Agile in the Embedded World

Stephen J. Mellor

StephenMellor@StephenMellor.com

www.StephenMellor.com

Agile practices can be applied to embedded systems, but the nature of the embedded world means that some practices require adjustment. We cover first why these adjustments are necessary and what those practices are. We then examine the practices in detail and the implications of these revised practices.

1. The Agile Alliance

To kick off the new millennium, Kent Beck wrote *eXtreme Programming eXplained*, [1] with all the appropriate capitalization designed to attract attention. This book and the practices behind it have had, and continue to have, a significant impact on the industry, but the book has also caused an extraordinary degree of vitriol, even bile. The reasons appear to include a focus on writing programs, not "analysis" or "design" (a.k.a. modeling); a disdain for documentation as such; and the Communist notion of working only forty hours a week.

XP is not the only "lightweight" method: others include Cockburn's Crystal Family (now merged with Highsmith's Adaptive Systems Development), SCRUM, and DSDM. In late 2000, Robert Martin from Object Mentor proposed that the various proponents of the lightweight methods meet to form a consortium that would promote our common ideas. This proposal came to fruition in February 2001 when 16 lightweight method proponents and I met together to form the Agile Alliance ("extreme" was viewed as too, well, extreme, and "lightweight" was considered too, sorry, lightweight.) During the meeting, a manifesto was born and a set of principles declared.

I introduced myself at this first meeting as "a spy" because my own interest has always been executable models: the notion that a model can be fully precise and detailed enough to be executed. Consequently, the idea that code is the only product of interest and that models are superfluous is strangely disturbing. Certainly, we should focus our attention on the final product, but that product can surely be an executable model—and that can be turned directly into code.

In the event, we agreed—after much discussion—that we could use the word "software" rather than "code." This would encompass executable models without diminishing the emphasis on execution. Consequently, I was able to become a signatory to the Agile manifesto, which can be found in full on the Agile Alliance web site. See [2].

While the principles that sprang forth from the Agile Alliance have sufficient merit for me to be a signatory to the Manifesto, some do not take into account the realities of real-time and embedded systems development. This paper will identify those principles in various agile methods that require some refinement in the context of real-time and embedded systems.

Copyright 2009-2013 © Stephen J. Mellor

2. Multiple Layers of Abstraction

It is traditional at this point to roll out some complex and convoluted definition of an embedded system, usually of limited value and relevance. However, what concerns us here is less what a real-time or embedded system *is* than the characteristics commonly found in embedded systems that cause difficulties in applying agile principles "out of the box."

It is implicit in much of the material written about "agile" that developers can build a complete, executable slice of a system, such as ComputeSalesTax, CreateInvoice, CallElevator and the like, and then deliver it. (Henceforth, I shall refer to such a slice as a "story.") It is also implicit that each story has the same size estimate whenever it is delivered. [1] But this is simply not true for many embedded systems. Before we can deliver the first story, we must build up necessary infrastructure such as—at a minimum—a way to communicate with the hardware.

As an example, let's say we decide to deliver the story OpenElevatorDoor. To make this happen, we need some application logic, perhaps a state machine that responds to commands, timers and signals such as someone standing in the doorway. But where do those commands come from? How are timers supplied? How does the mechanism that recognizes some fool standing in the doorway turn into a signal?

In an embedded system, delivering that first story involves a detour while we build this infrastructure ourselves. (Clearly, if we can buy or borrow that infrastructure from elsewhere then there are no additional issues to consider. We can simply deliver the story as demanded.) In short, one can only deliver a customer-centered vertical slice through the system when the infrastructure—the bottom of the slice—already exists (or can quickly be cobbled together.)

When building this infrastructure who is the Customer? Certainly, we know that we must report the position of the elevator in the shaft, but how does the "customer" want this to be presented? Should we report the position of the elevator in units of distance or floor number? How does our decision affect what the application must do, and how can we find someone who can make this decision? In short, where is the top of the slice? Who is the "Customer"?

Figure 1 shows the multiple layers of abstraction using a *domain chart*, which lays out the subject matters in the system and the relationships between them. Each oval represents a domain: a subject matter populated by a set of conceptual entities. For example, the Elevator domain will have conceptual entities (perhaps implemented as classes) Cabin, Door, Shaft and so on. The Device I/O domain may contain conceptual entities such as Digital Input Signal, Analog Output Signal etc.

---

[1] I chose the word "implicit" rather than "assumed" because I don't believe any one sat down and made any explicit simplifying assumptions. Rather, the case when you *cannot* deliver a complete, executable vertical slice directly is not considered.

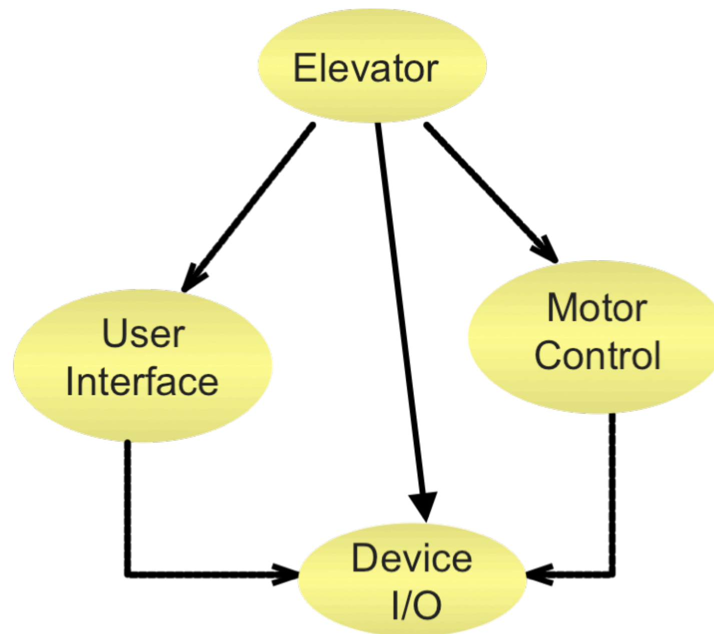

Figure 1: A domain chart for the Elevator System

The arrows represent a particular type of dependency. An arrow from one domain (the client) to another (the server) indicates that the client make certain assumptions about the services the service domain provides. In turn, the server domain has requirements placed upon it by the client domain by virtue of the assumptions. [2]

The domain chart helps us visualize the layers of abstraction in the system. If the application domain is the only domain that requires construction (as is often the case for IT systems), then we can build the system forthwith. But if we need to build some infrastructure, the domain chart helps us understand some required refinements starting with an understanding of who the customer is.

3. The Elusive Customer

Since the phrase "Customer on Site" was coined, there has been much discussion about what, exactly, constitutes a "Customer." In the end, there are three essential roles. The first is the end user. In the Elevator system that user is the passenger. What meaningful information could a passenger provide other than "Beam me up"? Fat lot of good it would do to have the passenger on site! In some cases, the customer can even be a hindrance. Who knew that a requirement for a cell phone was a 3 megapixel camera? Probably not the end user—at least not until s/he had seen the feature and decided it was useful. It is for this reason that the marketing department often acts as a proxy for the "customer" and is instrumental in making feature decisions.

The second is management function measuring bang-for-the-buck to ensure that the business gets what it wants. This role decides which stories should be developed in what order, irrespective of the detailed knowledge of the stories. This usually involves a collection of business-oriented

[2] This example is drawn from [6].

Copyright 2009-2013 © Stephen J. Mellor

people including the folk from Marketing. Business people provide value primarily in the application domain for the obvious reason that they care about what the system does, not the infrastructure that supports it. However, the business may impose general requirements on other domains, such as a requirement to use a particular standard or to reuse an existing (possibly suboptimal) implementation.

The third role is technical expert—the person who can answer technical questions of just how some particular story works, or is supposed to work. For all domains, it is important to have technical experts on site available to answer questions about desired functionality and about "how things work." In the case of the application domain, this is the customer or their proxy. For other domains, the developer him/herself may the technical expert.

Accordingly, I have separated the concept of a "Customer" into two categories: Business People (including the end user or their proxy) and Technical Experts. I have grouped them so as to distinguish between the planning aspect (properly the bailiwick of the business folk) and the necessary information about functionality that may be gleaned from technical experts. The roles may overlap.

4. Agile Principles

In this section, we list briefly thirteen agile principles and call out those that require further discussion (shown in italics). The names are my own, drawn from XP and the Agile Manifesto. They should be recognizable to an agile practitioner or a developer that has read up on agility.

1. *Business People on Site*
2. *Technical Experts on Site*
3. *Face-to-Face Communication*
4. Trust Motivated, Self-Organizing Teams
5. *Collective Ownership and Coding Standards*
6. Sustainable Development
7. Harness Change
8. *Metaphors and Simple Designs*
9. Technical Excellence and Refactoring
10. Testing
11. *Continuous Integration with Frequent Releases*
12. Estimate to Improve
13. Reflect and Tune (the process)

We now consider the agile principles that require some refinement.

4.1 Business People on Site

"Business people and developers must work together daily throughout the project."

Business people provide value primarily in the application domain for the obvious reason that they care about what the system does, not the infrastructure that supports it. However, the business may impose general requirements on other domains, such as a requirement to use a particular standard or to reuse an existing (possibly suboptimal) implementation.

As usual, the business people are involved in selection and prioritization of stories, and keep doing so over time. (See SCRUM [3] etc) Second, we must cost those stories in the broadest sense. If a 3meg camera increases the price of the cell phone beyond the price-point desired by marketing, then we need to establish that fact as early as we can, reordering and reprioritizing stories as we go. Third, Marketing needs to be able to demonstrate prototype products to customers. Again, this may lead to further reprioritization.

4.2 Technical Experts on Site

"Technical experts and developers must work together daily throughout the project."

For each domain, there are usually several experts. In the case of Device I/O for example, you will need to talk to the hardware engineers to find which bit twiddles what in the physical world. Similarly, when working on the Motor Control domain, there will no doubt be questions about safe acceleration and deceleration rates that must be answered by technical experts.

For each domain there are four kinds of customers. The first is the technical expert for the domain itself. If you are working on Motor Control, you will need to talk to (or be) an expert on Motor Control.

In addition, you must consider your client domain(s), and ensure that you meet the assumptions made by them. If, for example, the Elevator domain assumes it will be provided floor numbers, then that will become a requirement on the Motor Control domain. In this context, we can see that—at least in some measure—the "customer" for motor control is not a person at all, but a set of derived requirements of which the "customer" (a passenger, say) has little or no knowledge.

The third type of technical expert is the expert for the server domains. Motor Control will need to manipulate certain signals provided by Device I/O. If the Elevator domain demands floor numbers but Motor Control can only provide distances, where will the translation take place? This will require the developer of a domain to negotiate also with the technical experts of the server domain.

Finally, the business people may have requirements on server domains, perhaps on the layout of the user interface or what constitutes a comfortable acceleration or deceleration rate.

4.3 Face-to-Face

"The most efficient and effective method of conveying information with and within a development team is face-to-face conversation."

And so it is. But there is also value in permanence for *some* communications, so any project has to ask: What mix of documentation and conversation is needed to convey understanding, including to people not yet on the project? What are the right documents? What do the right documents contain?

In my view, two types of document are helpful. The first is the *technical note*. A technical note is a small (one to three page long) *informal* document that records interviews or data. For example, we could describe the registers and fields required to operate the elevators controls. (Often, this is available directly from the hardware engineers, though perhaps not in an immediately useful form.) Or we may capture the safe acceleration and deceleration curves for the elevator in a couple of figures and limited amount of text.

Note the focus on "informal" and "limited" amounts of text. The goal here is not to build a Document with a capital D, but to capture some information quickly and effectively for verification (Is that what you meant?) and for the future. Much of the "formal documentation" that appears in the code or model can be drawn from technical notes.

The second type of useful document captures the path-not-taken. It is agile dogma that documents are bad and code, because it tells the truth about what *really* executes, is good. However, the code does not tell why us we chose not to employ *other* plausible approaches.

Consider, for example, a system that reports and logs alarms. Should an alarm be considered as event ("This went wrong"), or as a level ("This is wrong.") It is appropriate to think of an alarm as an event when reporting it, but that does not address how persistently incorrect states are managed, or what happens when a fault is fixed. There are good reasons for choosing each alternative, but if the rationale is not documented "I wouldn't have done it that way" will rear its head and the system will become inconsistent.

4.4 Collective Ownership

"The code belongs to all of us."

This is a special case of the more general principle that information on a project should not be private. It also applies to information.

Privately held information is difficult to access. You have to find the person who has the information. That information needs to be "the latest." The person who holds the information has to be available when you are. And as the number of people increases on the project so does the number of communication paths, each of which is holding still more private conversation and producing privately held information.

To avoid an n-way shoutdown, be sure to put any information in a repository with a document identifier. This saves enormous amounts of time finding information and reduces the number of communication paths. It should be as simple as checking in code.

Collective ownership implies standards. Nothing new here, but standards apply also to models if you are using them.

4.5 Metaphors and Simple Designs

"A system's architecture can be described by metaphor."

The shape of the system is determined by shared metaphors between customer and developer. This can apply at the level of an application or in terms of the software architecture.

There are three main software architectures that we see in real-time and embedded systems: Monitor and Control, Transporters, and Transactions. Monitor and Control systems are those systems that are like a controller, PID loops and the rest. Examples are Fly-by-wire aircraft, aluminum/steel rolling mills, household temperature control, automobile controllers, such as automatic braking systems.

Transporter systems are like a telephone system, which stream data through without changing it. Other examples are telemetry, and—somewhat surprisingly—offline credit card transactions, which stream data through, sorting into customer and merchant buckets (as opposed to routing a call), and sum the values periodically.

Transaction systems are like banks: they maintain a picture of some abstract world and accept operations that queries or updates that data. Many engineering systems, such as simulations follow this pattern, manufacturing systems too.

Describing a system using metaphor provides a quick picture of what the system is "like," a far more effective approach than reading detailed documentation. Nonetheless, it is worth pointing out that a metaphor can be written down, itself becoming "documentation."

As a more concrete example, consider a chemical plant with a set of pipes, valves and pumps. This is "like" an electrical network with circuits and breakers. (Electrical potential obviates the need for pumps.) Similarly, an electrical network is "like" a graph. By thinking in terms of what a system is "like," we can simplify our designs significantly and ease communication about what it does. For an extended example, see [5]

4.6 Frequent Releases

"Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

Excellent advice. But how to do it in an embedded-system context? Again, the problem is the infrastructure: How can we deliver a story without the necessary infrastructure?

The answer, unsurprisingly, is to build the infrastructure. As a first cut, technically speaking, build from the bottom of the domain chart first. Because the lower-level domains often interface to the hardware, building these early allow the hardware engineers to make progress in checking that their hardware works as they advertise. However, this approach delays the delivery of the first application-centered story.

To address this problem, we may build only a portion of the lower-level domains. For example, instead of building the entire Device I/O domain, we may instead build only the digital input signals and ignore analog signals and output signals. This would allow us to link these signals to the Door (say), and demonstrate that if we set the signal indicating that the door is blocked, the door will indeed open. In other words: a complete user-centered story.

Of course, it's not quite complete because there is, as yet, no way to send a signal to the hardware actually to open the door. We can, however, stub this out and demonstrate that the correct call is made. Meanwhile, work can continue on building the code for outputting signals. Alternatively, we could link these services to the User Interface (assuming this consists of buttons linked to digital signals in the Device I/O domain), and then to link that to the application.

Selection of which stories to build first will be determined largely by (perceived) risk and by the stories that the customer would like to see first. Which neatly brings us back to where we started: Business people on Site.

4. Takeaways

Agile approaches have a place in embedded systems development, but they require some refinements to account for multiple levels of abstraction.

Because the customer for an application domain is neither an expert in the technology nor close to the majority of work involved, a proxy sometimes replaces the "customer." The proxy varies greatly, so we generalize and use the term "Business people" who take management decisions on which stories to implement first and what they should be.

Technical experts take on the role of the customer for domains other than the application. Because client domains place requirements on server domains, the technical expert for the client domain may also act as customer.

Face-to-face communication is generally best, but it imperative to document the design rationale, and in particular why one design was chosen over another. This significantly reduces time wasted remaking decisions. In addition, a short, informal technical note can be used to capture (sometimes incomplete) information for verification and for future incorporation into the product.

Collective ownership applies not only to code but also to information. Documents of all sorts should be under collective ownership and configuration control. Information bartering is evil.

Metaphors and simple designs allow us to communicate quickly the "shape" of the system. A good metaphor reduces the need for documentation.

Frequent releases are a necessity, but the nature of real-time and embedded systems means that much infrastructure needs to be built first. To deliver a user-centered story quickly, pare the amount of each domain that needs to be built to support its clients. This will require negotiation with the business people to determine whether to deliver stories that are easy to deliver quickly or whether to build more infrastructure and deliver stories that would otherwise be preferred by the business people.

5. References

[1] Kent Beck, *eXtreme Programming explained,* Addison-Wesley Publications Co; ISBN: 0201616416

[2] www.agilealliance.org

[3] Schwaber, Ken, *Agile Project Management with Scrum*. Microsoft Press. ISBN 978-0-735-61993-7.

[4] Leon Starr, *How to Build Shlaer-Mellor Models*, Prentice Hall

[5] *Designing for Change,* Stephen J Mellor, Design West 2013, Class 216

[6] Leon Starr, *Executable UML: A Case Study*, Model Integration, LLC.; ISBN: 0970804407 2 CDs included

Copyright 2009-2013 © Stephen J. Mellor