Common Pathologies in Hardware Product Development

And What to Do About Them

Kevin Thompson, Ph.D.
www.kevinthompsonphd.com

September 1, 2026

Abstract. Hardware product development is characterized by uncertainty, specialized engineering skills, physical dependencies, organizational and discipline boundaries, and complex interactions among components and subsystems. Traditional development systems can amplify these inherent challenges through sequential work, long feedback cycles, hidden work, rigid plans, and increasing costs of change. This paper examines these common pathologies and how they interact to increase development cost, schedule risk, and difficulty adapting as development progresses. It then describes how an appropriately adapted Agile development system can ameliorate these problems through cross-functional teams, shorter feedback cycles, incremental integration, visible work and dependencies, adaptive planning, and deliberate management of uncertainty and risk. The objective is not to make hardware development behave like software development or simply to accelerate engineering work, but to create a development system that enables organizations to manage complexity, adapt to changing circumstances, and address problems before their consequences compound.

Contents

1 Introduction

The development of complex hardware and integrated hardware-software products is challenging for many reasons. Some of these challenges are primarily engineering in nature, arising from the disciplines involved, the physical characteristics of the product, and the interactions among its components and subsystems. Others arise from how the work is organized, planned, coordinated, tracked, and managed. These characteristics can interact in ways that create friction, lengthen feedback cycles, obscure dependencies, increase the cost of change, and make problems more difficult to identify and address before their consequences compound. This article examines these common pathologies and how an appropriately adapted Agile development approach can help organizations manage them more effectively.

1.1 The Recurring Pattern

One useful way of viewing new hardware-product development is as three major phases, each producing a progressively more mature product definition.

Phase 1: Develop Operational Prototype
Develop a prototype that demonstrates the required functionality but is not yet designed for practical manufacture.

Phase 2: Develop Manufacturing Prototype
Develop a prototype that incorporates manufacturing considerations and represents a significant step toward a manufacturable product, but may still require further engineering before production.

Phase 3: Develop Manufacturable Design
Develop the final product design that is suitable for release to the manufacturing organization.

Within each of these phases lies a typical sequence of steps, namely

1. Develop requirements
2. Develop architecture
3. Develop the design
4. Complete the detailed engineering
5. Build a prototype
6. Perform integration work
7. Perform verification on the integrated prototype
8. Perform validation on the integrated prototype

Each phase of the development process has its own project plan, which identifies the schedule of major steps and associated activities, and the deliverables to be produced throughout the process.

At the beginning of development, much of the work can be planned with reasonable confidence because relatively few decisions have yet been made. As development progresses, however, the organization acquires new information, encounters dependencies, and makes additional commitments that can invalidate earlier assumptions.

Initially, shortly after the plan has been created, the work being performed generally aligns with the plan. The project appears to be progressing as expected and meeting its milestones. Individual engineering activities are being completed, and progress is visible.

As time passes, however, two things often happen:

1. Significant problems emerge that were not anticipated.

2. The actual work being done diverges from the plan.

When significant problems emerge after substantial work has been completed, correcting them can be expensive and disruptive. Examples of such problems include:

- Missing or erroneous requirements
- Components procured from suppliers that fail to meet performance expectations
- Hardware/software integration problems
- Communication problems across organizational boundaries
- Incompatible interfaces
- Architectural problems
- Unexpected interactions among components or subsystems
- Design changes required after significant downstream work has been completed

In the more extreme cases, problems may require another costly prototype or hardware iteration to resolve.

1.2 Why These Problems Matter

The problems described above matter because they increase the time, effort, and expense required to develop a product and make it more difficult for the organization to respond effectively when circumstances change. The consequences tend to compound as development progresses.

More decisions become dependent on earlier decisions. More work is completed that may need to be scrapped and replaced. More components may have been procured, only to be discarded. More designs may have been developed that now need to be replaced, potentially invalidating downstream plans and requiring costly changes to tooling or other production resources. Communication and coordination problems can create additional rework and delay. Long feedback cycles can allow problems to persist while downstream work continues to accumulate.

The result is that the cost of addressing problems tends to increase as development progresses. Problems that could have been resolved relatively easily early in development can become expensive and disruptive when discovered after substantial downstream work has been built upon them.

1.3 The Central Question

What makes hardware development difficult to manage effectively as development progresses? Typical answers include:

- The project is complex.
- Hardware takes a long time to develop.
- Specialized engineering skills are required.
- Requirements change.

- Teams and disciplines are organized separately.
- Communication and coordination are difficult.
- Someone made an incorrect assumption or decision.

While these explanations have some merit, they don't fully explain why problems can persist and compound during development.

The central thesis of this paper is that many of the difficulties encountered in hardware development are not simply inherent characteristics of the product or the engineering disciplines involved. They are amplified by the development system itself. Sequential activities, organizational and discipline silos, specialized skills, long feedback cycles, physical dependencies, communication and collaboration challenges, hidden work, rigid plans, and increasing costs of change can interact in ways that increase friction, delay feedback, and make the organization less able to adapt as development progresses.

An appropriately adapted Agile development system can ameliorate these pathologies by changing how work is organized, coordinated, planned, and executed.

2 The Traditional Hardware Development Model

The traditional model of Section 1.1 makes intuitive sense. The steps follow a logical sequence, and they provide useful milestones for measuring progress. They also provide logical opportunities for reviews or stage-gate decisions. While the actual development is rarely perfectly linear, the overall management system is often organized around these phases.

2.1 The Hidden Assumption

The difficulty with the linear sequence is that it assumes that sufficient knowledge can be obtained at each stage to make stable decisions about what subsequent stages will need, before proceeding to the next. That assumption breaks down in new-product development because an organization cannot know in advance everything it will need to learn. Developing a new product inevitably involves technical, manufacturing, integration, and other questions whose answers may not become apparent until development is underway.

Some questions cannot be answered until a sufficient level of system integration has been achieved. Depending on the question, this may require components to have been procured, subsystems to have been built and made capable of interacting, software to have been integrated with the hardware, and a sufficiently representative physical prototype to exist.

In other words, the development process needs information that the process itself may not make available until relatively late.

3 Why Problems Remain Hidden

Problems can remain hidden for multiple, interacting reasons. No single root cause explains all cases.

3.1 Overconfidence in Assumptions

Engineers and managers have to make assumptions in order to create plans and manage work. Many of these assumptions are reasonable and necessary, but some may prove to be incorrect. An assumption can seem obvious or well-founded, yet if it remains untested, discovering that it is wrong later can result in expensive delays and rework.

3.2 Sequential Development

The linear sequence of development steps described earlier leaves many assumptions untested until late in development. It also provides limited opportunity for feedback from later discoveries to influence earlier decisions and work. As a result, problems can propagate into later stages before they are recognized.

3.3 Silos, Communication, and Collaboration

Discipline-related silos are common. They include disciplines such as electrical engineering, mechanical engineering, optical engineering, antenna design, firmware development, and software development.

Organizational silos are also common, and reflect team boundaries or organizational divisions. In addition to silos associated with different engineering disciplines, there may be separate systems engineering, manufacturing, quality, supplier management, and supply-chain organizations.

Each group that forms a silo may optimize its own work while system-level problems remain unresolved. Organizational boundaries tend to introduce some degree of friction into communication and collaboration. Different organizational units may develop their own vocabulary, concepts of what planning means and what constitutes a plan, and expectations regarding timeliness. All of these differences introduce friction when two groups do not understand each other as well as they understand themselves.

Communication and collaboration across discipline and organizational boundaries is therefore generally more difficult than within a coherent group whose members share terminology, expectations, and ways of working. It is easy to make plans based on the assumption that communication and collaboration are frictionless, but that is rarely the case. Unless care is taken to prevent it, misunderstandings will occur. Dependencies that cross group boundaries may be overlooked, misunderstood, or poorly coordinated, sometimes leading to finger-pointing over who is at fault.

3.4 Specialized Skills

The development of a new product invariably requires multiple engineering skills. The variety of skills required can be large, and not all specialists will be involved in every phase of every product development effort. Typically, a subset of specialists is routinely involved in the day-to-day development work, while others contribute on an as-needed basis.

Coordinating the involvement of all of the engineering specialists required for a particular product is non-trivial. Their availability must be coordinated with the needs of the development effort, and those needs can change frequently as reality departs from the plan.

Here again, communication and collaboration can be difficult. For example, mechanical and electrical engineers may need to collaborate closely on a robotics project even though they have different technical vocabularies and may not understand each other's work in depth. Unless they are careful in their interactions, it is easy for either or both to make erroneous assumptions about what the other will provide, when it will be provided, or what constraints apply.

3.5 Physical Dependencies and Long Lead Times

Long lead times are common in hardware development, where lengthy dependency chains may intervene between making a design decision and obtaining something that can be tested.

For example, suppose the design of a custom circuit board takes four weeks and the supplier requires another two weeks to fabricate and deliver it. Six weeks may therefore pass between the beginning of the board design and having the completed board in hand for testing. Only then can assumptions that were made at the start of the design phase be tested against the actual hardware. If any of them fail, the process may need to repeat from the start.

The assembly of a prototype may be at the end of a substantial number of long dependency chains, which is an even more extreme case of the long lead time problem. Again, there can be a substantial interval between making a decision that informs the prototype design and obtaining evidence about its validity.

3.6 Uncertainty

Uncertainty has direct implications for cost, schedule, and development decisions. Engineering organizations inevitably encounter unknowns when developing new products.

Uncertainty can arise from almost anywhere. Examples include incomplete or contradictory requirements, interactions between components or subsystems, unexpected behavior at interfaces, reactions to environmental stresses such as heating and cooling, manufacturing and tooling issues, and so on.

Known unknowns can often be treated as risks and addressed through the development plan. Where possible, they should be investigated early, before assumptions associated with them have shaped substantial downstream work.

Unknown unknowns cannot be identified and planned for individually because, by definition, they are not known in advance. No amount of upfront investigation can eliminate them entirely. Nevertheless, their occurrence should be anticipated when planning the project. A development plan that assumes every activity will proceed as expected, with no capacity for unexpected technical problems, rework, or other disruptions, is unlikely to remain realistic as development progresses. The development plan should therefore reserve some capacity for unforeseen developments, with the appropriate amount depending on the nature of the project and the degree of uncertainty involved.

A capacity reserve, however, is not a substitute for addressing problems promptly. When an unexpected issue emerges, it should be investigated promptly and its appropriate response determined rather than simply deferred. The purpose of the capacity reserve is to provide the organization with capacity to respond to the unexpected, not to provide an excuse for postponing problems.

4 The Compounding Effect

The various pathologies described in Section 3 do not simply operate independently. They interact with one another, and their effects can compound over time.

The inevitable specialization of skills and organizational silos means that different teams and individuals are involved in different aspects of product development. The ability to communicate and collaborate across these boundaries becomes important, and failures slow the pace of work and may force rework of some deliverables. The friction of collaboration can then delay integration work, which in turn means that feedback arrives later and more uncertainty remains unresolved.

The result is that important decisions continue to be made while significant uncertainty remains, while work accumulates around assumptions that may later prove incorrect. As these effects compound, the organization has fewer options for responding without incurring additional cost, rework, and schedule disruption.

Figure 1 shows how pathologies compound and result in schedule slippage and late discovery of issues.

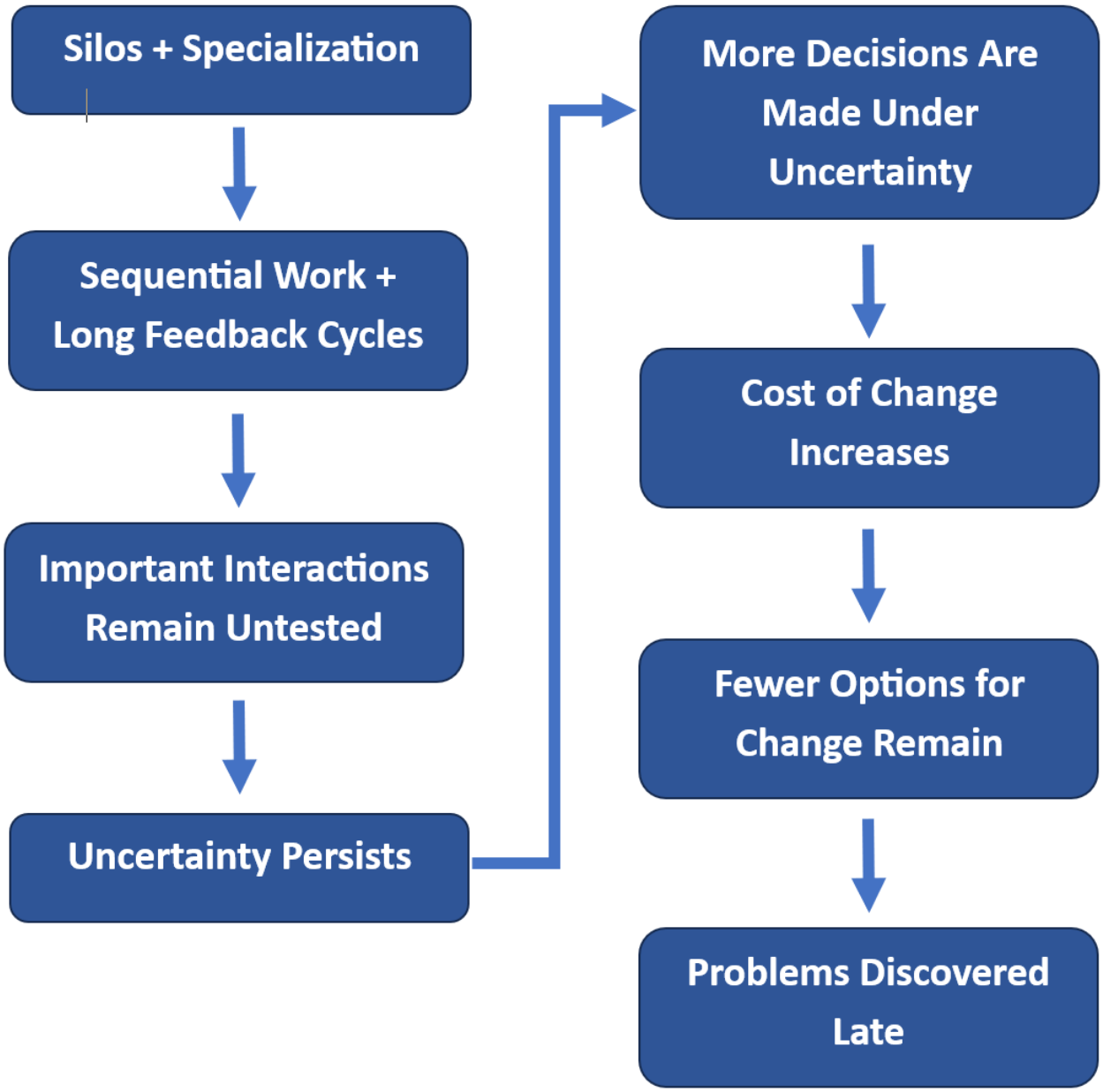

Figure 1. The interaction of development-process characteristics compounds their effects, increasing cost and reducing the ability to adapt

The longer these effects persist, the more they compound. As development progresses, more work becomes dependent on earlier decisions, the cost of change increases, and fewer alternatives remain available.

However, the organization cannot simply stop making decisions while uncertainty remains. Development continues, and new decisions are made on the basis of the information available at the time. Those decisions create additional dependencies and commitments, increasing the eventual cost of changing direction.

4.1 Where Problems Become Visible

Problems may surface at any point in development. However, there are particular points at which problems that have not been detected earlier are more likely to become visible.

The success of a set of subsystems does not guarantee system-level success. Mechanical, electrical, firmware, and software subsystems may all pass tests intended to reveal any faults, yet the integrated product can still fail. Thus, integration becomes a critical point for detecting problems.

Any hardware product of any complexity contains multiple interfaces, dependencies, and interactions, such as:

- Mechanical interfaces, where subsystems physically connect
- Electrical interfaces, where electrical connections are made
- Software interfaces, between software modules and device-resident firmware
- Firmware/hardware interfaces, where the firmware interfaces with the hardware in which it resides
- Timing dependencies, where subsystems that have different clock speeds must interface across boundaries
- Power dependencies, where subsystems draw power from the system's power supply
- Thermal interactions, where physical components couple to the ambient cooling solution in order to maintain appropriate operating temperatures
- Communication interfaces, where the product communicates with external devices, such as network-attached devices that control or use the product

All of these interfaces, dependencies, and interactions contribute to system-level behavior, and all must function correctly for the product to perform as intended.

Problems are often found when subsystems are first integrated with one another. The sooner integration testing is performed the sooner integration problems can be discovered and addressed. However, there is also no substitute for testing a completed prototype with all relevant subsystems present and integrated. Some system-level problems cannot be discovered until the necessary components and interactions actually exist.

Late discovery of integration problems can trigger a cascade of consequences. A problem may require redesign, which in turn creates rework and downstream changes. In more serious cases, another prototype iteration may be required, causing additional schedule disruption and cost.

In short, the cost of late discovery comes not only from the problem itself, but from everything that has been built on top of the assumption that turned out to be wrong.

5 What Agile Changes

An Agile development system built around the Scrum framework provides opportunities to ameliorate the problems described in the preceding sections. A key element of this approach is organizing the engineering work around stable, cross-functional Scrum Teams. The teams work in short Sprints, typically in the range of two to four weeks in length.

The Scrum Team working in a Sprint provides the basic organizational unit for the development effort. Planning and coordination then extend in two dimensions: time and scale.

The first dimension is time. While each Scrum Team develops a detailed plan for its next Sprint, teams can develop longer-term Release Plans which provide a broader view of the work expected over the coming months. These plans are periodically revised as experience is gained and new information becomes available.

The second dimension is scale. It is frequently the case that multiple Scrum Teams, each with a particular focus such as a subsystem, must work in parallel and coordinate their work over time to create the complete product. The Release Plan provides a mechanism for coordinating this work across teams and with other organizational groups involved in creating the product.

5.1 Sprint Planning

In brief, a Scrum Team plans its next Sprint at the beginning of the Sprint in a Sprint Planning meeting. The team has already identified a candidate set of deliverables to be considered for inclusion in the Sprint plan. The team estimates each deliverable and determines which deliverables can reasonably be accomplished during the Sprint, taking into account the estimated effort, available capacity, dependencies, and the specialties of the team members.

One parameter that can be useful in Sprint planning is Velocity, which is the amount of work the team has historically been able to accomplish during a Sprint. This provides a basis for forecasting how much work the team may reasonably be able to accomplish in the next Sprint.

5.2 Release Planning

A Release Plan is a longer-term plan that spans multiple Sprints and, commonly, multiple teams. It is the mechanism that expands in both the time and scale dimensions.

The span can be chosen based on organizational need; for hardware development, the span is often the length of a major development phase.

The structure of a Release Plan is shown in Figure 2.

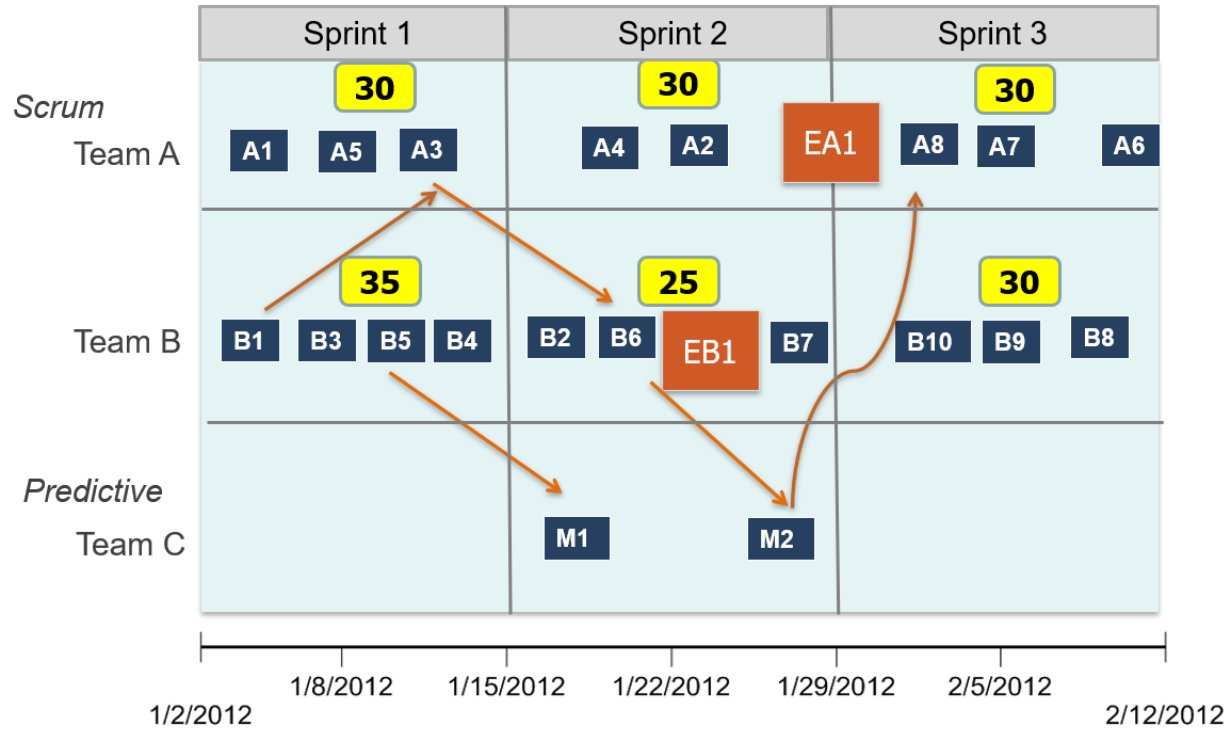

Figure 2. Sample Release Plan

Each team or department has a row in the plan. In the figure, the top two rows are for Scrum Teams A and B.

The bottom row represents Team C, which does not use Scrum to organize its work but instead uses a more classical predictive project-management approach. This illustrates a situation that commonly occurs in hardware development: Scrum Teams may need to coordinate their work with organizational groups that use different development and planning approaches.

The squares at the top of the top two rows show the amount of work forecast to be available in each Sprint, based on the team's historical Velocity and other relevant factors. The Release Plan uses a more conservative capacity assumption than the actual Velocity forecast for each Sprint, leaving some reserve for problems and disruptions that may affect the schedule. The appropriate amount of reserve depends on the nature of the work and the degree of uncertainty resolved.

The small squares in the middle of each row represent deliverables each team will produce over time, in time order. The larger squares represent deliverables that are too large to complete in a Sprint, and which must be decomposed into smaller deliverables before the Sprint in which work on them is intended to begin. Release Planning commonly includes such larger deliverables because it is not usually possible to decompose the entire scope of a development phase into small deliverables before Release Planning begins.

The arrows show dependencies between deliverables, and point from predecessor to successor. The sample plan shows cross-team dependencies, which require particular attention because they cross organizational boundaries and therefore introduce additional coordination requirements.

Dependencies can also be shown between deliverables within a single team, if desired.

The purpose of Release Planning is not to create a detailed long-term commitment that is expected to remain unchanged. Rather, it provides a framework for coordinating work across teams and organizational boundaries while allowing the plan to evolve as development proceeds. The plan makes dependencies visible, provides a basis for identifying potential conflicts before they become urgent, and allows new information to be incorporated without requiring the entire development plan to be recreated.

5.3 An Agile Hardware Case Study

The practices described above have also been applied in an Agile transformation of a hardware development organization that developed a mass spectrometer, as described in “Eleven Lessons Learned from Agile Hardware Development” [1]. In that effort, the development organization adapted Scrum practices to accommodate specialized engineering skills, component-oriented development, capacity constraints, and progressive product integration.

5.4 Different Agile Practices Address Different Problems

Additional characteristics of such an Agile development system and how those characteristics ameliorate the problems are described below.

5.4.1 The Problems of Sequential Work and Feedback Cycles

A major challenge of the sequential model described in Section 1.1 is that it contains no obvious mechanism for incorporating feedback from the discovery of problems into subsequent development work. In contrast, a problem encountered during one Sprint can be incorporated into the team's work for a subsequent Sprint.

The potential schedule impact is accounted for in the Release Plan through the use of a capacity reserve. The reserve shown in Figure 2 is an illustrative capacity assumption; the appropriate amount depends on the nature of the work and the degree of uncertainty involved. This reserve may be sufficient to absorb the impact of many problems, although corrective actions will still be incorporated into the Release Plan as new deliverables.

The status of work in the Release is shown in Burnup charts, as in Figure 3.

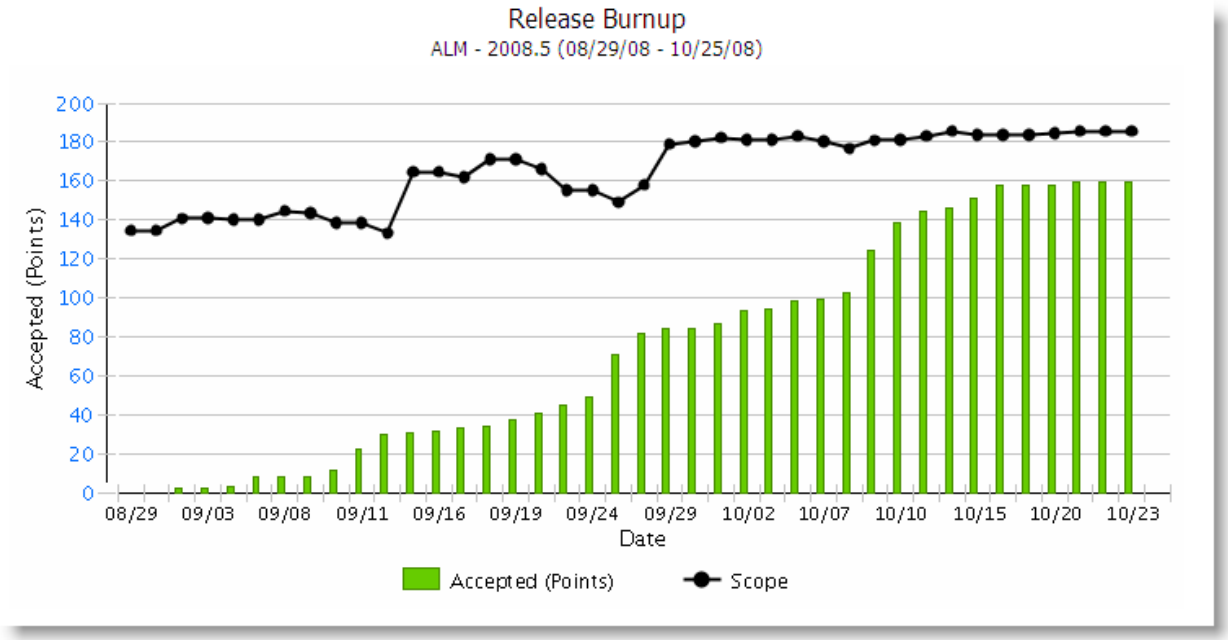

Figure 3. Sample Burnup Chart to Show Release Status

The black curve at the top is the scope line. It shows the amount of work represented by the current Release scope over time, as that scope evolves in response to unexpected difficulties and intentional changes.

The bars at the bottom show the amount of work completed over time.

Given those two curves, the expected completion date of the Release cycle can be forecasted by extrapolating both curves to the point where they intersect.

The important characteristic of this approach is that both the work completed and the work remaining can change as development proceeds. The Release Plan therefore provides a mechanism for incorporating new information rather than assuming that the original plan will remain valid throughout development.

5.4.2 The Problem of Silos

The Agile development system provides two mechanisms for reducing the problem of organizational and discipline silos. The mechanisms are cross-functional teams and Release Planning.

5.4.2.1 How Teams are Defined

For the purposes of this paper, a “team” is a group of people who

- Are typically 3—9 in number.
- Have some kind of focal area in which they work.
- Collaborate closely in doing their work.

- Collaborate via informal and as-needed conversations, as opposed to primarily through scheduled meetings.
- Share a common understanding of what they, as a team, will be working on in the near future; i.e., have some concept of a schedule that they have worked out and are following.
- Are preferably co-located.

Why the range of 3—9 people? First, because groups of fewer than three people provide little opportunity for the interaction-cost effects discussed here to become significant. Second, because the number of communication paths in a group of *N* people is $N(N-1)/2$, which grows as the square of *N*. As a practical matter, it becomes increasingly difficult for the team to coordinate and collaborate internally as the team size grows beyond about nine people.

The core concept behind team definition is that interaction costs are higher between teams than they are within a team. To collaborate on some deliverable across team boundaries requires planning the collaboration, scheduling it, and making the handoffs work. These steps are more formalized and time-consuming than is the case for collaboration within a team. Also, if some issue disrupts the plan, the re-planning work is again more costly than the informal adjustment that would occur if a similar situation arose within a single team.

An important economic objective in organizing people into teams is to minimize the interaction cost across the set of collaborating teams by minimizing the frequency with which interactions must occur across team boundaries.

In hardware development, this approach naturally leads to organizing teams around product capabilities or subsystems rather than engineering disciplines. For example, we can imagine a group of people who are responsible for working on a subsystem. If they are organized by engineering discipline, then much of their work will require frequent interaction with people in other disciplines, creating a relatively high interaction cost. If the team is cross-functional and aligned with the subsystem, then the interaction cost between the team members is lower.

Thus, when the work routinely requires multiple engineering disciplines, cross-functional teams are an important element of an interaction-cost-based approach to team organization. Scrum emphasizes cross-functional teams for much the same practical reason: a Scrum Team should contain the skills routinely required to produce meaningful deliverables representing portions of the product. For example, one Scrum Team might contain specialists in mechanical engineering, electrical engineering, optical engineering, and firmware because many of the deliverables that comprise the product require this combination of skills.

The friction that can arise from communicating across engineering disciplines is reduced when engineers from different disciplines work together routinely in a team, giving them the opportunity to develop a shared vocabulary and a common understanding of the interfaces between their areas of responsibility.

Skills that are required only intermittently are generally not included as permanent members of the Scrum Team, since they would otherwise be underutilized. Their involvement must instead be arranged when their skills are needed.

This does not eliminate organizational boundaries. The second mechanism addresses the organizational boundaries that remain outside the cross-functional Scrum Team: Release Planning.

5.4.2.2 Release Planning

Release Planning provides a mechanism for managing these remaining boundaries. A typical Release Plan will involve some mix of Scrum Teams and teams or departments that do not use Scrum. Scrum Teams will generally find it easier to coordinate with one another because they share a common process, concepts, and terminology. Coordination between a Scrum Team and a group using a different development model may require more deliberate communication, particularly to surface assumptions, dependencies, and differences in expectations.

The Release Planning meeting brings together representatives of all teams and departments that must be involved in the development phase being planned. Working together, they develop a Release Plan that incorporates the work each group must perform in order to complete the development phase successfully. Making cross-team dependencies visible is a primary focus of these planning meetings.

The conversations and learning that occur during Release Planning provide substantial value beyond the Release Plan artifact itself and are a major, intentional outcome of the meeting. Participants have an opportunity to understand how the other groups work, identify assumptions and dependencies, and establish working relationships before those dependencies become urgent.

5.4.3 The Problems of Hidden Work and Risk

In this approach, all work performed by the teams working on the product must appear in the Sprint and Release Plans. For example, while there is nothing wrong with keeping a risk register, the work required to investigate or mitigate those risks must show up explicitly in the Sprint and Release Plans. All research, supplier validation, and other work that must be done, even if it does not contribute a CAD drawing or a circuit element, must be represented in the plans as deliverables with associated estimates.

Making work visible is a central principle of Agile development and is valuable regardless of the development methodology being used.

5.4.3.1 The Problem of Latency

A common example of both making work visible and dealing with long lead times is that of ordering a part from a supplier, when there is a significant delay between placing an order and receiving the part. If this gap is longer than can reasonably be accommodated in a Sprint, the order placement should appear in the plans as a deliverable of its own. A second deliverable

should then be placed later in the plans for when the part is received, possibly followed by a deliverable for testing the part for acceptability.

In other words, latency—the delay between one action and another—should not be hidden within a single deliverable unless it is very short. Instead, the latency should be represented explicitly as the time between a deliverable that initiates the process and a subsequent deliverable that represents its completion.

5.4.4 The Problem of Rigid Plans

Traditional project planning, which emphasizes schedule artifacts such as Gantt charts and sequential processes such as that in Section 1.1, suffers from several common problems.

One problem is that they provide no natural mechanism for incorporating feedback from the work into the plan, as described above.

Another is that reality diverges from the plans fairly quickly, because unexpected problems appear early on in development and generally continue to appear throughout.

A useful plan should provide a reasonably reliable indication of what is likely to happen in the future. A plan that no longer reflects reality well ceases to serve that purpose and becomes a problem in its own right.

The obvious solution is to keep updating the plan to reflect the current understanding of reality, but this is easier said than done.

A classic project schedule contains a set of tasks and task estimates, assembled into a chronology based on resources, dependencies, and other constraints in addition to the tasks themselves. The creation and maintenance of such a schedule is typically the responsibility of a project manager.

The creation of such a schedule is itself a substantial amount of work. Changing the schedule is also commonly a substantial amount of work, as new tasks appear and disrupt it. Even obtaining the status information needed to update the plan can be a nontrivial effort, because the team members doing the work tend to be somewhat insulated from the details of the project schedule and may not readily have the information the project manager needs.

In short, the cost of change for the project schedule itself is high. Maintaining such a schedule to mirror a constantly changing reality can rapidly become impractical. The result is that the organization ends up with schedules that look convincing but are largely fictional.

The Agile development system has a very different concept of schedule. Each team has a row in the Release Plan. Each Scrum Team plans its next Sprint at the start of the Sprint by refining the Sprint plan that was originally formulated during Release Planning. It can put into that Sprint any deliverables that are appropriate, including work to address problems discovered in the

previous Sprint. The Release Plan therefore automatically updates as part of Sprint Planning, and remains current without requiring a separate replanning activity.

Little effort is therefore required to keep the Agile plan aligned with reality.

5.4.5 The Problem of Uncertainty

Because an Agile development system makes it relatively easy to incorporate new information into planning and execution, it can be used deliberately to reduce uncertainty throughout development. Not only can unanticipated problems be addressed shortly after their discovery, but uncertainty-reduction activities can be incorporated into the plan both at the beginning and as needed throughout development.

An Agile development process can expose and address uncertainty through

- Customer feedback
- Mitigation of known risks
- Early prototypes
- Frequent and incremental subsystem integration
- Frequent hardware/software integration
- Progressive verification
- Working demonstrations
- Short development increments
- Explicit tracking of technical risks

The goal is to make important unknowns visible while there is still time and money to respond to them.

5.4.6 The Problem of Expensive Late Changes

Late changes tend to be expensive changes because more downstream work and decisions may depend on the thing being changed. Late changes can result from problems discovered late in the development cycle, as well as from new information or changing requirements that emerge as development progresses.

The techniques described in this section do not eliminate the possibility of late changes, but they reduce the likelihood of encountering them and improve the organization's ability to respond when they do occur. They do so by reducing risk, uncertainty, and schedule unpredictability.

6 Agile for Hardware is not Software Scrum

The Agile development system described in this paper operates at two levels. At the team level, Scrum provides the framework for organizing and executing the engineering work. At the level

above the individual teams, Agile program management [2] provides the mechanisms needed to coordinate work across teams and organizational silos.

The Agile program management level is largely agnostic as to the type of product being developed. The same principles can be applied to software products, hardware products, and complex hardware/software products.

The same is not true for the team level, where Scrum is employed. It would be a mistake, however, to take a Scrum process designed around the characteristics of software development and apply its practices to hardware development without adaptation. Hardware development differs from software development in too many important ways for software-oriented Scrum practices to be transferred unchanged. The differences in development environment and appropriate practices are too numerous and detailed to describe here, but examples include:

- Work Decomposition: Hardware development often requires work to be decomposed around technical capabilities and product components or subsystems rather than simply around user-facing functionality.
- Planning: Specialized skills and types of work can make capacity planning and individual skill availability more important than simply planning work to a velocity-based capacity limit.
- Team collaboration: Hardware teams benefit from cross-functional organization, but specialized skills mean that assumptions about swarming and interchangeable team members that may work well in software development do not necessarily apply.

These and other adaptations are discussed in detail in “Agile Processes for Hardware Development” [3].

The objective is therefore not to apply software Scrum to hardware, but to adapt Agile practices to the characteristics of hardware development while retaining the underlying principles of short feedback cycles, visible work, incremental planning, and rapid response to new information.

7 A Practical Framework for Hardware Development Leaders

A useful starting point for hardware-development leaders is to ask where the development system makes it difficult to obtain information, coordinate work, or respond to problems. The earlier important information becomes available, the more options the organization generally has for responding to it, and the lower the resulting cost of change is likely to be. The objective is therefore to identify the characteristics of the development system that create the greatest friction or risk, and determine where they can be ameliorated.

7.1 Which Feedback Loops are Longest?

Examples of lengthy feedback loops include:

- Prototypes
- Supplier and part qualification
- Supplier and supply-chain dependencies and lead times
- Integration
- Testing
- Design-review cycles.

The key to addressing lengthy feedback loops is to create smaller feedback loops within them that provide useful information earlier, wherever practical. This does not make the underlying long feedback loop disappear, but it can allow problems and other important information to emerge earlier, when the organization has more options for responding to them.

7.2 Where are the Organizational Boundaries?

Identify where work crosses interfaces between organizational units. These boundaries may occur between engineering teams, across hardware/software interfaces, between engineering and other parts of the organization, or between the organization and its suppliers. Every interface introduces potential interaction costs and opportunities for misunderstanding or failure. Common root causes include:

- Communication failures caused by failure to plan together.
- Communication failures caused by differences in mindset and vocabulary.
- Different concepts around how to plan and track work.
- Different assumptions about timeliness.

These issues can lead to consequences such as:

- Broken dependencies: Missed handoffs because items are not available when needed.
- Confusion: Different groups fail to understand each other at a fundamental level, potentially causing a wide variety of problems.
- Incompatible planning assumptions: Different groups organize and communicate their work in ways that can cause synchronization failures.
- Dropped balls: Different groups have different assumptions about timeliness. One group thinks "as soon as possible" means anytime tomorrow, while another thinks it means anytime in the next four weeks.

Once these boundaries have been identified, examine whether the associated interaction costs can be reduced through the team-organization and Release Planning techniques discussed in Section 5.4.2.

7.3 Which Unknowns Have the Greatest Potential Impact?

Saying that every unknown should be addressed as early as possible is painting with too broad a brush. A more refined approach is to investigate first those unknowns that are most likely to lead to expensive consequences later, while deferring less consequential unknowns until later.

This is an important point, because it is unlikely that all unknowns can be addressed in the first Sprint. Some sequencing is required. In general, we want to investigate “high-leverage unknowns” before “localized unknowns.” A high-leverage unknown is one whose resolution can affect substantial amounts of downstream work or constrain many downstream decisions. A localized unknown primarily affects one component, activity, or relatively small set of decisions.

When assessing the impact of unknowns, which contain as-yet untested assumptions, it is important to ask questions such as:

- How uncertain is the assumption? What is the probability that it is wrong?
- What are the consequences if it is wrong? How much of the product development effort would be affected?
- What is the cost of learning late? How much additional work and expense will have accumulated, only to be scrapped, before we find out?

Consider the following examples.

Example A. Will the enclosure’s cosmetic finish meet the customer’s preference?

There may be substantial uncertainty as to the customer's preference, but the consequence of being wrong is relatively small. This would therefore be a localized unknown.

Example B. Can the thermal architecture keep the detector within its operating temperature range?

If there is substantial uncertainty about the thermal architecture, and failure could affect mechanical design, component selection, electronics, and system performance, then much of the product would be affected and the cost of learning late would be high. This would be a high-leverage unknown.

Example C. Will the firmware communicate correctly with the new sensor?

If the interface is well understood and based on a proven component, uncertainty may be relatively low, and its impact localized. However, if the sensor is new and the interface has never been demonstrated, this could become a high-leverage unknown. The degree of impact depends on both the uncertainty and the consequence.

7.4 Which Assumptions Have Not Yet Been Tested?

This question should be asked throughout the development cycle, as new uncertainties can appear even as earlier ones are resolved. The prioritization process described in the preceding section should be applied routinely as work progresses.

8 Conclusion

Every engineering organization wants work to proceed smoothly and quickly, resulting in the shortest practical time to market. This paper has described common pathologies encountered in complex hardware development and provided strategies for ameliorating them.

Hardware development organizations operate in an environment characterized by uncertainty, specialized disciplines, physical dependencies, long lead times, and complex interactions among components and subsystems. These characteristics make some degree of late discovery inevitable. However, the development process itself can either amplify this problem or help mitigate it.

Traditional development approaches tend to organize work into sequential phases and functional boundaries. As development progresses, decisions accumulate and become increasingly interconnected. Feedback from integration, testing, and other forms of empirical learning may therefore arrive only after substantial work has been completed. At the same time, organizational and discipline silos, communication and collaboration difficulties, and the increasing cost of changing established plans can make it difficult to respond effectively when new information does emerge.

The objective of an Agile approach to hardware development is not to eliminate these characteristics, nor is it simply to make engineering work proceed faster. Its more fundamental contribution is to create a development system that enables an organization to manage complexity more effectively: coordinating work across disciplines and organizational boundaries, exposing problems and dependencies sooner, adapting plans as circumstances change, and reducing the accumulation of rework, risk, and uncertainty as development progresses.

This requires more than applying software-development practices unchanged. Agile practices must be adapted to the realities of hardware development, including specialized engineering skills, physical dependencies, component and subsystem development, and the different ways in which progress can be demonstrated and validated. As described in “Agile Processes for Hardware Development,”[3] the principles underlying Scrum can transfer effectively to hardware development even though some software-oriented practices require significant adaptation.

The goal is not to make hardware development behave like software development. The goal is to create a development system that helps the organization manage complexity, coordinate

effectively, adapt as circumstances change, and address problems while there is still time and flexibility to act.

References

1. Thompson, K. "Eleven Lessons Learned about Agile Hardware Development." January 2016.
2. Thompson, K. *Solutions for Agile Governance in the Enterprise (Sage): Agile Project, Program, and Portfolio Management for Development of Hardware and Software Products*. Sophont Press, 2019.
3. Thompson, K. "Agile Processes for Hardware Development." August 2015.