# Applying Agile Methods to Embedded Systems Development

**Doug Dahlby, PhD**

**Principal Software Architect, ArrayComm, Inc.**

**2/13/2004**

# 1. Disclaimer

This paper consists purely of the author's opinions. The author does not purport to represent any formal group with his opinions. The assertions in this paper are based on speculation and common sense rather than hard data. The paper has not been edited, and thus is lengthy and meandering. But if you want to read it anyway, it will hopefully provide both a useful summary of information and new insights.

# 2. Introduction

Over roughly the past decade, iterative methods of software development have gained acceptance, largely displacing older methodologies such as the waterfall or V models of software development. The past few years have seen agile methods, in particular, gain a widespread following. Agile methods do indeed represent an advance in best practices of software development, but the benefits of agile methods vary across the different regimes of software development. Embedded systems is an example of a software development regime in which application of agile methods can be challenging, and the benefits of using agile methods may not be as pronounced as in other regimes of software development. This paper discusses how the distinguishing aspects of embedded systems software development affect the application of agile methods to embedded systems. Although the net result of using agile methods in embedded system software development is an improvement, careful consideration is required to achieve the potential benefits.

# 3. Background

## *3.1. Brief description of software development issues*

### 3.1.1. Software life cycle

The process of creating, deploying, and supporting software has several fairly distinct stages: system specifications, system architecture / design, component specification, design, coding, unit testing, integration, and maintenance. Different software development methodologies address the stages differently. Some focus on or neglect certain stages. Some proceed serially through the stages, while others allow for overlapping stages or cyclic iteration across stages. Regardless, each stage must be addressed at least in passing in any software development methodology.

### 3.1.2. Software coding tradeoffs

There are many metrics that can be used to evaluate the "quality" of software. Many code quality metrics are correlated, but some are anti-correlated. In particular, performance aspects such as cycles and memory usage are often anti-correlated with code clarity aspects such as readability, testability, modularity, and maintainability. To get good performance, the machine code must be well matched to the particular chip architecture. To have good clarity, the source code must be easily understandable by human reviewers. In theory, a compiler should be able to translate good-clarity source code into good-performance machine code, but in practice this translation is too complex to be completely feasible. Ideally, a compiler would generate highly tuned machine code from a natural language or simple graphical specification, but compilers will not have sufficient artificial intelligence to do this within the foreseeable future. Moore's law of exponential improvements in processor capability suggests that the performance / clarity trade-off should be weighted towards clarity, since machines continually improve their capacity to run machine code but humans don't appreciably improve their capacity to read source code. However, the optimal performance / clarity balance depends on the software regime.

## *3.2. Brief history of software development methodologies*

### 3.2.1. Chaos

The "chaos" model of software development jumps into coding and neglects requirements, design, and incremental testing. This model was used in the early days of computer programming, but works only for very small and simple systems. All subsequent development models have sought to improve on the chaos model by applying decomposition to make software development of a large system practical. Practical software development relies on decomposing the system into largely independent pieces, and building the system implementation by gradually accumulating working pieces. Thus, practical software development focuses at a high level on the whole system and at a detailed level on small components of the system. This is in contrast to the chaos method's unscalable focus at a detailed level on the whole system.

### 3.2.2. Functional and Waterfall

Functional programming (a.k.a. structured programming) focuses on the intended behavior of the software. The software's gross behavior is decomposed into functionally cohesive routines. The waterfall model of software development makes a single long serial pass through the life-cycle

phases, constructing all parts of the system at once and assembling them at the end. This model is illustrated in the figure below.

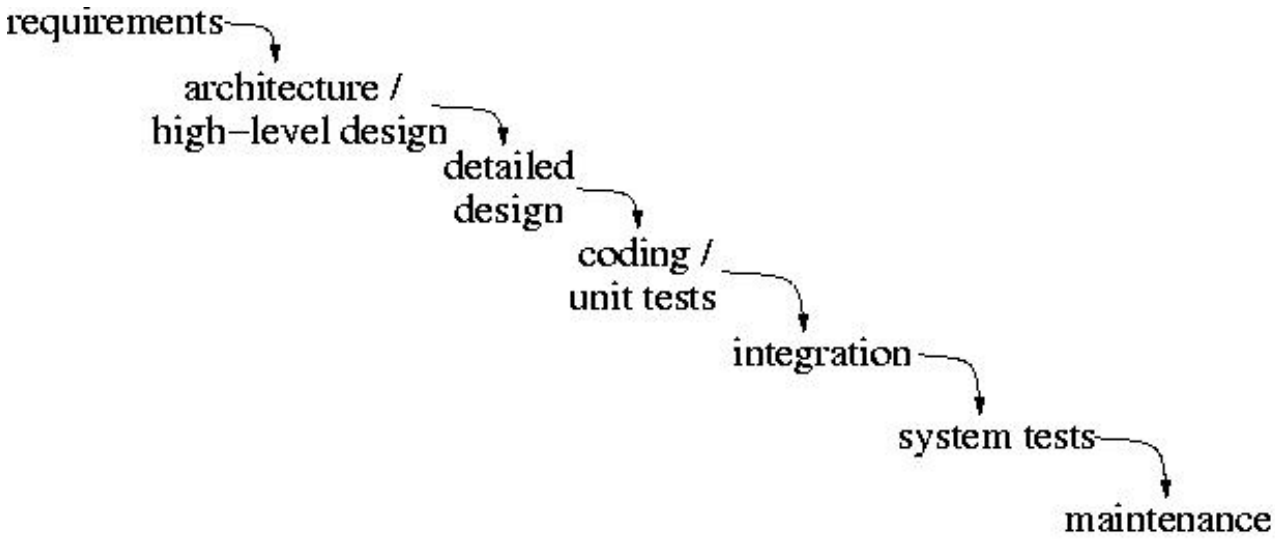


**Figure 1: Basic Waterfall Model**

Some versions of the waterfall model allow for iteration between adjacent phases of the life cycle, as is illustrated in the figure below.

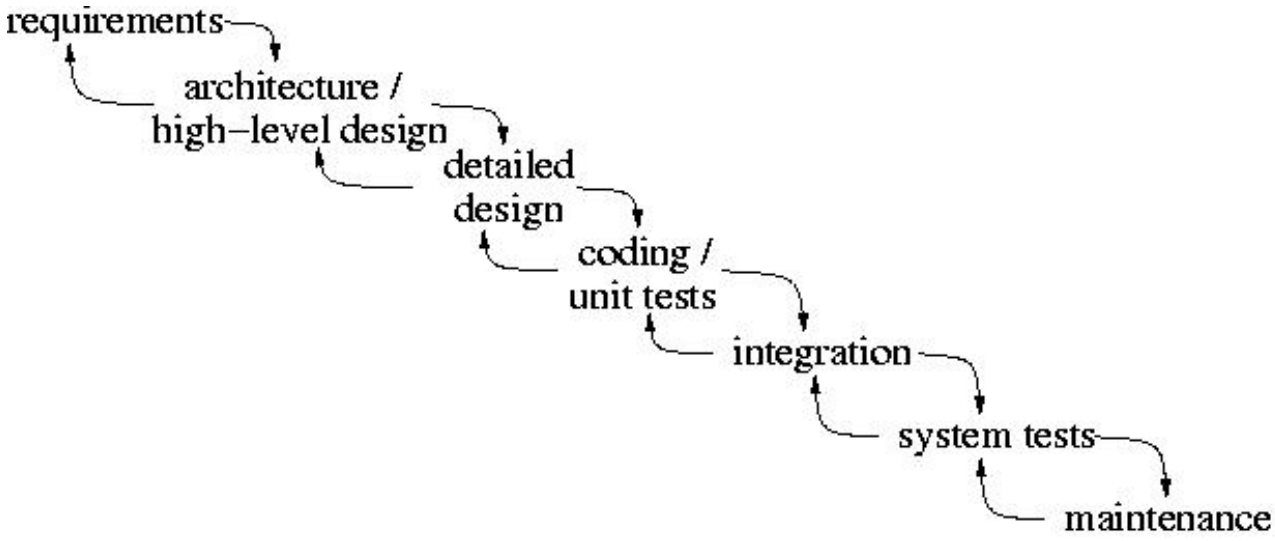


**Figure 2: Waterfall Model with Overlapping Phases**

In practice, it is virtually certain that in large systems some problems discovered during later stages will not be addressable by quick fixes, but rather will require re-architecting or even revisiting requirements. Thus, whether intentionally or not, in practice the waterfall model usually ends up being large-scale iterations that involve progressively more of the life-cycle phases. The potential resulting complexity is easily seen from the figure below.

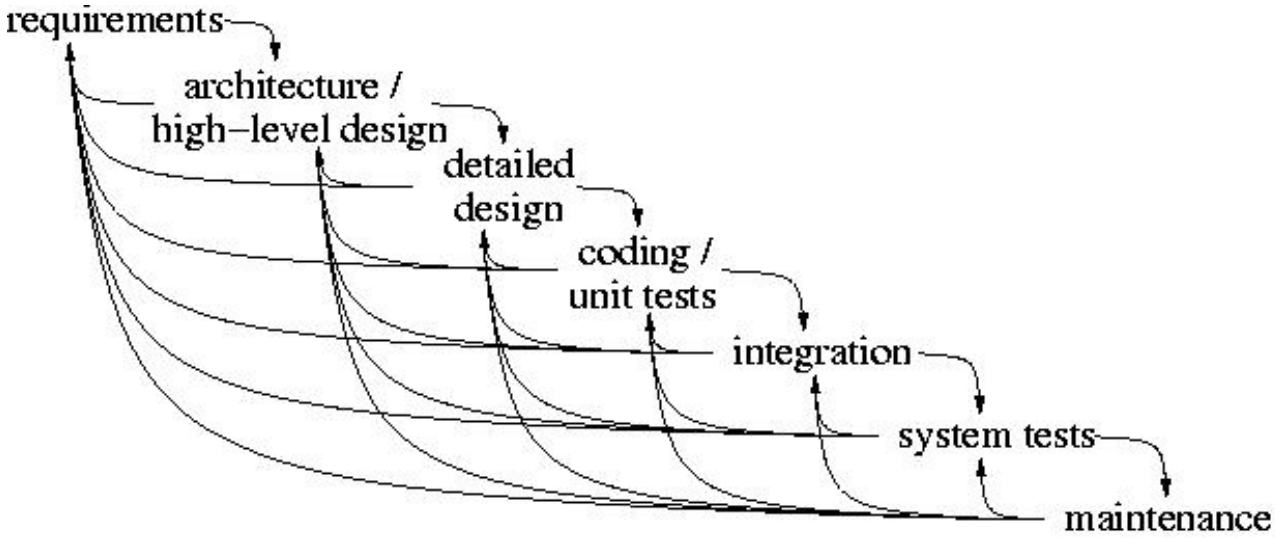


**Figure 3: Realistic Waterfall Model**

These paradigms were developed partially in reaction to the problems encountered with the chaos model, and were used extensively through the '70s and '80s.

### 3.2.3. Object-oriented and Iterative

Object-oriented programming focuses on the characteristics of components of the software. The iterative model of software development makes repeated serial passes through the life-cycle phases, constructing and incorporating small pieces of the system.

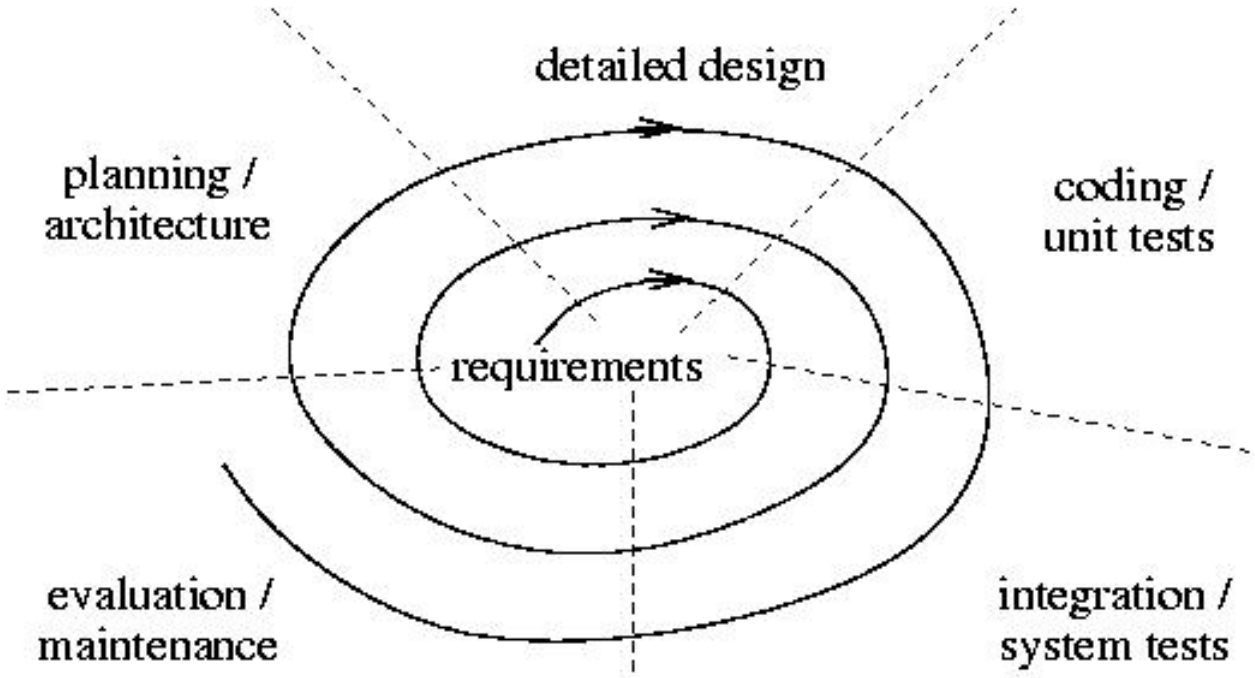


**Figure 4: Iterative Model**

These paradigms were introduced partially in response to problems encountered with the waterfall model, and were used extensively through the '90s.

### 3.2.4. Aspect-oriented and Agile

Aspect-oriented programming combines functional views of software (such as use cases) with object/component views (such as class diagrams). At different stages of the software life cycle, either a functional or object based view (or both) can be used, according to whichever is more appropriate at the moment. Generally speaking, a functional view is best for requirements gathering, since users care about what the system does rather than how it is organized. Both functional and component views are important during system architecture. Detailed design, coding, and unit testing tend to focus on component views so that the software can be constructed in manageable pieces. During integration, both component and functional views are important, culminating in acceptance tests' focus on function. Both functional and component views are important during maintenance, as the functionality of new features or fixes to existing features is considered, and impact analysis and regression testing are applied to components. Aspect-oriented programming not only adds a structured viewpoint back into component-based systems, but also provides a higher-level functional picture. The functional "aspects" often crosscut structured routines as well as components. Aspect-oriented programming relies on a framework of "join points", specifications of different aspects that could be implemented at the join points, and advice for which aspect(s) to actually use. An "aspect weaver" then augments the basic / default functionality by splicing in the advised behavior at the join points.
Agile software development processes are a refinement of iterative processes, in which continually changing requirements are accepted or even encouraged. Rapid iterations keep the system responsive to changes, and close communication both between customers and developers and among developers ensures that the growing and morphing system correctly accounts for the changes. Test-driven development ensures that only the minimum required software is constructed, that it is constructed correctly, and that it remains correct throughout on-going changes.

Strictly speaking, agile processes are a sub-type of iterative processes. However, in the traditional sense, iterative processes iterate on the design, coding, unit testing, and integration portions of the life cycle, but still perform requirements analysis and system architecture up front. In contrast, with agile methods each iteration revisits all stages of the software development life cycle, though with some sub-types of agile methods (e.g. the Rational Unified Process, RUP), more emphasis is given to particular "workflows" during iterations in different portions of the development cycle. That is, though early iterations incorporate some coding and integration, they focus on specifications and architecture. Though final iterations include some architecture and design, they focus on impact analysis, coding, and regression testing.

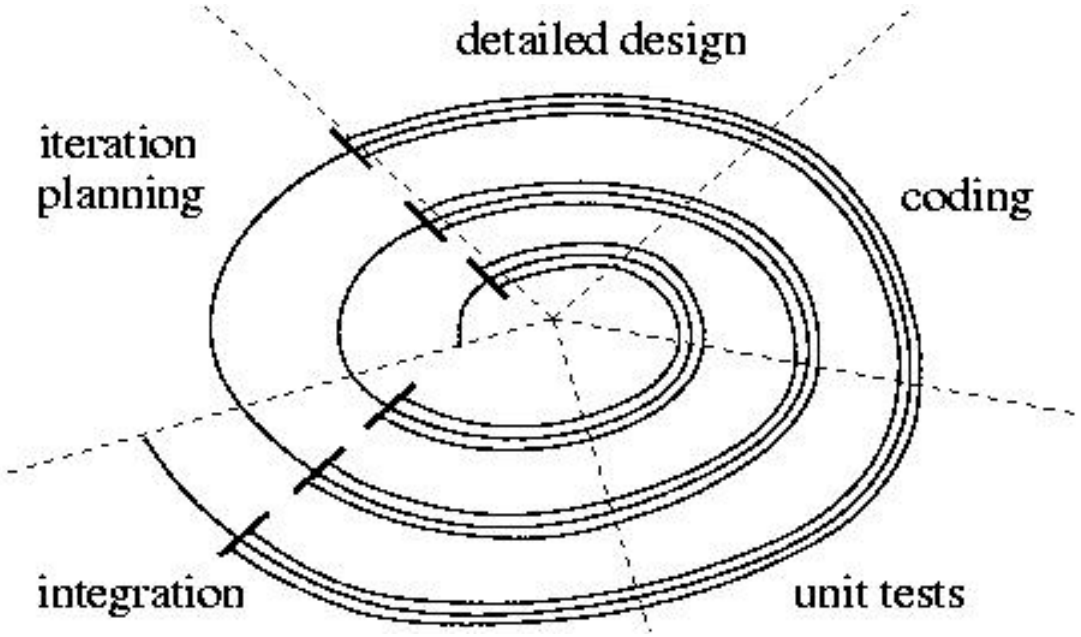


**Figure 5: Agile Model**

The distinction between traditional iterative processes and agile processes is primarily in the iteration planning phase, which first uses the metrics from past iterations to estimate the amount of work that can be done in the upcoming iteration, and then chooses what new features and components to include in the upcoming iteration. Thus, the agile method not only builds the system incrementally, as do traditional iterative methods, but also incrementally determines what kind of system to build.
Also, as is demonstrated by the parallel flow "swimlanes", during the design, coding, and unit testing phases, the iteration's work elements are assumed to be independent, and each proceeds through these phases at its independent pace.

These paradigms were introduced in the late '90s and early 2000s, and are coming into widespread practice.

# 4. What are agile methods?

The principles of agile methods are explained at the Agile Manifesto web site: http://www.agilemanifesto.org. In short, agile methods

- use continuous communication (customer/manager/developer, manager/developer, and developer/developer) to manage change
- keep close traceability between customer requirements ("stories") and the software
- deliver functional incremental software releases with a rapid turnaround
- improve ("refactor") old code selectively and continually

These key tenets are explained in more detail in the following section.

## *4.1. Agile method characteristics*

The principles of agile methods typically lead to specific practices such as the following:

1. use regular rapid cycles which create executable deliverables

2. focus on coding rather than planning or documentation
3. refactor continually to improve code
4. communicate continually and extensively within the engineering development team
5. communicate continually and extensively with customers
6. continually measure project progress, extrapolate projections, adjust long-term project goals (project end date and feature set), and set short-term goals (work elements for the next iteration)
7. use test-driven development to verify that code is initially correct, and emphasize regression testing to ensure that the code stays correct

## *4.2. Specific examples of agile methods*

Well-known agile methods include XP, RUP[1], Scrum, and Crystal. These agile methods are briefly explained below.

- eXtreme Programming (XP) – See http://www.extremeprogramming.org.
  The XP methodology focuses on small teams working very interactively to iteratively deliver working code. XP has four core values, which are addressed by core practices. The core values are communication, simplicity, feedback, and courage (empowerment and satisfaction). The core practices are
  - whole team (customers, management, and engineers all pulling together)
  - metaphor (shared high-level system picture to provide common context and vocabulary)
  - planning game (customers list "stories" (features), select stories to work on in each iteration, and specify acceptance criteria for each story)
  - incremental releases (executable artifacts delivered at each iteration, feedback provided by customer's evaluation)
  - simple design (build only what is needed right now)
  - pair programming
  - test driven development (write the test before coding the story, verify that the test initially fails, then passes after the story has been implemented, automate regression of unit tests and story acceptance tests)
  - refactoring (continual improvement of old code / architecture)
  - continuous integration (control software entropy growth through piecewise integration)
  - collective code ownership (anyone can write/rewrite any code, any time)
  - coding standards (agreement on common standards – allows refactoring to be substantive rather than stylistic)
  - sustainable pace (40 hour work week rather than death marches)
- RUP – See http://www-106.ibm.com/developerworks/rational/library/253.html.
  RUP divides a project into development cycles, and divides each development cycle into phases: inception, elaboration, construction, and transition. In turn, each phase consists of development iterations, where each iteration produces useful (ideally executable) artifacts. RUP identifies a series of "workflows", or topics, involved in software development: business modeling, requirements, analysis and design, implementation, testing, deployment,

[1] RUP is actually a "meta-process" that can be configured to form a wide variety of different software development processes. Frequently, though, RUP is presented with a configuration that makes it an agile software development process.

configuration and change management, project management, and environment management. All workflows are addressed in each phase, but each workflow's importance and effort ebbs and peaks differently across the phases.

- Scrum – See http://www.controlChaos.com/Scrumo.htm.
  The Scrum management process divides a project into short (30 day) iterations, or "sprints", where during each sprint there are short (15 minute) meetings ("scrums") between the development team and the team management to track progress, note current and imminent obstacles to progress, and decide what work to focus on until the next meeting. The development goals are kept constant during the sprint. Each sprint's goals are negotiated just prior to the sprint. Thus, the project goals adapt iteratively.
- Crystal – See http://alistair.cockburn.us/crystal/crystal.html.
  The Crystal methods consider not only what is theoretically optimal, but also what is actually practical, thus hoping to arrive at a good compromise that will have the buy-in needed to succeed. In particular, "markers and props"(documentation, discussions, meetings, etc.) are used where helpful to facilitate progress (and can be reusable "residue" for facilitating future progress). The formality of the process is scaled to the size and nature of the project.

### *4.3. Industry opinions regarding agile methods*

1. *XP eXcitement*
   XP is the most well known, most interesting, and most polarizing agile method. It has generated much buzz and fervor in the industry, and by now has attracted a sizeable share of practitioners. Many adherents view it as the messiah of development practices tainted by the waterfall model and strict ISO 9000/9001 compliance or CMM adherence. Others view it as a pendulum swing that has passed beyond a reasonable medium in its contrarianism to past overly-rigid development practices.
2. *Voices of moderation*
   Until recently, people debating the usefulness of XP were greeted with scorn, but now notable voices of moderation or dissent have spoken, pointing out potential problems with XP. Still, though, the industry as a whole continues to develop more interest in XP, and there is inevitable progress towards acknowledgement of the benefits of agile methods.

## 5. What are embedded systems?

Embedded systems can be roughly defined as "a system that is not primarily a computer but contains a processor". But rather than focusing on a definition, it is useful to consider aspects that most embedded systems share, at least to some degree.

- *Embedded systems are frequently price and size sensitive.*
  Many embedded systems such as PDAs or cell-phones are high-volume, low-cost and low-margin. This requires use of the cheapest components possible, which typically means simple processors and small memory (RAM and NVRAM/flash). This causes embedded systems software to trade off maintainability aspects such as portability, clarity, or modularity for performance optimization aspects such as a small boot image footprint, a small RAM footprint, and small cycle requirements. The increased up-front software development costs and periodic maintenance costs are amortized by the high-volume sales, and outweighed by the continuous hardware cost savings of cheaper components.

Many other embedded systems, though not so price-sensitive, have physical constraints on form factor or weight to use the smallest components possible. Again, this favors performance optimization at the cost of maintainability.
In addition to trading off portability, clarity, or modularity, embedded systems may also require optimization by using a low-level language, e.g. assembly rather than C, or C rather than code automatically generated from a UML model. However, this hand tuning is typically only applied to small portions of the software identified by the "90/10" guideline as being the major performance bottlenecks.

- *Embedded systems often have power limitations.*
  Many embedded systems run from a battery, either continually or during emergencies. Therefore, power consumption performance is favored in many embedded systems at the cost of complexity and maintainability.

- *Embedded systems are frequently real-time.*
  By nature, most embedded systems are built to react in real-time to data flowing to and through the system. The real-time constraints again favor performance aspects (particularly cycles usage) over maintainability aspects. There are generally both hard real-time constraints, which require an event to be handled by a fixed time, and soft real-time constraints, which set limits both on the average event response time and the permissible magnitude of outliers. Real-time operating systems use preemptive prioritized scheduling to help ensure that real-time deadlines are met, but careful thought is required to divide processing into execution contexts (threads), set the relative priorities of the execution contexts, and manage control/data flow between the contexts.

- *Embedded systems frequently use custom hardware.*
  Embedded systems are frequently comprised of off-the-shelf processors combined with off-the-shelf peripherals. Even though the components may be standard, the custom mixing and matching requires a high degree of cohesion between the hardware and the software -- a significant portion of the software for an embedded system is operating system and device driver software. Though this low-level software is often available for purchase, license, or free use, frequently a large portion of the operating system for an embedded system is custom-developed in-house, either to precisely match the hardware system at hand, or to glue together off-the-shelf software in a custom configuration.
  Often the functionality of an embedded system is distributed between multiple peer processors and/or a hierarchy of master/slave processors. Careful thought is required regarding the distribution of processing tasks across processors, and the extent, method, and timing of communication between processors.
  Furthermore, many embedded systems make use of specialized FPGAs or ASICs, and thus require low-level software to interact with the custom hardware.

- *Embedded systems are predominantly hidden from view*
  By nature, embedded systems typically have a limited interface with their "user" (real user or another component of the super-system). Thus, much of the system is developed to meet the software functional specifications developed during architecture and high-level design,

rather than the user requirements.

- *Embedded systems frequently have monolithic functionality.*
  Most embedded systems are built for a single primary purpose. They can be decomposed into components, and potentially the components could have low cross-cohesion and cross-coupling. That is, each component could serve a distinct purpose, and the interactions between components could be restricted to a few well-defined points. Nevertheless, the system as a whole will not function unless most or all of the components are operational. A system that requires all components to function before the system as a whole achieves useful functionality is a "monolithic system". This non-linear jump in system functionality as a function of component functionality is in contrast to some other types of software, where the system may be 50% functional (or more) when the software is 50% complete. For example, a space probe is built to travel by or to other planets and send back information about them. Though there are many lower-level responsibilities of the space probe components, such as targeting, landing, deploying sensors, deploying solar panels, and communications, each of these lower-level responsibilities is an indispensable component of the over-arching functionality. The space probe will be useless if any of these vital components is missing, even if all other components are completely functional. Another example is a cell phone, in which all the sub-features such as the user interface, the cellular base station selection, the vocoder, and the communications protocols are all vital aspects of the over-arching goal to transfer bi-directional audio information between the user and specific remote nodes.
  These are in contrast to other software regimes, such as web services or desktop tools, in which lower-level responsibilities are more likely to contribute independently to the aggregate system functionality rather than serving as indispensable parts of a monolithic whole.
  Though the software components of an embedded system are combined into a monolithic functionality, the components themselves are often very distinct. Embedded systems will frequently combine software components that perform signal processing, low-level device driver I/O, communications protocols, guidance and control, and user interfaces. Each of these specialized components requires a distinct developer skill set.

- *Embedded systems frequently have limited development tools.*
  Though some software regimes have a whole host of tools to assist with software development, embedded systems software development are more limited, and frequently use only basic compiler tools. This is in part because embedded systems often use custom hardware, which may not have tool support, and because embedded systems are often real-time and performance constrained, making it difficult to freeze the entire execution context under the control of a debugger or transfer control and data between the embedded target and a host-based tool, or capture extensive execution-tracing logs.
  Because of the limited choices of commercial tools for embedded systems software development, many embedded systems projects create their own tools to use for debugging and testing, or at least augment commercial tools with in-house tools.

- *Embedded systems frequently have stringent robustness requirements.*
  Embedded systems are often used in harsh environments and for mission-critical or medical

purposes. Therefore, requirements for reliability, correct exception handling, and mean time between failures are typically more stringent for embedded systems than for many other types of software. This translates into rigorous development processes and testing requirements. In turn, this increases the overhead needed to make a release of software. Some types of embedded systems are subject to regulatory requirements that purport to reduce fault rates by mandating the software development process, or at least specifying what documentation must accompany the embedded systems product.
Furthermore, for several types of embedded systems, it is difficult or even impossible to upgrade firmware, which emphasizes the need to "get it right" in the system's initial commercial release.

- *Embedded systems are frequently very long-lived.*
  Embedded systems often stay in use for many years. Frequently the duration of support for an embedded system is far greater than the turnover rate of the original software developers. This makes it paramount to have good documentation to explain the embedded systems software, particularly since the source code itself may have its self-documentation quality compromised due to performance trade-offs.

# 6. How well do agile methods fit embedded systems?

This section considers how well each agile method characteristics listed above applies to embedded systems software development.

1. *Agile aspect: use regular rapid cycles which create executable deliverables*
   *Implications:*
   In order to deliver new features within a short cycle (about 1-2 weeks), the feature set must be very granular. However, because embedded systems tend to have monolithic functionality, it is comparatively hard to decompose the functionality into small parallel pieces. Thus, a comparatively large amount of work is needed up-front to do detailed feature decomposition plans to allow for rapid cycles. Rather than obsessing over the decomposition, the cycle should be stretched to fit the size of the natural feature decomposition size.
   In particular, extra-long cycles may be required at the start of the project, while the simulation and/or OS infrastructure is being developed and the hardware bring-up is happening.
   Furthermore, the rush to fit a new feature within a cycle period may lead to the feature being initially implemented inefficiently, with the intent to refactor to a more efficient implementation during a later cycle. On the whole, this is an effective approach, but it has potential problems. First, the executable deliverable may turn out to not work at all due to cycle bloat. Secondly, by the time the product is almost out of cycles and optimization refactoring is required, some of the code that could have easily been improved when it was first written will take substantially more work later, when it is no longer fresh in mind. Finally, the product may end up suffering from a general lack of efficiency, rather than a few outstanding cycle-hungry functions/features. For example, debugging code may be freely used throughout, and end up dominating the cycles / memory usage of the product. If the debugging code has been well designed, it can easily be dynamically disabled or even compiled out. However, it if is developed ad hoc, it will take a lot of refactoring work to disable it. This "death by a thousand paper cuts" scenario is extremely difficult to fix

retrospectively. In contrast, a Big Design Up Front can set some specifications for which features need to be implemented efficiently from the start, and what coding practices should be observed universally to aid cycle and/or memory efficiency. Agile practices argue that up-front design should be minimized. This is a good practice to deal with volatile requirements and unforeseen issues. However, the top-level requirements for embedded systems are usually significantly more cut, dried, and frozen than requirements in the general software arena. Though embedded systems software development methods also need to account for change, the policy of responding to change should be based on the degree of change that is likely to happen.

Aggressive agile methods advocate creating a release for consumers as an output of each iteration. Even if an embedded system can deliver useful functionality in small pieces, making a customer release on every iteration is often not feasible for embedded systems, where the size, complexity, and required performance and reliability are large enough to require significant system testing in addition to the series of unit tests. To the extent that these system tests can be automated, the development process will greatly benefit. But continually running load tests will require additional test hardware systems, which can be costly. Furthermore, a large embedded system requires many iterations. Releasing each iteration to customers would cause an operations and support nightmare. Thus, for embedded systems it is a better policy to only invoke the overhead of extensive system testing for commercial release qualification and operations support only on selected iterations.

*Agile aspect benefit for embedded systems:* neutral

2. *Agile aspect: focus on coding rather than planning or documentation*
   *Implications:*

   Focusing on coding rather than design and documentation has some large benefits. In particular, it allows some early cycle and memory usage metrics to be taken. These early measurements, though still a poor predictor of the final product, are immensely better than back of the envelope calculations. Having cycle and memory usage projections available early allows for course corrections, or at least more accurate project planning. If it is discovered early that the cycles projection is far beyond the prototype processor's capability, perhaps a second prototype with a more powerful processor, faster bus, or better memory system can be developed. (Although, as is presented in the “Best Practices” section below, the initial prototype should already have the maximum possible capability.) Even if it is infeasible to introduce a second spin of the prototype, at least it will be recognized in advance that a lot of time will have to be spent in optimization.

   The benefits of early feedback have to be weighed against the extra importance of design and documentation in an embedded system. Up-front design is particularly important due to unavoidable limits on how much the design can be changed downstream. In other software development regimes, poor designs may result in a product that is slow, but still functional. In contrast, embedded systems tend to have real-time deadlines - if it's too slow, it's useless. Also, due to embedded systems’ emphasis on performance, portability may be compromised. This makes the initial choice of processors, operating system, compiler, and architecture particularly important, and also requires the portability / performance tradeoff to be well designed to account for system aspects that are likely to change within the product lifetime. The division of work among processors / tasks and the prioritization of

tasks cannot be lightly changed, so it is important to design the mapping of work to processors and tasks up front. Thus, design work merits extra attention in embedded systems, all the way from the system level down to the code component level. Embedded systems also often have extra documentation requirements. Due to the need to have optimized code, embedded systems can't rely as much on "self-documenting code" as software in other regimes. Furthermore, embedded systems' long-lived nature makes it highly likely that none of the original developers are available throughout the product's maintenance phase. Finally, certain embedded systems are subject to regulatory requirements regarding documentation.

Though agile methods give some thought to creation of other artifacts beyond working software, the agile viewpoint is that these other artifacts are often not necessary at all (the software should be self-documenting), or if necessary, should not be created until the relevant portions of software have been written, and then should be derived from the software. This viewpoint that working software is the overwhelming goal is simplistic. Though working software is indeed the primary deliverable of a software development project, there are other deliverables, which, though secondary, often cannot be neglected. These secondary goals include

a. *Objective proof that the software works correctly*
   This may require additional artifacts such as test / requirements traceability, test records, test coverage records, and source code quality metrics. Ideally, the majority of this documentation will be generated automatically by tools incorporated into the agile process. It is necessary to be sure that the emphasis on "working software" doesn't preclude this documentation, though.
b. *The ability to keep the software working correctly in the future*
   This may require additional artifacts such as architecture documentation to help code maintainers understand the high-level structure of the system, design documentation to help code maintainers understand the trade-offs, alternatives, and design decisions used when constructing the system, and source code quality metrics. These are artifacts that may not be requested by customers, because not all customers are enlightened enough to be aware of and plan against future needs. However, regardless of whether the customer explicitly requests such documentation, developers are bound by professionalism to produce it for projects whose complexity merits it.
c. *User documentation*
   Though agile methods are good at building good usability into software, software usability extends beyond the software itself. Additional artifacts of user documentation might include a user instruction manual, an installation guide, a change log / feature summary, errata list, and customer service. Again, these will be explicitly requested by customers if they are merited and the customers are sufficiently enlightened, but in many cases customers will not be aware of pending needs until they arise. To a large extent, these artifacts can be created within the agile process, derived from the already-working software that they document, but this needs to be carefully managed, and again is a departure from the agile philosophy that working software is all that really matters.

*Agile aspect benefit for embedded systems:* neutral

3. *Agile aspect: refactor continually to improve code*
   *Implications:*
   The practice of continual refactoring to continually do directed improvements of code modules is a great idea. In any software development project, there are challenges associated with refactoring, such as choosing which software modules to refactor on a given iteration, determining how extensive a chain of tightly-coupled modules to refactor at the same time, keeping the refactoring substantive rather than descending into religious wars about coding styles, getting management buy-in to improve "working" code, et cetera. Refactoring provides many key benefits, though. Foremost, it reverses, halts, or at least slows the "entropy growth" (complexity, obfuscation, and latent bug rate) of an aging software system. It can also serve as a very effective method for developers to improve their coding knowledge and style, and it disseminates familiarity with software modules across multiple developers.
   However, refactoring is more challenging in the realm of embedded systems than in other software development regimes.
   First, the relative emphasis on performance rather than clarity or modularity makes it more difficult for a prospective refactorer to correctly understand and update the old code. The decreased readability of the code is mitigated somewhat by a good architecture, with documentation available to the refactoring programmers. However, agile methods downplay architecture work, preferring to let the architecture grow ad hoc as the system is built from the ground up. Even if there is a solid architecture, it may have no good documentation due to agile methods' focus on coding at the expense of documentation.
   Secondly, the ability to refactor may be limited by previous architectural decisions which are immutable or at least very costly to change, such as the partitioning of work across multiple processors. In cases like this, refactoring-in-the-small may not be useful. For example, if one processor is out of cycles, it is more effective to make a refactoring-in-the-large architectural change of designating some of the overloaded processor's work to another processor (with the accompanying revision of data and control flow among the processors, and the work prioritization within the processors) than to make small, localized improvements within the original architecture. But to minimize these costly broad shifts in architectural paradigms, reasonable effort should be expended to make an initial best-guess architecture.
   Thirdly, due to the vastly distinct areas of software specializations such as operating systems, signal processing, control theory, communications protocols, and user interfaces, which are combined within an embedded system, it is impractical to have "universal code ownership", in which any developer can refactor code authored by any other developer. Most software developers have neither the interest nor the skill to master all the above areas of specialization. Thus, software modules falling within one of these areas of specialization will require refactoring by someone skilled in that specialization, rather than by a generic programmer. The idea of distributing code ownership is good, but differences in individual skills and specializations limit the extent to which the code ownership can be spread out. Universal code ownership also requires diligent programmers or code reviews (such as pair programming) to ensure that slacker programmers don't succumb to writing poor-quality code based on the assumption that the diligent programmers will feel obligated to improve it later.

In spite of these extra challenges, the notion of refactoring is very worthwhile in embedded systems, as long as the initial architecture and high-level design is good enough to limit the need for large-scale refactoring (rearchitecture), and if small-scale refactoring is approached pragmatically. One benefit refactoring particularly affords to embedded systems is a good approach to optimization, wherein during each iteration the worst cycle/memory hog software modules are identified and optimized, until the target performance is achieved. This iterative, incremental approach causes the optimization effort to stay focused on the software components with the most bang for the buck.
*Agile aspect benefit for embedded systems:* beneficial

4. *Agile aspect: communicate continually and extensively within the engineering development team*
   *Implications:*
   Agile methods call for software developers to communicate as personally as possible as much as possible, rather than relying on documentation or messages. This communication can range from code reviews to pair programming. Since the focus on performance in embedded systems tends to result in more obfuscated code, increasing the personal communication between developers pays large dividends.
   An extra complication often placed on in-person communication by embedded systems is the hardware-availability bottleneck. Due to cost or fragility, working prototype hardware systems are often in short supply. Furthermore, hardware reliability testing and software/hardware integration compete with software development for the use of the limited hardware systems. Hence, it is often important to keep the precious functional prototype systems fully utilized. To achieve this, developers may end up working as a tag-team, wherein one developer works an early schedule, then overlaps to hand off the development progress to a developer working a late schedule. This way, the development team can minimize the impact of hardware shortages, but at the cost of face-to-face time between the developers.
   Another potential pitfall of relying on in-person communication between developers is not developing adequate documentation for future developers who maintain the embedded system software. Frequently, documentation developed for communication between the original authors of the system is also invaluable to the system's maintainers. Often, relying on in-person communication results in neglect of the documentation needed by future maintainers. This is especially true in embedded systems, due to the long-lived nature of the system, and the extra difficulty to make the code self-documenting.
   Overall, in-person communication between developers is extremely beneficial, as long as it doesn't supplant generation of a reasonable amount of documentation for use by future system developers extending and maintaining the embedded system software.
   *Agile aspect benefit for embedded systems:* beneficial

5. *Agile aspect: communicate continually and extensively with customers*
   *Implications:*
   Because the majority of embedded systems software is not directly visible to the real customer, extending the communication between the development team and the customers has limited payback. It doesn't make sense for the customer or even marketing's customer proxy to pick the "stories" for the next iteration if the stories are "replace Gaussian

elimination with L-U decomposition", "hand-code the inner loop of foo() in assembly", and "add in-band control feedback to the channel X data stream". Though these are crucial decisions, it is counter-productive to spend the time and effort to educate the real customer to the point that they can make an informed decision. Instead, for embedded systems, a tech-savvy marketing team or a business-savvy engineering management team must stand in as the proxy for the real customer, to decide what features are essential in the near and long term, and what progression of stories over iterations will provide the optimal development path of these features. For embedded systems, it is particularly crucial that engineering management has exceptional technical and business skills, and that these skills are exercised proactively to set the course of the project, and continually to make corrections.

A counter note is that since embedded systems may require a customer proxy to stand in for the uneducated customer, it is useful to have the agile capability to frequently evaluate and reshape the way the system develops, to correct for disparities between the system visions of the real customer and the proxy.

*Agile aspect benefit for embedded systems:* unbeneficial

6. *Agile aspect: continual measurements, planning, projections, and adjustments by management*

   *Implications:*

   This practice is particularly useful for embedded systems, which face performance constraints in addition to the time, staff, and budget constraints common to all software development. In addition to actively managing the feature set, staffing, and projected delivery date, getting early and continuous metrics on actual performance of a partial system allows the manager to determine if, when, and where performance optimizations are required. The managerial projections need to take into account that embedded systems work is particularly unpredictable. Factors that cause this variability include potential need for extensive performance optimizations, and the unpredictable times required to find deeply hidden, rare, and complex bugs. Embedded systems bugs are often especially insidious, due to the close interactions and potential race conditions between multiple tasks in a real-time system. Furthermore, as is discussed in detail below, debugging methods are harder to apply within an embedded system. Nevertheless, the best way to account for this variability is to measure debugging effort required thus far, and statistically predict what is still required. These predictions are greatly facilitated in an agile method by the rapid progress to code development, and by combining testing, debugging, and integration with coding.

   Due to their size and complexity, embedded systems are particularly susceptible to Brook's conundrum that adding developers to a late project makes it later. The early feedback provided by agile methods makes it possible for managers to evaluate whether additional staff will be needed at a point early enough that the overhead of adding staff is less significant than the team's extended enhanced production rate.

   *Agile aspect benefit for embedded systems:* beneficial

7. *Agile aspect: test-driven development and regression testing*

   *Implications:*

   The agile practices of testing early and often are universally useful, due to the proven

wisdom that defect removal expense grows rapidly as a function of the time since defect introduction. It is easy to debug at the unit level while bugs are isolated to a small locale, but very difficult to debug at the system level. Debugging embedded systems is particularly difficult, for several reasons. Memory constraints may limit the use of debugging code and logs. Timing constraints and multi-tasking may make it impossible to use breakpoints to halt the system or to single-step through system execution. Even the insertion of debugging code may alter timing enough to hide the problem being debugged. The physical separation between the target processor being debugged and the host processor on which debugging tools may be running requires special interactions and possibly special hardware to pass debugging commands and data between the host and target processors. Frequently a special system must be dedicated for testing and debugging, which may result in access contention. Special test equipment may be needed to generate a system load that exposes bugs. Even with a good unit testing process, it is inevitable that significant debugging will be required during the integration of software components and the integration of the software onto the hardware. Nevertheless, by removing as many bugs as possible through unit testing and using incremental integration steps, the pain of debugging can be minimized.

A natural step to aid debugging and integration and to defer software / hardware integration off of the critical path is to develop a simulator for the embedded system. Use of a simulator involves significant extra effort to design, create, and maintain the simulator framework (particularly if the simulator handles multiple tasks within a processor, multiple processors within a system, and multiple systems), to design a good hardware abstraction interface where the simulator framework can interact with the application software, to compile the software for the simulator target as well as the real target, and to deal with the slow execution speed of the simulator. Nevertheless, a simulator provides huge returns on investment by decoupling software development from hardware bring-up, providing a convenient facility for debugging, making wide-scale software testing feasible, and allowing load testing that may not be feasible in a physical system.

Regression testing also has extra complexities in embedded systems, due to the need for specialized test harnesses and test equipment to exercise the system under test, the relative difficulty of probing the internal state of the system to verify correctness, and shortages of test systems. Naturally, regression testing should be performed on both the real system and the simulator, preferably from a framework that integrates test execution and test record logging across the real and simulated systems. In spite of the extra difficulties in performing testing and debugging in embedded systems, it is very important to follow the advice of agile methods by testing early and frequently, so that problems can be diagnosed more easily and fixed while there is still time to do so correctly and within the schedule.

*Agile aspect benefit for embedded systems:* beneficial

# 7. Best Practices

The overarching aim of agile methods is to improve a project's flexibility. There are many best practices related to making projects flexible enough to deal with unanticipated changes. Other important areas of best practices are productivity, enhancements, and quality. The best practices listed below overlap significantly with the practices recommended by the Rational Unified Process (RUP). Notable differences are that the RUP concentrates on a single development cycle (e.g. a single major release of a product) rather than considering how a sequence of development cycles throughout a product's lifetime are related. This distinction is emphasized in the figure below. The

RUP considers each cycle to be independent, but in reality the development cycles are closely connected. For example, creating coding conventions is a best practice that applies to the initial development cycle, but should be propagated largely unchanged into following development cycles. In fact, the product lifecycle looks strikingly similar to a single development cycle, but at a larger scope. The initial development cycles within a product lifecycle play much the same role as the initial iterations within a single development cycle.

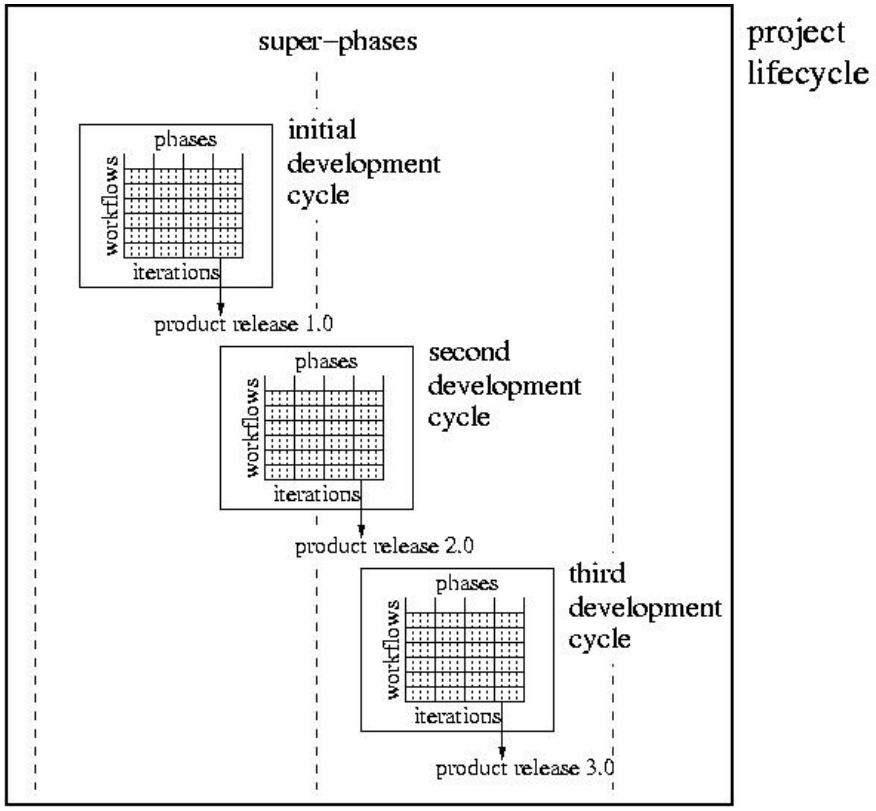


**Figure 6: Development Cycle Iterations**

Also, the RUP is a meta-process that is not inherently tailored for embedded systems development.

## *7.1. Flexibility*

- For the initial development cycle, do an up-front initial system architecture without obsessing. Identify key components and their interfaces, since interface refactoring is much more difficult than internal refactoring. Review the architecture periodically, especially as the initial functionality is implemented or significantly different functionality is added, to see if the architecture needs to be modified. For follow-on development cycles, assess whether the previous cycle's architecture is reasonable for the new cycle.
- For the initial development cycle, do an up-front product feasibility study that considers all potential future development cycles. For each development cycle, do an up-front feasibility study to address technical financial risks, and reassess the feasibility, especially at major cycle milestones. Periodically reassess the feasibility of continued support for the product as a whole.

- Establish coding standards and code templates at the start of the initial development cycle. Review the standards and templates at the start of each subsequent development cycle to see if modification is necessary.
- Use iterative development to get rapid prototype systems.
    - Use development rate metrics to extrapolate / refine projections of project completion.
    - Use prototypes' performance metrics to extrapolate / refine projections of the full system's performance.
    - Be careful to not invest too much effort in simple but inefficient algorithms.
- Do initial software development using a prototype hardware system which is as capable as possible
    - Use the fastest processor and bus speed possible (but try to choose something which is from the same family and is pin-compatible with the long-term goal processor).
    - Provide a generous amount of flash memory and RAM.
    - Use a FPGA prior to committing to an ASIC.
    - Store the FPGA image in a network location accessible to the micro-controller rather than in a boot ROM (or provide a boot ROM so the system can operate unplugged, but allow the system to select whether to load the FPGA from the boot ROM or network). Consider packaging the FPGA image inside the micro-controller's executable image.
    - Provide the option to boot the micro-controller with an image stored on the network rather than the boot ROM / flash memory's image.
- Create a simulator early on.
    - Use the simulator for initial software development until the hardware is ready for integration.
    - Use the simulator to develop new software components in parallel with integrating existing software components onto the hardware.
    - Use the simulator for debugging and regression testing.
- Defer optimization until it is clear how much is needed, and which software components are most suitable for optimization. Estimate based on code efficiency how much speed-up could be achieved with reasonable effort, and weight the speed-up by the fraction of cycles currently used by that code component. When considering how much a software component can be optimized, account for the extent to which the component needs to be left modular, portable, and generally maintainable. Optimize algorithms before optimizing code structure.
- Balance designing and coding components to be extensible for future features against focusing the design and code primarily on the current feature set. Write code that is easy to refactor, but don't make extensive predictions of what changes will arise in the future.

## *7.2. Productivity*

- Do proactive management planning.
    - Establish general long-term plans but focus on the decomposition of the project into small pieces (components / stories / work elements).
    - Carefully consider how work elements can be arranged to maximize parallelism. For example, create a simulator early on so that software development can subsequently proceed in parallel with hardware integration.

    - Focus first on "enabling technologies" such as process definition, coding conventions, commercial tool acquisition, in-house tools and component libraries, debugging and profiling facilities, operating system choice and abstraction, and hardware reliability verification.
    - As the "enabling technologies" start to emerge, use risk analysis and PERT analysis to schedule work elements into iterations. In addition to the risk innate in work elements, account for the risk of integration by scheduling work elements to extend and link chains of functional components, rather than developing components in isolation. If a low-risk component "connects" two high-risk components, develop the low-risk component (albeit possibly in a stub form) along with or shortly after the high-risk components it interfaces with.
    - Proactively facilitate in-person communication between developers working to integrate software components together, and software and hardware developers working to integrate the software onto the hardware. Having the relevant personnel work together as an integration team is better than having individuals tackle integration.
    - Create artifacts for internal use documenting how to use the software, how to use the hardware, and how to use the software on the hardware. Maintaining a concise document is much more effective than trying to have all personnel individually remember all information disseminated by relevant domain experts, or to spend time searching for the relevant expert when usage instructions are forgotten.
    - Use metrics from earlier iterations as feedback for revising plans and expectations for the upcoming iteration and the project as a whole.
- Provide useful software and hardware tools. Identify what features are useful, then get a tool that provides nearly the best-of-breed support for those features. Look first for a free-ware tool, but don't hesitate to purchase tools. It is penny wise but pound foolish both in terms of money and development schedule to avoid spending a few thousand dollars on a tool that would result in a 5% coding / merging / documenting / debugging gain for each of 20 engineers being paid $70K salaries. Also consider creating custom in-house tools.
- Expend the effort to make software components and tools developed in-house as generally useful as possible. Refactoring should not be relied on to improve tools at a future time. First, opportunities for tool reuse will be missed until the tool is refactored to be generally applicable. Secondly, changing the tool's interface during refactoring will necessitate wide-spread code changes, since a good tool by definition is used extensively.
- During early iterations of the initial development cycle, staff the project with only senior developers who are well qualified to do architecture, design, and coding. During subsequent iterations, gradually bring on more junior staff to serve as understudies to the senior staff. Produce documentation based on the process of introducing the initial junior staff to the project, and use this documentation to off-load some of the senior staff's work while introducing new junior staff to the project. Near the end of each development cycle, allow senior developers to investigate rearchitecture and high-level design for the upcoming development cycle. Proactively involve senior developers in the small-scale rearchitecture that may occur during an iteration's planning phase.

### *7.3. Quality*

- Evaluate the low-level design and code of at least the key software components with informal reviews. Pair programming, a mentor / trainee relationship, and peer reviews within a development group are various methods of accomplishing this.
- Encourage unit testing.
    - Make code templates for unit tests.
    - Have working examples of unit tests.
    - Store unit tests inside the code component they apply to.
    - Require unit tests for at least the key software components.
    - Automate unit tests so they can be easily reused during code maintenance.
    - Review unit tests to be sure they are high quality and thus easy to maintain.
    - Document unit tests so they can be easily maintained.
    - Primarily use a simulator environment for unit tests, as it may be infeasible for all developers to use scarce and complex hardware systems for unit testing.
    - Attempt to make the unit test able to execute on each variety of the real system, so compatibility / portability can be tested.
- Encourage regression testing.
    - Create an automated regression test hardware / software system. Make the regression test system as friendly as possible for interactive use by developers, but also able to be run in a bulk mode by scripts.
    - Keep logs of regression test results for each software build, noting which version of each regression test was in use at the time.
    - Encourage developers to regression test their changes on all varieties of hardware platforms and on the simulation platform before committing the changes to the master code repository.
    - Measure cycle usage and memory usage during regression tests.
    - Cover unit tests, feature tests and load tests as part of the regression testing.

## 8. Conclusions

The emphasis of agile methods is to stay flexible to account for unforeseen developments and a changing environment. This is a universally wise strategy, as long as it is applied within reason. Embedded systems development benefits from many of the key tenets of agile methods. However, because embedded systems are more rigid in some aspects than other software systems, agile methods don't apply universally, but rather need pragmatic adaptation.

## 9. On-line References


- http://www.embedded.com (the on-line version of *Embedded Systems Programming* magazine)
- http://www.embeddedstar.com (links and articles related to embedded systems programming)
- http://www.embedded.com/showArticle.jhtml?articleID=17602057 "Extreme Programming Without Fear", Dan Pierce
- http://www.objectmentor.com/resources/articles/EmbeddedXp.pdf "Extreme Programming and Embedded Software Development", James Grenning
- http://www.embedded.com/showArticle.jhtml?articleID=16700086 "XP deconstructed", Jack Ganssle

- http://www.therationaledge.com/content/dec_01/f_spiritOfTheRUP_pk.html "The Sprit of the RUP", Per Kroll

## About the author

Doug Dahlby has over 8 years of experience with all phases of the software life cycle in embedded systems development. Doug is currently employed at ArrayComm, Inc. (www.arraycomm.com) under the title of Principal Software Architect. Doug received his doctorate major in Aerospace Engineering and minor in Computer Science from Stanford University. Doug welcomes email comments regarding this article (dougdahlby@yahoo.com), but is likely to be too busy doing "useful work" to respond personally.