J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_8

8. Testing, Verification, and Test-Driven Development

Jacob Beningo[1]
(1) Linden, MI, USA

My experience from working with several dozen companies worldwide over the last 20 years has been that embedded software teams are not great at testing. Testing embedded software tends to be sporadic and ad hoc. Developers tend to spot-check the code they are writing and, once checked, assume that code is frozen and will never have a defect in it again. While I wish this were the case, the truth is that it is pretty standard for new code to break tested and verified code. In addition, many modern embedded systems have reached a complexity level where manually testing that code works is inefficient and nearly impossible without a team of engineers working around the clock.

The modern embedded software team should look to test automation to improve their software quality. Test automation integrated into a CI/CD pipeline allows developers to elevate the quality of their software, decrease development time, and test their software! There are many levels of testing and several modern processes that can be adopted, like Test-Driven Development (TDD). Unfortunately, like the promises of so many processes, the gap between achieving these results in practice vs. theory can be considerable and, in some cases, elusive.

This chapter will explore testing processes with a primary focus on TDD. First, we will see where it makes sense to use TDD and where developers should avoid it like the plague (or, perhaps, COVID?). Like with any other tool or process, we want to use it where it provides the most benefit. Once we understand TDD, I would recommend you dig deeper into Appendix C and test-drive TDD with the CI/CD example.

Embedded Software Testing Types

Testing should not be performed for the sake of testing. Every test and test type should have an objective to accomplish by creating the test. For example, the primary purpose of most embedded software testing is to verify that the code we have written does what it is supposed to under the conditions we expect the system to operate under. We might write tests designed to verify code behavior through an interface or write test cases to discover defects. We could even write tests to determine how well the system performs.

If you were to go and search the Internet for different types of testing that can be performed on software, you would discover that your list, at a minimum, contains several dozen test types. It is unrealistic for developers to attempt to perform every possible test on their code and system. The second goal of testing is to reduce risk. What risk? The danger is that a user will get hurt using the product. The risk is that the system will ruin the company's reputation. The risk is that the objectives of the system have not been met. We don't write tests or select test types to make management feel better, although perhaps that is the third goal. We trade off the amount and types of testing we perform to minimize the risks associated with our products.

There are several different ways to think about the types of testing that need to be performed. The first is to look at all the possible tests and break them into functional and nonfunctional testing groups. Functional tests are a type of testing that seeks to establish whether each application feature works per the software requirements.[1] Nonfunctional tests are tests designed to test aspects of the software that are not related to functional aspects of the code, such as[2]

- Performance
- Usability
- Reliability
- Scalability

Figure 8-1 shows some typical types of tests that one might encounter.

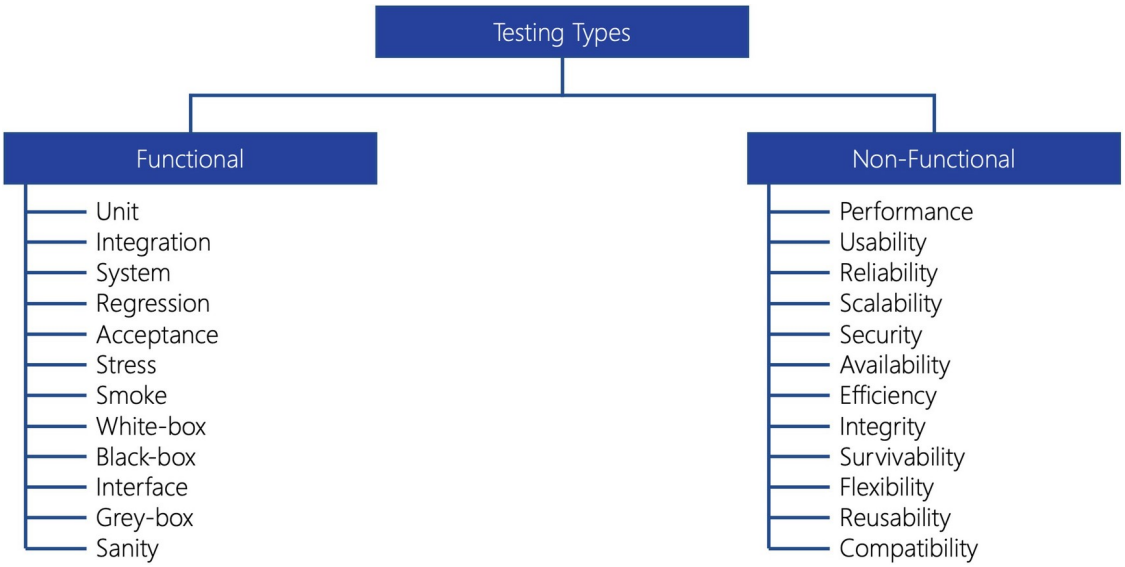

Figure 8-1 The types of testing are broken down into functional and nonfunctional tests

I'm confident that if you take the time to investigate and perform each type of testing on your product, by the end, you'll need your own sanity testing. As you can see, the potential types of tests that can be performed are huge, and I didn't even list them all in Figure 8-1. I got tired and bored and decided it was time to move on! While each of these tests has its place, it isn't required that every test type be performed on every system.

Testing Your Way to Success

With so many different types of testing available, it's more than enough to make any developer and team's head spin. However, all types of tests aren't required to design and build an embedded system successfully. When we start to look at the minimum testing types that are needed to ship a product successfully, you'll discover that there are, in fact, six tests that every team uses which can be seen in Figure 8-2. Let's examine each of these test types and how developers can use them.

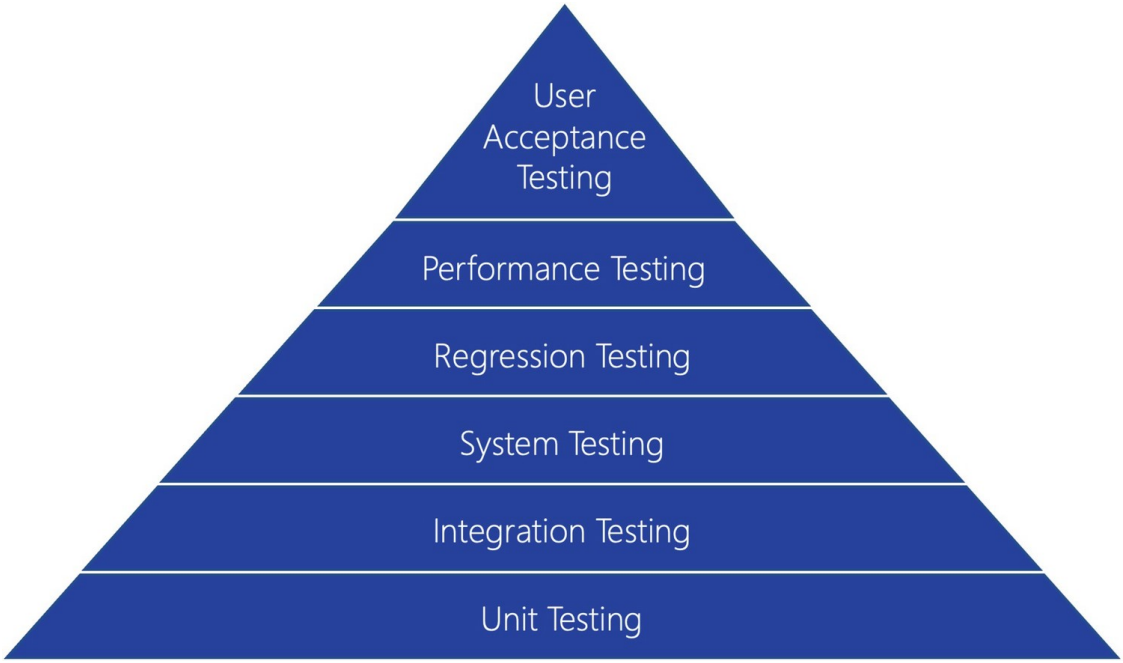

Figure 8-2 The core testing methodologies that every embedded developer should leverage

The first test developers should use, which forms the foundation for testing your way to success, is unit testing. Unit tests are designed to verify that individual units of source code, often individual functions, or objects, are fit for use in the application.[3] Unit tests are foundational because they test the smallest possible units in an application. Given that a (well written) function implements one behavior, a unit test ensures that the function does indeed correctly implement that behavior. You can't build a bridge with shoddy bricks, and you can't build a system with unstable functions. We test every function to ensure we are building our application on a solid foundation.

Integration tests are just a single level above unit tests. Once we have proven that the individual pieces of code work, we can start to integrate the pieces to build the application; undoubtedly, as components interact with each other, there will be defects and unexpected behaviors. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements for the application.[4] Integration testing shows that our building blocks are all working together as expected.

As an embedded system is developed, the end goal is to create specific features that meet the provided system requirements. To ensure those re-

quirements are met, we create system-level tests. System testing is conducted on a complete integrated system to evaluate its compliance with specified requirements.[5] In many cases, you may find that system-level tests are not written in C/C++. Instead, system testing often requires building out hardware systems that can automatically interact with the system and measure how it behaves. Today, many of these applications are written in Python.

So far, the first three tests we've discussed are everything you need to ensure that your code is checked from its lowest levels to the system level. However, we still need several types of tests to ensure that our system meets our clients' needs and that new features are breaking existing code.

Regression tests are automated tests that rerun the functional and nonfunctional tests to ensure that previously developed and tested software still performs after the software has been modified.[6] I can't tell you how many times early in my career I would manually test a feature, see that it worked, and move on to the next feature, only to discover later I had broken a feature I had already tested. Unfortunately, in those days, I was not using automated tests, so it was often weeks or months before I discovered something was broken. I'm sure you can imagine how long it then took to debug. Too long! Regression tests should be performed often and include your unit, integration, and system-level tests.

An essential and often-overlooked test that every developer should perform on their software is performance testing. Embedded developers work on a microcontroller (or low-end general-purpose processor) in resource-constrained environments. Understanding how many clock cycles or milliseconds the code takes to run is important. You may find that you only use about 10% of your processor early in your development cycle. However, as new features are added, the utilization often goes up. Before you know it, unexpected latencies can appear in the system, degrading performance. I highly recommend that developers take the time to measure their system performance before every merge. It'll tell you if you made a big blunder and help you understand how much bandwidth the processor has left.

The last minimum test that developers should make sure they use is user acceptance tests. A user acceptance test is where the team releases their product to their intended audience. User acceptance tests can shed a lot of light on whether the users like the product and how they use it and often discover defects with the product. I always joke that there are two times when a well-tested system is bound to fail: when giving a demo to your boss and releasing the product to your customer. In these situations, Murphy's law runs rampant. If you want to have a quality system, you must also let your users beat the thing up. They will use the system and exploit flaws in your product that you would never have imagined (because you designed and wrote it!).

What Makes a Great Test?

Edsger W. Dijkstra stated that "Testing shows the presence, not the absence of bugs." No matter how much we might try, we can never use testing to show that there are no bugs in our software. There is a limit to what our tests can tell us. Testing helps us to show that under specific circumstances the software behaves as we expect it to. If those conditions change, then all bets are off. Before we understand what makes a great test, we need to understand that tests have their limits and don't guarantee the absence of bugs or defects.

With that said, tests can in fact help us to show that a software system under specific conditions does meet the system requirements and behaves as we expect it to. Part of the problem is identifying what exactly to test. For any specific embedded system, we can define a nearly infinite number of tests. With an infinite number of tests, how do we decide what should be tested? After all, if we can create an infinite number of test cases, we'll undoubtedly bankrupt our company by never being ready to ship a product.

Every industry has their own requirements for how much and what testing should be performed. For example, the requirements for a consumer electronics product like an electronic light switch will have far less requirements than a Boeing 737 that includes safety-critical flight systems.

An initial decision on what needs to be tested can be gleaned from industry standards and even federal laws. However, there are also some general rules of thumb we can follow as well.

First, we should be fully testing our interfaces with unit tests. Unit tests can be written so that they run on hardware and exercise peripherals and drivers. Unit tests can also be run on host machines that test hardware-independent software. When we test our interfaces, we should be asking ourselves several questions such as

- What are the inputs to the interface?
- What are the outputs from the interface?
- Are there errors that are reported or can be tracked?
- What ranges can be inputs and outputs be within?

When I define an interface, for every function I document three to five key features of the function's interface:

- Inputs and their expected ranges
- Outputs and their expected ranges
- Preconditions – What should happen before calling the function
- Postconditions – What should happen after calling the function
- Range of function return values (typically error codes)

As you can see, these five documented features tell any developer exactly how the function can be used and what to expect from it. With this information, I can also design test cases that verify all those inputs, outputs, and postconditions.

Next, testing should also go well beyond our own code and apply to any libraries or third-party software that we use. It's not uncommon for this code to not be well tested. Library interfaces should be carefully examined and then test cases created to test the subset of features that will be used by the application. Again, this is helping to ensure that we are building our application upon a good, quality, foundation. Unless the library is coming a commercial source that has a test harness already available, develop tests for your library code, don't just assume that they work.

There are several characteristics that make up a great test. First, a great test should be limited in scope and test only one thing at a time. If a test fails, we shouldn't have to go and figure out what part of a test failed! Tests are specific and have just a singular purpose. Next, tests need to communicate their intent directly and clearly. Each test should clearly belong to a test group and have clear name associated with it. For example, if I have a communication packet decoding module, I might have a test group called PacketDecoding with a test that verifies the CRC algorithm produces the correct results:

```
TEST(PacketDecoding, crc_correct)
```

Next, tests should also be self-documenting. Since we are writing more code to test the code we already wrote, the last thing we need to do is write more documentation. A good test will have a few comments associated with it, but read very well. For example, if I am examining a test case, I should find that I can just read the code and understand what the test is doing. When I write a test, I need to choose clear variable names and be verbose in how I name things. Where things look complicated, I should refactor the test and add a few comments.

Finally, great tests should also be automated. Manual testing is a time-consuming and expensive process. At first, manual testing may seem like a good idea, but over time, as more and more tests are performed, it becomes an activity that has a diminishing return. Teams are better off putting time in up front to develop an automated test harness that can then report the results as to whether the tests have passed or failed.

Regression Tests to the Rescue

A major problem that often crops up with software development is that new features and code added to a project may break some existing feature. For example, I was working on a project once where we had some code that used a hardware-based CRC generator that used a direct memory access (DMA) controller to transfer data into the CRC calculator. It was a cool little device. A colleague made some changes to a seemingly unrelated area of code, but one that used an adjacent DMA channel. The CRC code was accidentally broken. Thankfully, I had several unit tests in place that, when executed as part of the regression tests, revealed that the CRC tests were no longer passing. We were then able to dig in and identify

what changed recently and found the bug quickly.

If I hadn't created my unit tests and automated them, it might have been weeks or months before we discovered that the CRC code was no longer working. At that point, I wouldn't have known where the bug was and very well could have spent weeks trying to track down the issue. Using regression testing, the issue appeared nearly immediately which made fixing the problem far faster and cheaper.

Regression testing can sound scary to set up, but if you have a good test set up, then regression testing should require no effort on the development team's part. When you set up testing, make sure that you use a test harness that allows you to build test groups that can be executed at any time. Those test groups can then be executed as part of an automated build process or executed prior to a developer committing their code to a Git repository. The important thing is that the tests are executed regularly so that any code changes that inject a bug are discovered quickly.

How to Qualify Testing

An interesting problem that many teams encounter when it comes to testing their embedded software is figuring out how to qualify their testing. How do you know that the number of tests you've created is enough? Let's explore this question.

First, at the function level, you may recall from Chapter 6 that we discussed McCabe's Cyclomatic Complexity. If you think back to that chapter, you'll remember that McCabe's Cyclomatic Complexity tells us the minimum number of test cases required to cover all the branches in a function. Let's look at an example.

Let's say that I have a system that has an onboard heater. The user can request the heater to be on or off. If the request is made to turn the heater on, then a heater timeout timer is reset and calculates the time that the timer should be turned off. The code has two parts: first, a section to manage if a new request has come in to turn the heater on or off and, second, a section to manage if the heater state should be HEATER_ON or HEATER_OFF. Listing 8-1 demonstrates what this function code might look like.

```
HeaterState_t HeaterSm_Run(uint32_t const SystemTimeNow)
{
    // Manage state transition behavior
    if (HeaterState != HeaterStateRequested || UpdateTimer ==
true)
    {
        if (HeaterStateRequested == HEATER_ON)
        {
            HeaterState = HEATER_ON;
            EndTime = SystemTimeNow + HeaterOnTime;
            UpdateTimer = false;
        }
        else
        {
            EndTime = 0;
        }
    }
    else
    {
        // Do Nothing
    }
    // Manage HeaterState
    if(SystemTimeNow >= EndTime)
    {
        HeaterState = HEATER_OFF;
    }
    return HeaterState;
}
```

Listing 8-1 An example heater state machine function that controls the state of a heater

How many test cases do you think it will take to test Listing 8-1? I'll give you a hint; the Cyclomatic Complexity measurements for HeaterSm_Run can be seen in Listing 8-2. From the output, you can see that the Cyclomatic Complexity is five, and there are nine statements in the function. Listing 8-2 is the output from pmccabe, a Cyclomatic Complexity metric analyzer for HeaterSm_Run.

The minimum number of test cases required to test HeaterSm_Run is five. The test cases that we need are

1. Two test cases for the first if/else pair

2. Two test cases for the if/else pair of the HeaterStateRequested conditional
3. One test case to decide if the heater should be turned off

```
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|       |    # Statements in function
|       |        |   First line of function
|       |        |       |   # lines in function
|       |        |       |       |  filename(definition line
number):function
|       |        |       |       |         |
5       5        9       1      29    heater.c(1):
HeaterSm_Run
```

Listing 8-2 The Cyclomatic Complexity results for HeaterSm_Run using the pmccabe application

Cyclomatic Complexity tells us the minimum number of test cases to cover the branches! If you look carefully at the HeaterSm_Run code, you may realize that there are several other tests that we should run such as

- A test case for the “HeaterState != HeaterStateRequested” side of the first if statement
- A test case for the “|| UpdateTimer == true” side of the first if statement
- At least one or maybe two test cases for what happens if “SystemTimeNow + CommandedOnTime” rolls over

At this point, we have five test cases just to cover the branches and then at least another three test cases to cover specific conditions that can occur in the function. The result is that we need at least eight test cases! That’s just for a simple function that manages a state machine that can be in the HEATER_ON or HEATER_OFF state.

From the example, I believe you can also see now why if you have tests that get you 100% test coverage, you might still have bugs in your code. A code coverage tool will analyze whether each branch of code was executed. The number of branches is five, not eight! So, if the first five test cases to cover the branches are covered by the tests, then everything can look like you have fully tested your function. In reality, there are at least

another three test cases which could have a bug hiding in them.

Now that we have a better understanding of testing, what makes a good test, and how we can make sure we are qualifying our tests, let's examine an agile process known as Test-Driven Development. I've found that following the principles and premises of TDD can dramatically improve not just testing, but code quality and the speed at which developers can deliver software.

Introduction to Test-Driven Development

My first introduction to Test-Driven Development for embedded systems was at the Boston Embedded Systems Conference in 2013. James Grenning was teaching a four-hour course based on his popular book ***Test-Driven Development for Embedded C***. (By the way, I highly recommend that every embedded developer and manager read this book.) My first impression was that TDD was a super cool technique that held a lot of promise, but I was highly skeptical about whether it could deliver the benefits that it was promising. TDD was so different from how I was developing software at the time that it felt awkward, and I shelved it for nearly half a decade. Today though, I can't imagine writing embedded software any other way.

Test-Driven Development is a technique for building software incrementally that allows the test cases to drive the production code development.[7] TDD is all about writing small test cases that are automated, fail first, and force developers to write the minimum amount of code required to make the test case pass. TDD is very different from traditional embedded software development because we typically bang out a whole bunch of code and then figure out how to test it (similar to what we did in the last section). The problem with this approach is that there are usually huge gaping holes in our test suites, and we are typically designing our tests in such a way that they may or may not catch a problem.

Test-Driven Development uses a simple, repeating process known as a microcycle to guide developers in writing their code. The TDD microcycle was first introduced in Kent Beck's ***Test-Driven Development***[8] book but also appeared

in James Grenning's book on page 7. A high-level overview of the TDD microcycle can be seen in Figure 8-3.

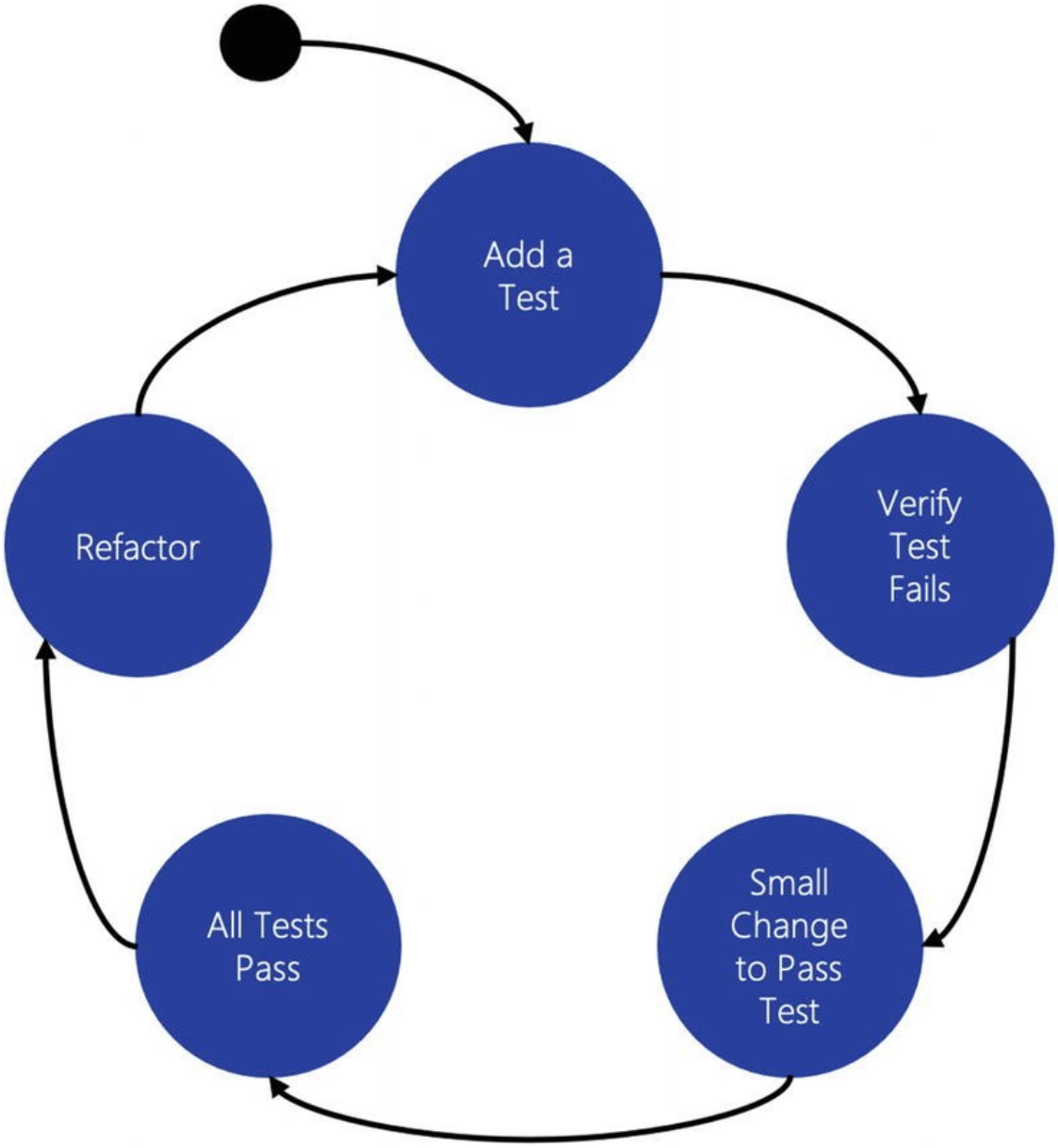

Figure 8-3 The TDD microcycle

The TDD microcycle involves five simple, repeating steps:[9]

1. Add a small test.
2. Run all the tests and see the new one fail. (The test might not even compile.)
3. Make the small change(s) needed to pass the test.
4. Run all the tests and see the new one pass.
5. Refactor to remove duplication and improve the expressiveness of the tests.

As you can see from this simple, repeating process, developers build their software one test case at a time. They can verify that their test case fails first, which means if something happens to the resulting production code, the test case has been verified that it can detect the problem. Developers are continuously running all their tests, so if new code added to the production code breaks something, you know immediately and can fix it. Catching a defect the moment you introduce it can dramatically decrease how much time you spend debugging.

Another exciting aspect of TDD is that developers can test a lot of code on a host environment rather than the intended embedded target. If the application is designed properly with hardware abstractions, the application functions can be executed on the host. Fake inputs and outputs can be generated in the test harness to more fully test the code than trying to duplicate the same behavior in hardware. Leveraging the host environment allows developers to work quickly, avoid long compile and deployment cycles, and continue development even if the hardware is not yet available! Now, developers do have to be careful with potential issues like endianness and other incompatibility issues with the target device. However, development can be dramatically accelerated.

> **Definition**Refactoring is the process of restructuring code while not changing its original functionality.[10]

TDD can now present additional problems, such as dealing with hardware and operating system dependencies. If you've followed the design philosophies discussed in this book, you'll discover that you can avoid quite a few of these issues. Where you can't, you can use tools to get around these issues, such as using

- Abstraction layers
- Test doubles
- Mocks

> **Definition**A mock allows a test case to simulate a specific scenario or sequence of events[11] that minimizes external dependencies.

At this point, I will warn you; I recommend being very careful about how far down the mock rabbit hole you go. TDD can dramatically improve the testability of your code, decrease debug times, and so forth, but don't get

so pulled into the idea that you spend lots of time and money trying to "do TDD perfectly." On the other hand, if you get to an area where it will be a lot of time and effort to write your tests, it might just make sense to write your test cases so that they run on your target device.

TipA test harness can run on a microcontroller. When hardware dependencies become too much to run on a host, run them on your target!

I have found that using TDD improves the software I write. I spend less time debugging. I also have found that it's fun! There is an almost addictive property to it. When you make a test case fail, write your production code, and then see your test case pass, I'm pretty sure the brain gives us a little dopamine hit. We are then encouraged to continue writing the code using a methodology that benefits our team, company, and end users.

Setting Up a Unit Test Harness for TDD

To get started with Test-Driven Development, embedded developers need a test harness that works in the language of their choice. Typically, embedded developers are going to be using C or C++. Several test harnesses are available to developers that range from free to commercial. A few examples of test harnesses that I've worked with include

- Unity
- CppUTest
- Google Test
- Ceedling
- Parasoft C/C++ test

In general, my go-to test harness is CppUTest. Undoubtedly, I am biased, though. Everything I've learned and know about TDD and test harnesses I've learned from my interactions with James Grenning. I mention this only so that you know to try several of your own and see which harness best fits your purposes.

CppUTest is a C/C++-based testing framework used for unit testing and test driving code. In general, CppUTest has been used for testing C and C++ applications. The framework provides developers with a test harness that can execute test cases. CppUTest also offers a set of assertions that can be used to test

assumptions such as

- CHECK_FALSE
- CHECK_EQUAL
- CHECK_COMPARE
- BYTES_EQUAL
- FAIL
- Etc.

If the result is incorrect, the test case is marked as having failed the test.

CppUTest provides a free, open source framework for embedded developers to build unit tests to prove out application code. With a bit of extra work, developers can even run the tests on target if they so desire. In general, I use CppUTest to test my application code that exists above the hardware abstraction layer. CppUTest allows developers to build test groups that are executed as part of the test harness. CppUTest comes with a set of assertions that can be used to verify that a test case has completed successfully or not which is then reported to the developer. CppUTest also includes hooks into gcov which can provide the status of branch coverage as well.

Installing CppUTest

Several different installation methods can be used to set up CppUTest that can be found on the **CppUTest website**.[12] The first is to install it prepackaged on Linux or macOS. (If you want to install on Windows, you'll need to use Cygwin, use a container, or a similar tool.) Alternatively, a developer can clone the **CppUTest git repository**.[13] Cloning the repository doesn't provide any real added benefit unless you are setting up the test harness within a Docker container as part of a CI/CD pipeline. In this case, it is just a simpler way to automatically install CppUTest as part of the Docker image.

I recommend a different approach if you want to get started quickly and experiment a bit. James Grenning has put together a **CppUTest starter project**[14] with everything a developer needs. The starter project includes

a Dockerfile that can be loaded and a simple command to install and configure the environment. If you want to follow along, clone the **CppUTest starter project** to a suitable location on your computer. Once you've done that, you can follow James' instructions in the README.md or follow along with the rest of this section.

Before getting too far, it's essential to make sure that you install Docker on your machine. Mac users can find instructions **here**.[15] Windows users can find the instructions **here**.[16] For Linux users, as always, nothing is easy. The installation process varies by Linux flavor, so you'll have to search a bit to find the method that works for you.

Once Docker is installed and running, a developer can use their terminal application to navigate to the root directory of the CppUTest starter project directory and then run the command:

```
docker-compose run cpputest make all
```

The first time you run the preceding command, it will take several minutes for it to run. After that, the command will download Docker images, clone and install CppUTest, and build the starter project. At this point, you would see something like Figure **8-4** in your terminal. As you can see in the figure, there was a test case failure in tests/MyFirstTest.cpp on line 23 along with an ERROR: 2 message. This means that CppUTest and James' starter project is installed and working correctly.

```
beningo@Jacobs-MacBook-Pro cpputest-starter-project-master % docker-compose run cpputest make all
Creating cpputest-starter-project-master_cpputest_run ... done
Running rename_me_tests
..
tests/MyFirstTest.cpp:23: error: Failure in TEST(MyCode, test1)
        Your test is running! Now delete this line and watch your test pass.

..
Errors (1 failures, 4 tests, 4 ran, 10 checks, 0 ignored, 0 filtered out, 1 ms)

make: *** [/home/cpputest/build/MakefileWorker.mk:458: all] Error 1
ERROR: 2
```

Figure 8-4 The output from successfully building the CppUTest Docker image

Leveraging the Docker Container

The docker-compose run command causes Docker to load the CppUTest container and then make all. Once the command has been executed, it will leave the Docker container. In the previous figure, that is why we get the ERROR: 2. It's returning the error code for the exit status of the Docker container.

It isn't necessary to constantly use the "docker-compose run CppUTest make all" command. A developer can also enter the Docker container and stay there by using the following command:

```
docker-compose run --rm --entry point /bin/bash cpputest
```

By doing this, a developer can simply use the command "make" or "make all." This advantage is that it streamlines the process a bit and removes the ERROR message returned when exiting the Docker image from the original command. So, for example, if I run the Docker command and make, the output from the test harness now looks like what is shown in Figure 8-5.

```
root@e0384cff4bf3:/home/src# make
compiling MyFirstTest.cpp
Linking rename_me_tests
Running rename_me_tests
..
tests/MyFirstTest.cpp:23: error: Failure in TEST(MyCode, test1)
        Your test is running! Now delete this line and watch your test pass.

..
Errors (1 failures, 4 tests, 4 ran, 10 checks, 0 ignored, 0 filtered out, 0 ms)

make: *** [/home/cpputest/build/MakefileWorker.mk:458: all] Error 1
```

Figure 8-5 The output from mounting the Docker image and running the test harness

To exit the Docker container, I need to type exit. This is because I prefer to stay in the Docker container to streamline the process.

Test Driving CppUTest

Now that we have set up the CppUTest starter project, it's easy to go in and start using the test harness. We should remove the initial failing test case before we add any tests of our own. This test case is in /tests/MyFirstTest.cpp. The file can be opened using your favorite text editor. You'll notice from the previous figure that the test failure occurs at line 23. The line contains the following:

```
FAIL("Your test is running! Now delete this line and watch
your test pass.");
```

FAIL is an assertion that is built into CppUTest. So, the first thing to try is commenting out the line and then running the "make" or "make all" command. If you do that, you will see that the test harness now successfully runs without any failed test cases, as shown in Figure 8-6.

```
root@001496277db5:/home/src# make
compiling MyFirstTest.cpp
Linking rename_me_tests
Running rename_me_tests
....
OK (4 tests, 4 ran, 9 checks, 0 ignored, 0 filtered out, 0 ms)
```

Figure 8-6 A successfully installed CppUTest harness that runs without failing test cases

Now you can start building out your unit test cases using the assertions found in the **CppUTest manual**. The developer may decide to remove MyFirstTest.cpp and add their testing modules or start implementing their test cases. It's entirely up to what your end purpose is.

I do have a few tips and recommendations for you to get started with CppUTest before you dive in and start writing your own code. First, I would create separate test modules for each module in my code. For example, if I have a module named heater, I would also have a module named heaterTests. All my heater tests would be located in that one module. I will often start by copying the MyFirstTest.cpp module and then renaming it.

Next, when creating a new module, you'll notice that each test module has a TEST_GROUP. These test groups allow you to name the tests associated with the group and also define common code that should be executed before and after each test in the group. The setup and teardown functions can be very useful in refactoring and simplifying tests so that they are readable.

You will also find that when you define a test, you must specify what group the test is part of and provide a unique name for your test. For example, Listing **8-3** shows an example test for testing the initial state of the heater state machine. The test is part of the HeaterGroup and the test is InitialState. We use the CHECK_EQUAL macro to verify that the return state from HeaterSm_StateGet is equal to HEATER_OFF. We then run the state machine with a time index of zero and check the state again. We can use this test case as an example to build as many test cases as we need.

```
TEST(HeaterGroup, InitialState)
{
  CHECK_EQUAL(HeaterSm_StateGet(), HEATER_OFF);
  HeaterSm_Run(0);
```

```
  CHECK_EQUAL(HeaterSm_StateGet(), HEATER_OFF);
}
```

Listing 8-3 An example test that verifies the initial state of the heater

As a first place to start, I would recommend writing a module that can manage a heater. I would list the requirements for the module as follows:

- The module shall have an interface to set the desired state of the heater: HEATER_ON or HEATER_OFF.
- The module shall have an interface to run the heater state machine that takes the current system time as a parameter.
- If the system is in the HEATER_ON state, it should not be on for more than HeaterOnTime.
- If the heater is on, and the heater is requested to be on again, then the HeaterOnTime will reset and start to count over.

Go ahead and give it a shot. I've provided my solutions to the code and my test cases in Appendix D.

> **Beware**Always make a test case fail first! Once it fails, write only enough code to pass the test. Then "rinse and repeat" until you have produced the entire module. (The tests drive what code you write, not the other way around.)

Final Thoughts

Testing is a critical process in developing modern embedded software. As we have seen in this chapter, there isn't a single test scheme that developers need to run. Instead, there are several different types of tests at various levels of the software stack that developers need to develop tests for. At the lowest levels, unit tests are used to verify individual functions and modules. Unit tests are most effectively written when TDD is leveraged. TDD allows developers to write the test, verify it fails, then write the production code that passes the test. While TDD can appear tedious, it is a very effective and efficient way to develop embedded software.

Once we've developed our various levels of testing, we can integrate those tests to run automatically as part of a CI/CD pipeline. Connecting the

tests to tools like GitLab is nearly trivial. Once integrated, developers have automated regression tests that easily run with each check-in, double-checking and verifying that new code added to the system doesn't break any existing tests.

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to start improving their Embedded DevOps:

- Make a list of the risks associated with the types of products you design and build. What types of testing can you perform to minimize those risks?
- What testing methodologies do you currently use? Are there additional methodologies that need to be implemented?
- What are some of the benefits of using Test-Driven Development? What are some of the issues? Make a pros and cons list and try to decide if TDD could be the right technique for you to use.
- Carefully examine Appendix D. Were you able to write a module and test cases that are similar to the solutions?
  - How many test cases were required?
  - Run a Cyclomatic Complexity scan on your code.
  - How does the complexity value compare to the number of test cases you wrote? Do you need more?
- If you have not already done so, read through Appendix C. There is an example on how to connect a test harness to a CI/CD pipeline.
  - Write your own cpputest.mk file.
  - Test-drive TDD by writing your own custom application module.

Footnotes

1 https://bit.ly/3zPKv6r

2 www.guru99.com/non-functional-testing.html

3 https://en.wikipedia.org/wiki/Unit_testing

4 https://en.wikipedia.org/wiki/Integration_testing

5 https://en.wikipedia.org/wiki/System_testing

6 https://en.wikipedia.org/wiki/Regression_testing

7 *Test-Driven Development for Embedded C*, James Grenning, 2011, page 4.

8 www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530

9 These steps are taken nearly verbatim from *Test-Driven Development for Embedded C*, page 7.

10 www.techtarget.com/searchapparchitecture/definition/refactoring

11 *Test-Driven Development for Embedded C*, James W. Grenning, page 177.

12 https://cpputest.github.io/

13 https://github.com/cpputest/cpputest.github.io

14 https://github.com/jwgrenning/cpputest-starter-project

15 https://docs.docker.com/desktop/mac/install/

16 https://docs.docker.com/desktop/windows/install/
