Evolving into Embedded Development

Matt Fletcher, William Bereza, Mike Karlesky, and Greg Williams
Atomic Object, LLC
{fletcher, bereza, karlesky, williams}@atomicobject.com

Abstract

In late 2005 we had the opportunity to start our first embedded development project. We apply agile practices to a variety of domains from web development to desktop applications to factory floor test equipment. The challenge for this new work was not learning the environment and technology. Our challenge was applying the practices of the agile world to the small and complex world of embedded systems. The hurdles were numerous: we battled the archaic state of many embedded tool sets, the lack of integration with tools like Rake that provide easy automation, and poor support for object oriented design. We've overcome each of these difficulties. This report is about our yearlong experience in introducing our development practices to embedded development.

1. Introduction

Atomic Object has been developing software for six years using Extreme Programming. We test a little, code a little, test a little more, and code a little more. Test-driven development (TDD) is not an optional practice in our office. We're test-infected and we're proud of it.

Two years ago Savant Automation hired us to rewrite the software for two of the half dozen boards controlling the automated guided vehicles they produce. These vehicles drive themselves through warehouses, carrying stock, following preprogrammed paths. Several of us at Atomic Object had been hoping to get into embedded programming. This project was something we jumped at.

TDD is uncommon in the embedded world. A few people like James Grenning [1], Micah Dowty [2], and Nancy Van Schooenderwoert [3] have introduced TDD and other agile practices to embedded development, but a lot of firmware engineers simply are not interested or do not believe agile practices are applicable to embedded development. So aside from simply working on some interesting embedded systems, we wanted to build upon previous work, further innovate on testing practices in the domain, and pro-

mote agile embedded development.

This report is about our yearlong endeavor on the Savant boards. Section 2 recounts our experience creating the firmware for the first system, the 'speed control' board. Section 3 describes both the second system, the 'battery monitor' board, and the changes and improvements we made based on our experience with the speed board. We conclude by discussing the most significant lesson we learned: there is nothing stopping a motivated developer from applying strong agile practices to embedded development.

2. Speed control

The board Savant had us start with was the 'speed control' board. Speed control's role in the system was to take the speed and direction requested by the vehicle's onboard computer and turn it into real motion of the vehicle. To accomplish this, speed control needed to translate the speed, in millimeters per second, into a specific voltage for the vehicle's drives. Further, speed control was required to gently ramp the actual speed to the desired speed—otherwise, the vehicle would not accelerate and decelerate smoothly. Speed control was also part of the vehicle's safety chain, so that if the board detected a problem related to speed (for example, not being able to attain the commanded speed because of an obstruction), then the board could trip the safety mechanism and halt the vehicle. Finally, speed control was responsible for reporting its status to the rest of the system via digital outputs and CAN bus [4] messages. We began working on the speed control board in the fall of 2005.

2.1. Initial development

The microprocessor controlling speed control was Microchip's PIC18F4480, which has 768 bytes of RAM and 16k bytes of flash [5]. At the beginning of the project, we chose to use Microchip's compiler and simulator for developing the firmware. Microchip's compiler constrained us to C and assembly language for development.

Ruby was introduced to the project in the form of a Rake [6] build script. We chose to use Rake instead of make

```
void testShouldCollectResultAverageItAndStartNextConversion()
{
  // mock expectations
  int result = 800;
  ADC_GetResult_ExpectAndReturn(result);
  Averager_AddResult_Expect(result);
  ADC_Start_Expect();

  // run the functional code
  ADCGatherer_CollectData();

  // mock verification is automatically done in tearDown()
}
```

Figure 1. An example unit test for the ADC gatherer

or Microchip's IDE because we've found Rake to be more effective, efficient, and useful than make. We've had a lot of success using Ruby for automation on previous projects; Rake made it easy to integrate Ruby code generation programs into our build process.

2.2. Unit testing

We started developing the firmware unit test-first, just like we do on any other project. Our environment was set up such that tests for each logical grouping of units (such as ADC functions, CAN input functions, CAN output functions, etc) were compiled into their own test executable. At test time, the build environment recompiled the necessary modules and then used Microchip's PIC simulator to execute each test file. Each test output its pass or failure status after executing; test output was collected by the simulator and analyzed to give us the overall red bar or green bar test status.

To further support unit testing, we integrated a Ruby script named Argent [7] into our build process. Argent is much like the C preprocessor in that it reads an input file and looks for specific directives to it. Generally, the directives request that Argent run another Ruby script, but it could be inline Ruby code. The output of the executed Ruby is then dumped back into the source file. We setup our build system to run Argent against each test file as it is compiled. The file directs Argent to scan the code, look for test function signatures, and generate the code in the `main` function to execute each test.

We used the C unit test framework embunit [8] for a little while, but eventually moved away from it because of efficiency. Specifically, because of the limited RAM and ROM in our microprocessor, we could not compile more than just a few unit tests into a single executable. Although we could have overcome this by breaking up our unit test suites into more and more executables, we chose to instead develop a unit test framework targeted directly at our needs. The result is a small and light unit testing environment we call Unity.

Being confined to C in an embedded environment did not stop us from developing the firmware like we would any other application. All of the code was developed test-first; our tests helped us define precisely what each unit was responsible for and what other units we were dependent upon. Whenever the testing for a unit became too difficult (say, because it took too much effort to set up the conditions for the test), we pushed the hard work out into another unit. The result is C code that had a look and feel just like the Java and Ruby code we develop in other projects.

Our unit tests made heavy use of mock objects. We've found mock objects to be extremely useful in our other development projects—they make it easy to separate and test the roles of the various components in the system [9]. In an object-oriented language, we'd normally insert mocks into our objects under test using constructor injection. Since we were using C, we couldn't use constructor injection, so instead we simply manipulated which object files were linked into our test files. In the production code, each component was linked against the real object files, but in test code, the module under test was linked against the mock object files.

Consider a unit responsible for gathering the last analog-to-digital conversion (ADC) result, accumulating it into the running average, and then starting the next analog sampling. In a situation like this, we found performing both the sampling of analog data and maintaining the average was too much responsibility for one unit—setting up an exhaustive set of tests for all of this work was too hard. The solution was to push the work of running the ADC peripheral into one unit and the average calculations into another. Figure 1

is a listing of an example unit test and Figure 2 is the corresponding functional code. Now the tests and the code for both ADC conversion and averaging are small, easy to understand, and thanks to the gatherer, have no knowledge of each other.

```
void ADCGatherer_CollectData(void)
{
  int result = ADC_GetResult();
  Averager_AddResult(result);
  ADC_Start();
}
```

Figure 2. Functional code for the ADC gatherer

2.3. Model-Conductor-Hardware

Throughout the course of creating the speed board firmware, we found ourselves following a common development pattern while implementing each of the firmware's features. For each feature (like ADC sampling, CAN bus input, digital output, etc), we broke the code up into three high-level components: a model implementing the feature's logic, the hardware making things happen, and a conductor in the middle. The conductor's role was to relay events and data between the model and hardware. We call this development pattern Model-Conductor-Hardware [10] (MCH). This doesn't mean there were only three components for every feature—the model and hardware were typically reliant on several helpers.

MCH improved our development experience because it made testing easy. The system logic and hardware control were cleanly separated and controlled by a third party. This allowed us to easily mock out each component; with each component mocked, setting up the conditions for each unit under test was trivial. The previously discussed test in Figure 1 demonstrates how mock objects (in this case, the mock ADC and averager) made our tests easy to understand.

2.4. Speed control experience

Developing speed control was not without its hiccups. Over the course of about nine months, we managed to introduce our basic practices. Mostly.

One of our missteps was not making the most of our mocks. Often times the mocks we created failed to make strong assertions; things like enforcing call count and ordering were not done. Even worse was that sometimes mocks were not even used (usually models in conductor tests). The tests for code that didn't use mocks sometimes became difficult to understand and maintain. Reflecting on this, the problem was that the mocks were generated by hand, not automatically. Creating mocks by hand is tedious and error-prone. Because hand-crafting mocks is so painful, it is tempting to avoid using them to their full potential. This is precisely what happened during the speed board development.

A constant source of frustration for us was our dependence on Microchip's compiler and simulator. The compiler worked in the sense that it generated code, but often times it would silently make unexpected assumptions. This led to some long and aggravating debugging sessions. Even worse was the simulator. First, the simulator was incapable of simulating the microcontroller's peripherals or the timer interrupts. This meant our tests could not be as thorough as we wanted. Second, the simulator could only be invoked through the GUI. So in order to run our automated unit tests, an AutoHotkey [11] script took over control of our system, ran the tests, and collected the results. Since AutoHotkey was controlling the GUI, any outside interruption (like a user shifting focus to another window) would disrupt the script and it would fail to finish. By the end of the project, running the entire test suite took dozens of minutes—which is a long time to wait for test results.

What we really missed out on with the speed control board was automated system tests. Like unit tests, system tests are useful in defining how a particular feature should work from beginning to end. For example, a simple, yet effective system test for speed control would have shown that when a speed command is issued, the board produces the correct output voltage to the drives. System tests push us to develop the minimal set of code needed to make the defined features complete.

The system tests also serve as automated regression tests. Our development speed board was hooked up to a nice test fixture: it had digital switches, analog joysticks, and LEDs that allowed the user to easily control the state of the board. The test fixture was great for demoing the board, but poor for demonstrating that every aspect of the system still worked. Having both unit and system tests as part of the continuous integration system would have alerted us to broken features as soon as possible. Our lack of automated system tests was the biggest regret we had coming away from the speed control board.

On the positive side, speed control proved to us that we could take the practices we've learned in application development and apply them to embedded development. We showed that we can:

1. Develop the software unit test-first

2. Automate building the software and running the tests

3. Use mock objects to support our unit testing

4. Write C code in a highly testable fashion

We also found the problems we solved both complicated and interesting. We got to learn about things like PID controllers, ADC, and CAN bus communications—the stuff that is often already done for you in application development. Speed control was a fun project.

3. Battery monitor

Development of the firmware for the 'battery monitor board' started in July 2006. The battery board's purpose was to report the overall state of charge for the vehicle's battery. Using a combination of two analog inputs from the battery and an internal lookup table, the battery board communicated the current charge level through digital output lines and CAN bus messages. The consumer of this information was the vehicle's main computer, which would make decisions about whether the vehicle should stop and recharge or if it was done charging and should go back to work. The battery board's microcontroller was a Microchip PIC18F2480. The 2480 model has the same specifications as speed control's 4480, but in a different form factor. We planned on introducing some of our new ideas to the battery board, so we were excited to get started.

3.1. System testing

The single most important goal for us, aside from developing good software, was to introduce automated system-test first development to our workflow. As discussed in the previous section, system testing provides the same kind of advantages that unit tests do, but at a higher, more user-oriented level. To make system testing possible, software and hardware to drive and support the tests was necessary.

Several pieces of hardware were required to set up an adequate test fixture. We purchased:

- a PIC firmware programming device

- miniLAB 1008, a USB digital and analog input/output device [12].

- PCAN-USB, a CAN bus communication device [13]

- a USB to serial converter

- a small enclosure and USB hub to tie everything together

Each of these devices was tucked neatly into the enclosure. Our sample battery board was mounted to the top of the enclosure with about a dozen wires running from the board to inside the box. [1] Compared to the speed board's test fixture, with its LEDs and switches, the battery board's fixture looked pretty boring. But it proved to be much more effective.

Software to drive the tests was easy to come by: we decided to use *Systir* [14], a Ruby system testing framework developed in-house. Systir capitalizes on Ruby's expressive syntax to help the user create and write tests using a domain-specific language. The language we created for the battery board tests was focused on simulating specific battery conditions and reading the output from the board, whether it be from the legacy digital lines or modern CAN messages. The miniLAB and PCAN-USB devices were chosen for our test fixture because they could be controlled programmatically via C libraries. We exposed their functionality to our system tests by using Ruby's native C extension facilities.

It took a bit of hardware and software to get our test fixture prepared, but once it was done we were able to create system tests like the one shown in Figure 3. This test shows that the battery board outputs the correct instantaneous charge level (as opposed to average charge level) when given a known battery voltage reading.

```
set_charge_level_output_to_instantaneous

set_battery_voltage_to 4.5
see_charge_level_of 7

set_battery_voltage_to 1.1
see_charge_level_of 2
```

Figure 3. System test for battery level

With the right combination of software and hardware, our system tests were easy to write, cleanly encapsulated the particular features the customer asked for, provided automated regression testing, and enabled our system test-first development approach.

3.2. Better toolchain

A second change we made at the beginning of battery board development was our choice of toolset. This time, we used the PIC version of IAR Systems' Embedded Workbench [15] instead of Microchip's tools. The suite includes a fast, modern compiler, a configurable simulator that can be executed from the command line, and a rich set of libraries. Further, one of the authors had a positive experience previously with IAR's ARM toolkit.

---

[1]Pictures are available at http://spin.atomicobject.com/2006/09/19/hardware-in-support-of-automated-system-testing/

We began to appreciate the new tools more and more each day. The compiler provided simple and easy access to intrinsic hardware functions and gave us meaningful error messages when there was a problem. The linker was intelligent and good at preserving precious code space. The header files for accessing hardware registers had some extremely convenient macros defined to help make the hardware-level code highly readable. But the real winner for us was the command line simulator. No longer did we need to wait minutes for our test suite to run! Substantial time and frustration were saved by the new simulator.

One downside to switching toolchains is that we lost access to Microchip's libraries for driving the on-board peripherals, like the ADC and CAN modules. The libraries would, for example, define simple functions for configuring and reading the ADC device. Despite this, between the Microchip data sheet [5] and our experience with the previous board, we had no trouble recreating the functionality we needed. Another downside was the cost for the new tools. But in the end, we found them to be worth every penny.

3.3. Automatically generated mocks

We spent the next few months developing the battery board system and unit-test first. The system tests instilled a lot more confidence in the team and the new tools helped streamline the development cycle. We also started developing 'Conductor First,' which was a take off of the 'Presenter First' [16] [17] technique used around our office on other projects. Conductor First pushed us to write the tests for the conductor modules first, because those tests defined what we needed, and only what we needed, from the hardware and model modules. Because of the limited scope of the battery board's features, we didn't get much opportunity to refine this practice. After a few model-conductor-hardware groupings were created, our high-level design needs were satisfied; most of the work was done in model and hardware helpers.

The need for a lot of model and hardware helpers exposed one weakness in our system: all of our mocks were still created by hand. As mentioned in Section 2.4, hand generated mocks had led to some poor testing habits in the speed control board. We were a lot more diligent about properly using mocks in the battery board, but creating them by hand was still a real nuisance.

To alleviate this, one of the authors spent a weekend creating a Ruby script to automatically generate a mock for each module in the system. The mock generation script worked by scanning a header file and using regular expressions to match the different types of function signatures. Depending on the type of function identified (such as one that takes no parameters and returns nothing, one that takes no parameters but returns something, etc.), the script would generate a mock version of the function that can be programmed with expectations much like jMock's [18]. All of the functions found in a single header file were turned into a mock that could be included by the tests as needed.

We tied the mock generation script into the build system so that the mocks would be regenerated whenever the system was built. As each test was compiled into its own executable, the build script linked the test to the module under test and each of the mocks. This way, as each test executed, any calls the code under test made to other modules was tracked by the mocks. If an unexpected or incorrect call to another module was made, the mocks would report the failure. This behavior paid off when we switched from the hand generated mocks to the new system: our mocks revealed that some of the code was making calls it shouldn't have been.

The mocks also helped us enforce single responsibility. Whenever the tests for a piece of code got too complicated, we could usually find a good way to break some of the complexity out into another module. Without the automatic mocks, there was always some pain associated with adding another module. But our scripts did all of the hard work by building the mocks for us.

3.4. Battery monitor experience

The battery monitor project had a short lifetime compared to speed control; we worked on the battery monitor for four months as opposed speed control for nine months. We learned more about good embedded development during the battery board project than we did on speed control, despite the shorter timeframe. We successfully introduced a better toolset, system test-driven development, and automatically generated mocks.

One thing that should have been improved were our build scripts. The scripts were not particularly intelligent; because of our setup, whenever a header file changed, almost the entire system would rebuild. This wasn't because of some crazy dependences within the code—it was because our build script was including every mock into every test. Most tests didn't need more than one or two mocks, so including every mock was inefficient. Our project was small, so this problem wasn't too painful, but going forward our build scripts will need to be reworked to include only the required mocks for any given test.

4. Future work and conclusion

The two Savant projects blessed us with systems that had clear inputs and outputs: status information from the vehicle battery and computer went in; directives to other vehicle components went out. This made the role of our system tests obvious. Next time, we'd like to get experience with

another board that doesn't have such easily testable features. We're sure that our unit testing strategy can remain about the same, but introducing system tests to a device without such an easily testable interface will be an interesting challenge.

We are currently involved in a new embedded project with another customer; this time the firmware is driving a color measurement device. The team has, so far, implemented minimal exception creation and handling routines in C, unit test support for their exceptions, and a Ruby extension for communicating with the device via USB. Each has made it easier for the team to develop and test their firmware at both the unit and system level.

Some time ago we were charged with applying agile practices to the embedded domain. Two years later, we've successfully carried over our set of standard practices with great results. Our combination of system and unit test-first development has really helped us attain the fabled 'clean code that works.' We've learned that there is nothing special about the embedded domain preventing us from using the same techniques we'd use during desktop or web application development. Going forward, we consider system tests, unit tests, and mocks an essential part of our the embedded development process. Each of these helped each of us become productive, happy programmers that are excited about embedded development.

5. Acknowledgements

References

[1] J. Grenning, "Extreme programming and embedded software development," Object Mentor," Article, Mar. 2004. [Online]. Available: www.objectmentor.com/resources/articles/EmbeddedXp.pdf

[2] M. Dowty, "Test driven development of embedded systems using existing software test infrastructure," University of Colorado at Boulder, Tech. Rep., Mar. 2004. [Online]. Available: svn.navi.cx/misc/trunk/docs/papers/embedded-test-driven-development.pdf

[3] N. Van Schooenderwoert, "Embedded agile: A case study in numbers," *Dr. Dobb's*, Nov. 2006. [Online]. Available: http://www.ddj.com/dept/architect/193501924

[4] *CAN Specification Version 2.0*, Robert Bosch GmbH, Sept. 1991.

[5] *PIC18F2480/2580/4480/4580 Data Sheet*, Microchip, Apr. 2007.

[6] J. Weirich. Rake – Ruby make. [Online]. Available: http://rake.rubyforge.org

[7] C. J. O'Neill. argent-codegen - a Ruby code generation tool. [Online]. Available: http://rubyforge.org/projects/argent/

[8] Embedded unit. [Online]. Available: https://sourceforge.net/projects/embunit/

[9] S. Freeman, N. Pryce, T. Mackinnon, and J. Walnes, "Mock roles, not objects," in *Proc. OOPSLA 2004*, 2004.

[10] M. Karlesky, W. Bereza, and C. Erickson, "Effective test driven development for embedded software," in *Proc. IEEE Electro/Information Technology Conference*, May 2006.

[11] Autohotkey. [Online]. Available: http://www.autohotkey.com/

[12] miniLAB 1008. Measurement Computing. [Online]. Available: http://www.measurementcomputing.com/

[13] PCAN-USB. PEAK System. [Online]. Available: http://www.peak-system.com/db/gb/pcanusb_gb.html

[14] D. Crosby, K. Fox, and M. Alles. System testing in Ruby. Atomic Object. [Online]. Available: http://atomicobject.com/pages/System+Testing+in+Ruby

[15] C/C++ compilers and debuggers for PIC18. IAR Systems. [Online]. Available: http://www.iar.com/p6058/p6058_eng.php

[16] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, and C. Steinstra, "Presenter First: Organizing complex GUI applications for test-driven development," in *Proc. Agile '06*, July 2006, pp. 276–288.

[17] D. Crosby and C. Erickson, "Big, complex, and tested? Just say 'when': Software development using Presenter First," *Better Software Magazine*, Feb. 2007. [Online]. Available: atomicobject.com/files/BigComplexTested_Feb07.pdf

[18] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. jMock - a lightweight mock object library for Java. [Online]. Available: http://www.jmock.org/