# Working With Legacy Projects

Yes, Test Driven Development is most helpful when applied to new projects, guiding your design and creating a robust framework. But, it's also useful to ease into an existing project that needs changes… Ceedling and friends are tools built for Unit Testing, of which TDD is just one flavor. While it might be more challenging to write tests for existing code (code that wasn't designed with testing in mind), it is often still worthwhile.

So how do we get there? How to we take an existing project, and start adding a little unit testing magic to it? Let's talk about some of the highlights!

## Updating a project to run Ceedling

The first step is to make your test tools work with your existing project. You are welcome to use Unity directly… if that's your direction, you're going to need to understand enough about how tests are built and managed to operate your own build system. If you have those skills and want to manage that process, awesome. For the rest of us, there's Ceedling.

Let's work with an example. Let's say we have an existing project called `YeOldProject`. It has a directory structure that looks something like this:

- `YeOldProject`
    - `RTOS`
    - `Hardware`
    - `Application`
    - `Config`
    - `Build`

Our project doesn't have any tests yet, because we're just now installing the test tools. So where should those tests go? Where do we even put our project.yml file? Let's cover the three most common options:

### Option 1: The Old School

The first option is the one that Ceedling runs with by default. I assumed that your project file lives in your main project folder, and that you will be piling all of your tests into a single test directory. This option is straightforward and it's easy to apply to an existing project.

If you have a lot of tests, you may also create subdirectories in the Test folder (which may even match the release code's directory structure).

- `YeOldProject`
    - `project.yml`
    - `RTOS`
    - `Hardware`
    - `Application`
    - `Config`
    - `Build`
    - `Test`
        - `Hardware`
        - `Application`
        - `Config`

## Option 2: Tests Are Secondary

The second option is similar. The tests are pushed into the Test directory (or subdirectories of it) again. This time, however, we also push the project yaml file into the test directory. Why would we do that? We're making a statement. We're saying that yes, this project has tests… but it's not the main show. We're pushing Ceedling and all of its mechanisms into the test directory. This option is common when it has been decided that Ceedling is never going to manage release builds… it is also very common in situations where a continuous integration server isn't being used..

- `YeOldProject`
    - `RTOS`
    - `Hardware`
    - `Application`
    - `Config`
    - `Build`
    - `Test`
        - `project.yml`

## Option 3: Modular

The last option is a great structure for new projects, but is a bit more work for existing projects. It requires us to insert an additional layer into all of the release folders. That layer contains `src` and `test` directories. When done well, this make an effective and portable structure where we can add a module to a project (both release and test) by dragging the parent folder to the project folder. When done this way, the project.yml file is almost always in the root directory.

- `YeOldProject`
    - `project.yml`

- `RTOS`
- `Hardware`
  - `Src`
  - `Test`
- `Application`
  - `Src`
  - `Test`
- `Config`
  - `Src`
  - `Test`
- `Build`

Of course there are many other options as well, but we illustrate these in order to give you and idea of the types of things to consider when adding to your project.

In any case, the next step is to add Ceedling to our project. Let's say we went with option 1. We'd open a command prompt and get ourselves to the directory which contains the `YeOldProject` folder (not the `YeOldProject` folder itself… it's parent). Once there, we're going to tell Ceedling that we want to create a new Ceedling project.

```
ceedling new YeOldProject
```

If you've read the `Readme.md` for Ceedling, you might already know that there are a number of options that can be applied at this step. We could add `--docs` to have Ceedling add Unity, CMock, and Ceedling documentation to our project for us. We could add `--local` to have Ceedling install a local copy of itself to our project for safe keeping. We could even ask Ceedling to add a `.gitignore` file for us automatically be adding `--gitignore`. If you're not sure about these options, we recommend leaving them off and running with defaults.

Don't worry about Ceedling overwriting anything in your project that exists already. It's really focused on adding a default `project.yml` file to your project, which is the core requirement for getting Ceedling to recognize it as a project that it can work with. It's unlikely that this `project.yml` file is going to be good enough as installed, so our next task is to edit it.

## Configuring project.yml to Test Your Project

This particular guide isn't really about configuring the project.yml file. Instead, we're just going to point out the things you should think about, and you can refer to the porting guides or the ceedlingpacket for more information.

The focus of getting our tests up and running is on the paths and the toolchain itself. Often, if you're working the a legacy project, the code has been written in a way that doesn't make it portable to other platforms. Many of the tricks we use for abstracting away registers aren't implemented. Instead, the release code has already been designed to directly read or write to those registers.

When this is the case, the best option is to find a compatible simulator for running our tests. We could consider running on the target itself, but usually this is very limiting (we don't have full control of the registers we want to test). We could consider running as a native executable on the host, but most often the registers are tied to specific memory addresses and therefore not portable. So a simulator, if it exists, is our best bet.

The plus side is that we can look at our tool configuration for our release code, and often use almost the exact same compiler and linker calls, settings, etc. If you're using an IDE, usually there is an option to show the compilation process in verbose mode. Then you can capture the calls to the compiler, assembler, and linker, and build the tool definitions based on those.

Again, we can refer to the porting documentation on more detail on getting this done, as well as configuring the simulator.

## API Headers

Perhaps the largest complication of making use of a legacy project, or even legacy modules or subsystems, is that they weren't *designed* for testing. They are sea of `#ifdef`'s and conditional compilation. There are modules which are only included in certain circumstances. CMock's simple parser can't handle such depth of complexity.

The best defense against such things is a central API header, which can be used to mock a subsystem. We discuss this situation during this class… but let's review here.

By funnelling both the release and the test code to API header files, we accomplish a few things:

- We document the parts of the API we are making use of.
- We abstract away the details of the subsystem, creating a black box.
- We have created a mockable interface for any consumers of this API.

The header itself may contain an `#ifdef TEST` which may include the real headers when not testing, but include the mockable API for testing purposes. This is also a good opportunity to simplify types where necessary, or give mockable function declarations in place of inlines or macros.

## Preprocessor

There are times where this is too large or too complex a task. It might make things easier to use your toolchains preprocessor to produce a squeaky-clean C file that can be more easily handled by CMock. Enabling it is simple… getting it to work cleanly can be more complex.

```
:use_test_preprocessor: true
```

Once enabled, a tool must be configured. Most often it's the same compiler as the compiler itself… yet it needs to be configured from scratch using the `:test_preprocessor` tool. The defines are likely going to be similar to the test compiler, but it will need to be configured to output the preprocessed C file.

## A Word about Coverage

It is a common temptation to aim for 100% test coverage when first getting into Unit Testing. While test coverage ***is*** a useful tool for pointing out areas of code that might require more attention, backfilling tests to reach 100% in all cases is rarely worth the time invested. Policies of 100% coverage tend to focus developers on "meeting their quota" instead of investing their attention on writing meaningful tests, so it's an unfortunate side effect that these policies often lead to lower quality tests in the long run.

This is especially true when working with legacy systems. When adding tests to a legacy system, use the tests to explain how the system works. Keep your brain engaged. Write tests that meaningfully show how modules are expected to behave. Test critical pieces of code and API's first. Don't focus on the percentage, but what is being tested and why.

## When to Refactor

Our final note is a reminder. We're using unit testing because we want to improve the overall quality of our code. Unit Testing is rarely used as a validation tool solely. The primary focus when working with legacy systems is to not spend time refactoring the system itself… instead focusing on the creation of good tests. This is an excellent goal, but occasionally this mindset will keep people from doing the right thing.

If a module or subsystem is proving to be incredibly hard to test, it's likely because it could be written in a more maintainable and testable way. Keep in mind that there are times when refactoring a module can actually save you time AND improve your quality.

*Happy Testing!*