Faster, more reliable, lower cost: agile product development for the IoT

Executive summary

The Internet of Things (IoT) offers immense possibilities for product development: not only for connecting existing products and harnessing the data they generate to deliver new value and functionality, but also for entirely new product and service concepts. However, the 'always-connected' nature of the IoT implies the constant generation of data and feedback. This, in turn, means that products must move from the traditional product development cycle—design, manufacture, deliver—to a new world of dynamic innovation driven by deep, often real-time, product-derived market and customer insight. Gaining competitive advantage in this new world means maximizing the speed and responsiveness of product-design processes. Agile approaches are proven to deliver these benefits for software development; this paper examines how they can be extended to provide the same benefits to the end-to-end product design cycle.

Product Development in the IoT

Systems of all kinds—in the home, in buildings, in cars, in public spaces—are becoming smarter. The first such smarter systems ran simple control programs that augmented the purely mechanical and electrical capabilities of their predecessors. They added more options, more features, more flexibility, and more *intelligence*. User controls and feedback devices changed from mechanical analog dials to digital displays, and it became possible to customize the behavior of systems; allowing them to make some simple decisions based on given inputs freed up users to focus their attention on higher-level concerns.

The advent of the internet enabled unprecedented collaboration—still at the "design level"—in engineering. But until fairly recently, manufactured devices did not use the internet, which was largely reserved for communication between humans rather than between devices.

Many devices we now use on a daily basis are systems capable of automating highly complex tasks on their own; it is not uncommon for a modern car to have dozens of processors running hundreds of megabytes of software. These formerly disconnected devices can now connect—and collaborate—on a global scale and with vast numbers of other systems both similar and disparate. The ability to send and receive data enables smart devices to diagnose their own faults, to upgrade their own software, to identify patterns of behavior and to respond dynamically to rapidly changing conditions.

The current generation of smart systems is enabled by a number of key technologies. First, the devices are not only smarter, they can also exchange information both rapidly and at low cost over the internet. Data mining—the ability to identify and determine subtle patterns in data streams—has progressed to the point where a machine can out-perform human experts in games like Jeopardy!® The combination of sensors connected via the internet, cloud-based analytics services and devices that can interact with their environment enables the creation of systems of systems at any scale, capable of delivering value through new functionalities and services.

Stand-alone smart devices can provide tremendous value to consumers and businesses alike. Connected smart devices promise to add significantly more value to users of all kinds by improving optimization, easing product updates, automatically determining when service or maintenance is required, and providing a new mode of communication to the service vendors. Interconnecting the devices offers even more opportunity for suppliers of devices and related services, because it enables the identification of usage trends and provides a better view of how products are actually used. It also highlights systemic problems and enables them to be addressed before they manifest themselves, manages interactions with customers, and opens new markets for services that would not otherwise be feasible.

Welcome to the Internet of Things.

Why Agile Product Development for the IoT?

Agile development is now mainstream for many software projects. It is particularly popular for small applications, developed by a collocated team, with continual verification and (very) frequent releases. This works well for environments that can be updated easily, where the requirements are not well understood at the start, or are likely to change, where there is not a contractually-fixed feature set, and where it is not necessary to pay significant attention to issues of regulatory approval and certification.

Two primary benefits provided by agile development approaches are *quality* and *efficiency*. Quality is improved through the insistence on "developer hygiene", which means that the software is continuously verified as it is developed, rather than at the end of the process (the traditional approach). The impact on final quality is analogous to the results of brushing one's teeth daily rather than attempting to clean them intensively just once a year. Test driven development (TDD) is a key enabling practice for continuous verification. Continuous integration (CI) practices bring together the work from multiple developers so that time-consuming integration concerns can be almost completely avoided. Frequent releases allow us to deliver partial functionality where it can be validated by the users to ensure that the systems actually meet their needs.

Efficiency is improved largely because quality is improved. In traditional waterfall or V-cycle processes, testing is carried out at the end. This results in time and effort spent at the end of the process (when the work is meant to be complete) identifying problems both large and small. However, these expenses are dwarfed by the cost of discarding significant work and redeveloping the software when defects are discovered at this late stage.

Agile methods clearly have value, but how to apply agile to product development for the IoT isn't obvious. First, agile methods are generally applied to small systems. Work has been done to make agile more suited to the demands of large-scale development, for example, Disciplined Agile Delivery[1] and the Scaled Agile Framework® (SAFe®)[2]. Second, relatively little has been written on applying agile methods to embedded software development, let alone other engineering disciplines. *Real-Time Agility*[3] discusses the use of agile methods for real-time and safety-critical systems development, while the new *SAFe for Lean Systems Engineering* (SAFe LSE)[4] and *Agile Systems Engineering*[5] approaches address the application of these techniques to disciplines other than software.

Any discussion of the IoT should consider its three primary constituent parts. First, the "Things"—the devices or elements that provide on-the-ground capabilities such as managing car engines, heating homes and offices, producing power for neighborhoods, and navigating aircraft. Second, the "Internet"—the vast collection of networks that enables the interconnection of all the devices into potentially many conglomerates. Third, the Cloud, which—although not explicit in the IoT name—is vital for its ability to virtualize the services that gather, integrate, analyze, interpret, and act on the data gathered from all the Things. In this paper, we will focus on the development of the Things—the smart, internet-connected devices that include far more than just software.

Continuous Engineering and the IoT

Continuous engineering is a new approach designed to support more efficient development for products within the IoT space—where operational insights provided through analytics can continuously inform product creation, update and improvement. It is supported by three conceptual pillars—strategic reuse, continuous verification, and engineering insight.

Strategic reuse means leveraging existing engineering data and intellectual property to construct new systems. This isn't just about reusing source code; it is about reusing requirements, architectures, interfaces, test cases, and implementation too. This pillar is implemented primarily by Product Line Engineering (PLE) practices, which will be discussed in the next section.

The second pillar, continuous verification, requires that we verify the completeness, consistency, accuracy and correctness of engineering data *as we create it*. This approach dovetails seamlessly with agile practices such as test-driven development (TDD) and continuous integration (CI). Like the strategic reuse pillar, continuous verification also applies not only to implementation but to all engineering data. It requires that we capture engineering data in ways that are fundamentally verifiable, for which there is already a solution: models.

The third pillar, engineering insight, implies among other things the need to acquire knowledge about the use – and possibly misuse – of deployed systems. This includes how well these systems function in their operational environments and the detection of defects or violations of assumptions in those deployments. These aspects can be managed through the Monitor Deployment workflow of Continuous Engineering. Another implication of this third pillar is that we must manage engineering knowledge as a key business asset. Like many assets, it loses value if unmanaged. The management of engineering skill is supported by the Assess and Improve Engineering Capability workflows.

Figure 1 shows the primary workflows that constitute Continuous Engineering.

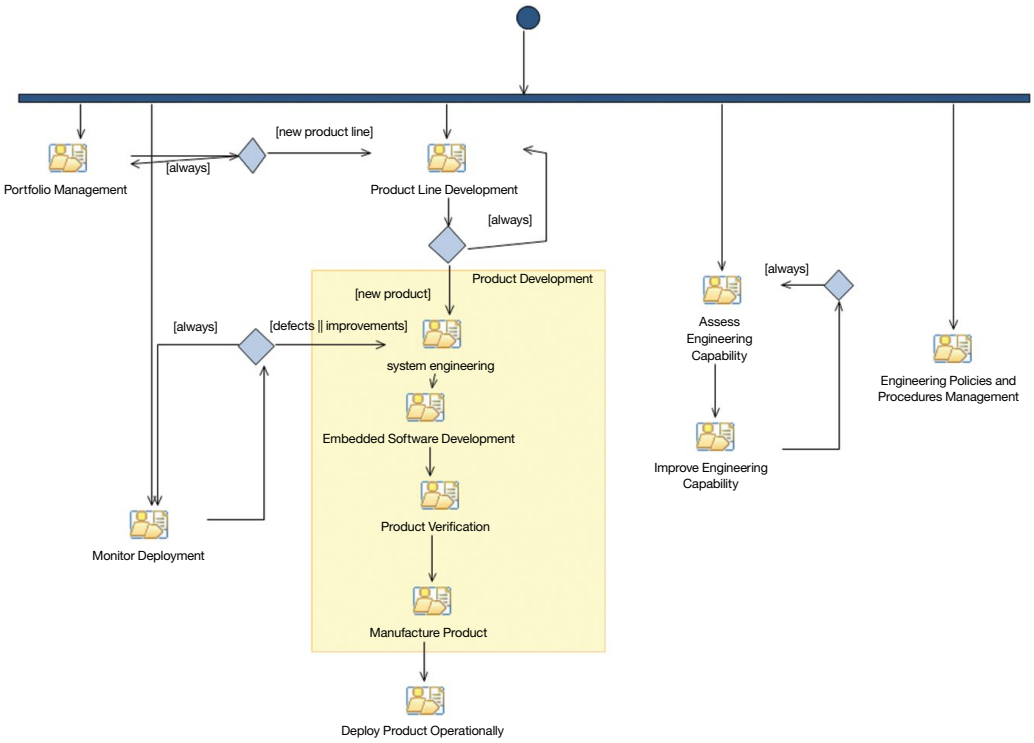

Figure 1. Continuous Engineering Workflows

Agile Product Line Engineering

Product line engineering (PLE) has been around for a long time both in the pure software realm and in certain industries (notably automotive). The basic idea is to create a family of products that differ in terms of features, platform, or targeted markets. The key term here is *family* because that implies that the products are highly similar. This means that it ought to be easier to create a product within the family (commonly known as a *variant*) than to create one from scratch, because much of the required engineering is already done and can be reused.

The primary barrier to effective PLE is that reuse is harder than it sounds because it involves not just reusing code. In product development, we produce many different sets of correlated, but distinct, engineering data. These data sets include vision, requirements, architecture, design, implementation, test cases, test environments, documentation, safety/reliability/security analyses, and so on. Complicating the process is the fact that we cannot just naively reuse the engineering data; these are, after all, *variants*, and so we will partially reuse the engineering data and partially create new engineering data. And doing that, without making mistakes, is *hard*.

Approach 1: "Clone and Own"

The most common way to reuse engineering data today is simply to copy and modify (also known as "clone and own"). This has the advantage of being conceptually simple, but results in divergent systems where identifying and fixing a defect shared by multiple products in the product line is difficult. The complexity of analyzing the impact of a feature change or repairing defects across variants with completely isolated data sets makes this approach very challenging.

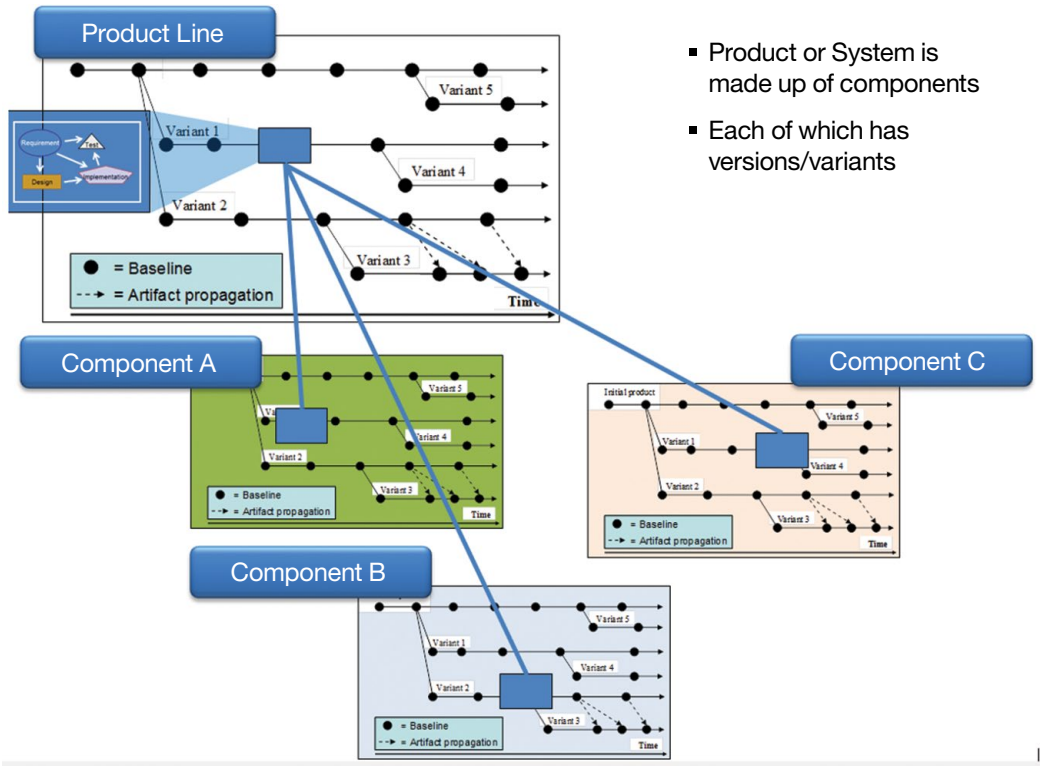

Figure 2. Managing PLE Variants

Approach 2: Streams

A more effective strategy is to use configured streams to manage both the core platform data and the variant-specific data. This strategy allows for ad hoc addition of new variants and is done primarily via branching streams. Because the core platform data is kept centrally, changes to elements isolated within the core can easily propagate into builds of the affected variants. Figure 2 shows a product line composed from three different components, each of which has streams for its engineering data (requirements, design, and so on). Specific products are shown as the variants within the product line.

Approach 3: Parametric Configuration

A rather more advanced strategy is to build in pre-defined extension points known as *variant parameters*. For example, a car might have a gasoline or diesel engine, three trim levels (Basic, Modern, and Luxury) and come with a manual or automatic gearbox. These three configuration parameters alone generate 12 variants (2 x 3 x 2). It would be fairly easy to then add a new kind of engine or a new trim level because those changes would align well with the predefined extension points. However, deciding late that you want to also produce variation around the roof type (hard, soft, sun roof and convertible) is relatively hard because this requires that all the other existing variants be retrofitted with that variation point. In this way, the parametric configuration approach simplifies the production of complex variations and generally has a higher level of reuse than the multiple stream approach, but makes it more difficult to product variants that do not align well with the predefined extension points.

Adding agility to the process helps avoid making decisions unnecessarily early in the project, keeping the process responsive to changes in market and customer needs. Here, the parametric approach (in which there are clear, predetermined extension points) is combined with the use of branching streams to allow the components to be shared. In addition, the development of a core platform and all associated data (with properties that are unlikely to be volatile) merged with branching streams for variant-specific data gives us the best of both worlds. Agile practices such as continuous verification via product simulation and frequent refactoring will benefit the development of product lines.

Agile Systems Engineering

Systems engineering focuses on the characterization of systems in terms of their essential properties without regard for which engineering discipline will implement those properties. For product development, systems engineering leads the way with a hand-off of engineering data to “downstream engineering” for detailed design and implementation. For product lines, there will be systems engineering at multiple sub-levels of abstraction, both at the final product level and at the core and variant levels.

The key activities for systems engineering include:

- System requirements specification
- Architectural trade-off analysis
- Architectural design
- Hand-off to downstream engineering

In traditional systems engineering, all systems engineering data must be complete before downstream engineering is allowed to begin. However, with an agile approach, we can focus the work on coherent clusters of requirements—usually use cases or user stories—develop the systems data around that, hand that data off to downstream engineering, and then work on the remaining requirements. Agile systems engineering is iterative and incremental. Figure 3 shows the high-level workflow for agile systems engineering. In it we see an iteration loop, in which a small number of use cases is developed, analyzed and handed off, before the next set of use cases is developed. In addition, there are ongoing activities, such as project management (in the Control Project activity), quality-assurance auditing when required, and management of changes.

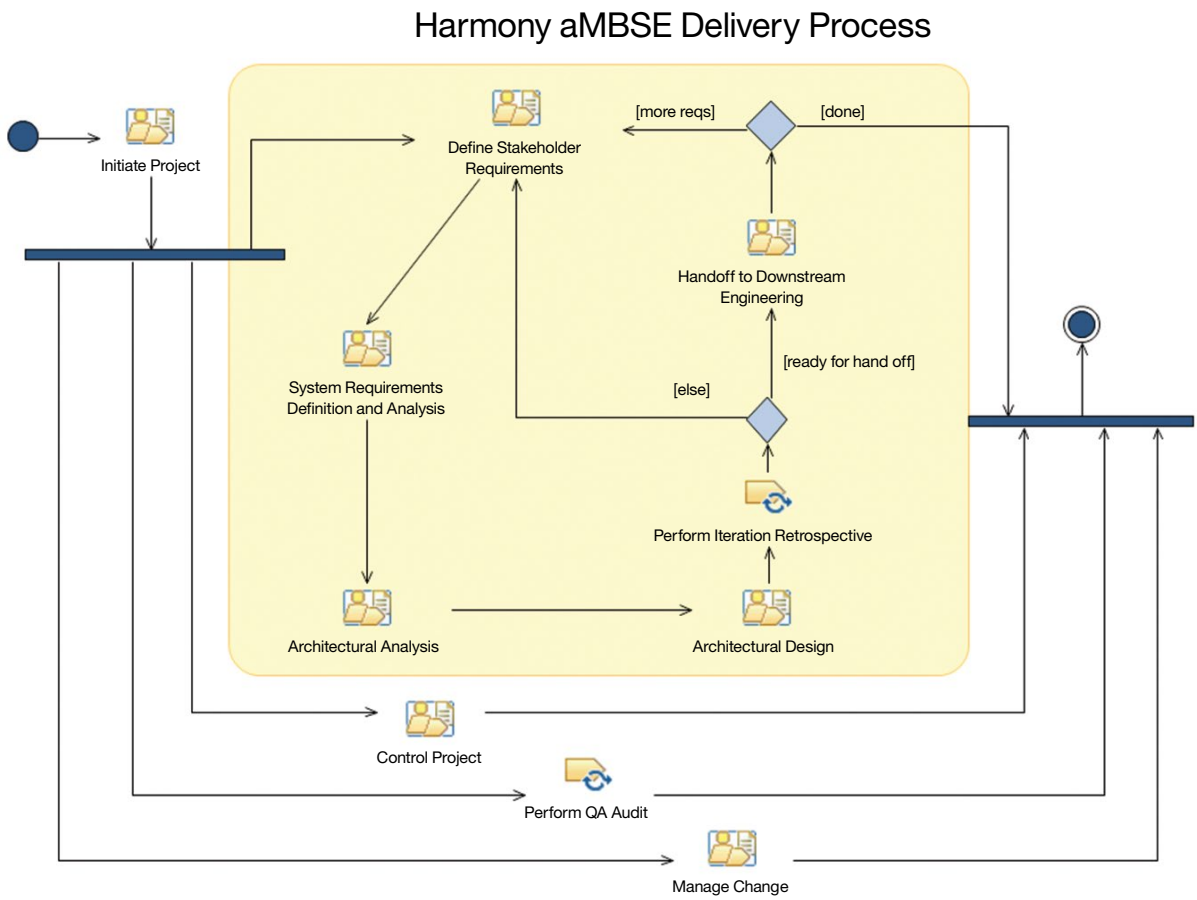

Figure 3. Harmony Agile Systems Engineering Workflow

Besides the large iteration loop for incrementally developing the systems engineering data, there are also agile practices going on within the activities. For example, an activityflow-based analysis of the use case within the System Requirements Definition and Analysis activity constructs *verifiable models* of the use cases that can be evaluated as they are being created for their consistency, correctness, completeness and accuracy. This workflow is shown in Figure 4.

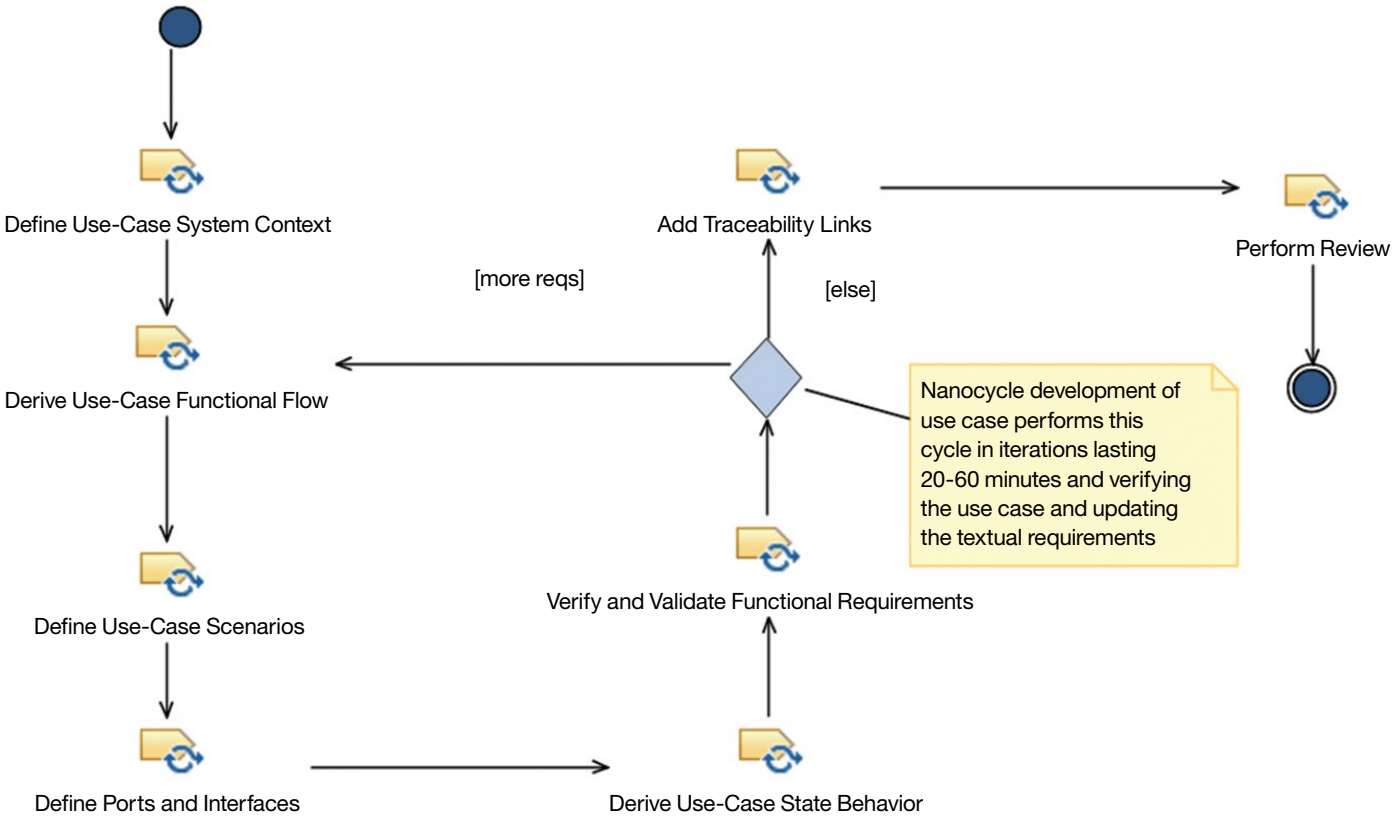

Figure 4. Flow-based Use Case Analysis

This figure shows that we define some functional flow—as described by the textual requirements—derive from this flow a small number of scenarios, construct a normative state machine, and then—through execution—verify the correctness of the requirements. This is done in very short nanocycles, which are typically 20 to 60 minutes in duration. A complex use case might require many such cycles before being fully understood (and specified) and each cycle will typically result in the identification of new or changed requirements. The point is that we are continuously verifying the correctness of the requirements as we specify them to ensure their high quality.

Agile Embedded Software Development

Virtually all of the literature on agile practices deals with applying these practices to the development of small applications running on general-purpose computers. A general-purpose computer, such as a personal computer, is designed to be flexible and to meet a wide range of end-user needs. By contrast, an embedded system is:

- *A computer system designed to perform one or a few dedicated functions, often with real-time computing constraints, embedded as part of a complete device that often includes hardware and mechanical parts.*
- *[A system] that contains at least one CPU but does not provide general computing services to the end-users. A cell phone is considered an embedded computing platform because it contains one or more CPUs but provides a dedicated set of services*[3]

Embedded systems control many devices in common use today[6]. Many embedded systems are very resource-constrained, performance-intensive, and have rigorous safety, reliability, and security needs. None of these issues are dealt with in the extensive literature on agile except in a very few cases. Nonetheless, agile can be—and has been—applied very successfully to the development of embedded systems. The Agile Harmony Embedded Software Development Process (Harmony ESW) uses iteration workflows to develop software design models and source code implementation while simultaneously applying the agile practices of Test-Driven Development and Continuous Integration. An example of an iteration workflow is shown in Figure 5. This workflow is executed on one or a small number of use cases (or user stories) and delivers a verified and working implementation. These iterations normally range from as little as a week to as long as a month to execute. Subsequent iterations elaborate additional use cases.

The detailed agile practices execute within the activities shown in Figure 5. For example, Figure 6 shows the High Fidelity Modeling workflow.

In this workflow, a single (analyzed) use case is designed, implemented, and subjected to continuous developer testing and frequent integration testing. First, a few software elements are identified, test cases are defined for them and these elements are added into the evolving collaboration of software elements. Then, in the "translate" step, source code is generated[7] and subjected to the tests just defined. Any identified defects are fixed immediately. If necessary, test coverage analysis is performed (this is required in safety critical environments) and the next set of software elements is identified. Similar to the discussion of requirements analysis in the section on Agile Systems Engineering, this work is done in a short nanocycle, generally ranging from 20 to 60 minutes in duration. Once testing reveals no defects, the software is released to team configuration management so that continuous integration testing can be done (see Figure 5).

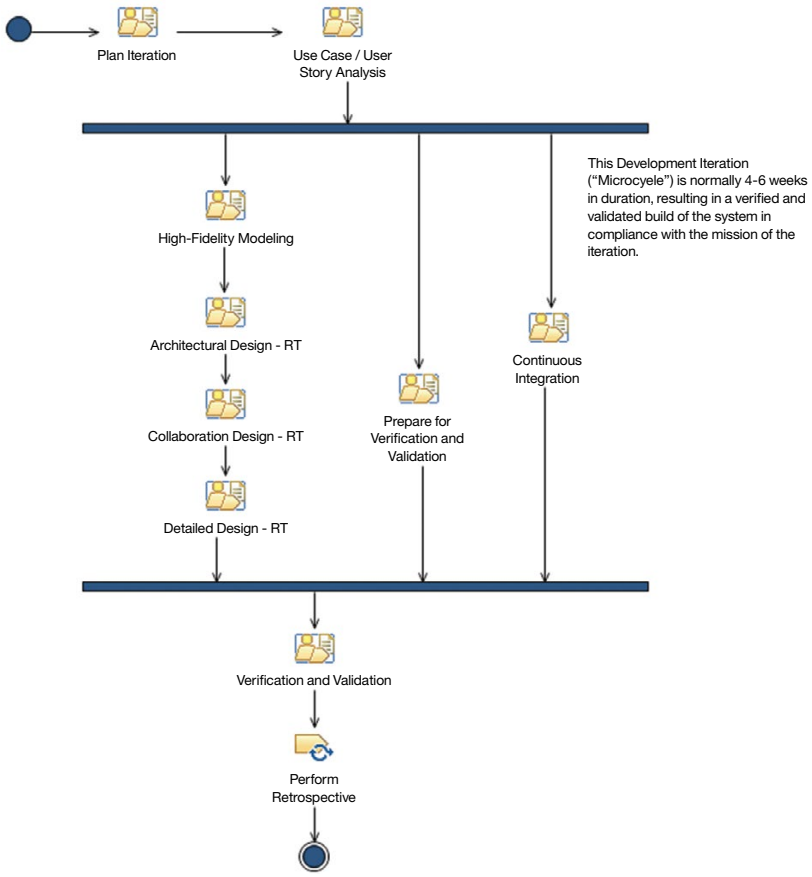

Figure 5. Harmony Embedded Software Development Iteration Workflow

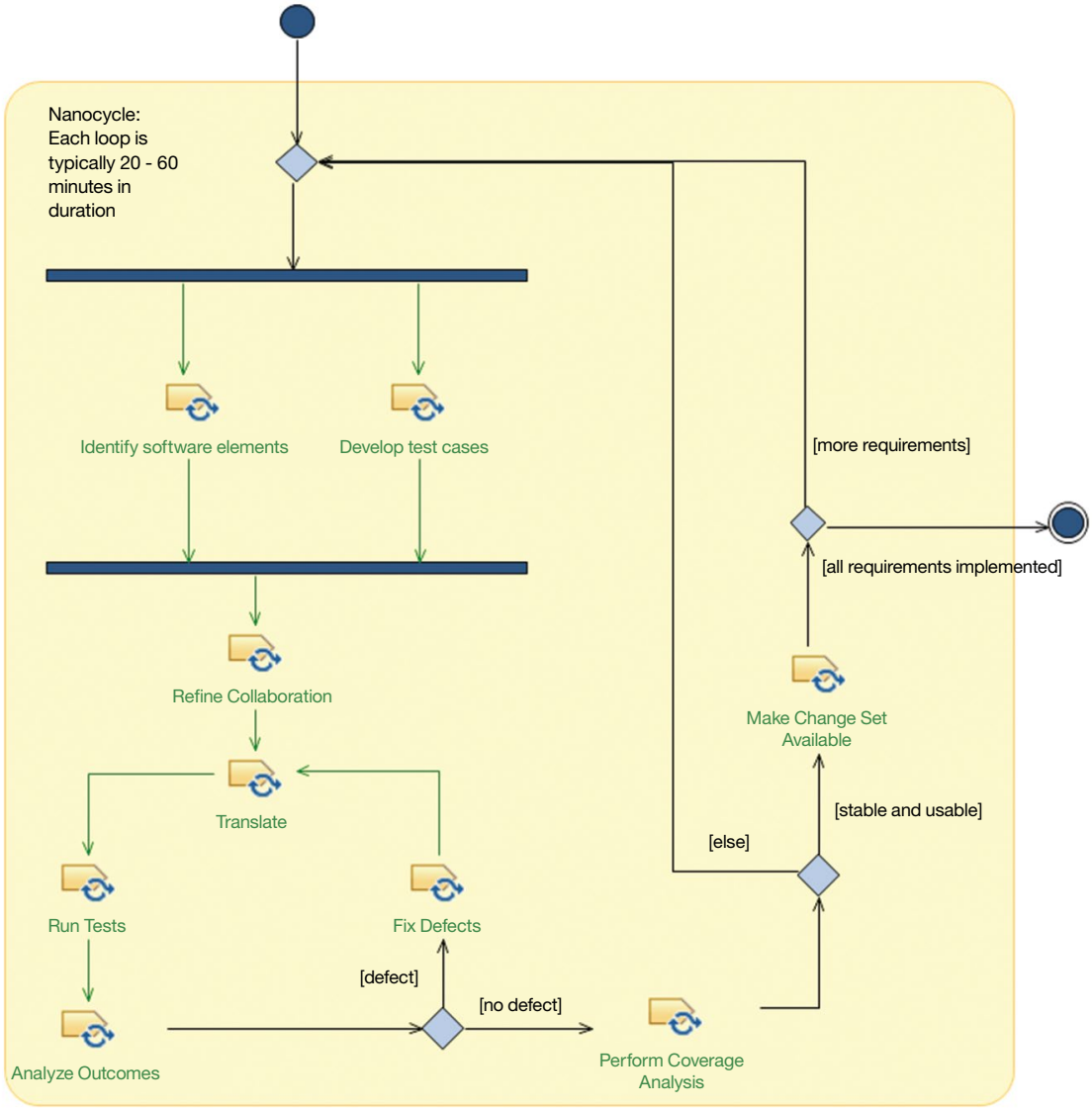

Figure 6. High Fidelity Modeling Workflow

Agile for Highly Dependable Systems

While it would be incorrect to say for IoT systems that "failure is not an option", it is certainly never a *good* option. And for many of these systems, it is a potentially dangerous and harmful option. IoT products control automotive braking systems, deliver electricity to major metropolitan areas and perform remote-controlled surgery. Failures in such systems cost more than money; they potentially cost lives. Can we really use agile methods to develop systems like these?

First, we need to understand more about what is involved. Dependability—literally defined as "our ability to depend upon the system"—has three primary aspects. *Safety* is defined as "freedom from harm". Specifically, even if the system malfunctions, people will not be harmed by a safe system. *Reliability* refers to the likelihood that a system will properly deliver its services. *Security* is defined as protection against intrusion, theft or interference. These aspects are distinct but do overlap to a degree. Each provides one pillar of dependability and all must be considered for highly dependable systems.

Systems do not become dependable by accident. Significant planning, analysis, and oversight is required during development to ensure that the system meets its dependability concerns. For example, safety analysis is required by all standards for safety critical systems, and normally safety-relevant data is captured with fault-tree diagrams and hazard analysis. Reliability is normally represented by means of Failure Modes and Effect Analysis (FMEA). There is no standard way of representing security analysis yet, although the development of a standard is underway[8].

Highly dependable systems generally require certification and, to become certified, the project must provide proof that the certification objectives have been met. This means that your development process must produce engineering data that demonstrates compliance. This includes not only the analytic data mentioned above, but also traceability to prove the consistency of the disparate data. For example, the requirements must represent (i.e. trace to) the safety analysis which must trace to the design, which must trace to the implementation, which must trace to the test cases, and so on. Figure 7 shows the required traceability between different sets of engineering data. Links labelled "R" are required for standard high-quality systems, "S" are required for safety-critical systems and "O" might be valuable but are considered optional.

These sets of data—the analyses and traceability—can be developed in an agile way. The keys to doing this are to develop the data incrementally and to continuously verify the correctness of that data. Looking back to Figure 4, it can be seen that traceability links are made as the requirements for a use-case stabilize. That is, the traceability data is developed incrementally along with the use-case data and requirements.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	Stakeholder Requirements		R			R					R				o	S	R	
2	System Requirements			R	o	S	R	R			o	R	R	S	S	S	R	R
3	Subsystem Requirements				R	S	R	R	R	S	o	R	R	S	S	S		R
4	SW/EE/ME Requirements					S	R	R	R	S	o	R	o	R	S	S		R
5	Dependability analyses						S	S	S	S	o	S	S	o	S	S	o	S
6	System Architecture							R	R	o		R	R		S	S		R
7	Control Model(s)								R	o	o	R	R	R	S	S		R
8	SW/EE/ME Design[1]									R		R	R	R	S	S		R
9	SW/EE/ME Implementation[1]										o	R	R	R	S	S		R
10	Validation Tests											R					R	
11	System Verification Tests												o	o	S	S		R
12	Component/Integration Tests													o	S	S		R
13	SW/EE/ME Tests														S	S		R
14	QA Records															S	S	R
15	CM Index																S	S
16	Validation Test Results																	
17	Verification Test Results[2]																	

[1] These are usually in separate work products, so this is really multiple items
[2] Tests occur at unit, component, integration, and system levels; again multiple items

Notation	Interpretation
R	Required for reliable, repeatable system development
S	Required for safety critical / high reliability / high security
o	Optional

Figure 7. Lifecycle Data Trace Matrix

Agile Product Development

Product development teams are coming under increasing pressure to reduce development costs and improve time-to-market while simultaneously improving product quality and capabilities. To achieve these outcomes, we need to more broadly apply the principles of agile methods to aspects of product development beyond just software. We need to develop electronic and mechanical aspects in a more agile way as well. Even more challenging, we need to improve the speed and quality with which we integrate the work products from different engineering disciplines.

Applying agile methods to whole-product development implies the ability to verify the engineering data, the engineering implementation within the various disciplines, and the integrated systems. Some good news is that new technology is available to support this kind of multi-disciplinary verification. High-fidelity UML and SysML modeling allows model development, execution and verification. Beyond that, the Functional Mockup Interface (FMI)[9] supports multi-modal co-simulation, so it is possible to build a single simulation that integrates UML for the software, SysML for the systems model, Simulink for the control model, and SimulationX for the mechanical physics model. These models may be developed both individually and collectively with agile practices, so that they can be developed incrementally with high quality and evolving capability.

Summary and Conclusion

Agile is a proven approach in software development. As software has become a primary differentiator between products, the use of agile has spread to embedded systems—which have their own challenges and needs[3].

The advent of the IoT brings with it shorter development cycles and a need for global and secure interconnection—but also new markets and opportunities. The IoT is accelerating not only the number of products that incorporate software and the amount of software they contain—but also the need for design that is responsive to operational insight, to drive both product updates and new and incremental designs.

These new conditions demand that agile is applied to other parts of the continuous engineering lifecycle, especially systems engineering and PLE. Applying agile in these domains is little discussed in the literature but it has been done. To achieve the promise of IoT and to meet its severe constraints, agile is a necessity.

Agile principles can also apply to non-software development disciplines—mechanical, electrical and so on—making it easier for practitioners in these disciplines to collaborate with agile software and systems engineering.

To successfully scale agile across the product development process will require teams to create and share information efficiently, and to have clear collaborative workflows and metrics-based insight into status and performance.

The IBM Internet of Things Continuous Engineering Solution comprises the tools, best practices and services needed to help you adopt an agile product development approach.

For more information

To learn more visit: **ibm.com**/continuousengineering

About the authors

Bruce Powel Douglass has a doctorate in neurocybernetics and over 35 years of experience designing safety-critical real-time applications in a variety of hard real-time environments. He has designed and taught courses in agile methods, object-orientation, MDA, real-time systems and safety-critical systems development, and is the author of over 6,000 book pages from a number of technical books, including *Agile Systems Engineering* (in press), *Real-Time UML*, *Real-Time UML Workshop for Embedded Systems*, *Real-Time Design Patterns*, *Doing Hard Time*, *Real-Time Agility* and *Design Patterns for Embedded Systems in C*. The Chief Evangelist at IBM Rational, he is a thought leader in the systems space, consulting with and mentoring IBM clients around the world; representing IBM at numerous conferences; and authoring tools and processes for the embedded real-time industry. Bruce contributed to both the UML and SysML specifications as well as a number of other standards. He can be followed on Twitter @BruceDouglass. Papers and presentations are available at his Real-Time UML Yahoo technical group (http://tech.groups.yahoo.com/group/RT-UML) and from his IBM page (**ibm.com**/software/rational/leadership/thought/brucedouglass.html).

Software Group
Route 100
Somers, NY 10589

Produced in the United States of America
June 2015

[1] Ambler, Scott and Lines, Mark *Disciplined Agile Delivery*, IBM Press, 2012

[2] Scaled Agile Framework http://www.scaledagileframework.com/

[3] Douglass, Bruce Powel *Real-Time Agility* Addison-Wesley, 2009

[4] Scaled Agile Framework for Lean Systems Engineering http://www.scaledagileframework.com/safe-for-lean-systems-engineering-safe-lse-news-update/

[5] Douglass, Bruce Powel *Agile Systems Engineering* Elsevier Press, 2015 (in press)

[6] http://en.wikipedia.org/wiki/Embedded_system

[7] This can be done manually with a code editor but, if the modeling is done in a highly capable tool such as IBM Rhapsody, it will be more convenient to automatically generate the code from the design.

[8] http://www.omg.org/hot-topics/threat-modeling.htm

[9] Functional Mockup Interface ps: www.fmi-standard.org

Please Recycle

WWW12345-USEN-00