Extracted from:

Test-Driven Development for Embedded C

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

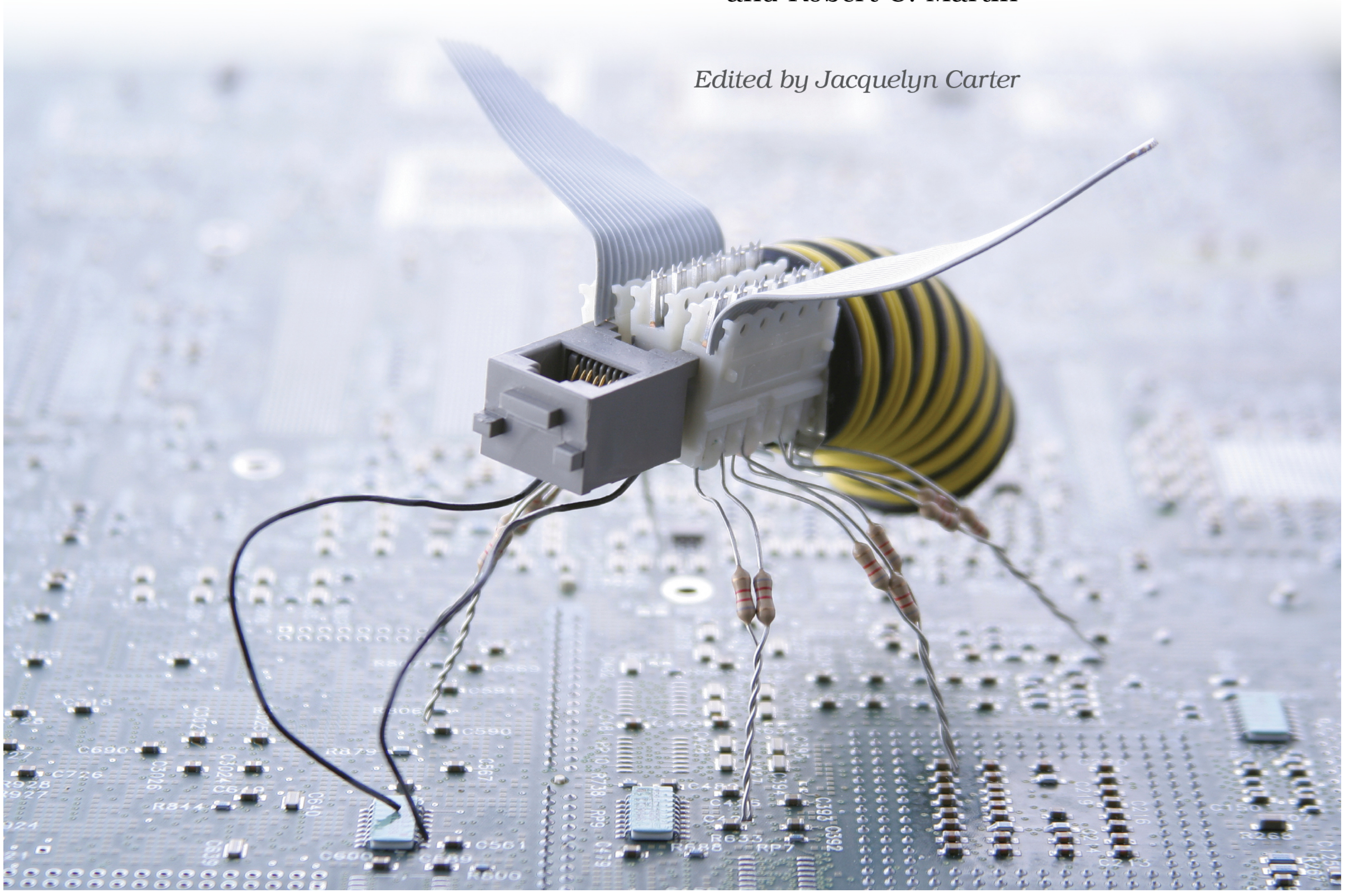
The Pragmatic Programmers

Test-Driven Development for Embedded C

James W. Grenning

Forewords by Jack Ganssle
and Robert C. Martin

Edited by Jacquelyn Carter



Test-Driven Development for Embedded C

James W. Grenning

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The spy is on a covert operation. It intercepts the inputs destined for the production code, later providing it to the test case. As part of its covert mission, it may also feed return results to the client code, getting the CUT to do the test's bidding. Very sneaky indeed.

After setting up the initial LightScheduler files (I use the shell scripts provided with CppUTest to create the initial files), write a test case that helps you envision the roles of the LightControllerSpy and FakeTimeService in the testing of LightScheduler. This tests helps you decide what facilities are needed in the test fixture.

tests/HomeAutomation/LightSchedulerTest.cpp
```
TEST(LightScheduler, ScheduleOnEverydayNotTimeYet)
{
    LightScheduler_ScheduleTurnOn(3, EVERYDAY, 1200);
    FakeTimeService_SetDay(MONDAY);
    FakeTimeService_SetMinute(1199);
    LightScheduler_Wakeup();

    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}
```

Dissecting this test case one line at a time, we see the test schedules the light with id equal to 3 to turn on every day at the 1,200th minute of the day (8 p.m.). The test takes control of the clock, telling the FakeTimeService that it should report that it is Monday at 7:59 p.m. Then the test simulates a callback to LightScheduler_WakeUp(), like the production TimeService will do every minute. Finally, the test checks the expected outcome. The literal constants for EVERYDAY and MONDAY, along with similar constants, will be part of the LightScheduler interface.

There are no scheduled events due, so LightController functions should not be called. The spy is debriefed after the covert mission by checking its secret test-only interface made up of LightControllerSpy_GetLastId() and LightControllerSpy_GetLastState(). LightControllerSpy_GetLastId() returns the id of the light that was controlled or LIGHT_ID_UNKNOWN when there has been no light control. LightControllerSpy_GetLastState() returns LIGHT_OFF, LIGHT_ON, or LIGHT_STATE_UNKNOWN. LIGHT_STATE_UNKNOWN means that the light has not been changed since initialization. If the mission is successful, the debrief should show that no light instructions were given.

TEST(LightScheduler, ScheduleOnEverydayNotTimeYet) looks like a decent test, but it's too big for a first test. There are two test doubles and skeletons of the production code to write. In addition, all the pieces must be wired together. Still, it is good to have this goal articulated. Comment out this test, and save it for later.

A natural first test is one that specifies what should happen during initialization, like this:

tests/HomeAutomation/LightSchedulerTest.cpp

```
TEST(LightScheduler, NoChangeToLightsDuringInitialization)
{
    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}
```

This test can be put together more quickly and keep the feedback coming rapidly. Nothing is needed for LightScheduler, letting us focus on the test double.

What if we have not yet chosen our OS, or the light-controlling hardware is still on the drawing board? Does that have to paralyze us while we make those decisions? No, it does not. We can treat those two areas as DOCs and define interfaces that perfectly meet the needs of the code under test. Interfaces let us break dependencies on unknowns. We develop right up to the unknown, defining what we want from it. The tests are helping to drive the design.

There is a subtle positive side benefit to programming when you don't know all the details—it can lead to a more abstract interface, one that does not get polluted with low-level implementation details. Since it does not reveal the implementation, it allows different potential implementations for different targets.

Write a few tests to drive the new fake, not so much to make sure the fake works—usually there is little that can go wrong—but to document the fake's behavior. You could drive the fake fully from the production code, but by writing tests for it, you also document the fake's behavior.

You non-C++ programmers are going to have to put up with a little bit of C++ . When CppUTest is used to test C code, you have to enclose C function declarations in an extern "C" block. You will see this in the following example. extern "C" tells the compiler to generate calls to the enclosed functions using C calling conventions. If you don't, you will get linker errors reporting that the linker cannot find functions you know are there. The names will look the same to you, but not to the linker.

tests/HomeAutomation/LightControllerSpyTest.cpp

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

extern "C"
{
```

```
#include "LightControllerSpy.h"
}

TEST_GROUP(LightControllerSpy)
{
    void setup()
    {
      LightController_Create();
    }

    void teardown()
    {
       LightController_Destroy();
    }
};

TEST(LightControllerSpy, Create)
{
    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}

TEST(LightControllerSpy, RememberTheLastLightIdControlled)
{
    LightController_On(10);
    LONGS_EQUAL(10, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_ON, LightControllerSpy_GetLastState());
}
```

The spy's header includes the header file of the interface it is replacing. The spy is an implementation of the LightController interface. Including that header is a way to stress that point.

tests/HomeAutomation/LightControllerSpy.h

```
#include "LightController.h"

enum
{
    LIGHT_ID_UNKNOWN = -1, LIGHT_STATE_UNKNOWN = -1,
    LIGHT_OFF = 0, LIGHT_ON = 1
};

int LightControllerSpy_GetLastId(void);
int LightControllerSpy_GetLastState(void);
```

You might wonder why the spy's header file defines the literal values for the light states instead of the production code header. Those values are used for interrogating the spy during test. They would pollute the production code.

The spy's implementation defines a *dead drop*,[2] made from file scope data, where the spy stores intelligence during its important covert operation.

tests/HomeAutomation/LightControllerSpy.c
```
#include "LightControllerSpy.h"

static int lastId;
static int lastState;
```

The create function initializes the spy's dead drop.

tests/HomeAutomation/LightControllerSpy.c
```
void LightController_Create(void)
{
    lastId = LIGHT_ID_UNKNOWN;
    lastState = LIGHT_STATE_UNKNOWN;
}
```

During the spy's mission, critical information is intercepted through the interface of the replaced collaborator.

tests/HomeAutomation/LightControllerSpy.c
```
void LightController_On(int id)
{
    lastId = id;
    lastState = LIGHT_ON;
}
void LightController_Off(int id)
{
    lastId = id;
    lastState = LIGHT_OFF;
}
```

The one being spied on suspects nothing. The intelligence is retrieved from the dead drop through secret accessor functions after the CUT is exercised.

tests/HomeAutomation/LightControllerSpy.c
```
int LightControllerSpy_GetLastId(void)
{
    return lastId;
}
int LightControllerSpy_GetLastState(void)
{
    return lastState;
}
```

As you can see, this spy is quite simple to write. Let's do something similar to take control of the clock in the next section.

2. A secret location where a spy leaves material to be picked up

8.5 Controlling the Clock

Time is usually a big deal in embedded systems. Time, a volatile input, makes testing a challenge. Waiting for timed events in tests makes the tests take too long, longer than they have to be. The bottom line is that the tests have to take over the clock.

Abstracting the clock is important too. Real-time operating systems often define nonstandard time functions, and this can lead to portability problems. If you want code that runs on more than one platform, abstract the clock. The abstraction provides a perfect place to insert your fake clock. In production, use a thin adapter to convert the embedded application's time API to the underlying OS calls. You get a shot of testability with a portability chaser.

needed for the TimeService is kind of the opposite of the LightControllerSpy. The test is not interested in what the CUT passes to the TimeService. But it is interested in controlling what the TimeService returns to the CUT. This is known as an indirect input. Take a look at its tests to reveal its behavior:

tests/HomeAutomation/FakeTimeServiceTest.cpp

```
TEST(FakeTimeService, Create)
{
    Time time;
    TimeService_GetTime(&time);
    LONGS_EQUAL(TIME_UNKNOWN, time.minuteOfDay);
    LONGS_EQUAL(TIME_UNKNOWN, time.dayOfWeek);
}

TEST(FakeTimeService, Set)
{
    Time time;
    FakeTimeService_SetMinute(42);
    FakeTimeService_SetDay(SATURDAY);
    TimeService_GetTime(&time);
    LONGS_EQUAL(42, time.minuteOfDay);
    LONGS_EQUAL(SATURDAY, time.dayOfWeek);
}
```

Now that the test doubles are assembled, let's look again at the test list in Figure 14, *Light Scheduler test list*, on page ?. Like I mentioned earlier, the list is roughly in the order we think we'll implement them. We'll start with the easiest tests first.

8.6 Make It Work for None, Then One

The fully implemented LightScheduler will have to manage a collection of scheduled items. Starting with a test case that involves many scheduled events makes for too much code. A good way to attack and conquer this problem is to start with the cases of *no* scheduled items and then *one* scheduled item, saving the *many* case for later. This is a common approach for engineering collection behavior using TDD.

The *do nothing* tests provides the shortest path to a passing test. All that is needed is the interface definitions for the production code calls. Don't worry that it seems that nothing is being tested. The objective here is to get these boundary tests right. Later when the full implementation is in place, these tests will continue to assure correct behavior of these boundary cases. You will be tempted to put in more code than an empty function body; don't do it. It is a path to untested code.

It looks like there are two areas of scheduler tests. Many of the scheduler tests will be concerned with the time and day matching, while other tests are concerned with the managing of multiple scheduled events. You really don't have to decide the whole test path up front, but I find it helpful to use the *0-1-N pattern* when there is collection behavior. First we handle the *zero* cases where there is nothing scheduled, or no events trigger. Then we handle the *one* event variation that drives the support for all the day and time variations. After the day and time variations are passing their tests, we will shift our focus to add support for the *N* cases where multiple scheduled events are test-driven.

Here's what the *no* scheduled items test looks like:

tests/HomeAutomation/LightSchedulerTest.cpp

```
TEST(LightScheduler, NoScheduleNothingHappens)
{
    FakeTimeService_SetDay(MONDAY);
    FakeTimeService_SetMinute(100);
    LightScheduler_Wakeup();
    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}
```

It may seem that there's no need to set the day and minute on the FakeTimeService; there are no scheduled events, so any time will do. We'll still set the day and minute so they are at least valid values. Only a skeleton of LightScheduler_Wakeup() is needed to satisfy this test.

Here is the initial TEST_GROUP():

tests/HomeAutomation/LightSchedulerTest.cpp

```
TEST_GROUP(LightScheduler)
{
    void setup()
    {
      LightController_Create();
      LightScheduler_Create();
    }

    void teardown()
    {
       LightScheduler_Destroy();
       LightController_Destroy();
    }
};
```

Now we're ready for that test we used to envision the test fixture capabilities.

tests/HomeAutomation/LightSchedulerTest.cpp

```
TEST(LightScheduler, ScheduleOnEverydayNotTimeYet)
{
    LightScheduler_ScheduleTurnOn(3, EVERYDAY, 1200);
    FakeTimeService_SetDay(MONDAY);
    FakeTimeService_SetMinute(1199);
    LightScheduler_Wakeup();

    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}
```

Because no lights are actually controlled during the previous test, only empty implementations are needed for the production code. The test drives the fixture development and the API for scheduling a light. Also, LightScheduler_ScheduleTurnOn() would just be a skeletal implementation like this:

src/HomeAutomation/LightScheduler.c

```
void LightScheduler_ScheduleTurnOn(int id, Day day, int minuteOfDay)
{
}

void LightScheduler_Wakeup(void)
{
}
```

Now for the moment we've been waiting for; let's turn on a light. I chose the EVERYDAY test case because that requires less production code to pass the test in this early stage.

tests/HomeAutomation/LightSchedulerTest.cpp
```
TEST(LightScheduler, ScheduleOnEverydayItsTime)
{
    LightScheduler_ScheduleTurnOn(3, EVERYDAY, 1200);
    FakeTimeService_SetDay(MONDAY);
    FakeTimeService_SetMinute(1200);

    LightScheduler_Wakeup();

    LONGS_EQUAL(3, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_ON, LightControllerSpy_GetLastState());
}
```

Let's think a little about how we'll implement the LightScheduler. We'll need a struct to hold the information about each schedule light control. With that struct we'll create an array to hold the 128 separate scheduled events (128 is the design limit from the requirements). LightScheduler_ScheduleTurnOn(), and similar operations, will use the next unused slot in the array. LightScheduler_Wakeup() will go through the array and see whether any scheduled lights need to be controlled.

Given that we have an idea of how to proceed, let's turn on a light, scheduled for every day at the right time. We need this struct and initialization:

src/HomeAutomation/LightScheduler.c
```
typedef struct
{
    int id;
    int minuteOfDay;
} ScheduledLightEvent;

static ScheduledLightEvent scheduledEvent;

void LightScheduler_Create(void)
{
    scheduledEvent.id = UNUSED;
}
```

You are probably wondering why ScheduledLightEvent only has fields for id and minuteOfDay. Also, where is the array to hold the different scheduled events? Before we get into the whys, take a look at the implementation and see that nothing else is yet needed.

src/HomeAutomation/LightScheduler.c
```
void LightScheduler_ScheduleTurnOn(int id, Day day, int minuteOfDay)
{
    scheduledEvent.id = id;
    scheduledEvent.minuteOfDay = minuteOfDay;
}
```

```
void LightScheduler_Wakeup(void)
{
    Time time;
    TimeService_GetTime(&time);

    if (scheduledEvent.id == UNUSED)
        return;
    if (time.minuteOfDay != scheduledEvent.minuteOfDay)
        return;

    LightController_On(scheduledEvent.id);
}
```

It is tempting to add more fields to ScheduledLightEvent and make scheduledEvent an array. But we don't need it yet. In Debug-Later Programming, we add all the stuff we think we'll need right away. There is virtually no end to "I'm going to need it soon" thinking. So in TDD, we generally only add what is needed by the current tests.

By that reasoning, you might question the need to introduce the ScheduledLightEvent data structure. I chose to introduce it now because it is not much extra syntactical weight to carry, and it will make converting to an array easier. I think ahead, but I act only on some things.

Why not add the array? I'd like to focus on the collection aspect of the scheduler separately and rather not carry around the array index syntax. These are judgment calls.

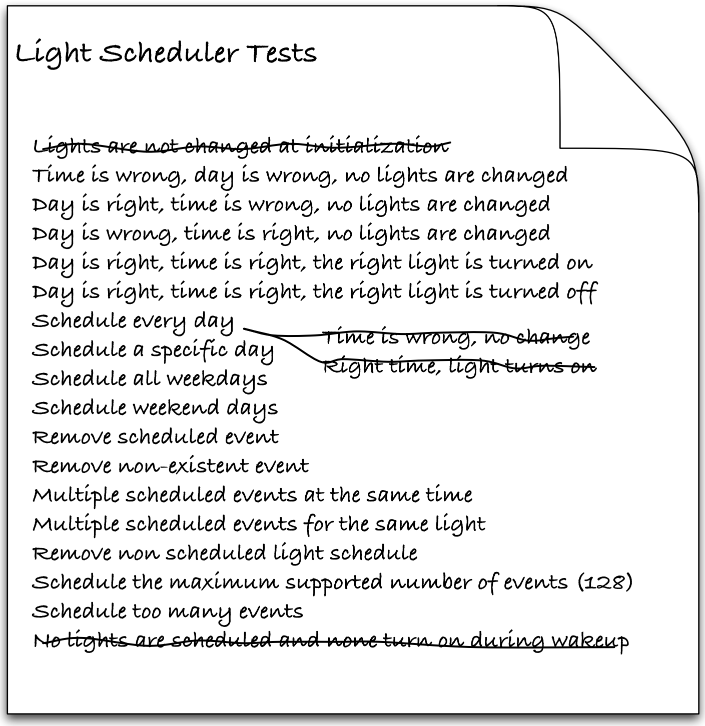
Now back to the code. Notice that the structure definition is in the .c file and not the .h file. We're purposely hiding those details so that the LightScheduler can manage them.

You can see what the test list would look like if I added some of the details of the specific tests written and scratched out the completed tests in Figure 18, *LightSchedulerTestList-revised1*, on page 14.

This test drives us to add the API for scheduling a light to turn off.

tests/HomeAutomation/LightSchedulerTest.cpp
```
TEST(LightScheduler, ScheduleOffEverydayItsTime)
{
    LightScheduler_ScheduleTurnOff(3, EVERYDAY, 1200);
    FakeTimeService_SetDay(MONDAY);
    FakeTimeService_SetMinute(1200);
    LightScheduler_Wakeup();

    LONGS_EQUAL(3, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_OFF, LightControllerSpy_GetLastState());
}
```



Light Scheduler Tests

~~Lights are not changed at initialization~~
Time is wrong, day is wrong, no lights are changed
Day is right, time is wrong, no lights are changed
Day is wrong, time is right, no lights are changed
Day is right, time is right, the right light is turned on
Day is right, time is right, the right light is turned off
Schedule every day
Schedule a specific day
Schedule all weekdays
Schedule weekend days
Remove scheduled event
Remove non-existent event
Multiple scheduled events at the same time
Multiple scheduled events for the same light
Remove non scheduled light schedule
Schedule the maximum supported number of events (128)
Schedule too many events
~~No lights are scheduled and none turn on during wakeup~~

~~Time is wrong, no change~~
~~Right time, light turns on~~

Figure 18— LightSchedulerTestList-revised1

Given our current tests, LightScheduler_Wakeup() can turn on or off a light that is scheduled for EVERYDAY. We are all set to continue adding scheduling scenarios and driving the implementation.

Here is the current state of LightScheduler_Wakeup():

src/HomeAutomation/LightScheduler.c

```
void LightScheduler_Wakeup(void)
{
    Time time;
    TimeService_GetTime(&time);
    if (scheduledEvent.id == UNUSED)
        return;
    if (time.minuteOfDay != scheduledEvent.minuteOfDay)
        return;
    if (scheduledEvent.event == TURN_ON)
         LightController_On(scheduledEvent.id);

    else if (scheduledEvent.event == TURN_OFF)
         LightController_Off(scheduledEvent.id);
}
```

The corresponding LightScheduler_ScheduleTurnOn() looks like the following, with LightScheduler_ScheduleTurnOff() being almost identical.

src/HomeAutomation/LightScheduler.c

```
void LightScheduler_ScheduleTurnOn(int id, Day day, int minuteOfDay)
{
    scheduledEvent.minuteOfDay = minuteOfDay;
    scheduledEvent.event = TURN_ON;
    scheduledEvent.id = id;
}

void LightScheduler_ScheduleTurnOff(int id, Day day, int minuteOfDay)
{
    scheduledEvent.minuteOfDay = minuteOfDay;
    scheduledEvent.event = TURN_OFF;
    scheduledEvent.id = id;
}
```

The LightScheduler's interface has evolved to this point:

include/HomeAutomation/LightScheduler.h

```
#ifndef D_LightScheduler_H
#define D_LightScheduler_H
enum Day {
    NONE=-1, EVERYDAY=10, WEEKDAY, WEEKEND,
    SUNDAY=1, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
};

typedef enum Day Day;

void LightScheduler_Create(void);
void LightScheduler_ScheduleTurnOn(int id, Day day, int minuteOfDay);
void LightScheduler_ScheduleTurnOff(int id, Day day, int minuteOfDay);
void LightScheduler_Wakeup(void);
#endif  /* D_LightScheduler_H */
```