The Embedded Muse 168

Editor: Jack Ganssle (jack@ganssle.com) November 4, 2008

EDITOR: Jack Ganssle, jack@ganssle.com

CONTENTS:
- Editor's Notes
- Book Review
- The Failure of Reuse
- Job Descriptions
- Jobs!
- Joke for the Week
- About The Embedded Muse

****

****

Editor's Notes

Did you know it IS possible to create accurate schedules? Or that most projects consume 50% of the development time in debug and test, and that it's not hard to slash that number drastically? Or that we know how to manage the quantitative relationship between complexity and bugs? Learn this and far more at my Better Firmware Faster class, presented at YOUR facility. See http://www.ganssle.com/classes.htm .

```
switch(city){
case San Jose:
case Phoenix:
case Austin:

Are you in the San Jose, Phoenix or Austin areas? I'll present a public
version of the Better Firmware Faster class in San Jose on December 8,
Phoenix on December 10 and Austin on December 12. Registration and
other info here: http://www.ganssle.com/classes.htm . You'll earn 0.7
Continuing Education Units, learn a lot, and have more than a little
fun. Sign up before November 8 and receive a $50.00 discount.
}
```

Book Review

Duane Mattern received one of the free books in the Great Book Giveaway. He submitted this review of Real-Time UML Workshop for Embedded Systems, by Bruce Powel Douglass. Duane's website is http://www.SampledSystems.com .

Thanks to The Ganssle Group for providing this book for free. The audience for the book is "practicing professional software developers and the computer science major". This book was timely because I recently finished an embedded UML project using iLogix's Rhapsody Version 7.1.1. The book comes with a CDROM that includes a 3-month license to Rhapsody along with Rhapsody UML projects of the examples covered in the book. I already had access to Rhapsody so I did not install the Rhapsody software from the CDROM.

I read the first chapter, which was a review of UML and UML diagrams. I skipped the second chapter on "The Harmony Process", (every organization has its own process). The third chapter on Requirements was rather short, but contains some homework using a traffic light and UAV examples. I was itching to get started, so I opened the first UAV "Coyote" project in Rhapsody. I enabled all of the instances in the configuration and attempted to generate the code. I got several errors: "actor is not a legal name" (Rhapsody didn't like the space in the name), and warnings about "non-behavioral ports". I jumped ahead to the last UAV "Coyote" example. I opened the project and used the Rhapsody Model Checker on the default configuration. I got two error and many warnings. Some of the errors were tool violations, such as "Multiple inheritance from reactive classes is not supported". The error "Multiple timeouts and duplicate triggers from the same state" were caused by missing details. You can generate code for specific packages. For example, I was able to look at the state machine code in the PedestrianLight class in the Roadrunner traffic light example. But, it became obvious that these models were not meant to execute.

The book has value if you are interested in using UML for requirements and system architectural design. There is also a discussion of the detail design of the state machines. You can do these things on a white board. In my opinion, the book missed a opportunity to really shine by taking advantage of the companion CDROM and providing Rhapsody models that were complete to the point of code compilation and execution.

The Failure of Reuse

Many commented on my rant about reuse in the last Muse. Michael Moedt wrote: To re-use something, it really needs to be reliable. Some ways to achieve this:

1. Automated [Unit] tests
This gives a degree of confidence in the module, as well as documentation on how it's used. As well, when shared between projects you get confidence that you're not breaking existing functionality.

2. Good debugging support
This includes things like assertions - if the module to be reused has a bug (or not - if it has bad input), it's good to know as soon as possible.

3. Encapsulated - so that you can use it as the module you want it to be
3b. Good Object-Oriented design (or the analogue in C)

4. Good documentation
I especially like a few good diagrams; forget all the text that won't get read anyways. If people don't know that a module exists and how to use it, and how it works when debugging, they're much less likely to use it.

Also, I totally agree with the quote that something is "not reusable until it has been reused three times". Perhaps then it will have gained some of the attributes above!

John Carter wrote: As one of the "custodians" of a single body of code that is currently shipping 4 distinct products with a fifth on the way I guess I have the right to say something about this.

* Project managers are the poachers of internal software quality. They steal it to buy "time to market" for their single pet project. Thus you need a proactive live "gamekeeper" to guard the whole company’s interests. (That's been my role)

* You need some Big Hard voices that insist on not rewriting, but refactoring.

* Unit Tests really really help. You can't unit test a Big Ball of Mud. http://www.laputan.org/mud/ . Thus Unit Testing forces you to design better systems.

* The notion of "Directory barriers" are a good clean way of defining variation points between products.

A Directory barrier is a directory name that is one of a short list system wide of directory names. A directory barrier must be explicitly listed for inclusion in a product and all other directory names in that list are excluded from the product.

It is called a "barrier" because it stops the build system for a product from even looking for files beyond (an inappropriate) one.

* Cohesion Good, Coupling Bad. Live it.

* Layer architectures are nice. They are simple to describe, simple to reuse, simple to test in layers. But Modern systems are too complex to be designed and delivered in a Layered Architecture.

However, an acyclic digraph architecture has all the simplifying advantages of a layered system and can scale to huge systems.

See http://en.wikipedia.org/wiki/Directed_acyclic_graph if you don't know what an acyclic digraph is. Nested sub and subsubsystems are a reuse disaster. Don't do that. Think rather of branches in a acyclic digraph.

* In a layered or DAG architecture the only excuse for an upward reference (a reference to a symbol implemented in a higher layer) is an asynchronous event originating in a lower layer. Otherwise use parameters and return values.

However, to enable reuse, the declaration and documentation for that symbol should reside in the interface of the lower layer.

* Small scale reuse fails when people don't write reusable software. Doh! If your software is tightly coupled to everything else, untested, undocumented, then Doh! reuse fails.

* Your Nirvana of "pluck a module" SEI style component based software engineering http://www.sei.cmu.edu/ is a bit like saying small scale software reuse obviously fails because we write crappy software. Therefore we will reuse large chunks of crappy software.... at least this approach has no obvious defects. :-)

Somebody somewhere has to work damn hard at getting the quality up and the coupling way down to make that work. (Refactoring the whole code base whilst the rest of the team are pouring 20 man days / day into the repository is a Herculean "clean the Aegean king's stables" type task.)

* He who owns the build script has the reins of the architecture in his hands.

John later wrote more in response to a question about DAG: In a way it has been around for decades...

For example many folk have said for ages... "Your header file should have all the #include's it needs to compile by itself, Don't have cycles in your #include's."

I.e. You shouldn't have a.h #include "b.h", which #include's "c.h" which #include's "a.h"

These notions exist (in perhaps more words) in John Lakos's classic "Large Scale C++ Design" (see chapter on Levelization).

Alas, most embedded programmers have never read that book, thinking it is only for "Big Iron" server folk.

Wakey Wakey Coffee Time! Since that book was written in 1991, Moore's Law has gifted us a factor 128x. The size of our wee handheld embedded systems is now the same size as the Big Iron programs of 1991.

The notion of a layered architecture has been around for ages. People like them.

Why? Because you can visualise them simply, you can build them up in layers, you can test at each layer boundary.

If it is a strict layering, you can replace one layer with another implementation, (Think ppp vs Ethernet under TCP/IP).

You can deploy the lower layers without having to drag along the upper layers if you don't need them.

Of course many folk claim they have a "layered architecture", when all they have is vague areas of a Big Ball of Mud which they think of as a layer.

The criteria for a true layered architecture is that symbols in an upper layer may only be declared in terms of and be implemented using symbols declared in the same layer or lower layer. (Where by symbol I mean, function, class, type, macro, variable, enum, struct, ...)

And a strictly layered architecture is one in which symbols in an upper layer may only be declared in terms of and be implemented using symbols declared in the same layer or the layer directly below.

So now instead of calling a subsystem a "layer". Call it a node. If layer A depends on lower layer B, say there is a directed edge from A to B. Then you have what mathematicians have for centuries called a Directed Graph.

Then insist, like in a layered architecture, that nobody warps space (and minds) by writing code where layer A is magically above AND below layer B. ie. It has no cycles. ie. A Directed Acyclic Graph or DAG.

Hey Presto! You have generalized the notion of a Layered Architecture to something scales to arbitrarily large systems, yet has all the same desirable properties.

How can I claim that it scales? Easy. You can always aggregate a collection of subsystems into packages, so the packages become the new nodes, and the dependencies between packages become the new edges. If you do this right, the new graph is also acyclic.

Do I have an extraordinary proof of this extraordinary claim? Yes.

Go ahead and install any Debian based Linux system. Watch how the apt package manager works. (Ubuntu 8.04 has 25000 available packages with 2400 installed on my box. I love watching it do a distribution upgrade! It gives me cold shivers of delight that something that complex just works!)

Gary Stringham said: I got to the point to where porting a driver for a given block on a new chip only took me an hour to do. My code was highly reusable, but primarily because I had collaborated with the hardware team. The way I did it was to work closely with the hardware teams so that I didn't have to throw out the drivers for each spin of the chip.

I do agree with the comment that it takes about three iterations to get it reusable, but once techniques are learned and understood, it was not that difficult to use them in other hardware drivers. Many of the techniques I developed were adopted and spread throughout the driver writers in our lab.

I think that one key component that is required to develop reusable code is to have the engineer stay in that job for at least three versions. When a different engineer is tasked with porting code to the new platform, they are not familiar with the old and tend to hack

and slash.

Kim Fowler sent an article titled Lessons Learned on Five Large-Scale System Developments by Nat Ozarin, IEEE Instrumentaiton & Measurement Magazine, Feb 2008, where he warns of excessive faith in code reuse. To quote: “New projects are often estimated by assuming that existing software can be plugged in like a piece of hardware. Unfortunately, experience has repeatedly shown that it’s better to follow a simple rule: Don’t assume reuse makes sense unless you do your homework.” Then he goes on to cite a system where reuse went awry when they discovered the reused code was totally inadequate.

Tony Gray submitted this: I'm one of those software engineers who never drank the code reuse Kool-aid, but I do take an occasional sip now and then. In my experience with embedded systems, I've found that re-using module interfaces can be much more productive than trying to reuse the code within the modules. For example, most of the projects I've worked on have had a variety of sensor inputs. After my first two or three projects I settled on a common way of handling sensors. First, I have digital and analog I/O layers that get the raw data from the sensors. Then I have a sensor layer that converts the raw digital and analog values into application values (temperature, pressure, distance, etc.). The code within my analog I/O module or sensor module may change for a particular project, but the function calls between layers (i.e., the architecture) remain the same.

I think the reason this works is because designing and debugging architectural issues is much more difficult than designing and debugging algorithms. Put another way, writing a temperature conversion algorithm from scratch is much easier than creating a code architecture from scratch.

Dave Kellogg wrote: When considering reuse, there are 3 levels of code:
Verbatim is code in the classic sense of reuse: The file is picked up and used with zero changes. This is the ideal, and hopefully includes some documentation to ease the deployment process.

Template code is source (often tables, etc) where the structure is used unchanged, but the contents are customized for each project. Very often the machinery that processes template code is itself verbatim code.

Custom code is just that custom to the project at hand.

Well-designed Template code qualifies as reused code. It avoids re-inventing mechanisms, and supports (to some extent) a fill-in-the-blank approach to reuse. Template coded often raises the level of solution from programming with code to programming with data. An example of template code is an interpolation look-up table: the structure of the table is reused, but the actual data may be custom. The engine that processes the table is usually Verbatim code.

There are two sides to the reuse coin:
- Producing reusable code.
- Consuming reused code.

In a continually improving organization, both faces of reuse must be present and increasing over time. Both types of reuse should be tracked with an appropriate metric. If you don't know how much reuse you have, you cant tell if you are getting better.

To increase reuse consumption, one of the check-list points during a code review should be identifying areas where prior code should have been reused but was not. Because reused code is already debugged, its use can accelerate a project schedule.

Reusable coded must be stored such that it is accessible, easily found, and readily deployable. There must be confidence that code in the reuse library is bug-free.

Reused code can (and should) accumulate over time. It does not all need to be invented all at one time. Hence, each successive project needs to bear only a portion of the burden of producing more reusable code.
To increase reuse production, each project needs to provide a segment
of time (usually at the end of the project)t for reviewing and identifying reusable code and packaging it as such. Some key points are:
- Small pieces of code are more readily reused. So files should be small, single-purposed, and well focused. Of course, this is also good software engineering practice.
- The file is the basis of reuse. If a function is potentially reusable, pull it out into its own file.
- Quality coding practices facilitate easier reuse. E.g., Low coupling and high cohesion; Uniform coding standards.
- Refactoring is a key activity in making code reusable. (see http://en.wikipedia.org/wiki/Refactoring)

As reuse is moved to higher and more abstract levels, having a well-designed architecture with clearly-defined interfaces becomes significantly more important. This is a key step in progressing to a product line type of software production.

Frequent correspondent Paul Bennett contributed: The real trick to getting easily re-usable components is to follow some simple rules:-

* Components perform one function only (part of KISS principle).
* Components interfaces are well defined.
* Components are simply testable and certifiably correct.

On the basis of this I have some components of systems that have been re-used, unchanged, in about 30 projects. They always go through re- certification within each project to ensure the results of the testing have not changed from the previous run. The re-use falls out quite naturally because the developer collects the necessary information about each function during the certification process and, so long as the storage of the code is arranged sensibly, it should be easy to retrieve for the next project that requires that specific function.

The one area that is not usually easy for re-use is the HMI end of things as that can be quite variable. Dealing with the lower levels of the system should be quite straight forward and be the ripest for re-use of code as developers are always slower to change the basic underlying hardware between projects.

Job Descriptions

Harold Teague had asked about job descriptions. Mike Barr had this to say: In the previous Muse, one of your readers asked for detailed job descriptions for firmware architect, firmware engineer, and firmware integration/systems engineer. That third title sounds like it is probably specific to his employer. The others are more generalizable--and related; to my mind, a firmware architect is simply a firmware engineer with sufficient years behind her to have prior first-hand experience with each of the range of design alternatives.

Although embedded software is deployed across a wide range of industries--including verticals as diverse in needs as medical devices and consumer electronics--the required knowledge, skills, and abilities are quite similar. Indeed the most knowledgeable embedded software developers find their skills transferable between industries and often move between them through their careers.

Here's how I would describe requirements when looking for a firmware engineer with ten or more years of on-the-job experience.

Formal Education
* Formal education with degree in electrical engineering, computer engineering, or

computer science.
* Ideally, an undergraduate degree in EE and a master's degree in CS or CE.

Practical Skills
* Knows how to use an oscilloscope, multimeter, logic analyzer, and other lab equipment to observe the proper functioning of software.
* Well acquainted with several version control systems.
* Comfortable reading a datasheet for a new processor or peripheral.
* An expert C programmer, well aware of its compiler-defined corner cases and too-rarely-used features such as bitfields and the volatile keyword.
* Knowledgeable of at least one major RTOS API (e.g., VxWorks, uC/OS- II, etc.).

Demonstrated Capabilities
* Able to independently develop and test interrupt-driven drivers from scratch given only limited (and sometimes erroneous) documentation of the specific peripheral and target board.
* Able to design and implement multithreaded programs using an RTOS, without resorting to the use of global variables.
* Able to ensure that critical deadlines will always be met through appropriate hardware/software partitioning, use of interrupts, and prioritization of tasks.
* Able to design and implement embedded software completely without an RTOS as appropriate.

Gary Stringham commented: While you are thinking about job roles within the firmware domain, I would suggest additional roles and responsibilities outside the firmware domain, that is, collaboration with your corresponding partners on the hardware side.

- Collaborate with the hardware team.
- Review and approve hardware design specifications.
- Be an ambassador from the firmware team to the hardware team.

I believe that one reason I was a successful firmware engineer was because of my frequent and regular collaboration with the hardware team.

Jobs!

Let me know if you're hiring firmware or embedded designers. No recruiters please, and I reserve the right to edit ads to fit the format and intents of this newsletter. Please keep it to 100 words.

Despite the downturn, embedded systems consultancy Netrino (http://www.netrino.com) is still growing fast. The company currently has open positions for embedded software developers as well as digital and analog electronics designers in many parts of the U.S. The company's mix of development, consulting, and training work will entertain your brain and makes for a collegial learning environment. If you may be interested, please send your current resume to careers@netrino.com.

Joke for the Week

Phil Ouellette responded to Guillermo Zepeda-Lopez's ideas about making software lighter: Additional ideas to save weight:

- Convert all variables to float type

- Use only null or void pointers (what could weigh less than nothing)

- Use only virtual functions, constructors and destructors (as apposed to real ones that must weigh something).

Not to be outdone, Tony Klein sent: He missed one thing we do to reduce code size, which would also have the side effect of reducing weight: we just use a smaller font. Say about 4 point. Makes the code really small.

Finally, John Johnson contributed this: I do not want take anything away from Guillermo's humorous contribution but the intent of the effort seems to follow the guidance of Colin Chapman, founder of Lotus Engineering, who urged his designers to "Simplify, and add lightness". Lotus Engineering, of course, created the famed Lotus sports car.

References to Colin Chapman and Lotus were found under http://en.wikipedia.org/wiki/KISS_principle where I also noted the following:

Instruction creep and function creep are examples of failure to follow the KISS principle in software development. This is known as "Creeping Featurism".

About The Embedded Muse

The Embedded Muse is an occasional newsletter sent via email by Jack Ganssle. Send complaints, comments, and contributions to him at jack@ganssle.com.

The Embedded Muse is supported by The Ganssle Group, whose mission is to help embedded folks get better products to market faster. We offer seminars at your site offering hard-hitting ideas - and action - you can take now to ***improve firmware quality and decrease development time***. Contact us at info@ganssle.com for more information.