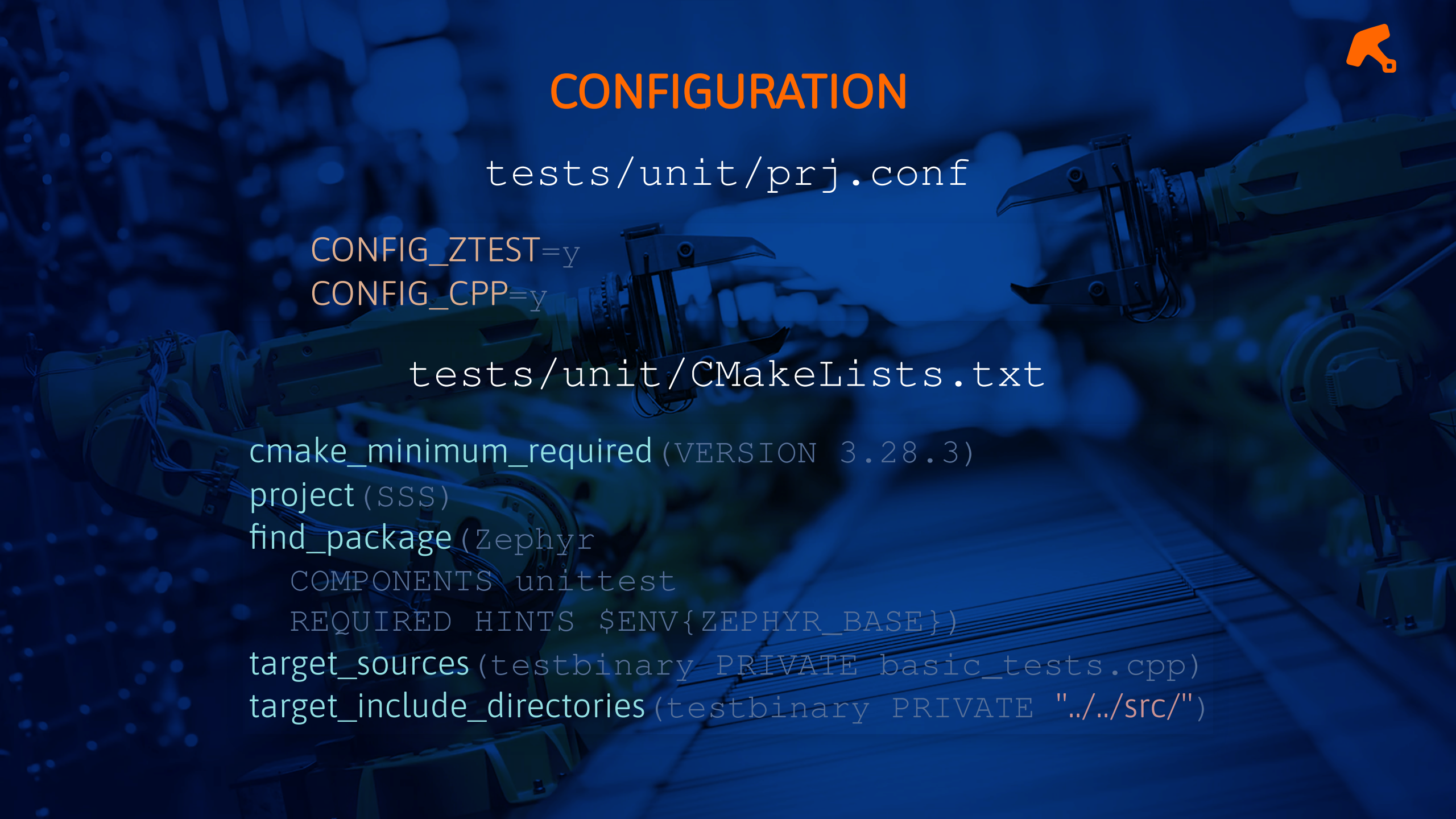


SETTING UP ZTEST

- ZTest is the Zephyr C based test framework
- Can execute with emulator, simulator and host based unit tests

PROJECT LAYOUT

```
├── CMakeLists.txt
├── Kconfig
├── prj.conf
├── west.yml
├── README.md
├── src/
├── tests/unit/
│       ├── basic_tests.cpp
│       ├── CMakeLists.txt
│       ├── prj.conf
│       └── testcase.yaml
```



CONFIGURATION

tests/unit/prj.conf

```
CONFIG_ZTEST=y
CONFIG_CPP=y
```

tests/unit/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28.3)
project(SSS)
find_package(Zephyr
  COMPONENTS unittest
  REQUIRED HINTS $ENV{ZEPHYR_BASE})
target_sources(testbinary PRIVATE basic_tests.cpp)
target_include_directories(testbinary PRIVATE "../../src/")
```

CONFIGURATION

tests/unit/testcase.yaml

```
tests:
sss.basic:
  tags: SSS_tests
  type: unit
```

tests/unit/basic_tests.cpp

```
#include <zephyr/ztest.h>
ZTEST_SUITE(SSS_tests, NULL, NULL, NULL, NULL,
ZTEST(SSS_tests, test_start)
{
  zassert_equal(true, false, "start with a failing test");
}
```

OUTPUT FROM TWISTER

```
(.venv) jacob@ewolf:~/src/SSS$ west twister -T SolarScrewSystem/tests/unit/ -O hest -n
Keeping artifacts untouched
INFO    - Using Ninja..
INFO    - Zephyr version: v4.0.99-ncs1-2
INFO    - Using 'zephyr' toolchain.
INFO    - Selecting default platforms per testsuite scenario
INFO    - Building initial testsuite list...
INFO    - JOBS: 12
INFO    - Adding tasks to the queue...
INFO    - Added initial list of jobs to queue
ERROR   - unit_testing/unit_testing sss.basic                                            FAILED: Failed (rc=1)
ERROR   - see: /home/jacob/src/SSS/hest/unit_testing_unit_testing/host/sss.basic/handler.log
INFO    - Total complete:    1/    1  100%  built (not run):    0, filtered:    0, failed:    1, error:    0
INFO    - 1 test scenarios (1 configurations) selected, 0 configurations filtered (0 by static filter, 0 at runtime).
INFO    - 0 of 1 executed test configurations passed (0.00%), 0 built (not run), 1 failed, 0 errored, with no warnings in 5.37 seconds.
INFO    - 52 of 53 executed test cases passed (98.11%), 1 failed on 1 out of total 929 platforms (0.11%).
INFO    - 1 test configurations executed on platforms, 0 test configurations were only built.
INFO    - Saving reports...
INFO    - Writing JSON report /home/jacob/src/SSS/hest/twister.json
INFO    - Writing xunit report /home/jacob/src/SSS/hest/twister.xml...
INFO    - Writing xunit report /home/jacob/src/SSS/hest/twister_report.xml...
INFO    - -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
INFO    - The following issues were found (showing the top 10 items):
INFO    - 1) sss.basic on unit_testing/unit_testing failed (Failed (rc=1))
INFO    -
INFO    - To rerun the tests, call twister using the following commandline:
INFO    - west twister -p <PLATFORM> -s <TEST ID>, for example:
INFO    -
INFO    - west twister -p unit_testing/unit_testing -s sss.basic
INFO    - or with west:
INFO    - west build -p -b unit_testing/unit_testing SolarScrewSystem/tests/unit -T sss.basic
INFO    - -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
INFO    - Run completed
```

IDE INTEGRATION

```
#include "fakes.hpp"
#include "zephyr/shell/shell.h"

ZTEST_SUITE(SSS_tests, NULL, NULL, NULL, NULL, NULL);

ZTEST(SSS_tests, test_start)
{
  zassert_equal(true, false, "start with a failing test");
}
```

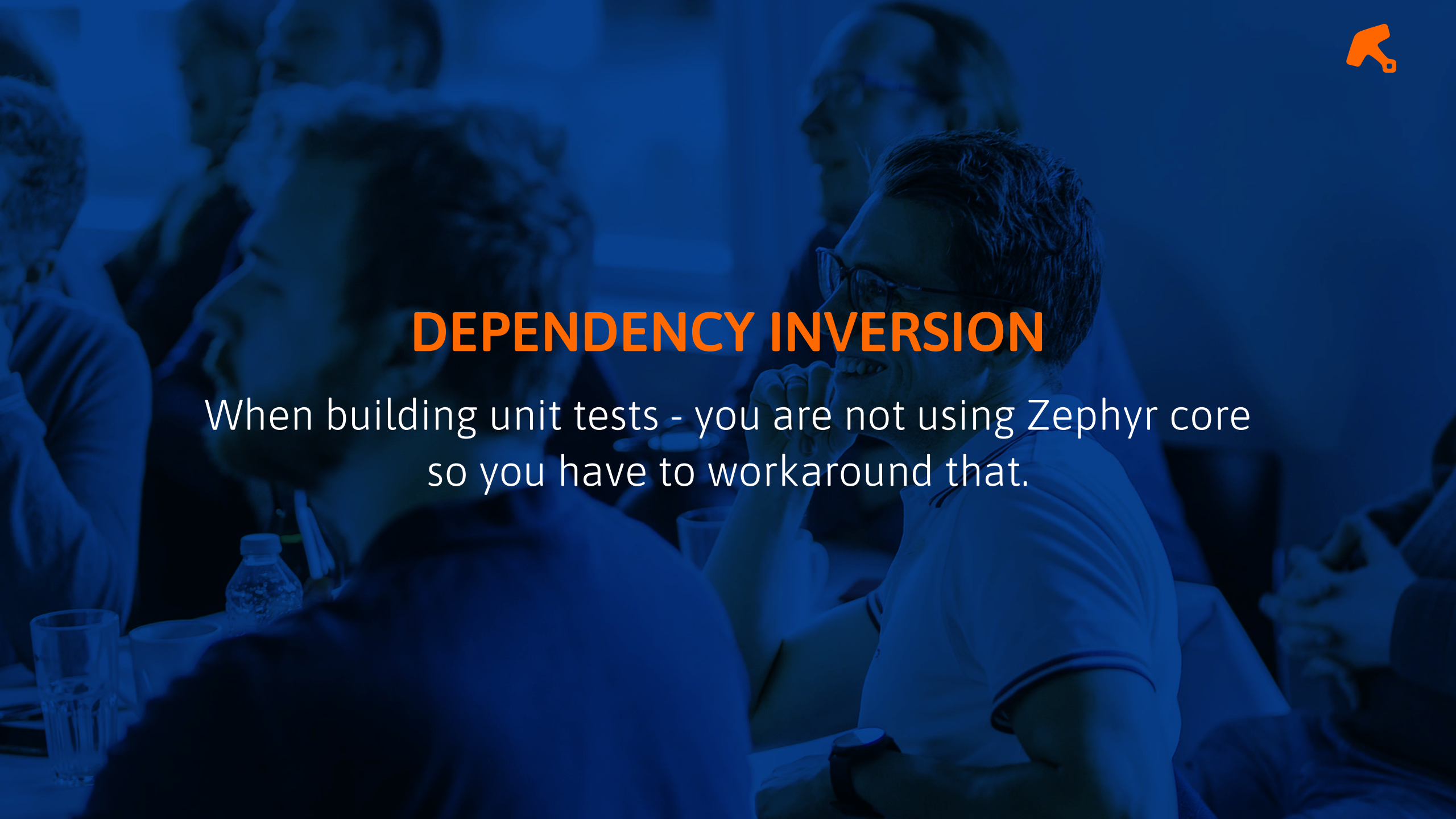
```
===================================================================
START - test_start

    Assertion failed at /home/jacob/src/SSS/SolarScrewSystem/tests/unit/basic_tests.cpp:
13: SSS_tests_test_start: (true not equal to false)
start with a failing test
 at test function
 FAIL - test_start in 0.000 seconds
===================================================================
```

GETTING FASTER FEEDBACK

`testbinary` is a Zephyr App, let's remove overhead from `twister`.

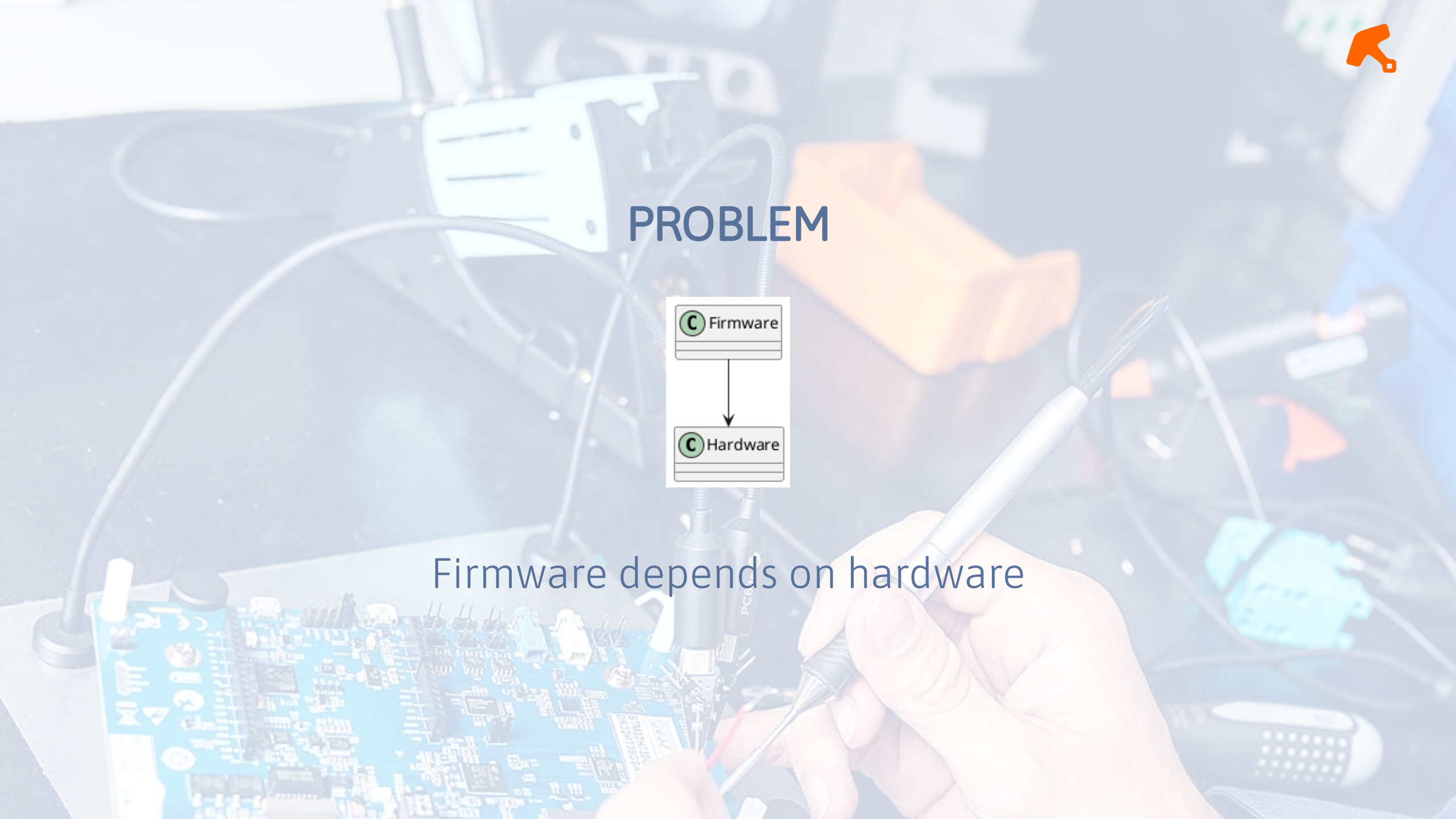
```
cmake -S XXX/tests/unit -B build_tests -DBOARD=unit_testing
cmake --build build_tests -t run-test
```

Method	Pristine build	Incremental build
Twister	0m8.584s	0m5.757s
Direct	0m6.965s	0m1.143s
Direct vs Twister	18.9% faster	80.2% faster



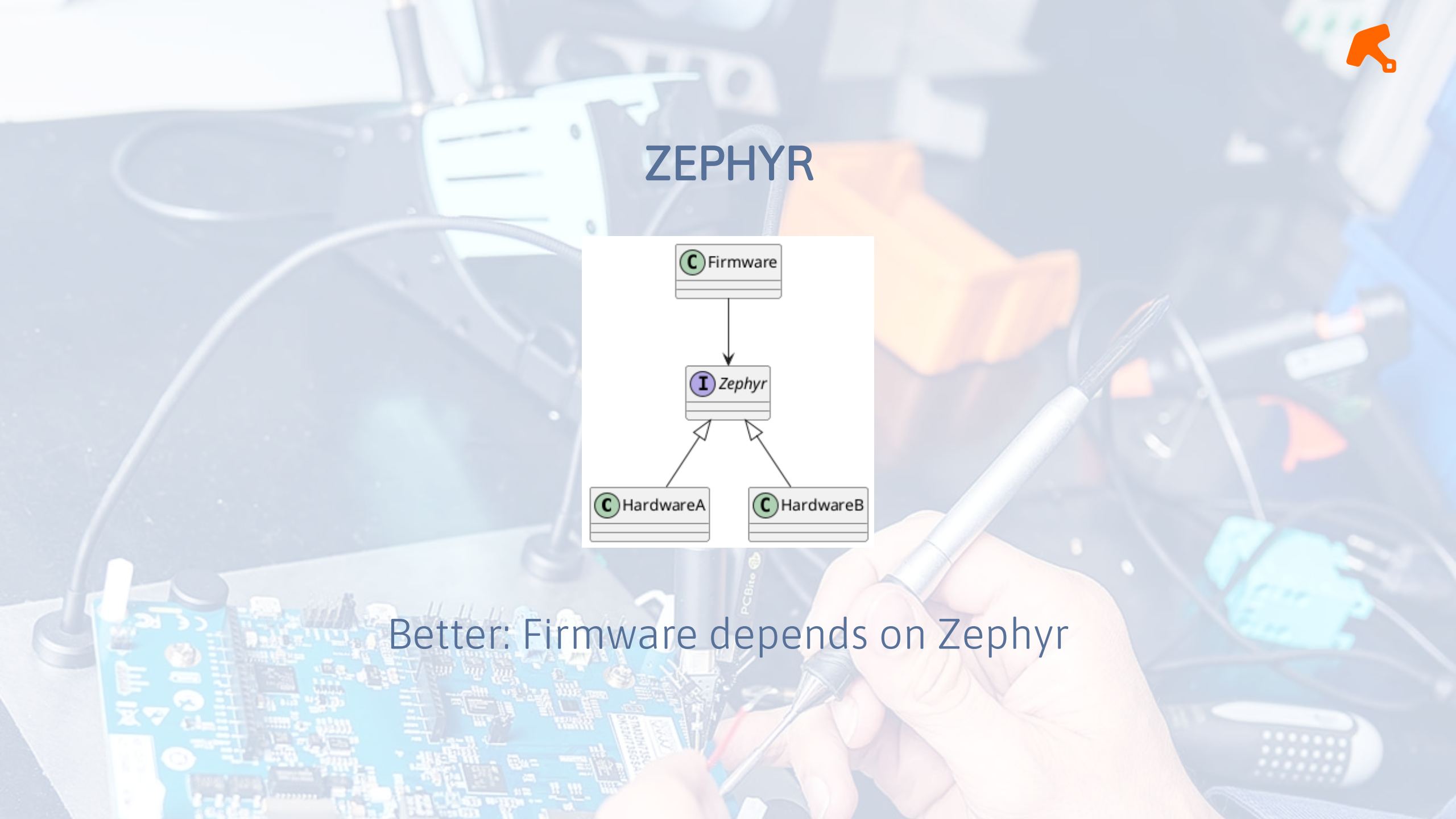
DEPENDENCY INVERSION

When building unit tests - you are not using Zephyr core
so you have to workaround that.



PROBLEM

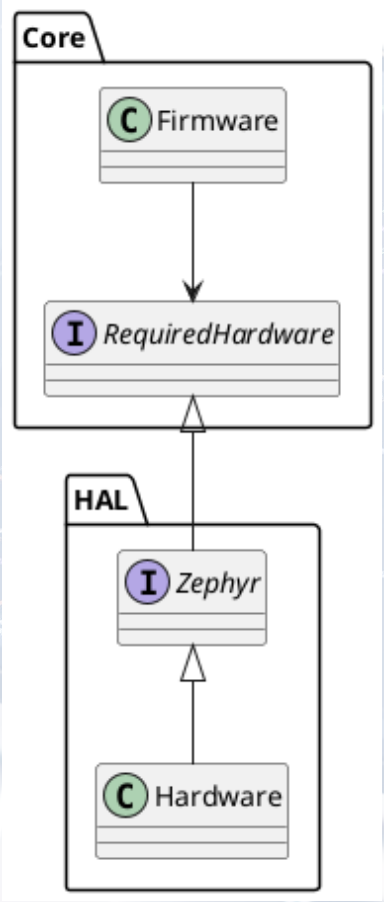
Firmware depends on hardware



ZEPHYR

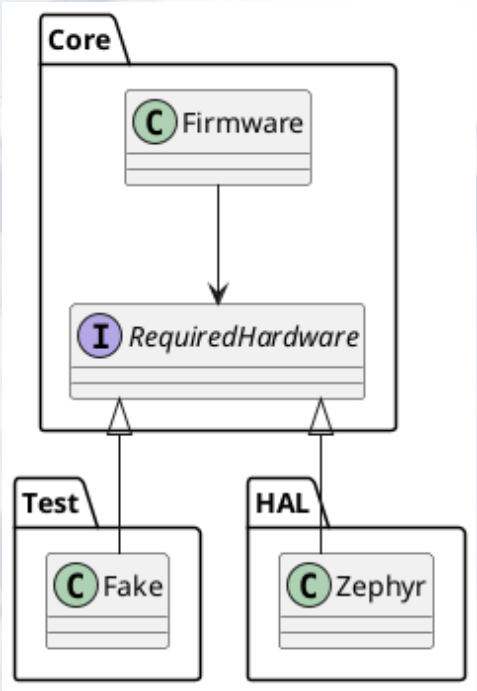
Better: Firmware depends on Zephyr

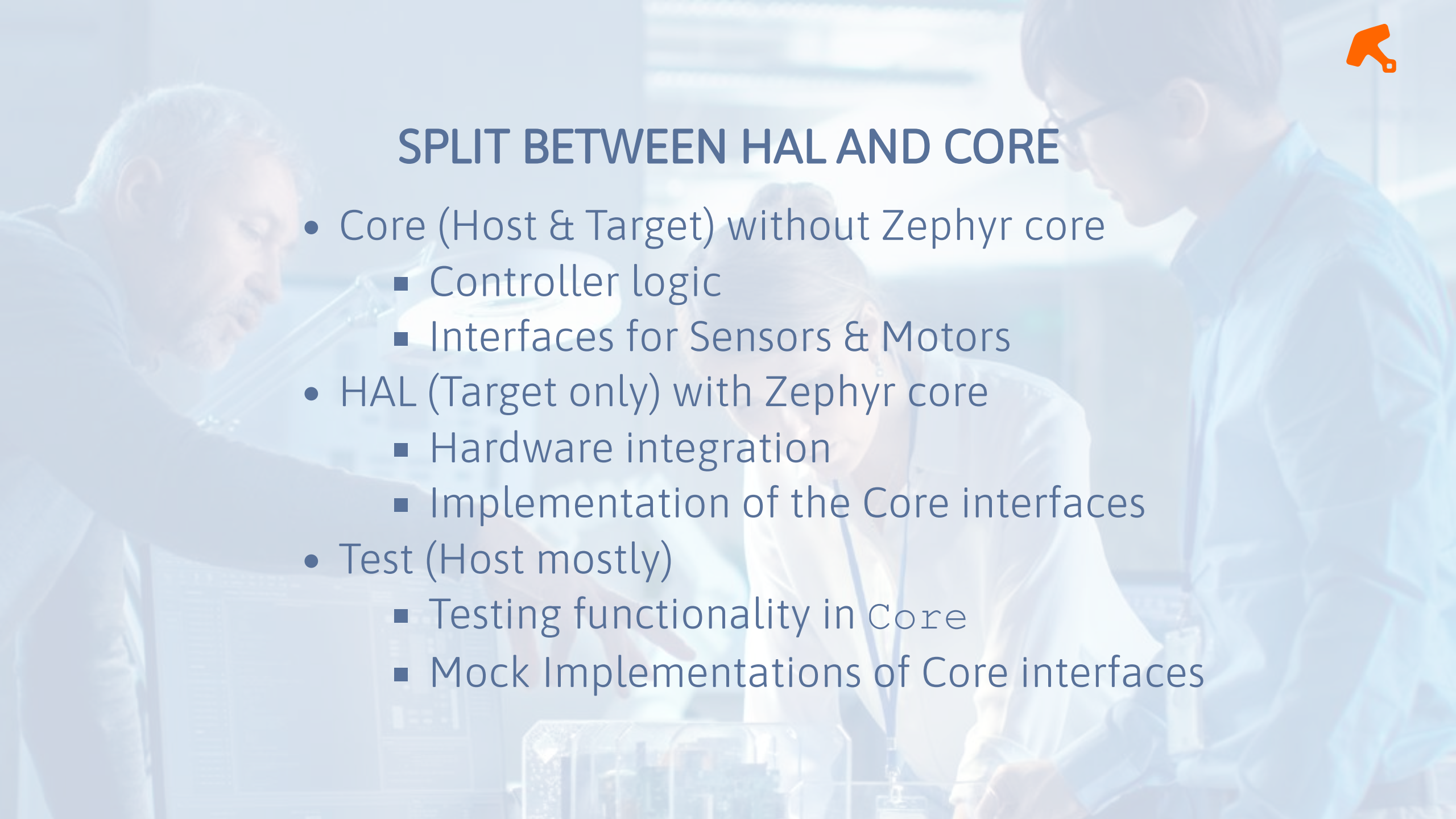
INVERSION



Inverted: Hardware depends on Firmware

TESTABILITY

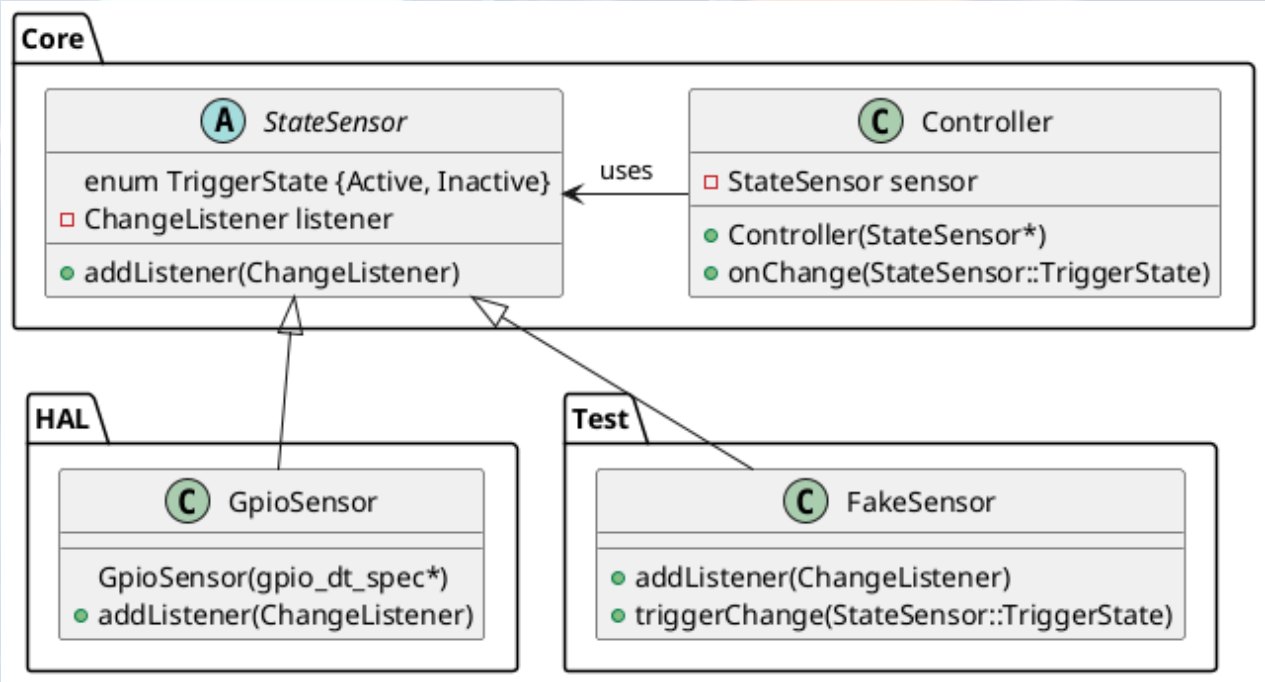




SPLIT BETWEEN HAL AND CORE

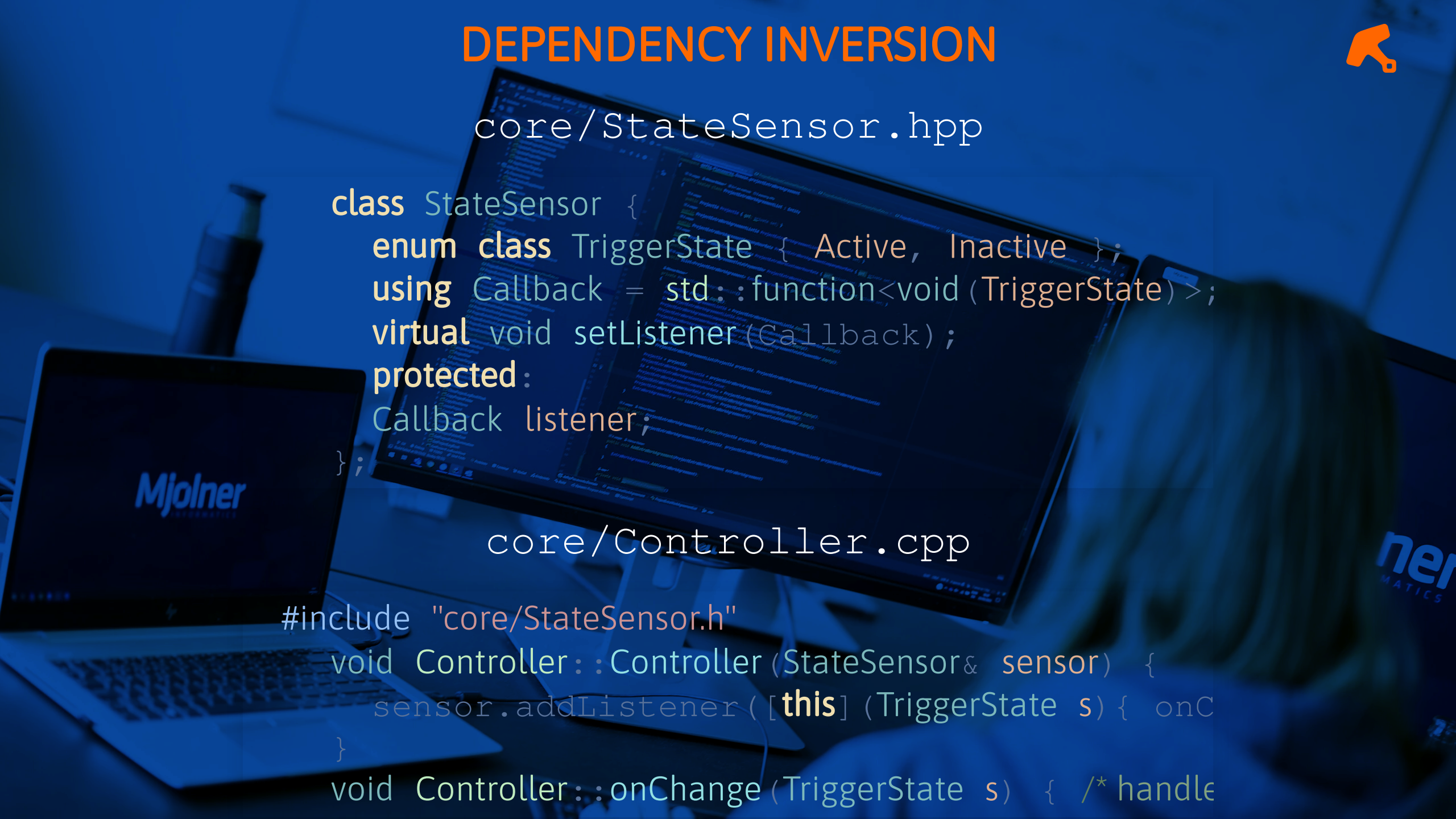
- Core (Host & Target) without Zephyr core
  - Controller logic
  - Interfaces for Sensors & Motors
- HAL (Target only) with Zephyr core
  - Hardware integration
  - Implementation of the Core interfaces
- Test (Host mostly)
  - Testing functionality in `Core`
  - Mock Implementations of Core interfaces

STATESENSOR



- Micro-switch
- Proximity Sensor
- Output from Controller Box

DEPENDENCY INVERSION



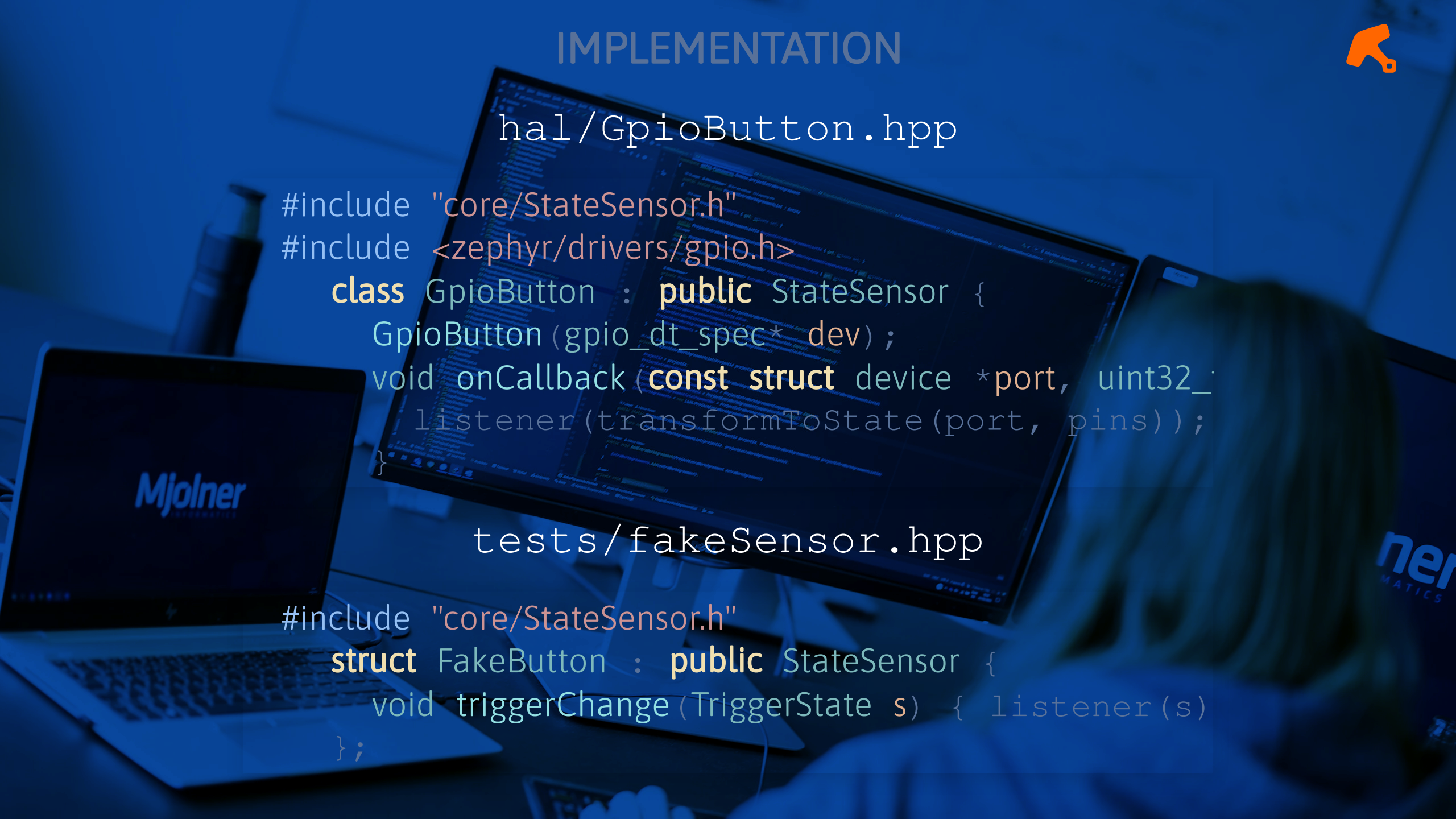
core/StateSensor.hpp

```
class StateSensor {
   enum class TriggerState { Active, Inactive };
   using Callback = std::function<void(TriggerState)>;
   virtual void setListener(Callback);
   protected:
   Callback listener;
};
```

core/Controller.cpp

```
#include "core/StateSensor.h"
   void Controller::Controller(StateSensor& sensor) {
      sensor.addListener([this](TriggerState s){ onC
   }
   void Controller::onChange(TriggerState s) { /* handle
```

IMPLEMENTATION



hal/GpioButton.hpp

```
#include "core/StateSensor.h"
#include <zephyr/drivers/gpio.h>
    class GpioButton : public StateSensor {
      GpioButton(gpio_dt_spec* dev);
      void onCallback(const struct device *port, uint32_
        listener(transformToState(port, pins));
      }
```

tests/fakeSensor.hpp

```
#include "core/StateSensor.h"
    struct FakeButton : public StateSensor {
      void triggerChange(TriggerState s) { listener(s)
    };
```

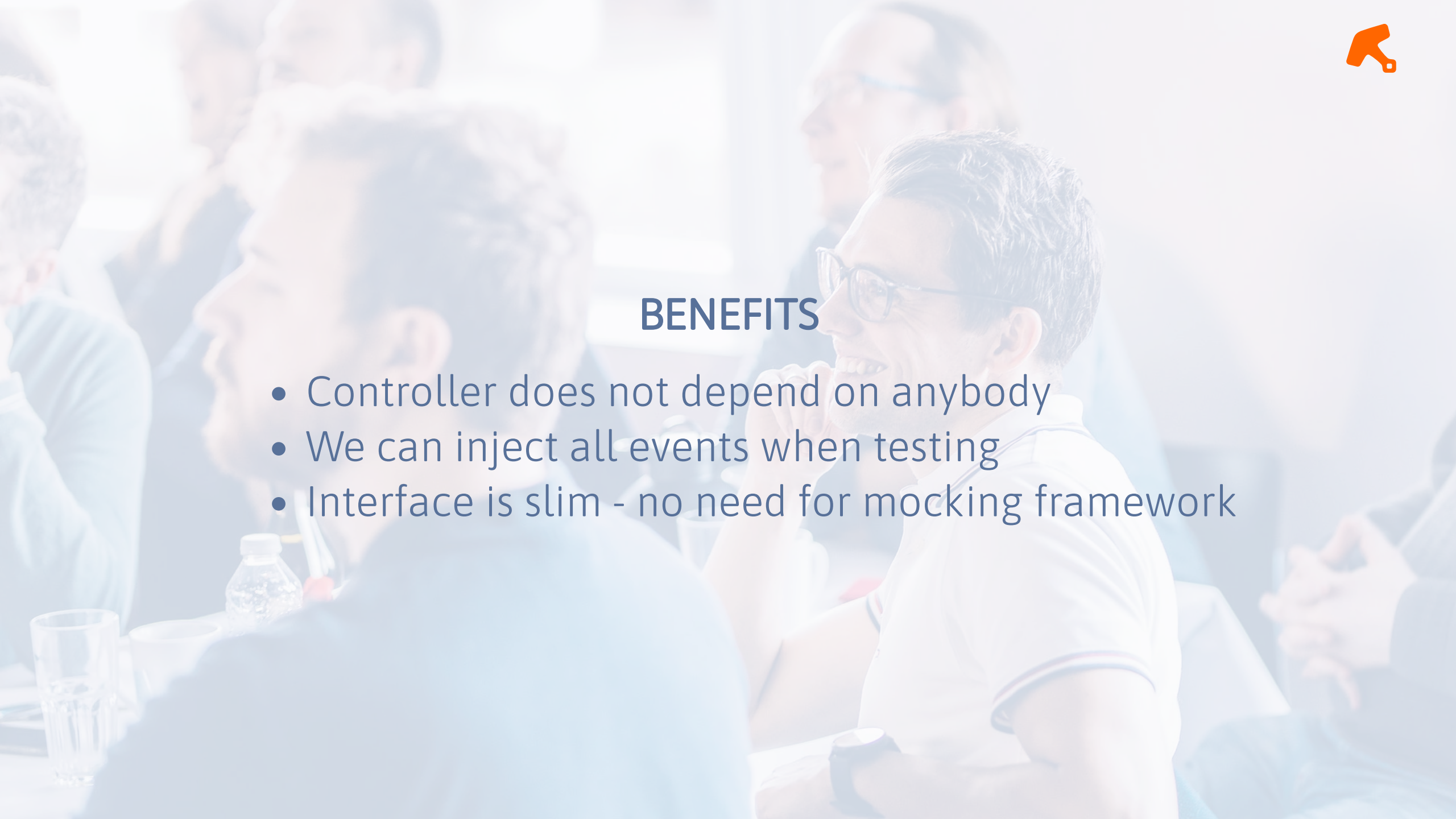
USAGE

main.cpp

```
static const struct gpio_dt_spec button_dt_spec =
    GPIO_DT_SPEC_GET(DT_NODELABEL(button), gpio

auto sensor = GpioButton(&button_dt_spec);
auto controller = Controller(sensor);
// World triggers sensor
```

test.cpp

```
auto sensor = FakeSensor();
auto controller = Controller(sensor);
sensor.triggerChange(..);
```



BENEFITS

- Controller does not depend on anybody
- We can inject all events when testing
- Interface is slim - no need for mocking framework

WHAT ABOUT THE HARD THINGS

Other things that stands as hard:

Logging

```
LOG_MODULE_REGISTER(ScrewDriverFactory,
CONFIG_LOG_DEFAULT_LEVEL);
```

Timing

```
k_work_init_delayable(&callback_data.delayable,
&onDebounceExpired);
```

Solution: *Dependency inversion and interfaces.*

LOGGING

core/logging.h

```
#ifndef ZTEST
#include <zephyr/logging/log.h>
#else
std::vector<std::string> loglines;
void logCapture(const char* level, const char* fmt, ...)
  //store logline
}
#define LOG_ERR(fmt, ...) \
    if (log_level >= LOG_LEVEL_ERR) testlog::log_ca
...
```


WE CAN TESTS THINGS
controller.powerDown();
zassert_false(TestLog::containsMessage(\
"I'm afraid I can't let you do that Dave"));

TIMING

```
screw_driver_->loosen();
context_->schedule(42ms,
                   [this]() {
                     screw_driver_->tight
                   });
```

Output is executed in *Context*-thread.

We used Zephyrs SystemWorkQueue.

CONTEXT BENEFITS

- No dependency on Zephyr
- Testing of timed events
  - `FakeContext.triggerNextEvent();`
- No need for mutexes in our core logic.
  - Every thing executes in order on the same thread.
- Caveat: All ISR/ShellCommands most be posted to Context.

TESTING THE *CONTEXT*

Here we added tests using an emulated target, to make sure that we did not create dead/live-locks with our `Context` implementation.

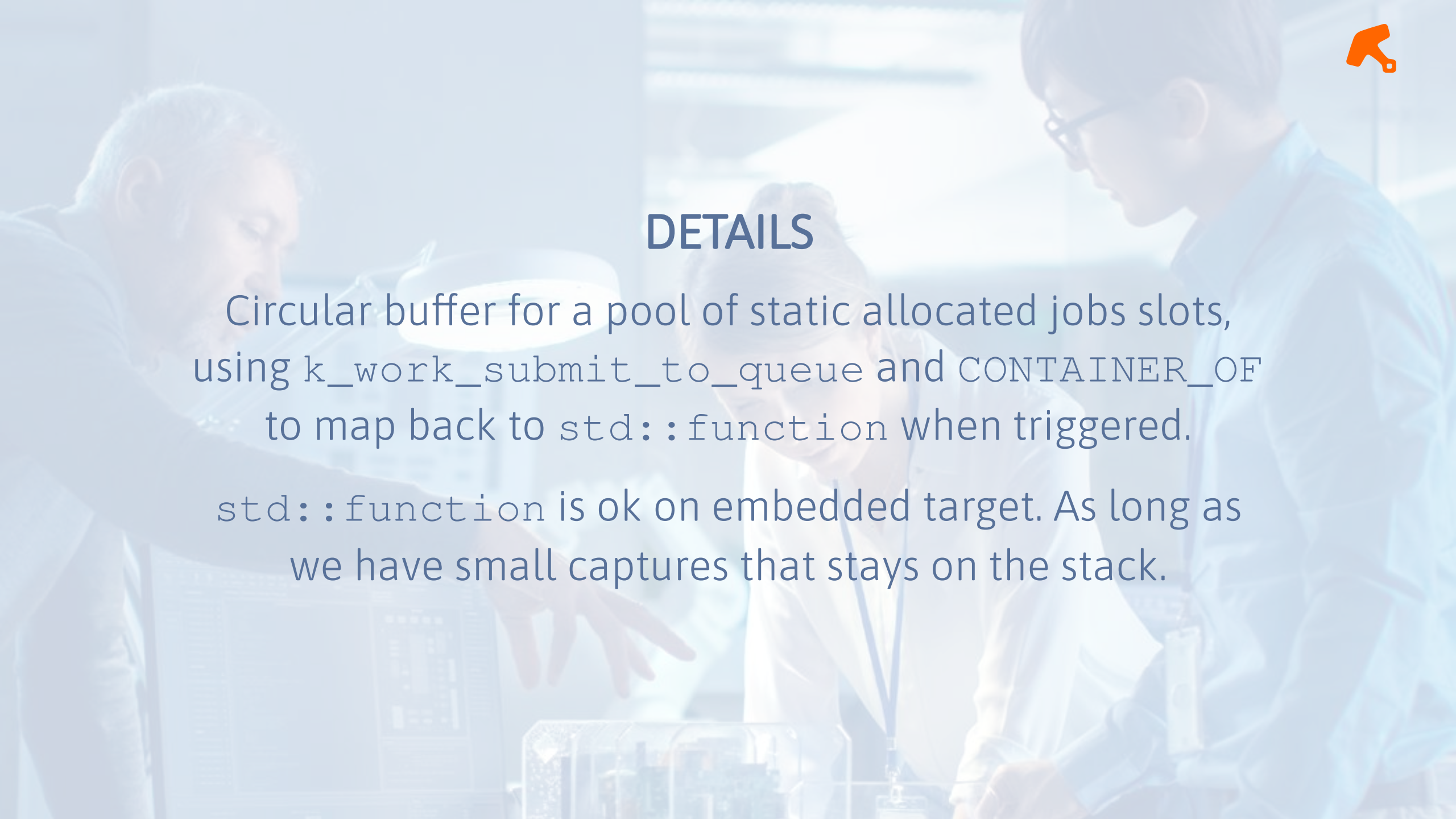
WRAP KWORKQUEUE WITH ASIO LIKE CONTEXT

core/Context.hpp

```
struct Context {
  virtual void post(std::function<void()> task) = 0;
  virtual void schedule(std::chrono::milliseconds,
                        std::function<void()> task)
};
```

hal/SystemWorkQueueContext.cpp

```
void SystemWorkQueueContext::post(std::function<void()>
  auto kworker = allocate();
  tasks[kworker->idx] = fn;
  k_work_init(&kworker->work, &workQueueHandler
  k_work_submit_to_queue(&k_sys_work_q, &kworke
}
```



DETAILS

Circular buffer for a pool of static allocated jobs slots, using `k_work_submit_to_queue` and `CONTAINER_OF` to map back to `std::function` when triggered.

`std::function` is ok on embedded target. As long as we have small captures that stays on the stack.

CONCLUSION

The Project is ~~still in development~~, good enough for now.

CHEAP REMOTE CONTROL

- ShellCommands is the "killer app"
- Really rapid prototyping
- NUS + BLE -> almost remote control for free

WHAT DID WE LEARN

- Zephyr worked as solid platform for Rapid Prototyping
- We can and will wrap C in abstractions:
  - Type safety
  - Testability
- TDD: Makes us go faster