



POLITÉCNICA

“Ingeniamos el futuro”

Laboratorio de Sistemas Electrónicos

Máster en Ingeniería de Sistemas
Electrónicos

Bloque 3: Tests unitarios
Código heredado (legacy)

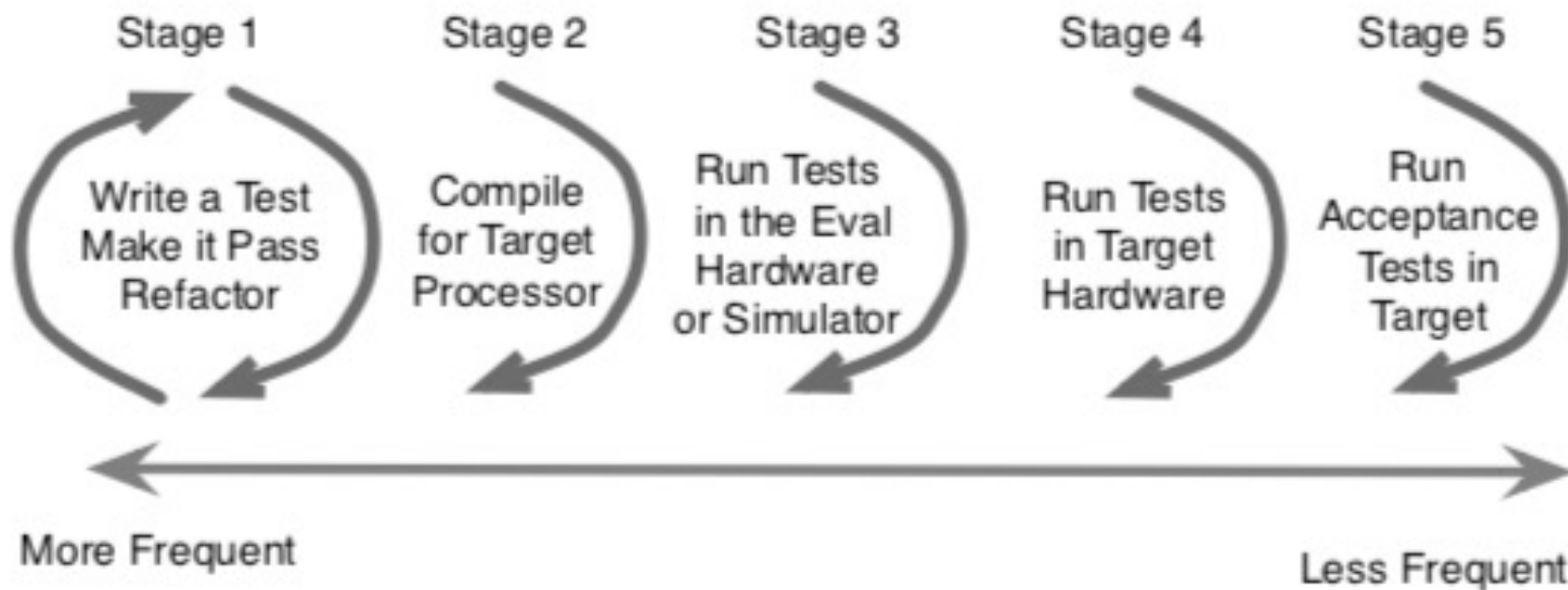
Índice

- Contexto y objetivo de las prácticas bloque 3
- Objetivo de la práctica de hoy
- Apoyo teórico: tests
- Pasos a seguir

Índice

- **Contexto y objetivo de las prácticas bloque 3**
- Objetivo de la práctica de hoy
- Apoyo teórico: tests
- Pasos a seguir

Flujo de TDD en empotrados



Objetivo de las prácticas

Mejorar librería fsm (vista en SEMP)

Tres fases:

- **Legacy: Añadir tests a código existente**
- TDD: Usar TDD para generar nuevo código
- CI/CD: Utilizar CI/CD en Github para:
 - Comprobar tests y mostrar en repositorio resultado
 - Publicar página web con los resultados y documentación

Librería fsm

Reserva memoria para crear una máquina de estados

```
fsm_t * fsm_new(fsm_trans_tt* tt);
```

Liberación de memoria de máquina de estados

```
void fsm_destroy(fsm_t* f);
```

Inicializa máquina de estados en variable ya creada

```
bool fsm_init(fsm_t* f, fsm_trans_tt* tt);
```

Librería fsm

Comprobación periódica de la máquina de estados

```
void fsm_fire(fsm_t* f);
```

Consulta de estado actual

```
int fsm_get_state(fsm_t *p_fsm);
```

Pone el estado actual

```
void fsm_set_state(fsm_t *p_fsm, int state);
```

Tabla de transiciones

```
bool funcion_check(fsm_t* f) {  
    return true;  
}
```

```
void funcion_update(fsm_t* f) {  
}
```

```
fsm_trans_t tt[] = {  
    {0, funcion_check, 1, funcion_update},  
    {1, funcion_check, 0, NULL},  
    {-1, NULL, -1, NULL}  
};
```


Tipos de test

- **State-based testing**

- Evalúa una funcionalidad aislada

1. Setup (estado del sistema)

2. Ejecución

3. Comprobación (valor devuelto y estado)

- **Interaction-based testing**

- Evalúa una funcionalidad muy relacionada con otras

1. Setup (dobles y estado del sistema, qué espera)

2. Ejecución (monitoriza interacción con dobles)

3. Comprobación (interacción esperada y resultado)

Usamos ceedling

- Unity: asserts y motor de tests
- CMock: interacción, dobles
- Automatización: ceedling es en ruby
- Resultados:
 - Compilan y ejecutan los tests
 - Tests pasados, tests con fallo, tests ignorados
 - gcov: Cobertura (código cubierto por tests)

Índice

- Contexto y objetivo de las prácticas bloque 3
- **Objetivo de la práctica de hoy**
- Apoyo teórico: tests
- Pasos a seguir

Legacy

- Completar los tests planteados en el fichero `test_fsm_legacy.c`
- El título de cada test y el comentario indican qué tiene que probar
- Evitar tocar el código a no ser que se detecte un error o haya que hacer un “truco” para poder probar algo
 - No está previsto que haya que tocar `fsm.c` o `fsm.h` en esta práctica

Índice

- Contexto y objetivo de las prácticas bloque 3
- Objetivo de la práctica de hoy
- **Apoyo teórico: tests**
- Pasos a seguir

Unity: test_fsm.c

```
#include "unity.h"

//Para que pueda haber tests parametrizables (con argumentos)
#define TEST_CASE(...)
#define TEST_RANGE(...)

//Módulo a probar (CUT o Code Under Test)
#include "fsm.h"

//Funciones de preparación y limpieza
//Comunes para los tests de este fichero
void setUp (void) { }
void tearDown (void) { }
```

Unity: test_fsm.c

```
/**
 * @brief Comprueba que la funcion de fsm_new devuelve NULL
 * y no llama a fsm_malloc si la tabla de transiciones es NULL
 */
void test_fsm_new_nullWhenNullTransition(void) {
    fsm_t *f = (fsm_t*)1;
    f = fsm_new(NULL);
    TEST_ASSERT_EQUAL (NULL, f);
}
```

<https://github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityAssertionsReference.md>

Unity: test_fsm.c

```
/**
 * @brief La máquina de estados devuelve NULL
 *        y no llama a fsm_malloc si el estado de origen
 *        de la primera transición es -1 (fin de la tabla)
 */
void test_fsm_nullWhenFirstOrigStateIsMinusOne (void) {
    fsm_trans_t tt[] = {{-1, is_true, 1, do_nothing}};
    fsm_t *f = (fsm_t*)1;
    f = fsm_new(tt);

    //TEST_ASSERT_EQUAL (XXX);
    TEST_FAIL_MESSAGE("Implement the test");
}
```

<https://github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityAssertionsReference.md>

Basic Fail and Ignore

TEST_FAIL()

TEST_IGNORE()

Boolean

TEST_ASSERT (condition)

TEST_ASSERT_TRUE (condition)

TEST_ASSERT_UNLESS (condition)

TEST_ASSERT_FALSE (condition)

TEST_ASSERT_NULL (pointer)

TEST_ASSERT_NOT_NULL (pointer)

Signed and Unsigned Integers (of all sizes)

TEST_ASSERT_EQUAL_INT (exp, act)

TEST_ASSERT_EQUAL_INT8 (exp, act)

TEST_ASSERT_EQUAL_INT16 (exp, act)

TEST_ASSERT_EQUAL_INT32 (exp, act)

TEST_ASSERT_EQUAL_INT64 (exp, act)

TEST_ASSERT_EQUAL (exp, act)

TEST_ASSERT_NOT_EQUAL (exp, act)

TEST_ASSERT_EQUAL_UINT (exp, act)

TEST_ASSERT_EQUAL_UINT8 (exp, act)

TEST_ASSERT_EQUAL_UINT16 (exp, act)

TEST_ASSERT_EQUAL_UINT32 (exp, act)

TEST_ASSERT_EQUAL_UINT64 (exp, act)

Unsigned Integers (of all sizes) in Hexadecimal

TEST_ASSERT_EQUAL_HEX (exp, act)

TEST_ASSERT_EQUAL_HEX8 (exp, act)

TEST_ASSERT_EQUAL_HEX16 (exp, act)

TEST_ASSERT_EQUAL_HEX32 (exp, act)

TEST_ASSERT_EQUAL_HEX64 (exp, act)

Masked and Bit-level Comparisons

TEST_ASSERT_BITS (mask, exp, act)

TEST_ASSERT_BITS_HIGH (mask, act)

TEST_ASSERT_BITS_LOW (mask, act)

TEST_ASSERT_BIT_HIGH (bit, act)

TEST_ASSERT_BIT_LOW (bit, act)

Integer Ranges (of all sizes)

TEST_ASSERT_INT_WITHIN (delta, exp, act)

TEST_ASSERT_INT8_WITHIN (delta, exp, act)

TEST_ASSERT_INT16_WITHIN (delta, exp, act)

TEST_ASSERT_INT32_WITHIN (delta, exp, act)

TEST_ASSERT_INT64_WITHIN (delta, exp, act)

TEST_ASSERT_UINT_WITHIN (delta, exp, act)

TEST_ASSERT_UINT8_WITHIN (delta, exp, act)

TEST_ASSERT_UINT16_WITHIN (delta, exp, act)

TEST_ASSERT_UINT32_WITHIN (delta, exp, act)

TEST_ASSERT_UINT64_WITHIN (delta, exp, act)

TEST_ASSERT_HEX_WITHIN (delta, exp, act)

TEST_ASSERT_HEX8_WITHIN (delta, exp, act)

TEST_ASSERT_HEX16_WITHIN (delta, exp, act)

TEST_ASSERT_HEX32_WITHIN (delta, exp, act)

TEST_ASSERT_HEX64_WITHIN (delta, exp, act)



Structs and Strings

```
TEST_ASSERT_EQUAL_PTR (exp, act)
TEST_ASSERT_EQUAL_STRING (exp, act)
TEST_ASSERT_EQUAL_MEMORY (exp, act, len)
```

Arrays

```
TEST_ASSERT_EQUAL_INT_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_INT8_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_INT16_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_INT32_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_INT64_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_UINT_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_UINT8_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_UINT16_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_UINT32_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_UINT64_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_HEX_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_HEX8_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_HEX16_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_HEX32_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_HEX64_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_PTR_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_STRING_ARRAY (exp, act, elem)
TEST_ASSERT_EQUAL_MEMORY_ARRAY (exp, act, len,
elem)
```

Each Equal (Comparing Arrays to a Single Val)

(these follow the pattern of the arrays, but are named like this)

```
TEST_ASSERT_EACH_EQUAL_INT8 (exp, act, elem)
```

Floating Point (If Enabled)

```
TEST_ASSERT_FLOAT_WITHIN (delta, exp, act)
TEST_ASSERT_EQUAL_FLOAT (exp, act)
TEST_ASSERT_EQUAL_FLOAT_ARRAY (exp, act, elem)
TEST_ASSERT_FLOAT_IS_INF (act)
TEST_ASSERT_FLOAT_IS_NEG_INF (act)
TEST_ASSERT_FLOAT_IS_NAN (act)
TEST_ASSERT_FLOAT_IS_DETERMINATE (act)
TEST_ASSERT_FLOAT_IS_NOT_INF (act)
TEST_ASSERT_FLOAT_IS_NOT_NEG_INF (act)
TEST_ASSERT_FLOAT_IS_NOT_NAN (act)
TEST_ASSERT_FLOAT_IS_NOT_DETERMINATE (act)
```

Double (If Enabled)

```
TEST_ASSERT_DOUBLE_WITHIN (delta, exp, act)
TEST_ASSERT_EQUAL_DOUBLE (exp, act)
TEST_ASSERT_EQUAL_DOUBLE_ARRAY (exp, act, elem)
TEST_ASSERT_DOUBLE_IS_INF (act)
TEST_ASSERT_DOUBLE_IS_NEG_INF (act)
TEST_ASSERT_DOUBLE_IS_NAN (act)
TEST_ASSERT_DOUBLE_IS_DETERMINATE (act)
TEST_ASSERT_DOUBLE_IS_NOT_INF (act)
TEST_ASSERT_DOUBLE_IS_NOT_NEG_INF (act)
TEST_ASSERT_DOUBLE_IS_NOT_NAN (act)
TEST_ASSERT_DOUBLE_IS_NOT_DETERMINATE (act)
```

Todos pueden tener `_MESSAGE` de sufijo, con una string de argumento extra al final

Unity: tests parametrizados

Se pueden dos macros para definir argumentos para tests:

TEST_CASE: Dar valores concretos de argumentos (**sin ; después**)

```
TEST_CASE(0, 1, 2)
```

```
TEST_CASE(2, 1, 500)
```

```
void test_fsmOrigStateIndependentOfNumber(int a, int b, int c) {}
```

TEST_RANGE: Definir rango (inicio, final y paso) (**con ; después**)

```
TEST_RANGE([0,2,1], [0,10,5], [1,5,2]);
```

```
void test_fsmOrigStateIndependentOfNumber(int a, int b, int c) {}
```

Dobles

Se usan para ...

- Evitar dependencias con hardware
- Evitar dependencias entre módulos (aislar CUT*)
 - Creados o no creados
 - Acelerar un módulo colaborador lento (red, BBDD)
- Conseguir situaciones difíciles de provocar
 - Error de memoria, de BBDD, valor extremo de sensor
- Forzar valor a variables volátiles, como el tiempo

* CUT: Code Under Test

Dobles: implementación

Objetivo:

- Que no haya referencias en el código al test
- Que no haya dependencia con el test
- Si la hay, existe la macro `#define TEST`

CMock: creando mocks

```
int funcToMock(int a, int b); //En el fileToMock.h
```

```
//En nuevos ficheros mock_fileToMock.h y mock_fileToMock.c
```

```
void funcToMock_ExpectAndReturn (int a, int b, int returnValue);
```

```
void funcToMock_ExpectAnyArgsAndReturn (int returnValue);
```

```
void funcToMock_IgnoreAndReturn (int returnValue);
```

```
void funcToMock_IgnoreArg_a (void);
```

```
void funcToMock_AddCallback (CMOCK_funcToMock_CALLBACK callback);
```

```
void funcToMock_Stub (CMOCK_funcToMock_CALLBACK callback);
```

```
//int func2ToMock(int* a);
```

```
//void funcToMock_ReturnThruPtr_a(int* returnValsPtr, int elements);
```

Dynamic interface

- En setUp o en test_ se configura la interfaz
- Puede ser una función para ello hecha por vosotros (fake)
- Puede ser una función inventada automáticamente (CMock)
 - Callback, se monitoriza y sustituye
 - Stub, se sustituye pero no se monitoriza

CMock: support/test_fsm.h

```
#ifndef _TEST_FSM_H
#define _TEST_FSM_H

#include "fsm.h"

int is_true(fsm_t* f);
void do_nothing(fsm_t* f);
void* fsm_malloc(size_t s);
void fsm_free(void* p);

#endif
```


CMock: test_fsm_legacy.c

```
#include "mock_test_fsm.h"
```

```
// Register Stub (_Stub) or Callback (_AddCallback) in tests where  
// real malloc and free are needed
```

```
// Callback/Stub for fsm_malloc that calls real malloc  
static void* cb_malloc(size_t s, int n) {  
    return malloc(s);  
}
```

```
// Callback/Stub for fsm_free that calls real free  
static void cb_free(void* p, int n) {  
    return free(p);  
}
```

CMock: test_fsm_legacy.c

```
void test_fsm_validTransitionCallsOutputFunction(void) {
    fsm_trans_t tt[] = {
        {0, is_true, 1, do_nothing},
        {-1, NULL, -1, NULL}
    };
    fsm_malloc_Stub(test_malloc_wrapper);           //Ignore, but do the malloc
    fsm_free_Stub(test_free_wrapper);              //Ignore, but do the free
    is_true_ExpectAnyArgsAndReturn(1);
    do_nothing_ExpectAnyArgs();
    fsm_t *f = (fsm_t*)NULL;
    f = fsm_new(tt);
    fsm_fire(f);

    TEST_ASSERT_NOT_NULL (f);
    fsm_destroy(f);
}
```

Ceedling

- Une Unity, CMock y CException
- Configurable con YAML
- Genera mocks para funciones en fichero “file.h” si se incluye en el test “mock_file.h”
- Posibles comandos
 - `ceedling help`
- Ejecución de tests
 - `ceedling test:all`
- Cobertura de código con tests
 - `ceedling gcov:all utils:gcov`

Ceedling: limitaciones

- Ceedling usa malloc y free internamente, por lo que no se puede hacer mock de ellos
 - Es recomendable usar xxx_malloc y xxx_free en el código, si se puede modificar
 - Se implementan en una función “weak” que llama a las originales
 - Añadiendo las funciones xxx_malloc y xxx_free a un fichero .h para mock, se implementará nuestra función

Ceedling: limitaciones

- No hay limitación en ficheros de test para el mismo módulo (empiezan por test_)
- Si el **módulo no compila**: ceedling falla
 - En la traza de la compilación se arregla
- Si el programa se queda en un **bucle infinito**, ceedling no termina
 - Depurando, se comprueba dónde está, se mata y se arregla
- Si el programa provoca una **excepción de Sistema Operativo**, ceedling falla
 - Se puede depurar hasta ver el punto en el que se provoca y se arregla

Índice

- Contexto y objetivo de las prácticas bloque 3
- Objetivo de la práctica de hoy
- Apoyo teórico: tests
- **Pasos a seguir**

Setup inicial

- El B-042 tiene herramientas preparadas (Linux)
 - Para vuestro ordenador, consultad material en Moodle
- Prepara código:
 - Descarga código MatrixMCU de Moodle
 - Cópialo a una nueva carpeta de tu repositorio
- En VSCode:
 - Abre la carpeta MatrixMCU/lib/fsm
 - Crea una nueva rama en tu repositorio: b3_legacy
 - Realiza un commit y push a la rama creada

VSCode: carpetas

- **.vscode**: configuración
 - Fichero settings.json: general
 - Fichero launch.json: depuración
 - Fichero c_cpp_properties.json: intellisense
- **ceedling**: tests, configuración de tests y resultados
 - Fichero **project.yml**: configuración de ceedling
 - Carpeta **test**: ficheros de test
 - Carpeta support: necesaria para ceedling incluso vacía
 - Fichero test_
 - Carpeta **build**: ficheros temporales (resultados)
- **include** y **src**: cabeceras (.h) y fuente (.c) de fsm
- Fichero **.gitignore**: ficheros ignorados por GIT

VSCode



Explorador de ficheros



Buscador



Gestión de GIT



Debug



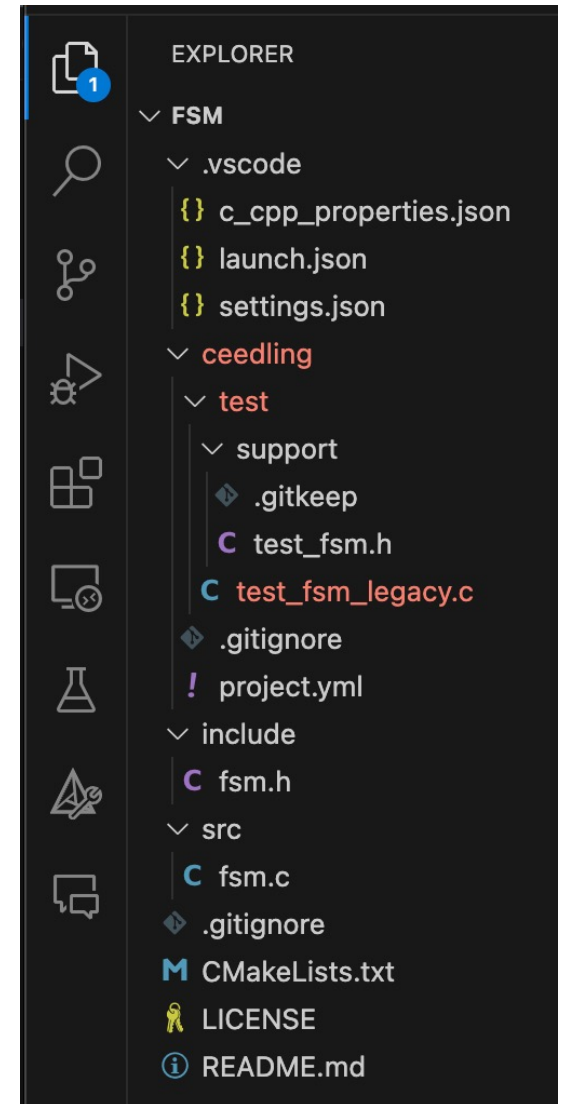
Conexión a remoto (SSH, WSL, Docker)



Extensiones




Test



VSCode

- Selecciona Test

- Aparece fsm_legacy
- Dentro, aparecen todos los tests (sin la palabra test delante)
- En cualquiera de ellos aparecen 3 iconos 
 1. Depuración (todos, se pueden poner puntos de parada)
 2. Ejecución (todos)
 3. Código fuente (del test concreto)
- Una vez ejecutado, salen 3 posibles resultados (ok)



1. Test ignorado
2. Test falla
3. Test pasa



1. No ejecutado
2. Error en ceecling

VSCode: implementar

- Ir al código fuente de los tests
- Preparar contexto
 - Crear e inicializar variables a usar
 - Crear e inicializar variables con resultado esperado
 - Preparar mocks
 - Si queremos comprobar que se llama a una función: Expect
 - Si queremos que una función devuelva valor, aunque puede que no se llame o sea muchas veces: Ignore
 - Si queremos personalizar una función, comprobando: Callback
 - Si queremos personalizar una función, sin comprobar: Stub
- Llamar a la función(es) a probar
- Comprobar estados con ASSERT

VSCode: errores

- Limpiar usando extensión de ceedling:
 - En Menú View > Command-Palette (Ctrl+Shift+P)
 - Ceedling: Clobber
- Limpiar desde terminal
 - Menú terminal > New terminal
 - Ir a la carpeta ceedling y ejecutar el comando de limpieza
ceedling clobber
- Se puede ejecutar en terminal el test (ver posibles errores de compilación)
ceedling test:fsm_legacy

Entrega

- Entrega en el repositorio, en la rama principal del repositorio
 - Ir a la rama main (o principal)
 - Hacer merge con rama b3_legacy
 - Hacer commit si es necesario
 - Hacer push
- Dos días de trabajo en laboratorio
 - 9 de abril
 - 16 de abril
 - Entregado 30 de abril antes de clase

Referencias

Creadores de CMock y Unity:

<http://www.throwtheswitch.org/>

<https://github.com/throwtheswitch>

Tutorial y ejemplos con ceedling

https://embetronicx.com/tutorials/unit_testing/unit-testing-in-c-introduction/

Empresa centrada en desarrollo seguro para empotrados

<https://interrupt.memfault.com/blog/a-modern-c-dev-env#adding-unit-tests>

Test Driven Development for Embedded C,

James W. Grenning. Ed. Pragmatic Bookshelf, 2011