Feature-Driven Design Using TDD and Mocks

Embedded Systems Conference Boston (Boston, Massachusetts)
ESC-427, October 2008

Mark VanderVoord	Greg Williams
X-Rite, Inc.	Atomic Object, LLC
http://xrite.com	http://atomicobject.com

ABSTRACT - Top-down design has long been talked about as a very efficient way of developing an embedded system. Although, many attempts at actually implementing a real product in this fashion has led to disaster since code interacting from a high-level down to a low-level interface to hardware registers begs to know how to interface to these hardware layers, thus sending the developer on a downward spiral toward implementing a bottom-up design, unfortunately breaking the original intent.

We present a design methodology, Feature-Driven Design, that focus on the implementation of *features* instead of modules of code that gives way to architectures that evolve naturally and are much easier to understand and maintain. This way of development is fueled by unit test frameworks and automatically generated mocks, so that that designs can materialize from high level ideas and iterative refinement.

The Evolving Embedded World

In the early days, embedded systems were mostly built using 8-bit micros with sub-kB amounts of memory, which largely ended up limiting the complexity of the system being developed. This made it somewhat reasonable to develop of full system even in assembly code and actually get it to work.

Nowadays, 32-bitters are the norm and memory is dirt cheap. So much so that it is being talked about in megabytes and even gigabytes! Our micros have been beefed up in other ways as well. Yeah, we still have serial ports, even though our PCs don't, but other whacky acronyms and names have sprouted up like USB, CAN, TCP/IP, WI-FI, Bluetooth, LCD, CDMA and DVI.

Although we can dream up crazy pie-in-the-sky ideas that take advantage of all these cool new things, it is getting increasingly more difficult to make them come to life and fulfill their full potential. They end up being a hodgepodge of cobbled-together-duct-taped-modules of brittle code that are nearly impossible to maintain.

The problem is that the complexity of these systems, due to complicated protocols and peripherals and with memory being so cheap we should be able to, theoretically, make just about

anything happen. Unfortunately, us lackeys that get to turn these dreams into reality are not commonly the ones who get to do the dreaming. It usually ends up coming to us from these people they say are called “Marketing” with all the tantalizing dollar signs attached so that the executives end up demanding that “Engineering” needs to make it happen by the end of the year. They even tell us they will give us whatever resources we need... except, of course, money. AND we can work as many hours over 50 in a week that we need to in order to make it happen.

Great!....Now what?

Our old ways of developing software just simply cannot keep up with the technology and the mad men (or women) that sign our paychecks.

Bottom-up design and implementation has long been known to result in poorly architected systems in the hands of the average embedded developer.

Top-down design has long been talked about as a very efficient way of developing an embedded system. Although, many attempts at actually implementing a real product in this fashion has led to disaster, since code interacting from a high-level down to a low-level interface to hardware registers begs to know how to interface to these hardware layers, thus sending the developer on a downward spiral toward implementing a bottom-up design, thus breaking the original intent and creating a big pile of spaghetti code.

It is easy to understand the motivation behind, so-called, top-down design. Unfortunately, the way we traditionally use programming languages doesn't naturally fit the direction of thinking that things are conceptualized in.

Therefore, we spend our days with cohorts and sticky notes or a whiteboard; if we're lucky, we get to use UML! Woo-hoo!! The best part is that we don't have to write any code while we're drawing pictures, and we can design the whole system that way... right?!?

Not likely...

At some point we have to write real code, and when we do, we will likely run into real integration issues and it's back to the drawing board.

A New Hope

What if we could create these interfaces without having to implement the underlying code? And test how our high level code works with the interface(s) we are dreaming up? This is where mocks come in. We use *CMock* [1], which is a free and an open-source mock generation tool which generates mocks for C modules using only their header files (or interfaces).

In its simplest form, a mock is a substitute for the *real* implementation of a module. For design verification purposes, we need a mock which we can setup to expect calls with specific

parameter values and return simulated responses. For verification of parameter values, it is most common to use a test framework. *CMock* has been adapted to use the *Unity* [2] unit test framework for verifying the passed parameter values to the methods of the mocked module. Upon an invalid parameter value, Unity will generate a failure which gives details to the developer as to what failed and what was expected. Furthermore, if we fully specify which calls are expected to occur of which mocks and which functions, *CMock* will report errors if anything outside of these expectations occurs.

Although Unity is used for mock call verification, it is basically a tool that allows various assertions to be easily specified and tested. The most basic use of Unity is to compare two values and an error will be reported if they do not match. Unity itself is written entirely in the C language, so it does not require the target compiler to support C++ extensions. This makes it adaptable to most any tool-chain for embedded development.

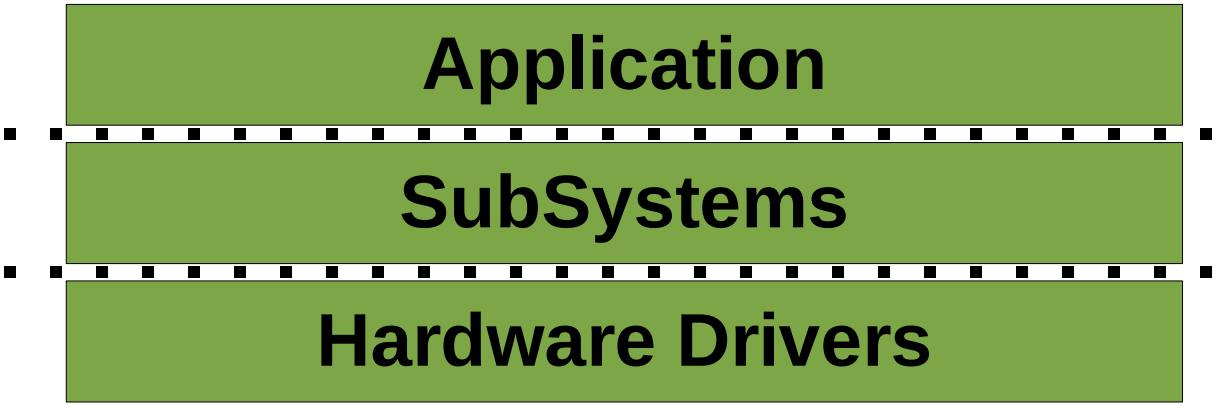
Unit testing focuses on running a single module of code in isolation from the rest of the system. With the power of these testing frameworks at our fingertips, we test how a given module's implementation processes data and interacts with the outside world. Since this is all specified in code we can automate all of our tests so that a entire suite of tests can be run at the press of a button. These low-level tests can be run on actual target hardware, but are most powerful and efficient to be executed in a simulator. That way, the developer doesn't need to have a real hardware prototype on hand at all times.

This sounds like a great deal of extra work to create these suites of tests for every single module in a system. That's where it makes sense to use the power of our gigahertz PCs to carry some of the weight by using higher level languages to handle generating, running and analyzing results of our tests. Numerous choices are available, but we chose to use Ruby [6], an open-source scripting language that has gained much popularity in recent years.

Ruby is used by *Unity* to piece together and run individual test functions into module test suites and run and report on the results. Ruby is a tremendous text processing language, and CMock takes advantage of these capabilities in order to parse function declarations in order to automatically generate the mock modules with identical interfaces of the real modules, or even modules that don't exist.

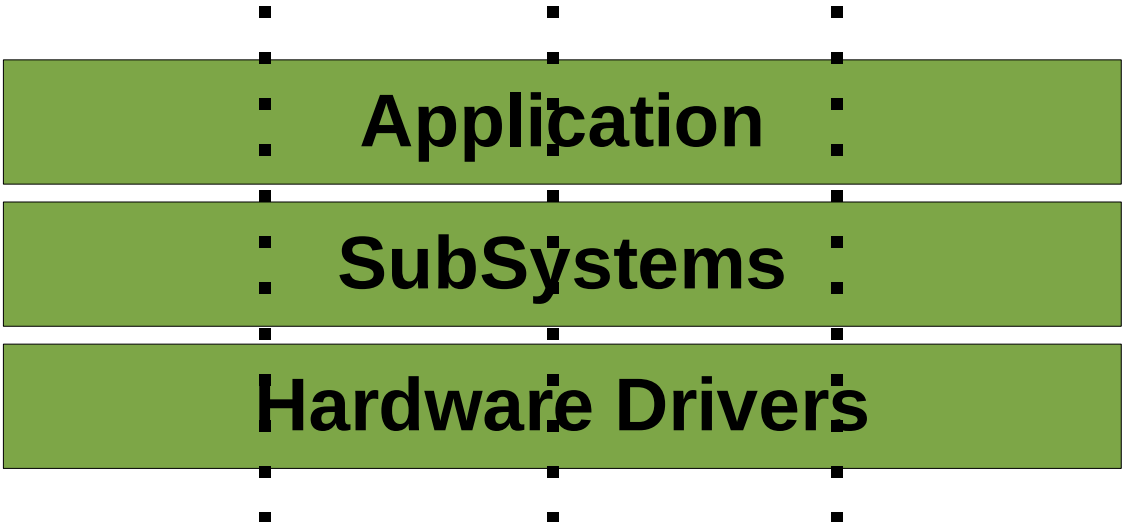
Feature-Driven Design - A New Way of Looking At Things

A common abstraction of a software system is to represent it in layers as in the following diagram:



The idea is that the lowest layer represents the interface to the hardware and the outside world. Each layer provides higher level abstractions and encapsulates the layer below it. The dotted lines show the separation intended by this way of thinking. Many developers try to implement their systems in this fashion. Although, this in itself does not create a well architected system. In fact, many times it results in high levels of coupling and messy dependency relationships that make the system very hard to comprehend and maintain.

Feature-driven development is intended to grow an architecture with a more natural approach. The resultant system can definitely be represented with this type of model, but we focus on implementing *features* instead of *layers*. To illustrate this, we view our task at hand in a different manner. The modified diagram below illustrates our new approach:



As you can see, the lines of separation traverse the system vertically instead of horizontally.

These realms between the dotted lines represent *features*. After all, customers and managers really only care about features and don't really have any interest in layers and architecture. By focussing on implementing *features* instead of *layers*, we can create something of value in a much shorter period of time.

By reorienting tasks to focus on implementing *features* rather than layers and/or modules, value and progress can be demonstrated much earlier and often, thereby keeping project managers, executives and customers at bay. Integration with the rest of the system tends to become a more natural phenomena, since we are modules and layers grow only as new functionality is needed.

This brings up another important benefit of feature-driven design. By keeping focus on the task at hand, we don't add functionality that *might* be needed by some other part of the system at a later time. Research has shown that most functionality that is added to software by speculation never ends up being utilized.

A Feature-Driven Example

Okay, this is all well and good, but how do we really get started?
Let's take a feature:

- **Monitor a voltage level**

In order to read a voltage level, we know that we have to get a sample from the A-to-D converter and provide it to the system at some specified rate. Okay, so instead of wasting our time planning it out, why don't we just write some code?

We will implement the feature by using a design pattern called Model-Conductor-Hardware, or MCH³. This pattern was inspired by the Model-View-Presenter pattern, which is a design pattern that is popular for implementing GUI applications.

In the instance of MVP, the Model, View and Presenter are 3 distinct modules. The Model contains the data and logic that provides an interface for the main application to push data into or extract data from the triad. The View contains an interface for feeding triggers and/or data into the View that causes updates to the GUI and also can contain event handling mechanisms that generate events and/or feed data into the system from the GUI. The Presenter acts as an isolation layer that ties the Model and View together in order to pass events and/or data back and forth through the triad in order makes things happen. In practice, the Presenter itself is stateless. All state data in only contained in the Model, and if needed, in the View.

An implementation method of MVP that has proven very effective for Test-Driven Development and Feature-Driven Development is simply called Presenter-First [4,5]. Which, oddly enough, means that we implement the Presenter first and let it drive the development of the Model and View according to what we deem is required to turn our Feature into reality.

In MCH, the Model plays the same role, but the Presenter is replaced by the Conductor, and the View is replaced by the Hardware module. The Hardware module is basically the point of entry into the hardware driver that implements the low-level details in order to talk to the hardware.

According to the MCH design pattern, there are usually 2 functions in a Conductor, Conductor_Init() and Conductor_Run().

Conductor_Init() function performs any initialization of the MCH triad, and since the Conductor itself is stateless, it basically calls the initialization functions of the Model and/or Hardware if needed. Conductor_Init() can also be used to map callback functions directly to events from either end of the triad.

Conductor_Run() basically acts as a crank that is turned in order to make the triad work if we want the triad to do its stuff only when desired. Under normal conditions, Conductor_Run() is called from another module, which we call Executor.

The Executor can be though of as the big daddy on the MCH triads that is the only one that calls the Executor_Init() and Executor_Run(). It too contains 2 functions, Executor_Init() and Executor_Run(). Executor_Init() is simply used to call the Conductor_Init() functions of each triad. As you can guess, Executor_Run() calls Conductor_Run() of each triad once. To make the system work, the Executor is usually called from the Main() function in the application. First Executor_Init() is called. Then, Executor_Run() is usually called repetitively in a loop while we want the system to operate.

We will implement an ADC triad for the Analog-to-Digital converter. By following the aforementioned pattern. To makes things easy to organize, we prepend each the triad name to each element of the triad. Therefore, let's give birth to ADCConductor_Init() and ADCConductor_Run():

AdcConductor.c

```
void AdcConductor_Init(void)
{
}

void AdcConductor_Run(void)
{
}
```

According to our design pattern, these functions should be called from the Executor. So let's add them:

Executor.c

```
...
#include “AdcConductor.h”

void Executor_Init(void)
{
      ...
      AdcConductor_Init();
}

bool Executor_Run(void)
{
      ...
      AdcConductor_Run();
}
```

Since we are practicing Test-Driven Development, we should be writing the tests first, right? Well, we cheated a little up to this point, but the full demo project contains tests for the Executor. We'll be a bit more diligent from here on out, I promise!

So, back to the AdcConductor...

We want to sample our voltage at a particular rate, and we'll assume something else will handle that timing and set a flag in the ADCModel to let us know when it is time to do a conversion. We need to start an ADC conversion, wait for it to complete, and then grab the conversion data, but since we don't want to waste time waiting in the Conductor, we will get the data from the last conversion and pass it on to the Model. Finally, we will trigger a new conversion, and we're done! In order to ensure we have a sample to start with, we will simply add our first conversion trigger to AdcConductor_Init(). Of course, before the ADC converter can be used, we need to first configure it.

TestAdcConductor.c

```
#include “AdcConductor.h”
#include “MockAdcHardware.h”

void
testInitShouldInitializeAdcConverterAndTriggerFirstConversion(void)
{
      AdcHardware_Init_Expect();
      AdcHardware_StartConversion_Expect();

      AdcConductor_Init();
}
```

In order to get this to compile, we obviously need to add the two expected function declarations to AdcHardware.h. We'll add TEST_IGNORE() macros as reminders to come back later and implement them.

AdcHardware.h

```
void AdcHardware_Init(void);
void AdcHardware_StartConversion(void);
```

AdcHardware.c

```
void AdcHardware_Init(void)
{
}

void AdcHardware_StartConversion(void)
{
}
```

TestAdcHardware.c

```
#include “AdcHardware.h”

void testNeedToImplementAdcHardware_Init(void)
{
     TEST_IGNORE();
}

void testNeedToImplementAdcHardware_StartConversion(void)
{
     TEST_IGNORE();
}
```

Now we have everything in place to run the unit tests on AdcConductor. Of course, the tests fail, because we haven't added any code yet. So let's do it:

AdcConductor.c

```
void AdcConductor_Init(void)
{
     AdcHardware_Init();
     AdcHardware_StartConversion();
}
```

Now, we run our tests, and all is good.

Next, we will implement AdcConductor_Run(). Since we are only doing conversions when needed, we assume that the AdcModel will know when it is time to collect data and start another conversion. Therefore, let's start with the simplest case.

TestAdcConductor.c

```
void testAdcConductor_Run_ShouldNotDoAnythingIfItIsNotTime(void)
{
      AdcModel_DoGetSample_ExpectAndReturn(FALSE);

      AdcConductor_Run();
}
```

AdcModel.h

```
bool AdcModel_DoGetSample(void);
```

AdcModel.c

```
bool AdcModel_DoGetSample(void)
{
      return false;
}
```

TestAdcModel.c

```
static void testNeedToImplement_AdcModel_DoGetSample(void)
{
      TEST_IGNORE();
}
```

Again, we run our tests, and they fail, since AdcModel_DoGetSample() was not called. Now we fix it.

AdcConductor.c

```
void AdcConductor_Run(void)
{
      if (AdcModel_DoGetSample())
      {
      }
}
```

We also should probably check to make sure that our ADC conversion is finished before grabbing the results.

TestAdcConductor.c

```
void testAdcConductor_Run_ShouldNotGetAdcResultIfConversionIncomplete(void)
{
      AdcModel_DoGetSample_ExpectAndReturn(TRUE);
      AdcHardware_GetSampleComplete_ExpectAndReturn(FALSE);

      AdcConductor_Run();
}
```

After running the test to make sure it fails, we then need to declare the new function and add call it.

AdcHardware.h

```
bool AdcHardware_GetSampleComplete(void);
```

AdcHardware.c

```
bool AdcHardware_GetSampleComplete(void)
{
	return false;
}
```

AdcConductor.c

```
void AdcConductor_Run(void)
{
	if (AdcModel_DoGetSample() && AdcHardware_GetSampleComplete())
	{
	}
}
```

Now we can get the conversion, pass it on and start another one.

TestAdcConductor.c

```
void
testAdcConductor_Run_ShouldGetLatestSampleFromAdcAndPassItToModelAndStartNe
wConversionWhenItIsTime(void)
{
	AdcModel_DoGetSample_ExpectAndReturn(TRUE);
	AdcHardware_GetSampleComplete_ExpectAndReturn(TRUE);
	AdcHardware_GetSample_Return(293U);
	AdcModel_ProcessInput_Expect(293U);
	AdcHardware_StartConversion_Expect();

	AdcConductor_Run();
}
```

Then we just need to follow this one through...

AdcConductor.c

```
void AdcConductor_Run(void)
{
      if (AdcModel_DoGetSample() && AdcHardware_GetSampleComplete())
      {
            AdcModel_ProcessInput(AdcHardware_GetSample());
            AdcHardware_StartConversion();
      }
}
```

So, now we've completed AdcConductor. In order to complete the feature, we simply follow the TEST_IGNORE() placeholders and drive toward completion. Let's dig into interfacing to the ADC converter next. Testing the interaction with processor registers is usually fairly trivial when using a simulator. We can simply initialize the registers to a particular value, and verify that the contain the proper value after executing the function under test. We'll tackle AdcHardware_Init(), which resets the ADC peripheral, configures the ADC clock rate, and enables the channel of interest.

TestAdcHardware.c

```
~~void testNeedToImplementAdcHardware_Init(void)~~
~~{~~
~~      TEST_IGNORE();~~
~~}~~

void testAdcHardware_Init_ShouldResetAndSetupForProperChannel(void)
{
      uint32 prescaler = (MASTER_CLOCK/(2*2000000))-1; // 5MHz  clock

      AT91C_BASE_ADC->ADC_CR = 0;
      AT91C_BASE_ADC->ADC_MR = 0;
      AT91C_BASE_ADC->ADC_CHER = 0;

      AdcHardware_Reset();

      TEST_ASSERT_EQUAL(AT91C_ADC_SWRST, AT91C_BASE_ADC->ADC_CR);
      TEST_ASSERT_EQUAL(prescaler,
            (AT91C_BASE_ADC->ADC_MR & AT91C_ADC_PRESCAL) >> 8);
      TEST_ASSERT_EQUAL(0x1 << 4, AT91C_BASE_ADC->ADC_CHER);
}
```

The implementation is straightforward:

AdcHardware.c

```
bool AdcHardware_Init(void)
{
      AT91C_BASE_ADC->ADC_CR = AT91C_ADC_SWRST;
      AT91C_BASE_ADC->ADC_MR = (((uint32)11) << 8) | (((uint32)4) << 16);
      AT91C_BASE_ADC->ADC_CHER = 0x10;
}
```

Alright, so you probably get the idea. Write a test, make sure it fails, modify the function under test until your tests pass. Iterate, iterate, iterate.

If you follow this methodology, you will see that you always have placeholders that will always let you know what you have left to do in order to reach completion. Therefore, you can set aside your your work at any point in time with high confidence that you know where to go next when you get back to it. Likewise, these placeholders can be used by a fellow developer if you need to divide up work for increased bandwidth.

Verbose naming of test cases provide living documentation of the system, thereby eliminating the need to maintain parallel design documentation that will inevitably get out of date. It is seldom necessary to provide further documentation on implementation details, since other developers should be able to get plenty of detail by investigating the test cases that have been defined and met by the actual implementation.

For complete details on implementation, including full unit tests, the entire demo project can be found at http://atomicobject.com/pages/Embedded+Software.

Conclusion

Firmware development can be a daunting task for many of the systems that we are challenged to build these days. The aforementioned development methodologies have proven to scale well from small systems with 8-bit micros with sub-kBytes of RAM and a handful of megabytes of program memory. It has also proved to scale well to very complex projects with many complicated internal and external peripherals and communication interfaces. Adopting these methodologies have increased our level of sanity in our endeavors, and we hope you can reap similar benefits.

Refererences

1. *CMock – Mock Generation Framework for C*
   http://cmock.sourceforge.net/

2. *Unity – Unit Test Framework*
   http://embunity.sourceforge.net/

3. M. Karlesky, G. Williams, *Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design* Patterns. April 2007.
   http://atomicobject.com/files/ESC-413Paper_KarleskyWilliams.pdf

4. M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra. *Presenter First: Organizing Complex GUI Applications for Test-Driven Development.* Agile 2006. Minneapolis, MN. July 2006.

5. D. Crosby and C. Erickson. *Big, Complex, and Tested? Just Say 'When': Software Development Using Presenter First,* Better Software Magazine. February 2007.

6. Ruby Programming Language
   http://www.ruby-lang.org/en/